Ordenamiento eficiente por ranking

Computación Paralela y Distribuida (CS4052)

Proyecto Final

Integrantes:
Sofía Valeria García Quintana (100 %)
Martin Gustavo Pérez Bonany Torrealva (100 %)
Paolo Vásquez Grahammer (100 %)

Universidad de Ingeniería y Tecnología

Facultad de Computación

Docente: Jose Antonio Fiestas Iquira

2024-2

Índice general

Capítulo 1

Introducción

1.1. Descripción general del problema

El problema de ordenamiento es una tarea fundamental en la computación, que consiste en organizar un conjunto de datos en un orden específico. Existen varios algoritmos para resolverlo como Quicksort y Mergesort, pero cuando los conjuntos de datos son muy grandes, el tiempo de ejecución puede volverse un obstáculo. Para solucionar esto, se busca paralelizar el proceso de ordenamiento, es decir, dividir el trabajo entre varios procesadores para realizar las operaciones de manera simultánea y así reducir el tiempo total de ejecución. El reto aquí es encontrar una manera eficiente de distribuir y organizar el trabajo entre los procesos.

1.2. Objetivos

El propósito de este proyecto es crear un algoritmo eficiente de ordenamiento por ranking en paralelo, utilizando el modelo PRAM y código C++ con MPI. Además, se medirá el tiempo de ejecución y se harán comparaciones de velocidad con Quicksort. Finalmente, se busca mejorar la escalabilidad del algoritmo probando diferentes configuraciones de procesos p y tamaños de datos n.

1.3. Solución planteada

La solución usa el modelo PRAM, que permite dividir el trabajo de ordenamiento entre múltiples procesos. Los datos se distribuyen en una cuadrícula de procesos, y se utilizan dos pasos de comunicación entre ellos: **gossip** para compartir los datos en las columnas y **broadcast** para compartirlos en las filas. Después, cada proceso ordena sus datos, calcula el ranking local y finalmente se suman todos los rankings para obtener el resultado final.

Este enfoque se implementa con MPI, que permite que los procesos se comuniquen de manera eficiente. De esta forma, el algoritmo se vuelve más rápido al aprovechar el paralelismo, y se compara con el rendimiento de Quicksort para ver cuál es más eficiente.

Capítulo 2

Método

2.1. Ordenamiento por ranking

El ordenamiento por ranking paralelo es una forma de organizar datos en la que se asigna a cada elemento un *ranking* según **cuántos elementos en el conjunto son menores que él**. Para hacerlo de manera eficiente, el trabajo se divide entre varios procesadores que trabajan al mismo tiempo.

Los datos se distribuyen entre $p = P \cdot P$ procesos organizando una cuadrícula de P filas y P columnas. Los elementos se reparten equitativamente (**scatter**), donde cada proceso recibe $N = n/p$ elementos. Los procesos comparten información entre ellos en dos etapas. En la primera, cada proceso comparte los N datos que recibió con los $P - 1$ procesos que están en su misma columna (**gossip**). Luego, en la segunda etapa, cada proceso que se encuentra en la diagonal principal de la cuadrícula ($p(i, j)$ donde $i = j$) toma los $N \cdot P$ elementos que tiene y los comparte con los demás procesos de su misma fila (**broadcast**).

Ahora, cada proceso **ordena** localmente los $N \cdot P$ elementos que acaba de recibir en el *broadcast*. Una vez ordenados, el proceso calcula el **ranking local** de cada elemento, que indica cuantos datos del conjunto que se recibió en el *gossip* son menores que el elemento. Por último, un proceso de cada fila se encarga de recibir los rankings calculados por lo demás procesos de su fila y los suma para generar un ranking global para cada elemento (**reduce**). Al terminar este proceso, todos los rankings están completos y cada proceso tiene el resultado correspondiente a su parte del conjunto de datos, por lo que recogemos el resultado en el proceso maestro (**gather**).

2.1.1. Modelo PRAM

Algorithm 1 Algoritmos de Comunicación

function SCATTER(d, p) ▷ $O(p \cdot (\alpha + \frac{n}{p} \cdot \beta))$
    Distribuir un array d entre $p = P^2$ procesos, cada uno recibe $\frac{n}{p}$ elementos
end function

function GOSSIP(p, P, N)
    for $j := 1, \ldots, P$ **do** ▷ $O(P) = O(\sqrt{p})$
        Compartir los $N = \frac{n}{p}$ elementos entre los P procesos de la misma columna
        ▷ Como un broadcast –>$O(\log P \cdot (\alpha + \frac{n}{p} \cdot \beta))$
    end for
end function

function BROADCAST(p, P, N) ▷ $O(\log P \cdot (\alpha + \frac{n}{\sqrt{p}} \cdot \beta))$
    Los procesos p_{ij} donde $i = j$ comparten sus $N \cdot P$ datos con los procesos en su fila
end function

function REDUCE($rankings, P$) ▷ $O(\log P \cdot (\alpha + \frac{n}{\sqrt{p}} \cdot \beta))$
    Enviar los $N \cdot P$ rankings locales al proceso p_{ij} donde $i = j$ de cada fila
end function

function GATHER($results, P$) ▷ $O(\sqrt{p} \cdot \alpha + \frac{n}{p} \cdot \beta)$
    Recoger los rankings globales de cada proceso p_{ij} donde $i = j$
end function

Algorithm 2 Algoritmos Locales

function LOCALSORT(a) ▷ $O(\frac{n}{\sqrt{p}} \log \frac{n}{\sqrt{p}})$
    Cada proceso ordena localmente los $N \cdot P$ elementos que recibió en el broadcast
end function

function LOCALRANKING(a', b) ▷ $O(\frac{n}{\sqrt{p}} \log \frac{n}{\sqrt{p}})$
    for $i := 1, \ldots, N \cdot P$ **do**
        $rank[i] \leftarrow$ lower_bound$(b, a'[i])$
            ▷ Búsqueda binaria para contar cuántos elementos en b son menores que $a'[i]$
    end for
end function

Algorithm 3 Ordenamiento por Ranking PRAM

1: **Input:** $d[1, \ldots, n]$
2: **Output:** Rankings de los elementos de d

3: SCATTER(d, p)
4: GOSSIP(p, P, N)
5: BROADCAST(p, P, N)

6: **for** $i = 1, \ldots, P^2$ **par do**
7: LOCALSORT(a) ▷ Cada proceso ordena localmente los datos
8: **end for**

9: **for** $i = 1, \ldots, P^2$ **par do**
10: LOCALRANKING(a', b) ▷ Cada proceso calcula el ranking local de los elementos
11: **end for**

12: REDUCE$(rankings, P)$
13: GATHER$(results, P)$
▷ Los rankings globales son recogidos por el proceso maestro

2.1.2. Complejidad teórica

Del PRAM anterior, se puede derivar que la complejidad algorítmica se reduce a lo siguiente:

$$T(n,p) = T_{\text{scatter}} + T_{\text{gossip}} + T_{\text{broadcast}} + T_{\text{sort}} + T_{\text{ranking}} + T_{\text{reduce}} + T_{\text{gather}}$$

Dependiendo de si la comunicación está optimizada, algunos de los términos anteriores varían. En general, se mantienen los siguientes:

- $T_{\text{scatter}} = p\left(\alpha + \frac{n}{p}\beta\right)$
- T_{sort} y $T_{\text{ranking}} = \frac{n}{\sqrt{p}}\log\left(\frac{n}{\sqrt{p}}\right)$
- $T_{\text{gather}} = \sqrt{p}\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)$

A continuación, veremos cómo varía la complejidad dependiendo de la optimización.

Ideal

Esta se consigue cuando las operaciones de *broadcast* y *reduce*, análogas, siguen un esquema de árbol para comunicar, haciendo que sean proporcionales a $\log(p)$. Con ello se tiene que:

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

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

Con ello se obtiene que:

$$\begin{aligned} T_{\text{ideal}}(n,p) = O\left(p\left(\alpha + \frac{n}{p}\beta\right)\right) + O\left(2\log\left(\sqrt{p}\right)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)\right) \\ + O\left(\sqrt{p}\cdot\log\left(\sqrt{p}\right)\left(\alpha + \frac{n}{p}\beta\right)\right) + O\left(2\frac{n}{\sqrt{p}}\log\left(\frac{n}{\sqrt{p}}\right)\right) \\ + O\left(\sqrt{p}\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)\right) \end{aligned}$$

Implementada

De manera sencilla, lo implementado se consigue obviando ese esquema de árbol, con los términos anteriores proporcionales a p. Tenemos que:

- $T_{\text{gossip}} = p(\alpha + \frac{n}{p}\beta)$
- $T_{\text{broadcast}}$ y $T_{\text{reduce}} = \sqrt{p}(\alpha + \frac{n}{\sqrt{p}}\beta)$

Se tiene entonces que:

$$T_{\text{imp}}(n,p) = O\big(2p(\alpha + \frac{n}{p}\beta)\big) + O\big(3\sqrt{p}(\alpha + \frac{n}{\sqrt{p}}\beta)\big) + O\big(2\frac{n}{\sqrt{p}}\log(\frac{n}{\sqrt{p}}))\big)$$

Expresión final

Para las siguientes secciones se tomará:

$$T(n,p) = O\big(2p(\alpha + \frac{n}{p}\beta)\big) + O\big(3\sqrt{p}(\alpha + \frac{n}{\sqrt{p}}\beta)\big) + O\big(2\frac{n}{\sqrt{p}}\log(\frac{n}{\sqrt{p}})\big)$$

Recordando que:

- $T_{\text{comp}} : O\big(\frac{n}{\sqrt{p}}\log(\frac{n}{\sqrt{p}})\big) + O\big(\frac{n^2}{p}\big)$
- $T_{\text{comm}} : O\big(2p(\alpha + \frac{n}{p}\beta)\big) + O\big(3\sqrt{p}(\alpha + \frac{n}{\sqrt{p}}\beta)\big)$
- α: Latencia
- β: Ancho de banda

2.1.3. Código MPI

La implementación del ordenamiento eficiente por ranking ha sido realizado en C++ con MPI. El recuento del desarrollo se encuentra en el siguiente repositorio. en GitHub [1]. De igual manera se consigan las tres versiones principales.

Versión inicial

Básicamente en esta etapa se tiene desde la repartición de la data (scatter) hasta el procedimiento Gossip, donde los procesos de una columna tienen los mismos datos.

```
01 |
02 | #include <mpi.h>
03 | #include <iostream>
04 | #include <vector>
05 | #include <string>
06 | #include <algorithm>
07 | #include <map>
08 | using namespace std;
09 | // Concat strings en orden por rank
10 | string concatenate(const map<int, string>& data_by_rank) {
11 |     string result;
12 |     for (const auto& entry : data_by_rank) {
13 |         result += entry.second;
14 |     }
15 |     return result;
16 | }
17 | int main(int argc, char** argv) {
18 |     MPI_Init(&argc, &argv);
19 |     int rank, size;
20 |     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
21 |     MPI_Comm_size(MPI_COMM_WORLD, &size);
22 |     const int rows = 2;
23 |     const int cols = 2;
24 |     if (size != rows * cols) {
25 |         if (rank == 0) {
26 |             cerr << "Error: This program requires exactly " << rows * cols <<
       " processes." << endl;
27 |         }
28 |         MPI_Finalize();
29 |         return 1;
30 |     }
31 |     int row = rank / cols;
32 |     int col = rank % cols;
33 |     map<int, string> local_data = {{rank, string(1, 'a' + rank) + string(1, '
       a' + rank + 13)}};
34 |     cout << "Process " << rank << " (Row " << row << ", Col " << col << ")
       starts with: "
35 |          << concatenate(local_data) << endl;
36 |     char recv_buffer[200];
37 |     for (int step = 0; step < rows - 1; ++step) {
38 |         int send_to = ((row + 1) % rows) * cols + col;      // misma col pero
        abajo
39 |         int receive_from = ((row + rows - 1) % rows) * cols + col; // arriba
       misma col
40 |         string send_buffer = concatenate(local_data);
41 |         MPI_Request send_request, recv_request;
42 |         MPI_Irecv(recv_buffer, 200, MPI_CHAR, receive_from, 0, MPI_COMM_WORLD
       , &recv_request);
43 |         MPI_Isend(send_buffer.c_str(), send_buffer.size() + 1, MPI_CHAR,
       send_to, 0, MPI_COMM_WORLD, &send_request);
44 |         MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
45 |         MPI_Wait(&send_request, MPI_STATUS_IGNORE);
46 |         string received_string(recv_buffer);
47 |         int source_rank = (rank - cols + size) % size; // basado en la pos,
```

```
    calcular el rank de quien envi  la data
48 |         local_data[source_rank] = received_string;
49 |     }
50 |     string result = concatenate(local_data);
51 |     cout << "Process " << rank << " (Row " << row << ", Col " << col << ")
    ends with: " << result << endl;
52 |     MPI_Finalize();
53 |     return 0;
54 | }
55 |
56 |
```

Listing 2.1: commit: Gossip funcional

Versión preliminar

Se agrega el resto del algoritmo, que consiga el broadcast, ordenamiento y ranking local, la reducción en los procesos de la diagonal, y el gather final en el maestro, teniendo ahí el ordenamiento final.

```
01 |
02 | #include <map>
03 | #include <mpi.h>
04 | #include <string>
05 | #include <vector>
06 | #include <iterator>
07 | #include <iostream>
08 | #include <algorithm>
09 | using namespace std;
10 | string concatenar(const map<int, string>& data_by_rank) {
11 |     string result;
12 |     for (const auto& entry : data_by_rank) {
13 |         result += entry.second;
14 |     }
15 |     return result;
16 | }
17 | vector<int> local_rank(const string& local_A, const string& A) {
18 |     vector<int> rank_counts(A.size(), 0);
19 |     for (size_t i = 0; i < A.size(); i++) {
20 |         for (size_t j = 0; j < local_A.size(); j++) {
21 |             if (local_A[j] <= A[i]) {
22 |                 rank_counts[i]++;
23 |             }
24 |         }
25 |     }
26 |     return rank_counts;
27 | }
28 |
29 | void gossip_step(int rank, int rows, int cols, int size, map<int, string>&
    local_data) {
30 |     int row = rank / cols;
31 |     int col = rank % cols;
32 |     char recv_buffer[200];
33 |     for (int step = 0; step < rows - 1; ++step) {
34 |         int send_to = ((row + 1) % rows) * cols + col;      // same col but
    below
```

```
35 |         int receive_from = ((row + rows - 1) % rows) * cols + col; // above
       same col
36 |         string send_buffer = concatenar(local_data);
37 |         MPI_Request send_request, recv_request;
38 |         MPI_Irecv(recv_buffer, 200, MPI_CHAR, receive_from, 0, MPI_COMM_WORLD
       , &recv_request);
39 |         MPI_Isend(send_buffer.c_str(), send_buffer.size() + 1, MPI_CHAR,
       send_to, 0, MPI_COMM_WORLD, &send_request);
40 |         MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
41 |         MPI_Wait(&send_request, MPI_STATUS_IGNORE);
42 |         string received_string(recv_buffer);
43 |         int source_rank = (rank - cols + size) % size; // calculate the rank
       of the data sender
44 |         local_data[source_rank] = received_string;
45 |     }
46 | }
47 |
48 | void reverse_broadcast_step(int rank, int rows, int cols, const string&
       starting_data, map<int, string>& resulting_data) {
49 |     int row = rank / cols;
50 |     int col = rank % cols;
51 |     char recv_buffer[200];
52 |     if (col == row) {
53 |         for (int c = 0; c < cols; ++c) {
54 |             if (c != col) {
55 |                 int send_to = row * cols + c;
56 |                 MPI_Send(starting_data.c_str(), starting_data.size() + 1,
       MPI_CHAR, send_to, 0, MPI_COMM_WORLD);
57 |             }
58 |         }
59 |         resulting_data[0] = starting_data;
60 |     } else {
61 |         int send_from = row * cols + row;
62 |         MPI_Recv(recv_buffer, 200, MPI_CHAR, send_from, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
63 |         string received_string(recv_buffer);
64 |         resulting_data[0] = received_string;
65 |     }
66 | }
67 | string sort_and_print_by_rank(const vector<int>& aggregated_ranks, const
       string& result) {
68 |     vector<pair<int, char>> rank_with_indices;
69 |     for (size_t i = 0; i < result.size(); ++i) {
70 |         rank_with_indices.emplace_back(aggregated_ranks[i], result[i]);
71 |     }
72 |     sort(rank_with_indices.begin(), rank_with_indices.end());
73 |     string sorted_result;
74 |     for(const auto& rank : rank_with_indices) {
75 |         sorted_result += rank.second;
76 |     }
77 |     cout << "\n========================== Aggregated Ranks: ";
78 |     copy(aggregated_ranks.begin(), aggregated_ranks.end(), ostream_iterator<
       int>(cout, " "));
79 |     cout << endl;
80 |     cout << "========================== Reordered String:" << sorted_result
       << endl;
81 |     return sorted_result;
82 | }
83 | pair<string, vector<int>> calculate_and_print_ranks(int rank, int rows, int
       cols, const string& starting_data, const string& result) {
```

```
84 |     vector<int> local_ranking = local_rank(starting_data, result);
85 |     string sorted_result;
86 |     cout << "\n-=-=-=-=-=-= Process " << rank << " (Row " << (rank / cols) << "
   |     , Col " << (rank % cols) << ") has:\n";
87 |     cout << "Local ranks: ";
88 |     copy(local_ranking.begin(), local_ranking.end(), ostream_iterator<int>(
   |     cout, " "));
89 |     cout << endl;
90 |     MPI_Barrier(MPI_COMM_WORLD);
91 |     int row = rank / cols;
92 |     int col = rank % cols;
93 |     int diagonal_process = row * cols + row;
94 |     char recv_buffer[200];
95 |     string recv_word;
96 |     if (col != row) {
97 |         MPI_Send(local_ranking.data(), local_ranking.size(), MPI_INT,
   |     diagonal_process, 0, MPI_COMM_WORLD);
98 |         cout << "Process " << rank << " sent ranks to diagonal process " <<
   |     diagonal_process << endl;
99 |     } else {
100|         vector<int> aggregated_ranks(local_ranking.size(), 0);
101|         for (int c = 0; c < cols; ++c) {
102|             if (c != col) {
103|                 vector<int> received_ranks(local_ranking.size());
104|                 MPI_Recv(received_ranks.data(), received_ranks.size(),
   |     MPI_INT, row * cols + c, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
105|                 for (size_t i = 0; i < aggregated_ranks.size(); ++i) {
106|                     aggregated_ranks[i] += received_ranks[i];
107|                 }
108|             } else {
109|                 for (size_t i = 0; i < aggregated_ranks.size(); ++i) {
110|                     aggregated_ranks[i] += local_ranking[i];
111|                 }
112|             }
113|         }
114|         sorted_result = result;
115|         if (rank != 0) {
116|             MPI_Send(aggregated_ranks.data(), aggregated_ranks.size(),
   |     MPI_INT, 0, 1, MPI_COMM_WORLD);
117|             MPI_Send(sorted_result.c_str(), sorted_result.size() + 1,
   |     MPI_CHAR, 0, 1, MPI_COMM_WORLD);
118|             cout << "Diagonal process " << rank << " sent aggregated ranks to
   |      Process 0" << endl;
119|         } else {
120|             vector<int> global_ranks(aggregated_ranks);
121|             for (int r = 1; r < rows; ++r) {
122|                 vector<int> received_ranks(aggregated_ranks.size());
123|                 MPI_Recv(received_ranks.data(), received_ranks.size(),
   |     MPI_INT, r * cols + r, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
124|                 MPI_Recv(recv_buffer, 200, MPI_CHAR, r * cols + r, 1,
   |     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
125|                 string received_string(recv_buffer);
126|                 recv_word = received_string;
127|                 cout << "RECV STRING: " << received_string << endl;
128|                 cout << "Process 0 received aggregated ranks from diagonal
   |     process " << (r * cols + r) << endl;
129|                 global_ranks.insert(global_ranks.end(), received_ranks.begin
   |     (), received_ranks.end());
130|             }
131|             sorted_result = sort_and_print_by_rank(global_ranks, (result +
```

```
    recv_word));
132 |         }
133 |     }
134 |     return {sorted_result, local_ranking};
135 | }
136 | int main(int argc, char** argv) {
137 |     MPI_Init(&argc, &argv);
138 |     int rank, size;
139 |     const int rows = 2;
140 |     const int cols = 2;
141 |     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
142 |     MPI_Comm_size(MPI_COMM_WORLD, &size);
143 |     string local_string;
144 |     vector<int> local_vector;
145 |     pair<string, vector<int>> final_output;
146 |     if (size != rows * cols) {
147 |         if (rank == 0) cerr << "Error: This program requires exactly " <<
    rows * cols << " processes." << endl;
148 |         MPI_Finalize();
149 |         return 1;
150 |     }
151 |     int row = rank / cols;
152 |     int col = rank % cols;
153 |     map<int, string> local_data = {
154 |         {rank, string(1, 'a' + rank) +
155 |                 string(1, 'a' + rank + 13) +
156 |                 string(1, 'a' + rank + 8) +
157 |                 string(1, 'a' + rank + 21)}
158 |     };
159 |     map<int, string> resulting_data;
160 |     cout << "Process " << rank << " (Row " << row << ", Col " << col << ")
    starts with: " << concatenar(local_data) << endl;
161 |     // ------------------------------------------------
162 |     gossip_step(rank, rows, cols, size, local_data);
163 |     // ------------------------------------------------ Local data contiene la
    informaci n compartida en el gossip.
164 |     string gossip_result = concatenar(local_data);
165 |     // ------------------------------------------------
166 |     reverse_broadcast_step(rank, rows, cols, gossip_result, resulting_data);
167 |     // ------------------------------------------------ Resulting data contiene
     la informaci n compartida en el bcast.
168 |     string result2 = concatenar(resulting_data);
169 |     // ------------------------------------------------ Realiza el paso de sort
     previo al local ranking. Utiliza quick sort.
170 |     sort(gossip_result.begin(), gossip_result.end());
171 |     // ------------------------------------------------ Local ranking y Reduce
172 |     MPI_Barrier(MPI_COMM_WORLD);
173 |     final_output = calculate_and_print_ranks(rank, rows, cols, gossip_result,
     result2);
174 |     MPI_Finalize();
175 |     cout << endl;
176 |     return 0;
177 | }
178 |
179 |
```

Listing 2.2: commit: gossip, bcast, sort, local_ranking y reduce

Versión final

Se hacen añadidos a lo anterior: mediciones de tiempo, se optimiza el cálculo del local ranking, haciendolo $O\left(\frac{n}{\sqrt{p}} \cdot \lg\left(\frac{n}{\sqrt{p}}\right)\right)$, y cuestiones de input. Se usará esta en las siguientes secciones.

```
01 |
02 | #include <map>
03 | #include <mpi.h>
04 | #include <string>
05 | #include <vector>
06 | #include <iterator>
07 | #include <iostream>
08 | #include <algorithm>
09 | #include <random>
10 | #include <iomanip> // Para std::setprecision
11 | using namespace std;
12 |
13 | float t1,t2,t3,t4,t5,t6,t7,t8;
14 | float t9, t10, t11, t12, t13, t14, t15, t16;
15 | float t_inicial, t_final;
16 |
17 | string generateRandomString(size_t length) {
18 |     const string_view characters = "
    |     ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
19 |     random_device rd;
20 |     mt19937 generator(rd());
21 |     uniform_int_distribution<size_t> distribution(0, characters.size() - 1);
22 |
23 |     string randomString;
24 |     randomString.reserve(length);
25 |
26 |     generate_n(back_inserter(randomString), length, [&]() {
27 |         return characters[distribution(generator)];
28 |     });
29 |
30 |     return randomString;
31 | }
32 |
33 | string concatenar(const map<int, string>& data_by_rank) {
34 |     string result;
35 |     for (const auto& entry : data_by_rank) {
36 |         result += entry.second;
37 |     }
38 |     return result;
39 | }
40 |
41 | vector<int> local_rank(const string& local_A, const string& A) {
42 |     vector<int> rank_counts(A.size(), 0);
43 |
44 |     for (size_t i = 0; i < A.size(); i++) {
45 |         rank_counts[i] = lower_bound(local_A.begin(), local_A.end(), A[i]) -
    |     local_A.begin();
46 |         }
47 |
48 |     return rank_counts;
49 | }
50 | void gossip_step(int rank, int rows, int cols, int size, map<int, string>&
    |     local_data) {
```

```
51 |     int row = rank / cols;
52 |     int col = rank % cols;
53 |     char recv_buffer[10000];
54 |
55 |     for (int step = 0; step < rows - 1; ++step) {
56 |         int send_to = ((row + 1) % rows) * cols + col;      // same col but
       below
57 |         int receive_from = ((row + rows - 1) % rows) * cols + col; // above
       same col
58 |
59 |         string send_buffer = concatenar(local_data);
60 |
61 |         MPI_Request send_request, recv_request;
62 |         MPI_Irecv(recv_buffer, 10000, MPI_CHAR, receive_from, 0,
       MPI_COMM_WORLD, &recv_request);
63 |         MPI_Isend(send_buffer.c_str(), send_buffer.size() + 1, MPI_CHAR,
       send_to, 0, MPI_COMM_WORLD, &send_request);
64 |
65 |         MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
66 |         MPI_Wait(&send_request, MPI_STATUS_IGNORE);
67 |
68 |         string received_string(recv_buffer);
69 |         int source_rank = (rank - cols + size) % size; // calculate the rank
       of the data sender
70 |         local_data[source_rank] = received_string;
71 |     }
72 | }
73 |
74 | void reverse_broadcast_step(int rank, int rows, int cols, const string&
       starting_data, map<int, string>& resulting_data) {
75 |     int row = rank / cols;
76 |     int col = rank % cols;
77 |     char recv_buffer[10000];
78 |
79 |     if (col == row) {
80 |         for (int c = 0; c < cols; ++c) {
81 |             if (c != col) {
82 |                 int send_to = row * cols + c;
83 |                 MPI_Send(starting_data.c_str(), starting_data.size() + 1,
       MPI_CHAR, send_to, 0, MPI_COMM_WORLD);
84 |             }
85 |         }
86 |         resulting_data[0] = starting_data;
87 |     } else {
88 |         int send_from = row * cols + row;
89 |         MPI_Recv(recv_buffer, 10000, MPI_CHAR, send_from, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
90 |         string received_string(recv_buffer);
91 |         resulting_data[0] = received_string;
92 |     }
93 | }
94 | string sort_and_print_by_rank(const vector<int>& aggregated_ranks, const
       string& result) {
95 |     vector<pair<int, char>> rank_with_indices;
96 |
97 |     for (size_t i = 0; i < result.size(); ++i) {
98 |         rank_with_indices.emplace_back(aggregated_ranks[i], result[i]);
99 |     }
100 |
101 |     sort(rank_with_indices.begin(), rank_with_indices.end());
```

```
102 |     string sorted_result;
103 |
104 |     for(const auto& rank : rank_with_indices) {
105 |         sorted_result += rank.second;
106 |     }
107 |     return sorted_result;
108 | }
109 |
110 | string calculate_and_print_ranks(int rank, int rows, int cols, const string&
        starting_data, const string& result) {
111 |     string sorted_starting_data = starting_data;
112 |
113 |     //SORT (4)
114 |
115 |     t7 = MPI_Wtime();
116 |     sort(sorted_starting_data.begin(), sorted_starting_data.end());
117 |     t8 = MPI_Wtime();
118 |
119 |     // LOCAL RANKING (5)
120 |
121 |     t9 = MPI_Wtime();
122 |     vector<int> local_ranking = local_rank(sorted_starting_data, result);
123 |     t10 = MPI_Wtime();
124 |
125 |     string sorted_result;
126 |
127 |     MPI_Barrier(MPI_COMM_WORLD);
128 |
129 |     int row = rank / cols;
130 |     int col = rank % cols;
131 |     int diagonal_process = row * cols + row;
132 |     char recv_buffer[10000];
133 |     string recv_word;
134 |
135 |     // REDUCE (6)
136 |     // Enviar los datos a las diagonales
137 |
138 |     t11 = MPI_Wtime();
139 |     if (col != row)
140 |     {
141 |         MPI_Send(local_ranking.data(), local_ranking.size(), MPI_INT,
        diagonal_process, 0, MPI_COMM_WORLD);
142 |         // cout << "Process " << rank << " sent ranks to diagonal process "
        << diagonal_process << endl; (COMENTADO)
143 |     }
144 |     else
145 |     {
146 |         // Los ranks de la diagonal
147 |         vector<int> aggregated_ranks(local_ranking.size(), 0);
148 |         for (int c = 0; c < cols; ++c)
149 |         {
150 |             // recibir desde la diagonal por cada uno y sumarlo al proceso
        principal
151 |             if (c != col)
152 |             {
153 |                 vector<int> received_ranks(local_ranking.size());
154 |                 MPI_Recv(received_ranks.data(), received_ranks.size(),
        MPI_INT, row * cols + c, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
155 |                 for (size_t i = 0; i < aggregated_ranks.size(); ++i)
156 |                 {
```

```
157 |                     aggregated_ranks[i] += received_ranks[i];
158 |                 }
159 |             }
160 |             else
161 |             {   // si es el diagonal, se le suma igual
162 |                 for (size_t i = 0; i < aggregated_ranks.size(); ++i)
163 |                 {
164 |                     aggregated_ranks[i] += local_ranking[i];
165 |                 }
166 |             }
167 |         }
168 |         t12 = MPI_Wtime();
169 |
170 |         // GATHER (6)
171 |         // Despu s cada diagonal env a su ranking y string al proceso 0
172 |
173 |         t13 = MPI_Wtime();
174 |         if (rank != 0)
175 |         {
176 |             MPI_Send(aggregated_ranks.data(), aggregated_ranks.size(),
    MPI_INT, 0, 1, MPI_COMM_WORLD);
177 |             MPI_Send(result.c_str(), result.size() + 1, MPI_CHAR, 0, 1,
    MPI_COMM_WORLD);
178 |             // cout << "Diagonal process " << rank << " sent aggregated ranks
     to Process 0" << endl; (COMENTADO)
179 |         }
180 |         else
181 |         {
182 |             vector<int> global_ranks(aggregated_ranks);
183 |
184 |             for (int r = 1; r < rows; ++r)
185 |             {
186 |                 int d_proc = r * cols + r; // proceso diagonal en base al
    iterador r
187 |                 vector<int> received_ranks(aggregated_ranks.size());
188 |                 // Recibe la info de cada diagonal
189 |                 MPI_Recv(received_ranks.data(), received_ranks.size(),
    MPI_INT, d_proc, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
190 |                 MPI_Recv(recv_buffer, 10000, MPI_CHAR, d_proc, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
191 |                 // Tranforma la informaci n del buffer en una string
192 |                 string received_string(recv_buffer);
193 |                 recv_word += received_string;
194 |
195 |                 // cout << "Process 0 received ranks from diagonal process "
    << d_proc << endl; (COMENTADO)
196 |                 // Expande global ranks
197 |                 global_ranks.insert(global_ranks.end(), received_ranks.begin
    (), received_ranks.end());
198 |             }
199 |
200 |             t14 = MPI_Wtime();
201 |
202 |             t15 = MPI_Wtime();
203 |             sorted_result = sort_and_print_by_rank(global_ranks, (result +
    recv_word));
204 |             t16 = MPI_Wtime();
205 |         }
206 |     }
207 |     return sorted_result;
```

```
}

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);

    int sqrt_size = static_cast<int>(sqrt(size));
    if (sqrt_size * sqrt_size != size) {
        if (rank == 0) cerr << "Error: Number of processes must be a perfect square." << endl;
        MPI_Finalize();
        return 1;
    }

    const int rows = sqrt_size;
    const int cols = sqrt_size;

    if (argc < 2) {
        if (rank == 0) cerr << "Usage: mpiexec -n <num_processes> ./program <message_size>" << endl;
        MPI_Finalize();
        return 1;
    }

    int msg_size = atoi(argv[1])/size;
    if (msg_size <= 0) {
        if (rank == 0) cerr << "Error: Message size must be a positive integer." << endl;
        MPI_Finalize();
        return 1;
    }

    int total_elements = rows * cols;
    string input;

    if (rank == 0) {
        input = generateRandomString(msg_size * total_elements);
        if (input.size() % total_elements != 0) {
            cerr << "Input Size [" << input.size() << "] doesn't match row * col size [" << total_elements << "]" << endl;
            MPI_Finalize();
            return 1;
        }
    }

    char* local_string = new char[msg_size + 1];
    local_string[msg_size] = '\0';

    //SCATTER (1)

    t_inicial = MPI_Wtime();

    t1 = MPI_Wtime();
    MPI_Scatter(input.c_str(), msg_size, MPI_CHAR, local_string, msg_size, MPI_CHAR, 0, MPI_COMM_WORLD);
    t2 = MPI_Wtime();

```

```
263 |     string local_data_str(local_string);
264 |     delete[] local_string;
265 |
266 |     // cout << "Process " << rank << " received: " << local_data_str << endl;
      |       (COMENTADO)
267 |
268 |     map<int, string> local_data = {{rank, local_data_str}};
269 |     map<int, string> resulting_data;
270 |
271 |     // GOSSIP (2)
272 |
273 |     t3 = MPI_Wtime();
274 |     gossip_step(rank, rows, cols, size, local_data);
275 |     t4 = MPI_Wtime();
276 |     string gossip_result = concatenar(local_data);
277 |
278 |     // BROADCAST (3)
279 |
280 |     t5 = MPI_Wtime();
281 |     reverse_broadcast_step(rank, rows, cols, gossip_result, resulting_data);
282 |     t6 = MPI_Wtime();
283 |
284 |     string result2 = concatenar(resulting_data);
285 |
286 |     // SORT, LOCAL, RANKING, REDUCE Y GATHER
287 |     string final_output = calculate_and_print_ranks(rank, rows, cols,
      |     gossip_result, result2);
288 |
289 |     t_final = MPI_Wtime();
290 |
291 |     if (rank == 0) cout << "Final result: " << final_output << endl;
292 |
293 |     if (rank == 0)
294 |     {
295 |         cout << fixed << setprecision(10);
296 |
297 |         cout << "Ejecucion: " << ((t_final - t_inicial) - (t16 - t15)) <<
      |     endl;
298 |         cout << "Computo: " << ((t8 - t7) + (t10 - t9)) << endl;
299 |         cout << "Comunicacion: " << ((t_final - t_inicial) - (t16 - t15) - ((
      |     t8 - t7) + (t10 - t9))) << endl;
300 |     }
301 |
302 |     MPI_Finalize();
303 |     return 0;
304 | }
305 |
306 |
```

Listing 2.3: commit: ranking optimized and correct calcs

2.2. Quicksort en paralelo

Toda la explicación del *quicksort* en paralelo está basada en el repositorio *Quicksort-Parallel-MPI* del siguiente repositorio en GitHub [2]. Se está tomando la implementación *Parallel Merge Quicksort*, basada en división y unión.

En esta implementación, se divide el arreglo de tamaño n en p subarreglos, cada uno de tamaño n/p, para que cada proceso p ejecute un *quicksort* secuencial en su subarreglo. Luego, se deben combinar estos arreglos. El primer subarreglo se une con el segundo, el tercero con la unión de los dos anteriores, y así sucesivamente, y esto ocurre $(p-1)$ veces.

2.2.1. Complejidad teórica

Para derivar la complejidad, el quicksort en paralelo tiene dos partes principales: ordenamiento local y *merge*. Para lo primero, cada proceso realiza el quicksort en su arreglo de tamaño n/p, lo que se traduce en $n/p \cdot \log(n/p)$, y para lo segundo, se realiza un *merge* $(p-1)$ veces, donde el tamaño promedio de los subarreglos es $(n/2 + n/p)$, teniendo $(n/2 + n/p)(p-1)$. Finalmente, la complejidad es:

$$T(n,p) = \frac{n}{p} \cdot \log\left(\frac{n}{p}\right) + \left(\frac{n}{2} + \frac{n}{p}\right)(p-1)$$

2.2.2. Código MPI

La implementación tomada es *quicksort_ merge_ mpi.c*, y sigue la explicación anterior.

```
01 | #include "mpi.h"
02 | #include<stdio.h>
03 | #include <stdlib.h>
04 | #include "math.h"
05 | #include <stdbool.h>
06 | #define SIZE 1000000
07 |
08 | int hoare_partition(int *arr, int low, int high){
09 |     int middle = floor((low+high)/2);
10 |     int pivot = arr[middle];
11 |     int j,temp;
12 |     // move pivot to the end
13 |     temp=arr[middle];
14 |     arr[middle]=arr[high];
15 |     arr[high]=temp;
16 |
17 |     int i = (low - 1);
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 |     // move pivot back
27 |     temp=arr[i+1];
28 |     arr[i+1]=arr[high];
29 |     arr[high]=temp;
30 |
31 |     return (i+1);
32 | }
33 |
```

```
34 |
35 | /*
36 |     Simple sequential Quicksort Algorithm
37 | */
38 | void quicksort(int *number,int first,int last){
39 |     if(first<last){
40 |         int pivot_index = hoare_partition(number, first, last);
41 |         quicksort(number,first,pivot_index-1);
42 |         quicksort(number,pivot_index+1,last);
43 |     }
44 | }
45 |
46 | /*
47 |     Function that handles the merging of two sorted subarrays
48 |   and returns one bigger sorted array
49 | */
50 | void merge(int *first,int *second, int *result,int first_size,int second_size
    |     ){
51 |   int i=0;
52 |   int j=0;
53 |   int k=0;
54 |
55 |   while(i<first_size && j<second_size){
56 |
57 |     if (first[i]<second[j]) {
58 |       result[k]=first[i];
59 |       k++;
60 |       i++;
61 |     }else{
62 |       result[k]=second[j];
63 |       k++;
64 |       j++;
65 |     }
66 |
67 |     if(i == first_size){
68 |       // if the first array has been sorted
69 |       while(j<second_size){
70 |         result[k]=second[j];
71 |         k++;
72 |         j++;
73 |       }
74 |     } else if (j == second_size){
75 |       // if the second array has been sorted
76 |       while(i < first_size){
77 |         result[k]=first[i];
78 |         i++;
79 |         k++;
80 |       }
81 |     }
82 |   }
83 | }
84 |
85 | int main(int argc, char *argv[]) {
86 |
87 |     int *unsorted_array = (int *)malloc(SIZE * sizeof(int));
88 |     int *result = (int *)malloc(SIZE * sizeof(int));
89 |     int array_size = SIZE;
90 |     int size, rank;
91 |     int sub_array_size;
92 |
```

```
93 |   MPI_Status status;
94 |   // Start parallel execution
95 |     MPI_Init(&argc, &argv);
96 |     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
97 |     MPI_Comm_size(MPI_COMM_WORLD, &size);
98 |
99 |     if(rank==0){
100|       // --- RANDOM ARRAY GENERATION ---
101|       printf("Creating Random List of %d elements\n", SIZE);
102|       int j = 0;
103|       for (j = 0; j < SIZE; ++j) {
104|           unsorted_array[j] =(int) rand() % 1000;
105|       }
106|         printf("Created\n");
107|   }
108|
109|   // Number of Clusters to be run
110|   int iter_count = size;
111|   // Determine the size of the subarray each Cluster receives
112|   sub_array_size=(int)SIZE/iter_count;
113|
114|   // Cluster 0 (Master) splits the array and sends each subarray to the
       respective machine
115|   if( rank == 0 ){
116|     double start_timer;
117|     start_timer=MPI_Wtime();
118|     int i =0;
119|     if(iter_count > 1){
120|       // ============================================SENDING DATA
       ============================================
121|       for(i=0;i<iter_count-1;i++){
122|         int j;
123|         //send the subarray
124|         MPI_Send(&unsorted_array[(i+1)*sub_array_size],sub_array_size,MPI_INT
       ,i+1,0,MPI_COMM_WORLD);
125|       }
126|
127|       // =======================================CALCULATE FIRST SUBARRAY
       =======================================
128|       int i =0;
129|       int *sub_array = (int *)malloc(sub_array_size*sizeof(int));
130|       for(i=0;i<sub_array_size;i++){
131|         // Passing the first sub array since rank 0 always calculates the
       first sub array
132|         sub_array[i]=unsorted_array[i];
133|       }
134|       // Sequentially sorting the first array
135|       quicksort(sub_array,0,sub_array_size-1);
136|
137|       // ============================================RECEIVING DATA
       ============================================
138|       for (i=0;i<iter_count;i++){
139|         if(i > 0){
140|           int temp_sub_array[sub_array_size];
141|           // Receive each subarray
142|           MPI_Recv(temp_sub_array,sub_array_size,MPI_INT,i,777,MPI_COMM_WORLD
       ,&status);
143|           int j;
144|           int temp_result[i*sub_array_size];
145|           for(j=0;j<i*sub_array_size;j++){
```

```
146 |             temp_result[j]=result[j];
147 |           }
148 |           int temp_result_size = sub_array_size*i;
149 |           // Merge it back into the result array
150 |           merge(temp_sub_array,temp_result,result,sub_array_size,
    temp_result_size);
151 |
152 |         }else{
153 |           // On first iteration we just pass the sorted elements to the
    result array
154 |           int j;
155 |           for(j=0;j<sub_array_size;j++){
156 |             result[j]=sub_array[j];
157 |           }
158 |           free(sub_array);
159 |         }
160 |       }
161 |     }else{
162 |       // if it runs only in a single Cluster
163 |       quicksort(unsorted_array,0,SIZE-1);
164 |       for(i=0;i<SIZE;i++){
165 |         result[i]=unsorted_array[i];
166 |       }
167 |     }
168 |     double finish_timer;
169 |     finish_timer=MPI_Wtime();
170 |     printf("End Result: \n");
171 |     printf("Cluster Size %d, execution time measured : %2.7f sec \n",size,
    finish_timer-start_timer);
172 |   }else{
173 |     // All the other Clsuters have to sort the data and send it back
174 |     sub_array_size=(int)SIZE/iter_count;
175 |     int *sub_array = (int *)malloc(sub_array_size*sizeof(int));
176 |     MPI_Recv(sub_array,sub_array_size,MPI_INT,0,0,MPI_COMM_WORLD,&status);
177 |     quicksort(sub_array,0,sub_array_size-1);
178 |     int i=0;
179 |     MPI_Send(sub_array,sub_array_size,MPI_INT,0,777,MPI_COMM_WORLD);//sends
    the data back to rank 0
180 |     free(sub_array);
181 |   }
182 |
183 |   if(rank==0){
184 |         // --- VALIDATION CHECK ---
185 |         printf("Checking.. \n");
186 |         bool error = false;
187 |         int i=0;
188 |         for(i=0;i<SIZE-1;i++) {
189 |             if (result[i] > result[i+1]){
190 |                 error = true;
191 |             printf("error in i=%d \n", i);
192 |           }
193 |         }
194 |         if(error)
195 |             printf("Error..Not sorted correctly\n");
196 |         else
197 |             printf("Correct!\n");
198 |     }
199 |   free(unsorted_array);
200 |   // End of Parallel Execution
201 |   MPI_Finalize();
```

```
202 | }
```

Listing 2.4: quicksort_merge_mpi.c

Capítulo 3

Resultados

Para esta sección, y para en general todas las mediciones se han realizado tres pruebas:

- Prueba 01: $n = 36$ y $p = 1, 4, 9$
- Prueba 02: $n = 576$ y $p = 1, 4, 9, 16$
- Prueba 03: $n = 14400$ y $p = 1, 4, 9, 16, 25$

3.1. Tiempos generales

Todos los tiempos de ejecución, comunicación y cómputo de las expresiones teóricas e implementaciones se resumen en la tabla 3.1. Se ha considerado lo siguiente:

- Ordenamiento por ranking (implementación)
- Quicksort en paralelo (implementación)
- Ordenamiento por ranking (teórico-ideal)
- Ordenamiento por ranking (teórico-implementado)

Note lo siguiente:

- $T_{\text{ideal}}(n,p) = O\left(p\left(\alpha + \frac{n}{p}\beta\right)\right) + O\left(2\log\left(\sqrt{p}\right)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)\right) + O\left(\sqrt{p}\log\left(\sqrt{p}\right)\left(\alpha + \frac{n}{p}\beta\right)\right) + O\left(2\frac{n}{\sqrt{p}}\log\left(\frac{n}{\sqrt{p}}\right)\right) + O\left(\sqrt{p}\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)\right)$
- $T_{\text{imp}}(n,p) = O\left(2p\left(x + \frac{n}{p}\beta\right)\right) + O\left(3\sqrt{p}\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)\right) + O\left(2\frac{n}{\sqrt{p}}\log\left(\frac{n}{\sqrt{p}}\right)\right)$

Para las mediciones se ha tomado tanto α como β igual a uno.

Size (n)	Procesos	Ranking - Implementado			Ranking - Teo (ideal)	Ranking - Teo (imperfecto)	Quicksort
		tejec	tcomp	tcomm	tejec	tejec	tejec
36	1	0.000059	0.000050	0.000009	382.0134	622.0134	0.000002
	4	0.000078	0.000017	0.000061	391.3184	504.0534	0.000934
	9	0.000373	0.000006	0.000367	412.3741	515.6378	0.000183
576	1	0.000211	0.000188	0.000023	9246.236	12906.236	0.000033
	4	0.000189	0.000084	0.000105	7594.9271	8881.8652	0.000043
	9	0.000507	0.000057	0.000450	7137.8677	7694.8782	0.000368
	16	0.000664	0.000041	0.000623	7001.2167	7183.3062	0.000288
14400	1	0.006122	0.005736	0.000386	323763.5244	414975.5244	0.001023
	4	0.004589	0.002796	0.001793	235803.9044	267150.4428	0.002006
	9	0.005974	0.001819	0.004154	208508.2344	220681.1635	0.001916
	16	0.008558	0.001835	0.006723	196847.0718	198342.5617	0.002095
	25	0.014683	0.001442	0.013240	191302.8796	185361.5425	0.002552

Tabla 3.1: Resultados de experimentacion

3.2. Tiempos teóricos: ideal e implementado

Dado los tiempos de las tablas anteriores, para que las mediciones teóricas tengan sentido ahora, y también para las posteriores comparaciones, estas se han normalizado siendo divididas por el promedio de los cocientes entre los tiempos teóricos y experimentales. Es decir, si se tienen n mediciones, para cada medición $t_{i:\text{teórico}}$, se ha calculado $t'_{i:\text{teórico}}$, como:

$$t'_{i:\text{teórico}} = \frac{\sum \frac{t_{i:\text{teórico}}}{t_{i:\text{empírico}}}}{n}$$

Se pueden observar los resultados en las gráficas 3.1 y 3.2. En líneas generales, los resultados tienen sentido siguiendo las expresiones teóricas. Los tiempos de ejecución no se reducen mucho exactamente al aumentar los procesos, ello debido al tiempo de cómputo liderado por $O\left(\frac{n}{\sqrt{p}} \log\left(\frac{n}{\sqrt{p}}\right)\right)$, término en el cual no se nota mucha reducción al aumentar los procesos. En cambio, la comunicación es proporcional a p y $\sqrt{p}$, y el aumento es mucho más rápido. Por ello, en las gráficas, si se siguieran aumentando procesos, el rendimiento empeoraría.

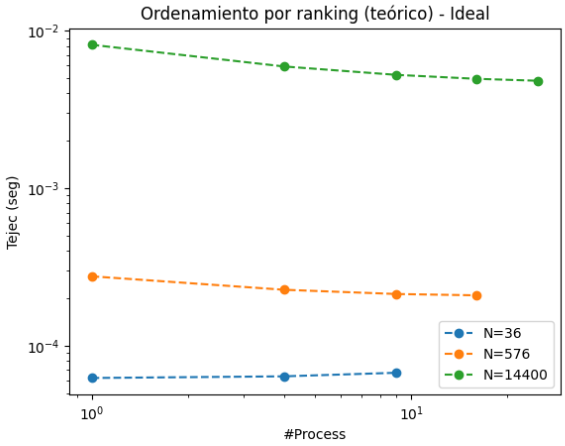

Figura 3.1: Tiempos teóricos ideales del ordenamiento por ranking

En cuanto a las discrepancias con respecto a la gráfica ideal e implementada, no resultan muy distintas, pero vemos que la ideal, claramente, es mejor debido a su optimización. Dado que estamos probando con un p máximo de 25, la mejora $\log(p)$ no resulta tan evidente con respecto a $\sqrt{p}$.

3.3. Rendimiento del código

Las mediciones de la implementación del código en MPI, de tiempos de ejecución, cómputo y comunicación, y respectiva eficiencia, se pueden ver en las gráficas 3.3, 3.4, 3.5, 3.6. En

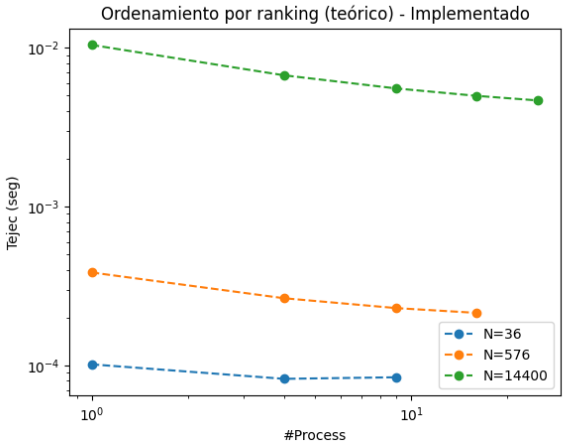

Figura 3.2: Tiempos teóricos implementados del ordenamiento por ranking

cuanto a los tiempos de cómputo y comunicación, se obtiene lo esperado: uno disminuye y el otro aumenta al aumentar los procesos. Siguiendo la idea de las gráficas teóricas, vemos que el cómputo disminuye muy poco al aumentar los procesos, mientras que la comunicación crece bastante siguiendo lo explicado anteriormente.

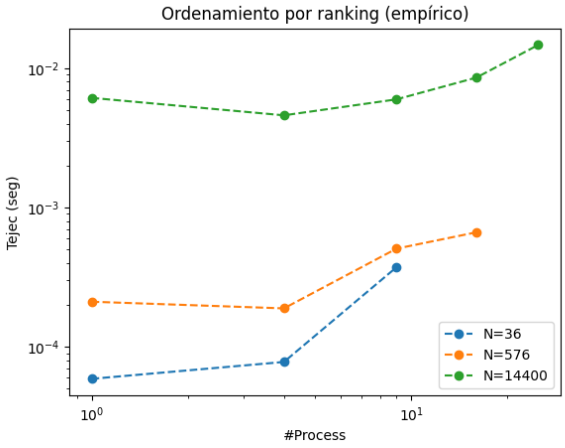

Figura 3.3: Tiempos empíricos de ejecución del ordenamiento por ranking

Con respecto a diferentes n, el óptimo de procesos varía; mientras mayor sea n, mayor será este óptimo. La mejora en el término $O\left(\frac{n}{\sqrt{p}}\log\left(\frac{n}{\sqrt{p}}\right)\right)$ contra el aumento de la comunicación al incrementar los procesos será más notable mientras más grande sea n.

La gráfica de eficiencia cobra sentido con lo anterior: a mayor n, veremos una caída

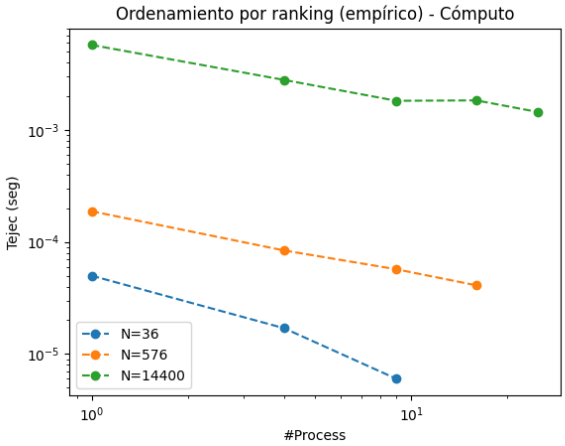

Figura 3.4: Tiempos empíricos de cómputo del ordenamiento por ranking

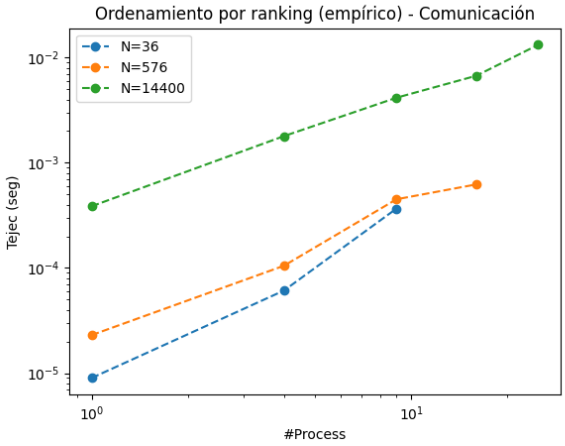

Figura 3.5: Tiempos empíricos de comunicación del ordenamiento por ranking

menor en la eficiencia al aumentar los procesos. También se nota que el óptimo de los procesos para los casos registrados está alrededor de $p = 4$, y este irá creciendo al aumentar n. En sí, vemos escalabilidad: al aumentar n, no se requieren tantos procesos para hallar mejora.

3.4. Validación teórica

Se estará comparando el rendimiento del código implementado con la expresión $T_{\text{imp}}(n, p)$, es decir, la complejidad derivada de lo implementado; ello se hará con las gráficas 3.2 y 3.3.

Vemos cierta discrepancia con respecto a lo teórico, y se debe particularmente a la

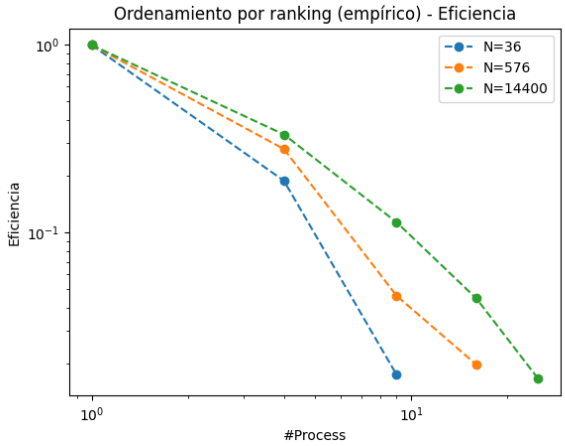

Figura 3.6: Eficiencia empírica del ordenamiento por ranking

implementación de la comunicación. La expresión teórica asume que, específicamente, las operaciones de *gossip*, *broadcast*, *reduce* y *gather* son directas, mientras que en la práctica no son realmente así. En todos los métodos explicados anteriormente, cada proceso debe saber exactamente a quién comunicar, y calcular ello toma cierto tiempo, por lo que prácticamente el tiempo de comunicación resulta mayor al teórico. Por ello, el óptimo de procesos se halla mucho antes, debido a que la caída del cómputo no logra compensar dicha subida en la comunicación.

3.5. Comparación con quicksort en paralelo

Básicamente estaremos comparando la implementación del ordenamiento por ranking con el Parallel Merge Quicksort [2]. Para ello, recordemos sus complejidades:

- $T_{\text{imp}}(n,p) = O\left(2p\left(x + \frac{n}{p}\beta\right)\right) + O\left(3\sqrt{p}\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)\right) + O\left(2\frac{n}{\sqrt{p}}\log\left(\frac{n}{\sqrt{p}}\right)\right)$
- $T_{\text{quick}}(n,p) = \frac{n}{p}\cdot\log(\frac{n}{p}) + \left(\frac{n}{2} + \frac{n}{p}\right)(p-1)$

Directamente de estas expresiones, podemos intuir que el ordenamiento por ranking escala mejor que el Parallel Merge Quicksort, por el término en este último $\frac{n}{2}$, y también un poco por el $(p-1)$. Directamente, al aumentar n, empeora, y si aumentáramos p para compensar ello, el término $(p-1)$ aumenta bastante el resultado teórico.

Ello se puede comprobar con las gráficas 3.7 y 3.8. Si comparamos los tiempos de ejecución y eficiencia, vemos que el Parallel Merge Quicksort muestra peor desempeño por lo explicado anteriormente, viendo que el ordenamiento por ranking muestra mejor escalabilidad.

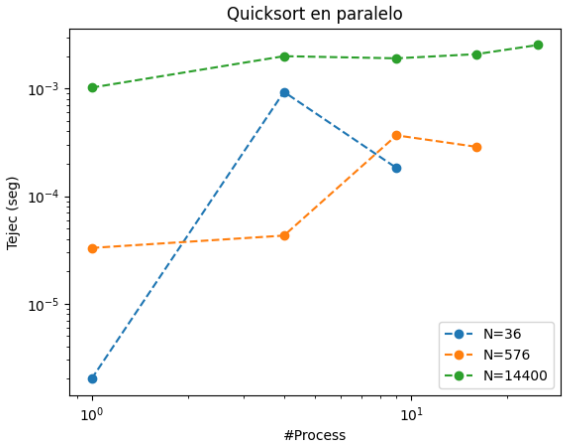

Figura 3.7: Tiempos empíricos de ejecución de quicksort en paralelo

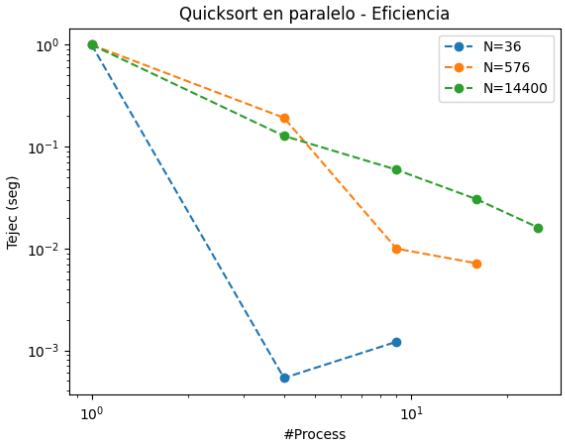

Figura 3.8: Eficiencia empírica de quicksort en paralelo

Sin embargo, de la expresión se puede intuir, y los autores del repositorio hacen énfasis en que para n muy grandes, se obtienen mejores resultados, y los plasman en las gráficas 3.9 y 3.10. Viendo que el ordenamiento por ranking también obtiene mejores resultados para n grandes, la escalabilidad de la implementación resulta mejor que en las gráficas anteriores.

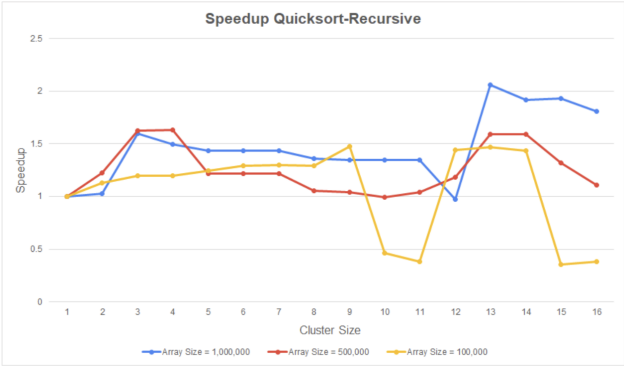

Figura 3.9: Speedup empírico de quicksort en paralelo [2]

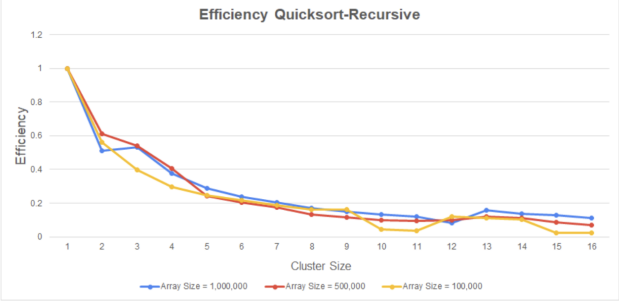

Figura 3.10: Eficiencia empírica de quicksort en paralelo [2]

Capítulo 4

Conclusiones

Básicamente, se hace un recuento de lo desarrollado. En líneas generales, el diseño y planeación teórica se vieron reflejados en la implementación, por lo que se puede validar. Sin embargo, entraremos en más detalle.

4.1. Revisiones Generales

De la implementación y de todo el análisis, se rescata lo siguiente:

- La comunicación no optimizada muestra resultados similares a la optimizada para valores de p relativamente no muy grandes.
- El ordenamiento por ranking obtiene mejores resultados para n muy grandes, lo que se traduce en una buena escalabilidad.
- Se obtienen mejores resultados que el Quicksort en paralelo, validando el rendimiento de la implementación.

4.2. Mejoras

Dentro de lo desarrollado, se consignan las siguientes mejoras:

- Optimizar la comunicación con el esquema de árbol, acercando así el rendimiento de la implementación a la expresión teórica.
- Probar con valores de n y p mucho más grandes, comprobando así explícitamente la escalabilidad del código.
- Englobar los procesos de comunicación, con el objetivo de reducir el costo de estos.

Bibliografía

[1] Owzok. Parallel sort by ranking, 2024. Repositorio de GitHub, consultado el 1 de diciembre de 2024. URL: https://github.com/Owzok/Parallel-Sort-By-Ranking.

[2] Samuel Trias Amo. Quicksort-parallel-mpi, 2020. Accedido: 2024-12-01. URL: https://github.com/triasamo1/Quicksort-Parallel-MPI.