UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CARRERA DE CIENCIAS DE LA COMPUTACION

ORDENAMIENTO EFICIENTE POR RANKING EN PARALELO

Computación Paralela CS(4052)

AUTOR(ES)

Marcelo Mario Zuloeta Salazar

PROFESOR(ES)

José Antonio Fiestas Iquira

Lima - Perú

Diciembre, 2024

TABLA DE CONTENIDO

Pág.

INTRODUCCIÓN

Descripción del problema

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.

Formulación del problema

Para formular el problema podemos plantearnos las siguientes interrogantes: ¿Cómo dividir el trabajo de manera equitativa entre múltiples procesadores? ¿Cómo sincronizar los procesos para evitar conflictos?

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, 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.

Objetivos

Con lo anterior en mente, podemos establecer que este proyecto tiene como objetivo diseñar e implementar un algoritmo de ordenamiento por ranking paralelo eficiente, basado en el modelo PRAM y utilizando C++ con MPI. Se evaluará su rendimiento mediante el tiempo de ejecución y comparaciones con Quicksort. Además, se explorará la escalabilidad del algoritmo variando el número de procesos (p) y el tamaño de los datos (n).

Planteamiento de la solución

La solución propuesta se basa en el modelo PRAM (definido posteriormente en la descripción del método), distribuyendo los datos en una matriz de procesos. La comunicación entre procesos se realiza mediante dos fases: gossip para intercambiar datos dentro de las columnas y broadcast para compartirlos entre las filas. Tras el ordenamiento local y el cálculo del ranking parcial en cada proceso, se combinan los resultados para obtener el ranking global. La implementación con MPI facilita la comunicación entre los diferentes procesos. Se llevará a cabo una evaluación del rendimiento comparando este enfoque con el algoritmo secuencial Quicksort.

CAPÍTULO I

MÉTODO

Para el método se va a presentar el PRAM teórico, detallando la complejidad esperada para cada sección. Además se va a detallar el código final en C++ utilizando MPI. Adicionalmente,

1.1 Ordenamiento por Ranking

En el ordenamiento por ranking paralelo, cada elemento de un conjunto de datos obtiene una posición única que refleja su magnitud relativa respecto a los demás elementos. Para lograr esto, la tarea se descompone en subtareas que son distribuidas y ejecutadas simultáneamente en múltiples procesadores.

Los datos se distribuyen entre $p = P \cdot P$ procesos organizando una cuadrícula, por lo que tenemos esta restricción de que la cantidad de procesos debe ser un cuadrado perfecto. Los elementos se reparten equitativamente al hacer scatter, donde cada proceso recibe $N = n/p$ elementos. Para más detalle, podemos definir que los procesos comparten información entre ellos en dos etapas:

1. **Scatter:** Cada proceso recibe $N = n/p$ elementos de forma equitativa.
2. **Gossip:** Los procesos intercambian sus N datos con los $P - 1$ procesos que están en su misma columna.
3. **Broadcast:** Los procesos diagonales ($p(i, j)$, $i = j$) toman los $N \cdot P$ elementos que tienen y los comparten con los demás procesos de su misma fila.

El proceso comienza cuando cada nodo recibe $N \cdot P$ elementos durante el *broadcast*. Una vez completada esta etapa, cada nodo realiza un ordenamiento local de los elementos recibidos.
A continuación, se calcula el *ranking* local de cada elemento, lo que permite determinar cuántos datos del conjunto recibido durante el *gossip* son menores que el elemento en cuestión.

En la siguiente fase, un nodo representante en cada fila recopila los *rankings* locales calculados por los demás nodos de la misma fila. Este nodo los combina sumándolos para obtener un *ranking* global de los elementos en la etapa de *reduce*. Finalmente, en la etapa de *gather* el proceso maestro reúne los resultados parciales desde cada nodo y genera el conjunto completo de resultados .

1.1.1 PRAM del ordenamiento por ranking en paralelo

Para definir el PRAM hay que definir los algoritmos locales y de comunicación:

1. Scatter(d, p): Distribuye un array d entre $p = P^2$ procesos, de manera que cada proceso reciba $\frac{n}{p}$ elementos. Por tanto se utiliza para distribuir partes del inputData a todos los procesos (esto tendrá más sentido al presentar el código). La complejidad teórica va a estar definida asumiendo que α es el tiempo de inicio de la comunicación y β es el tiempo por unidad de datos transmitida. La cantidad de datos $\frac{n}{p}$ será consistente con la partición de datos entre los procesos.

2. Gossip(p, P, N): Cada proceso comparte sus $N = \frac{n}{p}$ elementos con los $P - 1$ procesos en su misma columna. Esta realiza intercambios de datos entre procesos en una configuración de anillo, lo que implica múltiples pasos de

comunicación.Ya que asumimos una topología de cuadrícula, los pasos serán determinados en base a $\sqrt{p}$

3. Broadcast(p, P, N): Los procesos p_{ij}, donde $i = j$, comparten sus $N \cdot P$ datos con los procesos de su fila. Ya que esta operación se realiza desde procesos diagonales a todos los procesos en su fila, la complejidad va a estar determinada por una subred de tamaño $\sqrt{p}$

4. LocalSort(a): Cada proceso ordena localmente los $N \cdot P$ elementos recibidos en el broadcast. Se trabaja sobre $\frac{n}{\sqrt{p}}$ elementos.

5. LocalRanking(a', b): Para cada elemento de a', se calcula cuántos elementos en b son menores o iguales utilizando mapas para contar y acumular frecuencias.

6. Reduce(rankings, P): La reducción de rangos se realiza recogiendo datos en el proceso maestro p_{ij} donde $i = j$ y combinados en un ranking global. Es una reducción típica en MPI.

7. Gather(results, P): Similar a Reduce, los datos finales son recogidos en el proceso maestro.

Con esto claro, podemos definir el PRAM.

Algorithm 1 PRAM

```
1: Input: Array de datos d[1, ..., n]
2: Output: Rankings de los elementos de d
3: Scatter(d, p)
4: Gossip(p, P, N)
5: Broadcast(p, P, N)
6: for cada proceso paralelo p_ij par do
7:    LocalSort(a)
8:    LocalRanking(a', b)
9: end for
10: Reduce(rankings, P)
11: Gather(results, P)
```

1.1.2 Complejidad teórica

El análisis de complejidad del modelo PRAM permite descomponer el tiempo total de ejecución $T(n, p)$ en términos de las operaciones principales que realiza el algoritmo. Estas operaciones incluyen la distribución de datos (*scatter*), las etapas de comunicación (*gossip* y *broadcast*), el ordenamiento local, el cálculo de rankings, la combinación de resultados (*reduce*), y la recolección final de resultados (*gather*).

Ahora, para aclarar: en la implementación sin optimización basada en árbol, las operaciones de *gossip* son proporcionales a p, mientras que *broadcast* y *reduce* son proporcionales a $\sqrt{p}$.

Cada término tiene una contribución específica basada en el número de elementos n, el número de procesos p, y los parámetros de comunicación α (latencia a.k.a. tiempo de inicio de comunicación) y β (ancho de banda a.k.a. tiempo por unidad de datos transmitida):

$$T_{\text{scatter}} = p \cdot \left(\alpha + \frac{n}{p} \cdot \beta \right)$$

$$T_{\text{gossip}} = \sqrt{p} \cdot \left(\alpha + \frac{n}{p} \cdot \beta \right)$$

$$T_{\text{broadcast}} = \sqrt{p} \cdot \left(\alpha + \frac{n}{\sqrt{p}} \cdot \beta \right)$$

$$T_{\text{sort}} = \frac{n}{\sqrt{p}} \cdot \log \left(\frac{n}{\sqrt{p}} \right)$$

$$T_{\text{ranking}} = \frac{n}{\sqrt{p}}$$

$$T_{\text{reduce}} = \sqrt{p} \cdot \left(\alpha + \frac{n}{\sqrt{p}} \cdot \beta \right)$$

$$T_{\text{gather}} = \sqrt{p} \cdot \left(\alpha + \frac{n}{\sqrt{p}} \cdot \beta \right)$$

De esta forma, la complejidad total se expresa como la suma de las complejidades del método scatter, gossip, broadcast, sort, ranking, reduce y gather.

La complejidad varía dependiendo de si las operaciones de comunicación están optimizadas o no.

Ahora bien, idealmente cuando se utiliza un esquema de comunicación basado en un árbol, las operaciones de *broadcast* y *reduce* son proporcionales a $\log(p)$. En este caso, las complejidades se expresan como:

$$T_{\text{gossip}} = \sqrt{p} \cdot \log(\sqrt{p}) \cdot \left(\alpha + \frac{n}{p} \cdot \beta \right)$$

$$T_{\text{broadcast}} \text{ y } T_{\text{reduce}} = \log(\sqrt{p}) \cdot \left(\alpha + \frac{n}{\sqrt{p}} \cdot \beta \right)$$

1.1.3 Código del algoritmo en C++

El código final implementado y presentado es el siguiente. Incluye el scatter inicial, los métodos de gossip y broadcast, ordenamiento y ranking, el reduce diagonal y el gather al maestro:

(a) (b)

FIGURA 1.1: Scheme showing the architecture of a generic kinematic task. (a) Logo en tamaño de 3 centímetros. (b) Logo en tamaño de 2 centímetros.

```
1
2  #include <map>
3  #include <mpi.h>
4  #include <string>
5  #include <vector>
6  #include <iostream>
7  #include <algorithm>
8  #include <random>
9  #include <iomanip>
10 #include <fstream>
11 #include <iterator>
12 using namespace std;
13
14 float startTime, endTime;
15 float tiempoComputoInicio, tiempoComputoFin,
     tiempoComunicacionInicio, tiempoComunicacionFin;
16 float timePoints[8];
17
18 string retrieveString(size_t length) {
19     ifstream file("characters.txt");
```

```
20      string chars;
21
22      if (file.is_open()) {
23          getline(file, chars);
24          file.close();
25      } else {
26          cerr << "Error opening characters.txt" << endl;
27          return "";
28      }
29
30      random_device rd;
31      mt19937 generator(rd());
32      uniform_int_distribution<size_t> distribution(0,
            chars.size() - 1);
33
34      string result;
35      result.reserve(length);
36
37      generate_n(back_inserter(result), length, [&]() {
38          return chars[distribution(generator)];
39      });
40
41      return result;
42  }
43
44  string mergeData(const map<int, string>& dataMap) {
45      string combined;
46      for (const auto& entry : dataMap) {
```

```
47          combined += entry.second;
48      }
49      return combined;
50  }
51
52  vector<int> computeLocalRank(const string& localData,
        const string& fullData) {
53      map<char, int> frequencyMap;
54      for (char c : localData) {
55          frequencyMap[c]++;
56      }
57      map<char, int> cumulativeFrequency;
58      int runningTotal = 0;
59      for (auto& pair : frequencyMap) {
60          runningTotal += pair.second;
61          cumulativeFrequency[pair.first] = runningTotal;
62      }
63      vector<int> ranks(fullData.size(), 0);
64
65      for (size_t i = 0; i < fullData.size(); i++) {
66          char currentChar = fullData[i];
67          auto it = cumulativeFrequency.upper_bound(
              currentChar);
68          if (it != cumulativeFrequency.begin()) {
69              --it;
70              ranks[i] = it->second;
71          }
72      }
```

```
73
74      return ranks;
75  }
76
77  void performGossip(int rank, int rows, int cols, int size
        , map<int, string>& data) {
78      int row = rank / cols;
79      int col = rank % cols;
80      char buffer[10000];
81
82      for (int step = 0; step < rows - 1; step++) {
83          int target = ((row + step + 1) % rows) * cols +
                col;
84          int source = ((row + rows - step - 1) % rows) *
                cols + col;
85
86          string toSend = mergeData(data);
87          int sendSize = toSend.size() + 1;
88
89          MPI_Sendrecv(toSend.c_str(), sendSize, MPI_CHAR,
                target, 0,
90                       buffer, 10000, MPI_CHAR, source, 0,
91                       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
92
93          data[(rank - cols + size) % size] = string(buffer
                );
94      }
95  }
```

```
96
97  void broadcastReverse(int rank, int rows, int cols, const
        string& data, map<int, string>& results) {
98      int row = rank / cols;
99      int col = rank % cols;
100     char buffer[10000];
101
102     if (col == row) {
103         for (int c = 0; c < cols; c++) {
104             if (c != col) {
105                 MPI_Send(data.c_str(), data.size() + 1,
                        MPI_CHAR, row * cols + c, 0,
                        MPI_COMM_WORLD);
106             }
107         }
108         results[0] = data;
109     } else {
110         MPI_Recv(buffer, 10000, MPI_CHAR, row * cols +
            row, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
111         results[0] = string(buffer);
112     }
113 }
114
115 string sortAndDisplayRanks(const vector<int>& ranks,
       const string& data) {
116     vector<pair<int, char>> indexedRanks;
117
118     for (size_t i = 0; i < data.size(); i++) {
```

```
119             indexedRanks.emplace_back(ranks[i], data[i]);
120         }
121         sort(indexedRanks.begin(), indexedRanks.end(), [](
                const pair<int, char>& a, const pair<int, char>& b
                ) {
122             return a.first < b.first || (a.first == b.first
                    && a.second < b.second);
123         });
124 
125         string sortedData;
126         for (const auto& rank : indexedRanks) {
127             sortedData += rank.second;
128         }
129 
130         return sortedData;
131 }
132 
133 string processAndRankData(int rank, int rows, int cols,
        const string& initialData, const string& processedData
        ) {
134         string sortedData = initialData;
135 
136         tiempoComputoInicio = MPI_Wtime();
137         sort(sortedData.begin(), sortedData.end());
138         tiempoComputoFin = MPI_Wtime();
139 
140         vector<int> localRanks = computeLocalRank(sortedData,
                processedData);
```

```
141
142     MPI_Barrier(MPI_COMM_WORLD);
143
144     int row = rank / cols;
145     int col = rank % cols;
146     int diagProc = row * cols + row;
147     char buffer[10000];
148     string combinedData;
149
150     tiempoComunicacionInicio = MPI_Wtime();
151     if (col != row) {
152         MPI_Send(localRanks.data(), localRanks.size(),
               MPI_INT, diagProc, 0, MPI_COMM_WORLD);
153     } else {
154         vector<int> totalRanks(localRanks.size(), 0);
155         for (int c = 0; c < cols; c++) {
156             if (c != col) {
157                 vector<int> receivedRanks(localRanks.size
                   ());
158                 MPI_Recv(receivedRanks.data(),
                   receivedRanks.size(), MPI_INT, row *
                   cols + c, 0, MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
159                 for (size_t i = 0; i < totalRanks.size();
                    ++i) {
160                     totalRanks[i] += receivedRanks[i];
161                 }
162             } else {
```

```
163                 for (size_t i = 0; i < totalRanks.size();
                        i++) {
164                     totalRanks[i] += localRanks[i];
165                 }
166             }
167         }
168 
169         if (rank != 0) {
170             MPI_Send(totalRanks.data(), totalRanks.size()
                    , MPI_INT, 0, 1, MPI_COMM_WORLD);
171             MPI_Send(processedData.c_str(), processedData
                    .size() + 1, MPI_CHAR, 0, 1,
                    MPI_COMM_WORLD);
172         } else {
173             vector<int> globalRanks(totalRanks);
174 
175             for (int r = 1; r < rows; r++) {
176                 int diagProc = r * cols + r;
177                 vector<int> receivedRanks(totalRanks.size
                        ());
178                 MPI_Recv(receivedRanks.data(),
                        receivedRanks.size(), MPI_INT,
                        diagProc, 1, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
179                 MPI_Recv(buffer, 10000, MPI_CHAR,
                        diagProc, 1, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
180                 string receivedData(buffer);
```

```
181                 combinedData += receivedData;
182
183                 for (size_t i = 0; i < receivedRanks.size
                        (); i++) {
184                     globalRanks[i] += receivedRanks[i];
185                 }
186             }
187
188             sortedData = sortAndDisplayRanks(globalRanks,
                    processedData + combinedData);
189         }
190     }
191     tiempoComunicacionFin = MPI_Wtime();
192     return sortedData;
193 }
194
195 int main(int argc, char** argv) {
196     MPI_Init(&argc, &argv);
197
198     int rank, size;
199     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
200     MPI_Comm_size(MPI_COMM_WORLD, &size);
201
202     int totalElements = rows * cols;
203     string inputData;
204
205     if (rank == 0) {
```

```
206         inputData = retrieveString(msgSize *
                totalElements);
207         if (inputData.size() % totalElements != 0) {
208             MPI_Finalize();
209             return 1;
210         }
211     }
212
213     char* localData = new char[msgSize + 1];
214     localData[msgSize] = '\0';
215
216     startTime = MPI_Wtime();
217
218     timePoints[0] = MPI_Wtime();
219     MPI_Scatter(inputData.c_str(), msgSize, MPI_CHAR,
            localData, msgSize, MPI_CHAR, 0, MPI_COMM_WORLD);
220     timePoints[1] = MPI_Wtime();
221
222     string localDataStr(localData);
223     delete[] localData;
224
225     map<int, string> dataMap = {{rank, localDataStr}};
226     map<int, string> resultData;
227     timePoints[2] = MPI_Wtime();
228     performGossip(rank, rows, cols, size, dataMap);
229     timePoints[3] = MPI_Wtime();
230     string gossipResult = mergeData(dataMap);
231     timePoints[4] = MPI_Wtime();
```

```
232     broadcastReverse(rank, rows, cols, gossipResult,
           resultData);
233     timePoints[5] = MPI_Wtime();
234
235     timePoints[6] = MPI_Wtime();
236     string finalResult = processAndRankData(rank, rows,
           cols, gossipResult, mergeData(resultData));
237     timePoints[7] = MPI_Wtime();
238     endTime = MPI_Wtime();
239
240     if (rank == 0) {
241         cout << fixed << setprecision(10);
242         cout << "Execution Time: " << (endTime -
              startTime) << endl;
243         cout << "Scatter Time: " << (timePoints[1] -
              timePoints[0]) << endl;
244         cout << "Gossip Time: " << (timePoints[3] -
              timePoints[2]) << endl;
245         cout << "Broadcast Time: " << (timePoints[5] -
              timePoints[4]) << endl;
246         cout << "Process Time: " << (timePoints[7] -
              timePoints[6]) << endl;
247         cout << "Tiempo de computo: " << (
              tiempoComputoFin - tiempoComputoInicio) << "
              segundos." << endl;
```

```
248         cout << "Tiempo de comunicacion: " << (
                tiempoComunicacionFin -
                tiempoComunicacionInicio) << " segundos." <<
                endl;
249     }
250
251     MPI_Finalize();
252     return 0;
253 }
```

1.2 Ordenamiento usando Mergesort

La implementación del algoritmo de quicksort paralelo en MPI que se encuentra en el repositorio de GitHub Quicksort-Parallel-MPI que utiliza un enfoque recursivo y divide el problema entre diferentes procesos para lograr paralelismo. A continuación, se describe el funcionamiento paso a paso:

1. Inicialización y Distribución de Datos: El proceso maestro (rank 0) genera un arreglo de números aleatorios. Este arreglo es el que se va a ordenar utilizando múltiples procesos en un entorno MPI.

2. División Recursiva y Paralelización: El proceso se inicia con el proceso maestro que tiene el arreglo completo. Utiliza una función de partición (Hoare Partition) para dividir el arreglo en dos subarreglos alrededor de un pivote. El proceso maestro mantiene una mitad del arreglo y envía la otra mitad a otro proceso para su procesamiento paralelo. Este proceso se repite recursivamente, donde cada proceso que recibe un subarreglo lo divide nuevamente y envía una de las mitades a otro proceso, si es posible (es decir, si hay procesos

disponibles). La asignación de subarreglos a procesos sigue una estructura de árbol binario, donde el índice del proceso que recibe los datos se calcula como currProcRank + pow(2, rankIndex).

3. Ordenamiento Local: Cada proceso, al final de la cadena de divisiones, ordena su subarreglo asignado localmente utilizando el algoritmo de quicksort secuencial.

4. Recolección y Combinación: Una vez que un proceso termina de ordenar su subarreglo, envía los datos ordenados de vuelta al proceso que se los había enviado. Este proceso de envío de vuelta también sigue la estructura de árbol binario en reversa, donde cada proceso combina los subarreglos ordenados que recibe con su propio subarreglo ordenado.

5. Finalización: El proceso maestro recibe las partes ordenadas del arreglo y las combina para formar el arreglo ordenado completo. Se realiza una verificación final para asegurarse de que el arreglo está correctamente ordenado.

1.2.1 Complejidad de Mergesort

La complejidad teórica del algoritmo de quicksort paralelo se puede describir de la siguiente manera:

- **Ordenamiento Local:** Cada proceso realiza un quicksort en su subarreglo, lo cual tiene una complejidad promedio de $O(\frac{n}{p} \log \frac{n}{p})$ para un subarreglo de tamaño n.

- **Paralelismo:** La profundidad del árbol de procesos es idealmente $\log p$, donde p es el número de procesos. Esto implica que en el mejor caso, la partición se realiza en $O(\frac{n}{p} \log p)$.

- **Comunicación:** La comunicación entre procesos introduce un overhead que puede ser significativo, especialmente en sistemas con alta latencia de comunicación. Este overhead no es trivial de cuantificar en términos simples, pero afecta el rendimiento general del algoritmo.

Por lo tanto, la complejidad total del algoritmo, considerando el paralelismo y la comunicación, puede ser expresada como:

$$O(\frac{n}{p}\log\frac{n}{p} + \frac{n}{p}\log p)$$

donde n es el número total de elementos a ordenar y p es el número de procesos.

1.2.2 Código de Mergesort

Este es el código de Mergesort, extraído directamente del siguiente repositorio: Link al repositorio

Código de quicksort_mpi.cpp:

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "math.h"
5  #include <stdbool.h>
6  #define SIZE 1000000
7
8  /*
9      Divides the array given into two partitions
10         - Lower than pivot
11         - Higher than pivot
```

```
12      and returns the Pivot index in the array
13  */
14  int partition(int *arr, int low, int high){
15      int pivot = arr[high];
16      int i = (low - 1);
17      int j,temp;
18      for (j=low;j<=high-1;j++){
19    if(arr[j] < pivot){
20          i++;
21                temp=arr[i];
22                arr[i]=arr[j];
23                arr[j]=temp;
24    }
25      }
26      temp=arr[i+1];
27      arr[i+1]=arr[high];
28      arr[high]=temp;
29      return (i+1);
30  }
31
32  /*
33      Hoare Partition - Starting pivot is the middle point
34      Divides the array given into two partitions
35          - Lower than pivot
36          - Higher than pivot
37      and returns the Pivot index in the array
38  */
39  int hoare_partition(int *arr, int low, int high){
```

```
40      int middle = floor((low+high)/2);
41      int pivot = arr[middle];
42      int j,temp;
43      // move pivot to the end
44      temp=arr[middle];
45      arr[middle]=arr[high];
46      arr[high]=temp;
47
48      int i = (low - 1);
49      for (j=low;j<=high-1;j++){
50          if(arr[j] < pivot){
51              i++;
52              temp=arr[i];
53              arr[i]=arr[j];
54              arr[j]=temp;
55          }
56      }
57      // move pivot back
58      temp=arr[i+1];
59      arr[i+1]=arr[high];
60      arr[high]=temp;
61
62      return (i+1);
63  }
64
65  /*
66      Simple sequential Quicksort Algorithm
67  */
```

```
68 void quicksort(int *number,int first,int last){
69     if(first<last){
70         int pivot_index = partition(number, first, last);
71         quicksort(number,first,pivot_index-1);
72         quicksort(number,pivot_index+1,last);
73     }
74 }
75
76 /*
77     Functions that handles the sharing of subarrays to
       the right clusters
78 */
79 int quicksort_recursive(int* arr, int arrSize, int
   currProcRank, int maxRank, int rankIndex) {
80     MPI_Status status;
81
82     // Calculate the rank of the Cluster which I'll send
       the other half
83     int shareProc = currProcRank + pow(2, rankIndex);
84     // Move to lower layer in the tree
85     rankIndex++;
86
87     // If no Cluster is available, sort sequentially by
       yourself and return
88     if (shareProc > maxRank) {
89         MPI_Barrier(MPI_COMM_WORLD);
90       quicksort(arr, 0, arrSize-1 );
91         return 0;
```

```
92      }
93      // Divide array in two parts with the pivot in
           between
94      int j = 0;
95      int pivotIndex;
96      pivotIndex = hoare_partition(arr, j, arrSize-1 );
97
98      // Send partition based on size(always send the
           smaller part),
99      // Sort the remaining partitions,
100     // Receive sorted partition
101     if (pivotIndex <= arrSize - pivotIndex) {
102         MPI_Send(arr, pivotIndex , MPI_INT, shareProc,
               pivotIndex, MPI_COMM_WORLD);
103       quicksort_recursive((arr + pivotIndex+1), (arrSize
             - pivotIndex-1 ), currProcRank, maxRank,
             rankIndex);
104         MPI_Recv(arr, pivotIndex , MPI_INT, shareProc,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
105     }
106     else {
107         MPI_Send((arr + pivotIndex+1), arrSize -
               pivotIndex-1, MPI_INT, shareProc, pivotIndex +
                1, MPI_COMM_WORLD);
108         quicksort_recursive(arr, (pivotIndex),
               currProcRank, maxRank, rankIndex);
```

```
109         MPI_Recv((arr + pivotIndex+1), arrSize -
                pivotIndex-1, MPI_INT, shareProc, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
110     }
111 }
112
113
114 int main(int argc, char *argv[]) {
115     int unsorted_array[SIZE];
116     int array_size = SIZE;
117     int size, rank;
118     // Start Parallel Execution
119     MPI_Init(&argc, &argv);
120     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
121     MPI_Comm_size(MPI_COMM_WORLD, &size);
122     if(rank==0){
123         // --- RANDOM ARRAY GENERATION ---
124         printf("Creating Random List of %d elements\n",
                SIZE);
125         int j = 0;
126         for (j = 0; j < SIZE; ++j) {
127             unsorted_array[j] =(int) rand() % 1000;
128         }
129         printf("Created\n");
130   }
131
132     // Calculate in which layer of the tree each Cluster
        belongs
```

```
133     int rankPower = 0;
134     while (pow(2, rankPower) <= rank){
135         rankPower++;
136     }
137     // Wait for all clusters to reach this point
138     MPI_Barrier(MPI_COMM_WORLD);
139     double start_timer, finish_timer;
140     if (rank == 0) {
141       start_timer = MPI_Wtime();
142         // Cluster Zero(Master) starts the Execution and
143         // always runs recursively and keeps the left
              bigger half
144         quicksort_recursive(unsorted_array, array_size,
              rank, size - 1, rankPower);
145     }else{
146         // All other Clusters wait for their subarray to
              arrive,
147         // they sort it and they send it back.
148         MPI_Status status;
149         int subarray_size;
150         MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,
              MPI_COMM_WORLD, &status);
151         // Capturing size of the array to receive
152         MPI_Get_count(&status, MPI_INT, &subarray_size);
153       int source_process = status.MPI_SOURCE;
154         int subarray[subarray_size];
```

```
155         MPI_Recv(subarray, subarray_size, MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
156         quicksort_recursive(subarray, subarray_size, rank
                , size - 1, rankPower);
157         MPI_Send(subarray, subarray_size, MPI_INT,
                source_process, 0, MPI_COMM_WORLD);
158     };
159
160     if(rank==0){
161         finish_timer = MPI_Wtime();
162       printf("Total time for %d Clusters : %2.2f sec \n",
             size, finish_timer-start_timer);
163
164         // --- VALIDATION CHECK ---
165         printf("Checking.. \n");
166         bool error = false;
167         int i=0;
168         for(i=0;i<SIZE-1;i++) {
169             if (unsorted_array[i] > unsorted_array[i+1]){
170             error = true;
171                 printf("error in i=%d \n", i);
172             }
173         }
174         if(error)
175             printf("Error..Not sorted correctly\n");
176         else
177             printf("Correct!\n");
```

```
178     }
179
180     MPI_Finalize();
181     // End of Parallel Execution
182     return 0;
183 }
```

CAPÍTULO II

EXPERIMENTACIÓN

Para la experimentación se han realizado 4 tipos de pruebas:

Datos(s)	Procesos(m)
1024	1,4,9
4096	1,4,9
10000	1,4,9,16
40000	1,4,9,16,25

Tabla 2.1: Datos versus procesos.

2.1 Comparativa de tiempos

Para medir los tiempos de la experimentación con los teóricos, se ha hecho un análisis a nivel de eficiencia. Suponemos que el tiempo de ejecución disminuye idealmente con el aumento del número de procesos, lo cual raramente es el caso en la práctica debido a la sobrecarga de comunicación y sincronización. También asumimos que el trabajo se divide de manera uniforme entre los procesos.

Podemos entonces modelar el tiempo ideal usando una función que simule la disminución de la eficiencia con el aumento de procesos. Una forma simple de hacer esto es usando una función logarítmica o una raíz cuadrada para moderar la reducción del tiempo. Esto es prácticamente el tiempo paralelo igual al tiempo secuencial sobre la raíz de p.

De esta manera podemos mostrar la gráfica de tiempos en comparativa:

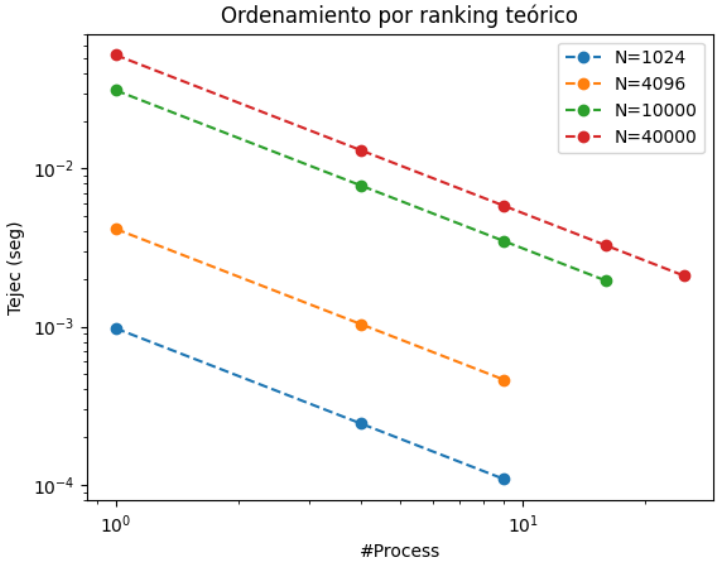

FIGURA 2.1: Gráfica de tiempos ideal.

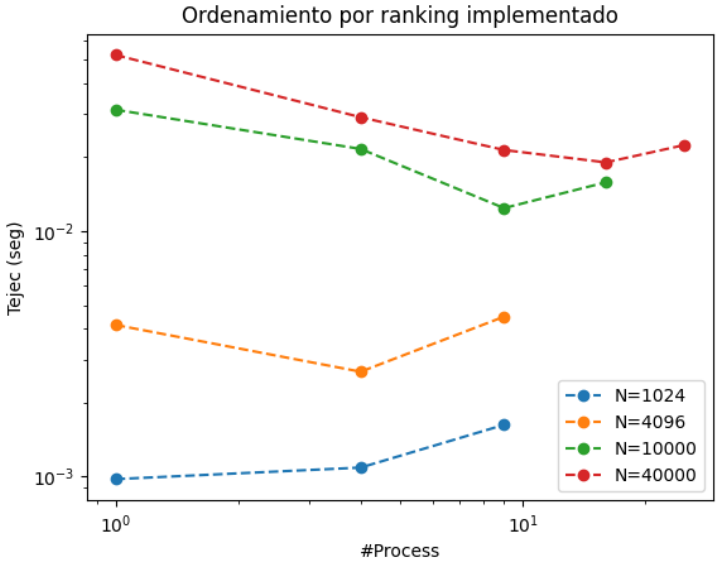

FIGURA 2.2: Gráfica de tiempos de la implementación.

```
marce@DESKTOP-CIH1MKB:~/Paralela/Proyecto/Ranking$ mpiexec -n 1 ./rankingsort 40000
Execution Time: 0.0522460938
Scatter Time: 0.0000000000
Gossip Time: 0.0000000000
Broadcast Time: 0.0000000000
Process Time: 0.0517578125
Tiempo de cómputo: 0.0104980469 segundos.
Tiempo de comunicación: 0.0256347656 segundos.
```

FIGURA 2.3: Captura de medición de tiempos

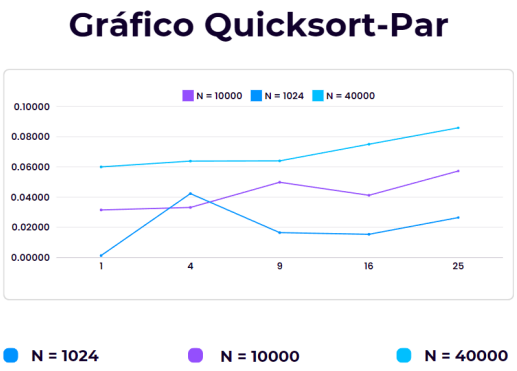

FIGURA 2.4: Quicksort Paralelo

Cabe aclarar que también se ha hecho una medición de los tiempos de cómputo y comunicación, a la vez que de Scatter, Gossip, Broadcast y de procesamiento. En la figura que muestra la captura de pantalla se puede apreciar la medición a nivel de consola. En este caso, los tiempos de Gossip, Scatter y Broadcast son cero por utilizar un sólo proceso. Vea la figura 2.3.

Comparando con los tiempos que da el código de Quicksort, tenemos lo referente a la gráfica de quicksort en la figura 2.4.

Análisis de la Experimentación con `rankingsort_mpi.cpp`

La experimentación con el código `rankingsort_mpi.cpp` ha revelado diferencias significativas entre los tiempos de ejecución teóricos e ideales. Estas diferencias

se pueden atribuir a varios factores inherentes a la computación paralela y la implementación específica del algoritmo de ordenamiento por ranking.

Factores que Afectan los Tiempos de Ejecución

1. **Sobrecarga de Comunicación:** En un entorno de MPI, la comunicación entre procesos puede introducir una sobrecarga significativa. Cada vez que un proceso envía o recibe datos, se incurre en un costo de tiempo que no está presente en los cálculos teóricos. En el código `rankingsort_mpi.cpp`, operaciones como `MPI_Sendrecv` y `MPI_Scatter` son críticas y su eficiencia depende de la latencia y el ancho de banda de la red subyacente.

2. **Desbalance de Carga:** Aunque el algoritmo intenta distribuir la carga de manera uniforme entre los procesos, diferencias en la distribución de datos o en la capacidad de procesamiento pueden llevar a que algunos procesos terminen su trabajo más rápidamente que otros. Esto resulta en tiempos de espera (idle times) donde algunos procesos están inactivos, esperando que otros terminen.

3. **Costos de Sincronización:** Las barreras y otras sincronizaciones (`MPI_Barrier`) aseguran que todos los procesos alcancen ciertos puntos de manera simultánea. Estas sincronizaciones son necesarias para la coherencia de los datos pero introducen retrasos que no son considerados en los modelos teóricos ideales.

4. **Eficiencia del Algoritmo:** La eficiencia del algoritmo de ordenamiento y la manera en que se manejan los datos (por ejemplo, la función `mergeData` y `computeLocalRank`) también influyen en el rendimiento. La complejidad computacional de estas funciones impacta directamente en el tiempo total de ejecución.

CONCLUSIONES

2.2 Conclusiones generales

El desarrollo y análisis del proyecto han permitido validar tanto el diseño teórico como la implementación práctica del algoritmo de ordenamiento por ranking paralelo. A continuación, se destacan los puntos clave y los aspectos a mejorar.

Para empezar, es evidente que la comunicación no optimizada muestra un rendimiento similar a la optimizada para valores pequeños de p, aunque las diferencias se hacen evidentes al aumentar el número de procesos. El algoritmo de ordenamiento por ranking muestra una escalabilidad prometedora, especialmente para conjuntos de datos de gran tamaño (n grandes), debido a su estructura paralela eficiente. Comparado con el algoritmo Quicksort paralelo, el ordenamiento por ranking ofrece un mejor rendimiento en términos de escalabilidad, validando así la eficacia de la implementación desarrollada.

2.2.1 Posibles mejoras

A pesar de los buenos resultados obtenidos, se identifican varias áreas de mejora que podrían optimizar aún más el rendimiento del algoritmo. Además se debería haber probado con una cantidad de datos más grandes, pero las limitaciones técnicas fueron bastante fuertes.

Además, el local ranking no está del todo optimizado. Se puede lograr una mejor implementación, pudiendo llegar a una complejidad óptima logarítmica.

En general, los resultados obtenidos validan el enfoque teórico planteado, demostrando que el algoritmo de ordenamiento por ranking paralelo es una solución eficiente y escalable para grandes volúmenes de datos. Con las mejoras propuestas,

se espera que el rendimiento del algoritmo pueda acercarse aún más a los límites teóricos establecidos.