UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CARRERA DE CIENCIAS DE LA COMPUTACIÓN

Algoritmo de Ordenamiento por Ranking en Paralelo
Computación Paralela (CS4052)

INTEGRANTES

Leandro Blas, Juan José

Choqque Mejia, Fernando Adriano

Zuloeta Salazar, Marcelo Mario

PROFESOR(A)

Fiestas Iquira, Jose Antonio

Lima - Perú

Diciembre, 2024

TABLA DE CONTENIDO

Pág.

I Introduccion **1**

1.1 Motivacion 2

1.2 Explicación Breve del Algoritmo 3

II Metodo **5**

2.1 PRAM 5

2.2 Código en C MPI 7

III Resultados **13**

I

Introduccion

El ordenamiento de datos es una operación fundamental en la computación y se aplica en una variedad de contextos, desde la organización de grandes bases de datos hasta la optimización de algoritmos en sistemas distribuidos. Sin embargo, los algoritmos de ordenamiento tradicionales, secuenciales, presentan limitaciones en cuanto a su escalabilidad y eficiencia cuando se aplican a grandes volúmenes de datos o en sistemas con múltiples procesadores. Este problema se agrava en situaciones que requieren el procesamiento de grandes conjuntos de datos en tiempo real, como en aplicaciones científicas, financieras y de análisis de grandes volúmenes de información. En este contexto, se vuelve crucial desarrollar soluciones paralelizadas que puedan aprovechar el potencial de los sistemas multi-core y distribuidos.

Una de las soluciones para mejorar la eficiencia del ordenamiento de grandes conjuntos de datos es el uso de algoritmos paralelos, que dividen la carga de trabajo entre múltiples procesadores. El Parallel Ranking Sort (Parallel Ranking Sort), implementado utilizando el modelo de programación de paso de mensajes MPI (Message Passing Interface), ofrece una manera efectiva de abordar estos desafíos. El algoritmo de Parallel Ranking Sort se basa en la idea de que, en lugar de ordenar los elementos de forma secuencial en un solo hilo de ejecución, los datos se distribuyen entre múltiples procesos. Cada proceso calcula su ranking local y luego se realiza un intercambio de información entre los procesos para generar un ranking global.

1.1 Motivacion

El algoritmo parallel ranking sort ha ganado relevancia en el campo de la informática debido a su capacidad para gestionar de manera eficiente grandes cantidades de datos a través de un enfoque distribuido. El algoritmo no solo se beneficia de la paralelización, sino que también presenta ventajas frente a otros algoritmos de ordenamiento secuenciales, como Quicksort o Merge Sort, especialmente cuando se implementa en arquitecturas paralelas. Por ejemplo, en un sistema con N procesos, el algoritmo de ordenamiento por ranking divide los datos de manera eficiente entre los procesos, lo que reduce el tiempo total de ejecución al distribuir las tareas de manera más equitativa.

La comparación teórica con algoritmos secuenciales demuestra que, en un entorno paralelo, el algoritmo de ranking por MPI puede superar a los enfoques secuenciales tradicionales, que tienen una complejidad temporal O(n log n). En sistemas distribuidos, donde el tiempo de comunicación entre los procesos puede ser un factor significativo, el algoritmo de ranking por MPI puede aprovechar la localización de los datos y la comunicación eficiente entre procesos para lograr un ordenamiento más rápido.

Estudios han demostrado que los algoritmos paralelos, como el ordenamiento por ranking, son particularmente útiles en situaciones de alto rendimiento y big data, donde las soluciones secuenciales no son viables debido a los recursos limitados. Este enfoque paralelo ha demostrado ser más escalable y eficiente que los algoritmos secuenciales clásicos, especialmente cuando se trata de grandes volúmenes de datos distribuidos a través de múltiples nodos en un clúster de computadoras. Justificación

La implementación de algoritmos de ordenamiento en paralelo es esencial para resolver problemas de procesamiento de datos en tiempo real, que son cada vez más comunes en áreas como la simulación científica, el análisis de datos masivos

y el procesamiento de imágenes. El algoritmo de Parallel Ranking Sort utilizando MPI presenta varias ventajas en este sentido. En primer lugar, al distribuir los datos entre múltiples procesos, el algoritmo no solo mejora la velocidad de ordenamiento, sino que también maximiza la utilización de los recursos disponibles en sistemas multi-core. Esto lo convierte en una opción atractiva para aplicaciones en las que el tiempo de procesamiento es crítico.

Además, la capacidad de MPI para gestionar la comunicación entre procesos en un entorno distribuido permite que el algoritmo escale eficazmente a medida que se incrementa el número de procesos o el tamaño de los datos. Esto es crucial en aplicaciones donde la cantidad de datos a procesar puede ser extremadamente grande, y las soluciones secuenciales no pueden satisfacer los requisitos de tiempo.

El algoritmo de Parallel Ranking Sort tiene aplicaciones prácticas en áreas como la minería de datos, el procesamiento de grandes volúmenes de datos en redes sociales, y en la gestión de bases de datos distribuidas. En estos casos, la eficiencia en el ordenamiento no solo mejora el rendimiento general del sistema, sino que también permite realizar análisis en tiempo real, lo que puede ser decisivo para la toma de decisiones en aplicaciones críticas.

1.2 Explicación Breve del Algoritmo

El algoritmo de Parallel Ranking Sort utilizando MPI se implementa en varias fases. Primero, los datos se distribuyen entre los procesos, y cada proceso realiza un ordenamiento local de los datos que le corresponden. Luego, se lleva a cabo un intercambio de información entre los procesos en forma de un paso de "Gossip", donde los datos de una columna se replican en todos los procesos de esa columna. A continuación, se realiza un Broadcast, en el que los procesos comparten sus datos con los procesos de las filas correspondientes. Una vez que los datos están correctamente

distribuidos, cada proceso ordena localmente su arreglo. Posteriormente, se evalúa el ranking local de cada proceso y, finalmente, se realiza un paso de Reduce para consolidar los resultados y obtener el orden final de los elementos.

Este enfoque de ordenamiento no solo distribuye el trabajo entre los procesos de manera eficiente, sino que también optimiza la comunicación y el intercambio de información entre ellos, lo que resulta en un tiempo de ejecución significativamente reducido en comparación con los algoritmos secuenciales.

II

Metodo

En esta sección describiremos el diseño del algoritmo a través de un PRAM teórico. Posteriormente, definiremos el algoritmo en C++ utilizando MPI para lograr la implementación requerida.

2.1 PRAM

Se define el siguiente PRAM teórico para calcular el ranking de elementos distribuidos en una matriz bidimensional de procesos. Cada proceso contiene una porción de los datos y colabora con otros para realizar el ranking global.

```
Input: d[1...n]
Output: ranking de d(i,j,k) almacenado en b' en el proceso (i,j)

1. Gossip:
   forall (i, j) pardo
      a[i, j] = gather a[1, j], ..., a[P, j]
                (from all processes in column j)

2. Broadcast:
   forall (i, j) pardo
      a'[i, j] = broadcast a[i, j]
                 (to all processes in row i)

3. Sort:
```

```
forall (i, j) pardo
    sort a'[i, j]
```

4. Local Ranking:

```
forall (i, j) pardo
    b[i, j] = rank elements in a'[i, j]
```

5. Reduce:

```
forall (i, j) pardo
    b'[i, j] = reduce sum of b[i, j]
            (across all columns to process (i, 0))
```

A continuación se detalla cada sección del PRAM:

- Gossip: Se encarga de ordenar los datos dentro de cada proceso. Cada proceso (i, j) ordena los datos en a'[i, j].
  El método consiste en utilizar un algoritmo de ordenamiento estándar (como quicksort o mergesort) para ordenar los datos dentro de cada proceso. Esto prepara los datos para el cálculo de ranking local.
  La complejidad algoritmica es: $O(\sqrt{P})$

- Broadcast: Se encarga de asignar un ranking local a los datos ordenados. Cada proceso (i, j) asigna un ranking a los elementos en a'[i, j], almacenando los rankings en b[i, j].
  El método consiste en que después de ordenar, cada elemento en el conjunto de datos de un proceso se le asigna un rango basado en su posición. Por ejemplo, el menor elemento recibe el rango 1, el siguiente el rango 2, y así sucesivamente.
  La complejidad algorítmica es: $O\left(\log P \cdot \left(\alpha + \frac{n}{\sqrt{P}} \cdot \beta\right)\right)$

- Sort: Se encarga de ordenar los datos dentro de cada proceso. Cada proceso (i, j) ordena los datos en a'[i, j].

  El método consiste en que se utiliza un algoritmo de ordenamiento estándar para ordenar los datos dentro de cada proceso. Esto prepara los datos para el cálculo de ranking local.

  La complejidad algorítmica es: $O\left(\frac{n}{\sqrt{P}} \log\left(\frac{n}{\sqrt{P}}\right)\right)$

- Local Ranking: Se encarga de asignar un ranking local a los datos ordenados. Cada proceso (i, j) asigna un ranking a los elementos en a'[i, j], almacenando los rankings en b[i, j].

  El método consiste en que después de ordenar, cada elemento en el conjunto de datos de un proceso se le asigna un rango basado en su posición. Por ejemplo, el menor elemento recibe el rango 1, el siguiente el rango 2, y así sucesivamente.

  La complejidad algorítmica es: $O\left(\frac{n}{\sqrt{P}} \log\left(\frac{n}{\sqrt{P}}\right)\right)$

- Reduce: Se encarga de sumar los rankings locales para obtener un ranking global. Los rankings en b[i, j] son sumados a través de todas las columnas y el resultado es almacenado en el proceso (i, 0) como b'[i, j].

  El método consiste en que se utiliza una operación de reduce que suma los valores de ranking de todos los procesos en una columna y envía el resultado al proceso en la posición (i, 0). Esto proporciona un ranking final que considera los rankings locales de todos los procesos.

  La complejidad algorítmica es: $O\left(\log P \cdot \left(\alpha + \frac{n}{\sqrt{P}} \cdot \beta\right)\right)$

2.2 Código en C MPI

```
    #include <mpi.h>
#include <iostream>
#include <vector>
```

```
#include <algorithm>
#include <cmath>
#include <string>

template <typename T>
void print_array(int rank, const std::vector<T>& arr, const std::string&
    std::cout << "Process " << rank << " - " << label << ": ";
    for (const T& elem : arr) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;
}

void validate_processes(int rank, int size) {
    if (rank == 0) {
        int sqrt_size = static_cast<int>(std::sqrt(size));
        if (sqrt_size * sqrt_size != size) {
            std::cerr << "Error: The number of processes must be a perfec
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
}

std::vector<char> scatter_data(const std::vector<char>& data, int elemen
    std::vector<char> local_data(elements_per_process);
    MPI_Scatter(data.data(), elements_per_process, MPI_CHAR,
```

```
                local_data.data(), elements_per_process, MPI_CHAR,
                0, MPI_COMM_WORLD);

    // print_array(rank, local_data, "Initial local data");

    return local_data;
}

std::vector<char> perform_gossip(const std::vector<char>& data, int eleme
    std::vector<char> column_data(elements_per_process * grid_size);
    int column = rank % grid_size;
    MPI_Comm column_comm;
    MPI_Comm_split(MPI_COMM_WORLD, column, rank, &column_comm);

    MPI_Allgather(data.data(), elements_per_process, MPI_CHAR,
                  column_data.data(), elements_per_process, MPI_CHAR,
                  column_comm);

    // print_array(rank, column_data, "Column data after Gossip");

    MPI_Comm_free(&column_comm);
    return column_data;
}

std::vector<char> perform_broadcast(const std::vector<char>& data, int el
    int row = rank / grid_size;
    int diag_process = row * grid_size + row;
```

```
    MPI_Comm row_comm;
    MPI_Comm_split(MPI_COMM_WORLD, row, rank, &row_comm);

    std::vector<char> diagonal_data(elements_per_process);
    if (rank == diag_process) {
        diagonal_data = data;
    }

    MPI_Bcast(diagonal_data.data(), elements_per_process, MPI_CHAR, row,

    // print_array(rank, diagonal_data, "Diagonal data after Broadcast");

    MPI_Comm_free(&row_comm);
    return diagonal_data;
}

std::vector<char> perform_sort(const std::vector<char>& data) {
    std::vector<char> sorted_data = data;
    std::sort(sorted_data.begin(), sorted_data.end());

    return sorted_data;
}

std::vector<long> perform_local_rank(const std::vector<char>& data, const
    std::vector<long> local_rank(other_data.size());

    for (size_t i = 0; i < other_data.size(); i++) {
        for (size_t j = 0; j < data.size(); j++) {
```

```
            if (other_data[i] >= data[j]) {
                local_rank[i]++;
            }
        }
    }

    return local_rank;
}



std::vector<long> perform_reduce(const std::vector<long>& local_rank, int

}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    validate_processes(rank, size);

    std::vector<char> data = {'c', 'j', 'k', 'h', 'o', 'a', 'g', 'm', 'e'
    int grid_size = static_cast<int>(std::sqrt(size));
    int elements_per_process = static_cast<int>(data.size()) / size;
```

```
    std::vector<char> local_data = scatter_data(data, elements_per_proces

    std::vector<char> column_data = perform_gossip(local_data, elements_p
    local_data.clear();

    std::vector<char> diagonal_data = perform_broadcast(column_data, elem

    std::vector<char> sorted_data = perform_sort(diagonal_data);
    column_data.clear();

    std::vector<long> local_rank = perform_local_rank(sorted_data, diagon

    std::vector<long> reduced_rank = perform_reduce(local_rank, rank, gri

    // Print
    if (true) {
        print_array(rank, sorted_data, "Sorted data");
        print_array(rank, diagonal_data, "Diagonal data");
        print_array(rank, local_rank, "Local rank");
        print_array(rank, reduced_rank, "Reduced rank");

        std::cout << std::endl;
    }

    MPI_Finalize();
      return0       ;
}
```

III

Resultados

Los resultados se presentan en el siguiente archivo:

https://docs.google.com/spreadsheets/d/1VRmVDYLhbXeF5xWSWwjvouL4yXCpjz41ImFF

REFERENCIAS BIBLIOGRÁFICAS