**ЗМІСТ**

[ВСТУП 3](#_Toc107620592)

[РОЗДІЛ 1. ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ 5](#_Toc107620593)

[РОЗДІЛ 2. РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ 8](#_Toc107620594)

[РОЗДІЛ 3. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС 11](#_Toc107620595)

[РОЗДІЛ 4. РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ 12](#_Toc107620596)

[РОЗДІЛ 5. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ (ПОРІВНЯЛЬНИЙ АНАЛІЗ ШВИДКОСТІ ОБЧИСЛЕНЬ) 15](#_Toc107620597)

[ВИСНОВКИ 16](#_Toc107620598)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 17](#_Toc107620599)

[ДОДАТКИ 18](#_Toc107620600)

[Додаток А 18](#_Toc107620601)

[Додаток Б 20](#_Toc107620602)

# **ВСТУП**

Обчислювальний процес - це послідовність цілеспрямованих дій з обробки інформації, що призводять до досягнення поставленої мети.

Обчислювальний процес характеризується трьома параметрами:

* складом подій, з яких він складається;
* тривалістю кожної події (або операцій з обробки події);
* послідовністю виникнення подій.

Найпростіший вид паралелізму реалізований в мультипрограмних системах - цей вид паралелізму називається "суміщення в часі різних етапів різних завдань".

Мультипрограмна обробка можлива і в однопроцесорній системі, так як реалізує поєднання в часі виконання різних частин різних завдань. Для такого суміщення лише необхідно, щоб однопроцесорна система складалася з відносно незалежних, автономних частин, кожна з яких може виконувати свою роботу одночасно з роботою інших. При цьому кожна задача все-таки буде вирішуватися послідовно, але різні частини обчислювальної системи будуть одночасно працювати над різними завданнями, не заважаючи один одному.

В обчислювальних системах, що містять кілька обробних пристроїв, можливий інший тип паралелізму: одночасне рішення різних завдань або одночасне рішення незалежних гілок однієї задачі.

Якщо в систему надходить безперервний потік не пов'язаних між собою задач (тобто таких завдань, для яких характерним є те, що рішення будь-якої задачі не залежить від результатів вирішення інших завдань), використання декількох обробних пристроїв дозволяє поєднати вирішення цих завдань в часі. Це - "природний паралелізм незалежних завдань".

При вирішенні великого завдання в ній так само можуть бути виділені окремі незалежні частини - гілки, які за наявності декількох оброблювальних пристроїв можуть виконуватися паралельно і незалежно один від одного. Це - "паралелізм незалежних гілок" [1].

При виконанні курсової роботи буде досліджено алгоритм сортування злиттям, надано його короткий опис та наведено відому паралельну реалізацію. Також буде розроблено послідовний алгоритм та його паралельну реалізацію з використанням технології МРІ на мові програмування Java. Розроблені алгоритми буде досліджено на швидкодію і обчислено показник прискорення з варіюванням характеристик алгоритму (к-ть потоків).

# **РОЗДІЛ 1. ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ**

**1.1 Опис послідовного алгоритму**

Сортування злиттям – алгоритм, що використовується для сортування масиву, який реалізований за принципом "розділяй і владарюй". Завдання сортування масиву розбивається на кілька підзадач сортування масивів меншого розміру, після виконання яких результат комбінується, що і призводить до вирішення початкової задачі.

Алгоритм сортування злиттям виглядає так:

1) Вхідний масив розбивається на дві частини однакового розміру;

2) Кожен з підмасивів сортується окремо, за цим же принципом, тобто проводиться повторне розбиття до тих пір, поки довжина підмасиву не досягне одиниці. У разі кожен одиничний масив вважається відсортованим;

3) Після цього здійснюється злиття всіх підмасивів в один, у результаті отримуємо відсортовані дані [2].

Сортування злиттям часто застосовують при сортуванні масивів, списків, потоків. Під час сортування дані розділюються на невеликі частинки а потім відбувається збірка усіх частинок в правильному порядку.

Приклад роботи алгоритму наведено на рисунку 1.1.

Складність сортування злиттям оцінюється логарифмічно [3].

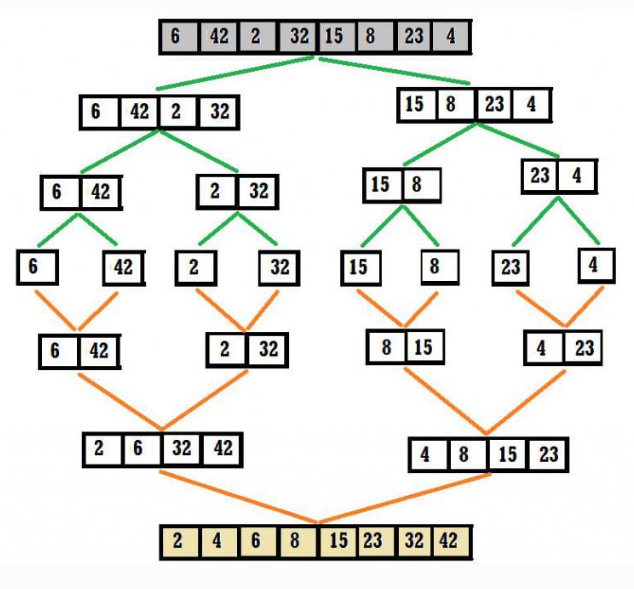


Рис. 1.1 – Приклад роботи алгоритму злиттям [3]

**1.2 Опис паралельного алгоритму**

Операція порівняння-обміну об’єднує дві відсортовані послідовності довжини M , що містяться в задачах A і B . Після завершення операції обидві задачі мають M даних, і всі елементи завдання A менші або рівні всім елементам завдання B. Як показано на малюнку 1.2, кожне завдання надсилає свої дані іншій задачі. Завдання A визначає M найнижчих елементів і відкидає залишок; цей процес вимагає щонайменше M/2 і щонайбільше M порівнянь. Аналогічно, завдання B визначає M найвищих елементів.

Алгоритм паралельного злиття виконує операцію злиття для двох відсортованих послідовностей довжини M2d, кожна з яких розподілена за 2d завданнями, щоб створити єдину відсортовану послідовність довжини M2d+1, розподілену на 2d+1 завдання. Як показано на рисунку 1.3, це досягається за допомогою шаблону зв’язку гіперкуб. Кожне із 2d+1 завдань включає d+1 кроки порівняння-обміну, по одному з кожним сусідом.

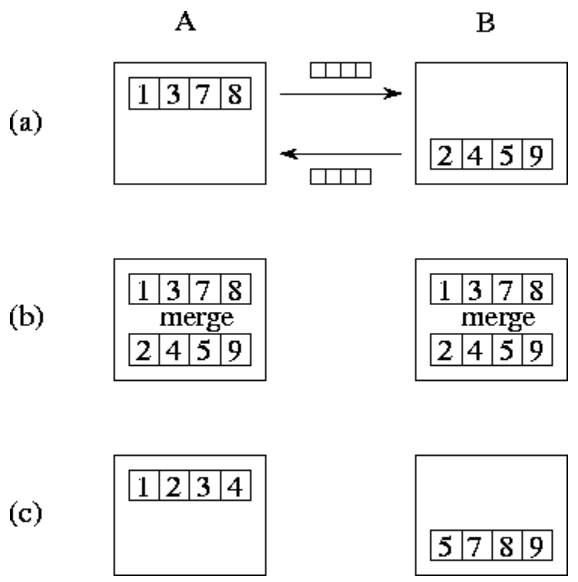


Рис. 1.2 - Алгоритм сортування злиттям з M=4 [4]

(a) Завдання A і B обмінюються відсортованими підпослідовностями.

(b) Вони виконують операцію злиття для визначення найнижчого та найвищого M елементів відповідно.

(c) Інші елементи відкидаються, залишаючи єдину відсортовану послідовність, розділену на два завдання.

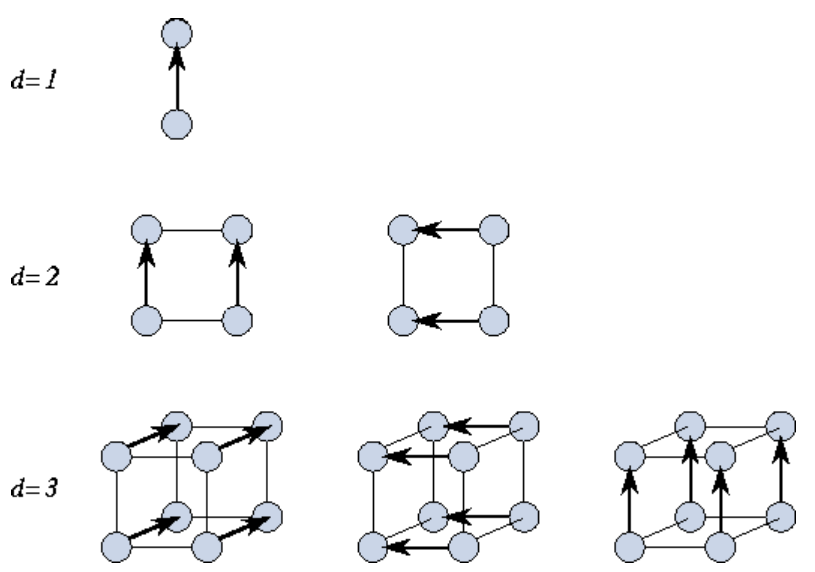


Рис. 1.3 - Операція паралельного злиття, що виконується

в гіперкубах розмірності 1, 2 і 3. У гіперкубі розмірності

d кожне завдання виконує d операцій порівняння-обміну [4].

# **РОЗДІЛ 2. РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ**

**2.1 Розробка послідовного алгоритму**

Створимо функцію merge сортування підмасивів (рис. 2.1), m\_sort для об’єднання підмасивів (рис. 2.2) та non\_growth\_check для перевірки незростання елементів масиву (рис. 2.3). Основна функція наведена на рис. 2.4.

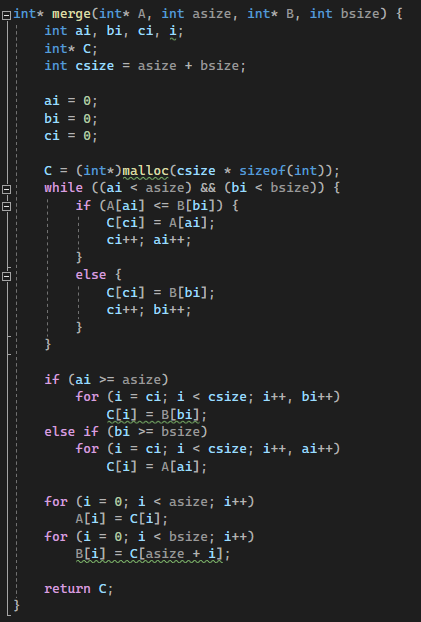


Рис. 2.1 – Функція сортування підмасивів

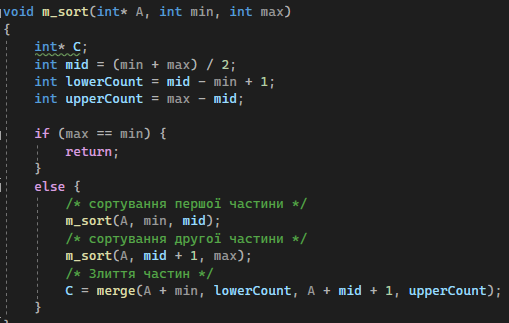


Рис. 2.2 – Функція для злиття підмасивів

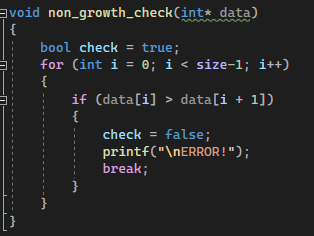


Рис. 2.3 – Функція для перевірки незростання масиву

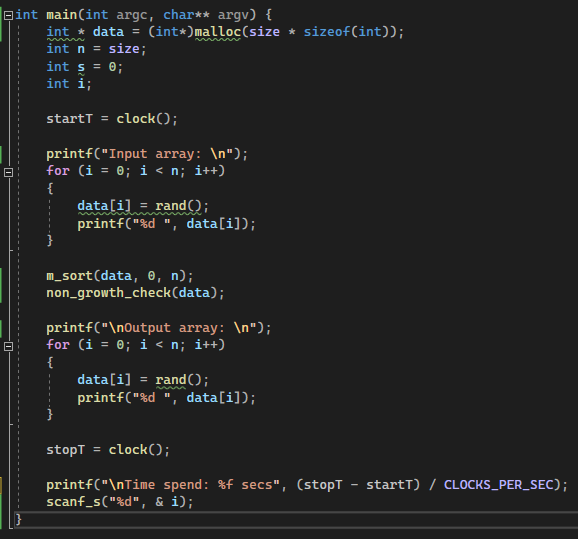


Рис. 2.4 – Основна функція програми

Запустимо програму та перевіримо її роботу (рис. 2.5).

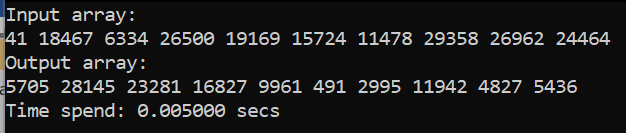


Рис. 2.5 – Перевірка роботи програми

на маленькому масиві

Перевіримо правильність сортування масиву за допомогою допоміжної функції перевірки на незростання на масивах великого розміру (рис. 2.6).



Рис. 2.6 – Перевірка роботи програми на великому масиві

**2.1 Аналіз швидкодії послідовного алгоритму**

Проведемо тестування програми на масивах різного розміру (масиви заповнюються довільними значеннями). Результати тестування програми наведені на рисунку 2.7. Перенесемо отримані дані на графік на рис. 2.8.











Рис. 2.7 – Результати тестування програми

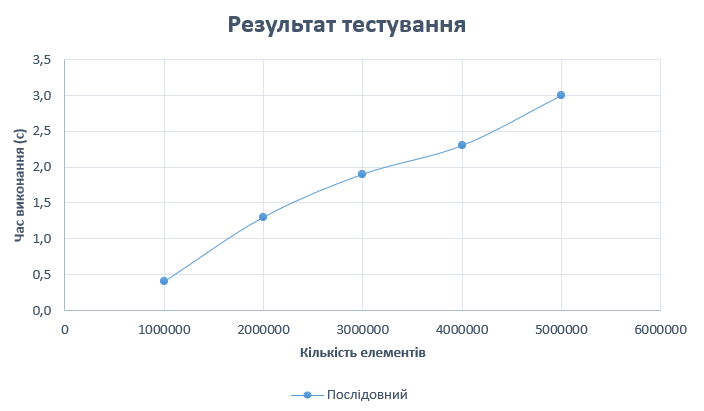


Рис. 2.9 – Залежність часу виконання програми

залежно від кількості елементу масиву

На рисунку 2.9 бачимо, що теоретична оцінка алгоритму не підтвердилась, фактично виміряні дані показують лінійну залежність O(n). Для пришвидшення роботи алгоритму і отримання кращих результатів часу виконання програми застосуємо технологію МРІ. Послідовний алгоритм наведено у додатку А.

# **РОЗДІЛ 3. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС**

MPI — це інтерфейс який дозволяє обмінюватись повідомленнями між декількома комп’ютерами, що виконують паралельну програму в розподіленій пам’яті, іншими словами це інтерфейс для забезпечення зв’язку між «гілками» паралельної програми. Стандарт MPI визначає синтаксис і семантику бібліотечних підпрограм, корисних для написання переносних програм для передачі повідомлень. Цілями MPI є висока продуктивність, масштабованість і портативність. На сьогодні використання MPI залишається домінуючою моделлю, яка використовується у високопродуктивних обчисленнях. MPI не схвалений як офіційний стандарт будь-якою організацією зі стандартів, наприклад IEEE або ISO, але, як правило, вважається галузевим стандартом. Також стандарт MPI визначає синтаксис і семантику бібліотечних підпрограм, корисних для написання переносних програм для передачі повідомлень. Цілями MPI є висока продуктивність, масштабованість і портативність. MPI залишається домінуючою моделлю, яка використовується сьогодні у високопродуктивних обчисленнях.

Відправлення та отримання є двома основними концепціями MPI. Якщо процес A хоче надіслати повідомлення процесу B, процес A збирає всі необхідні дані в буфер і видає відправку процесу B, використовуючи свій ранг. Пристрій зв'язку часто мережа відповідає за маршрутизацію повідомлення в належне місце. Процес B повинен визнати, що він хоче отримати дані A. Як тільки процес B підтвердить, дані будуть передані, а процес A буде надіслано підтвердження про передачу даних.

MPI дозволяє відправникам і одержувачам вказувати ідентифікатор повідомлення з повідомленням, відомим як теги. Коли процес B запитує повідомлення з певним номером тега, повідомлення з різними тегами будуть буферизуватися мережею, доки B не буде готовий до них [5].

# **РОЗДІЛ 4. РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ**

**4.1 Проектування паралельного алгоритму**

Оптимізуємо послідовний алгоритм, розроблений у розділі 2. Оптимізуємо код шляхом розпаралелення – пересилання повідомлень процесором.

Основною метою оптимізації алгоритму є мінімізація затраченого часу на виконання програми.

Розділимо на процеси алгоритм сортування відповідно до схеми, наведеної на рис. 1.3. Використаємо MPI\_Send та MPI\_Recv для відправки та отримання повідомлень.

**4.2 Реалізація паралельного алгоритму**

Спочатку ініціалізуємо МРІ. Створимо додаткові змінні та заповнимо масив довільними значеннями (рис. 2.1).

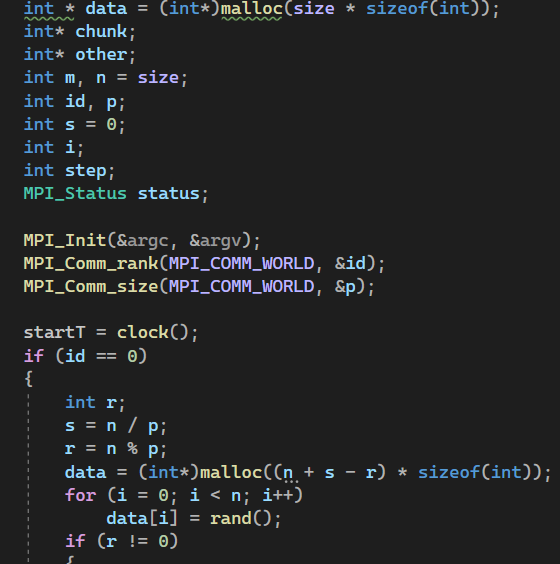


Рис. 4.1 – Ініціалізація МРІ та змінних

Створимо транслятор повідомлень (MPI\_Bcast) та скеттер (MPI\_Scatter) що відповідає за пересилання даних від одного процесу до усіх інших в комунікаторі (рис. 4.2).

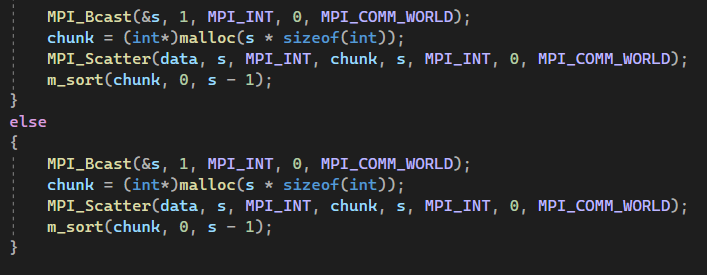


Рис. 4.2 –Пересилання даних між процесами

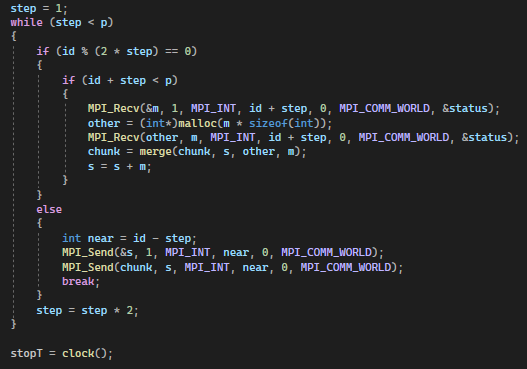


Рис. 4.3 – Відправка та отримання повідомлень

**4.3 Тестування паралельного алгоритму**

Проведемо тестування програми змінюючи кількість процесів. Результати тестування для 10 процесів наведено на рисунку 4.4. Для 20 процесів – на рисунку 4.5.











Рис. 4.4 – Результати тестування алгоритму для 10 процесів











Рис. 4.5 – Результати тестування алгоритму для 20 процесів

# **РОЗДІЛ 5. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ (ПОРІВНЯЛЬНИЙ АНАЛІЗ ШВИДКОСТІ ОБЧИСЛЕНЬ)**

Визначимо швидкодію паралельного алгоритму. Результати досліджень з рис. 4.4 та 4.5 перенесемо на графік на рис. 5.1.



Рис. 5.1 – Графік часу виконання програми відносно

кількості елементів масиву

З рисунку 4.5 можна зробити висновок, що вдалося добитися прискорення роботи алгоритму.

Порівняємо, наскільки збільшився час виконання програми. Створимо таблицю 5.1, в яку додамо всі отримані результати. Обчислимо в ній показник прискорення, якого вдалося досягти. Також обрахуємо загальні показники прискорення.

Таблиця 5.1 – Загальні результати тестування

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кількість елементів | Послідовний час виконання, сек | 10 процесів | 20 процесів | Показник 10 | Показник 20 |
| 1000000 | 0,4 | 0,4 | 0,3 | 1,0 | 1,3 |
| 2000000 | 1,3 | 0,9 | 0,6 | 1,4 | 2,2 |
| 3000000 | 1,9 | 1,5 | 1,1 | 1,3 | 1,7 |
| 4000000 | 2,3 | 1,5 | 1,4 | 1,5 | 1,6 |
| 5000000 | 3,0 | 2,1 | 1,8 | 1,4 | 1,7 |

Було отримано максимальне прискорення у 2,17 разів. В середньому показник прискорення склав 1,5.

# **ВИСНОВКИ**

При виконанні курсової роботи було виконано проектування, розробка та тестування послідовного та паралельного алгоритмів. Також було виконано аналіз швидкодії розроблених алгоритмів зі зміною к-ті процесів та розміру масиву.

При дослідженні швидкодії алгоритму було отримано наступне прискорення: Максимальне значення прискорення було отримано для 20 потоків та 2 млн. елементів масиву. В середньому показник прискорення склав 1,52.

Під час розробки паралельного алгоритму вдалося зменшити час виконання алгоритму завдяки використанню технології МРІ.

З цього можемо зробити висновок, що використання паралельних обчислень є досить обґрунтованим та його варто використовувати для оптимізації певних обчислювальних процесів.

# **СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. ni.biz.ua - Навчальна Інформація для українських студентів паралельні обчислення. URL: http://ni.biz.ua/3/3\_11/3\_114340\_parallelnie-vichisleniya.html
2. programm.top. Сортировка слиянием. URL: https://programm.top/c-sharp/algorithm/array-sort/merge-sort/
3. Кучер В. Г. Сортування злиттям: алгоритм, переваги і особливості. URL: <https://kafedra.com.ua/sortuvannya-zlyttyam-algorytm-perevagy-i-osoblyvosti/>
4. Foster, I.. Mergesort. URL: [https://www.mcs.anl.gov/~itf/dbpp/text/node 127.html#:~:text=A%20parallel%20merge%20algorithm%20performs,of%20length%20distributed%20over%20tasks](https://www.mcs.anl.gov/~itf/dbpp/text/node%20127.html#:~:text=A%20parallel%20merge%20algorithm%20performs,of%20length%20distributed%20over%20tasks)
5. Pavani Panakanti. What is Message Passing Interface. UPL: https://medium.com/@getting.better.everyday/what-is-message-passing-interface-mpi-e5cf61d2bcde

# **ДОДАТКИ**

## **Додаток А**

Лістинг послідовного коду програми

/\* merge sort \*/

#include <stdio.h>

#include <mpi.h>

#include <time.h>

#include <malloc.h>

#include <random>

#define size 5000000

double startT, stopT;

int\* merge(int\* A, int asize, int\* B, int bsize) {

int ai, bi, ci, i;

int\* C;

int csize = asize + bsize;

ai = 0;

bi = 0;

ci = 0;

C = (int\*)malloc(csize \* sizeof(int));

while ((ai < asize) && (bi < bsize)) {

if (A[ai] <= B[bi]) {

C[ci] = A[ai];

ci++; ai++;

}

else {

C[ci] = B[bi];

ci++; bi++;

}

}

if (ai >= asize)

for (i = ci; i < csize; i++, bi++)

C[i] = B[bi];

else if (bi >= bsize)

for (i = ci; i < csize; i++, ai++)

C[i] = A[ai];

for (i = 0; i < asize; i++)

A[i] = C[i];

for (i = 0; i < bsize; i++)

B[i] = C[asize + i];

return C;

}

void swap(int\* v, int i, int j)

{

int t;

t = v[i];

v[i] = v[j];

v[j] = t;

}

void m\_sort(int\* A, int min, int max)

{

int\* C;

int mid = (min + max) / 2;

int lowerCount = mid - min + 1;

int upperCount = max - mid;

if (max == min) {

return;

}

else {

/\* сортування першої частини \*/

m\_sort(A, min, mid);

/\* сортування другої частини \*/

m\_sort(A, mid + 1, max);

/\* Злиття частин \*/

C = merge(A + min, lowerCount, A + mid + 1, upperCount);

}

}

void non\_growth\_check(int\* data)

{

bool check = true;

for (int i = 0; i < size-1; i++)

{

if (data[i] > data[i + 1])

{

check = false;

printf("\nERROR!");

break;

}

}

}

int main(int argc, char\*\* argv) {

int \* data = (int\*)malloc(size \* sizeof(int));

int n = size;

int s = 0;

int i;

startT = clock();

//printf("Input array: \n");

//for (i = 0; i < n; i++)

//{

// data[i] = rand();

// printf("%d ", data[i]);

//}

m\_sort(data, 0, n);

non\_growth\_check(data);

//printf("\nOutput array: \n");

//for (i = 0; i < n; i++)

//{

// data[i] = rand();

// printf("%d ", data[i]);

//}

stopT = clock();

printf("\nTime spend: %f secs", (stopT - startT) / CLOCKS\_PER\_SEC);

scanf\_s("%d", & i);

}

## **Додаток Б**

Лістинг паралельного коду програми

#include <stdio.h>

#include <mpi.h>

#include <time.h>

#include <malloc.h>

#include <random>

#define size 5000000

double startT, stopT;

int\* merge(int\* A, int asize, int\* B, int bsize) {

int ai, bi, ci, i;

int\* C;

int csize = asize + bsize;

ai = 0;

bi = 0;

ci = 0;

C = (int\*)malloc(csize \* sizeof(int));

while ((ai < asize) && (bi < bsize)) {

if (A[ai] <= B[bi]) {

C[ci] = A[ai];

ci++; ai++;

}

else {

C[ci] = B[bi];

ci++; bi++;

}

}

if (ai >= asize)

for (i = ci; i < csize; i++, bi++)

C[i] = B[bi];

else if (bi >= bsize)

for (i = ci; i < csize; i++, ai++)

C[i] = A[ai];

for (i = 0; i < asize; i++)

A[i] = C[i];

for (i = 0; i < bsize; i++)

B[i] = C[asize + i];

return C;

}

void swap(int\* v, int i, int j)

{

int t;

t = v[i];

v[i] = v[j];

v[j] = t;

}

void m\_sort(int\* A, int min, int max)

{

int\* C;

int mid = (min + max) / 2;

int lowerCount = mid - min + 1;

int upperCount = max - mid;

if (max == min) {

return;

}

else {

/\* сортування першої частини \*/

m\_sort(A, min, mid);

/\* сортування другої частини \*/

m\_sort(A, mid + 1, max);

/\* Злиття частин \*/

C = merge(A + min, lowerCount, A + mid + 1, upperCount);

}

}

void non\_growth\_check(int\* data)

{

bool check = true;

for (int i = 0; i < size-1; i++)

{

if (data[i] > data[i + 1])

{

check = false;

printf("\nERROR!");

break;

}

}

}

int main(int argc, char\*\* argv) {

int \* data = (int\*)malloc(size \* sizeof(int));

int\* chunk;

int\* other;

int m, n = size;

int id, p;

int s = 0;

int i;

int step;

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);

startT = clock();

if (id == 0)

{

int r;

s = n / p;

r = n % p;

data = (int\*)malloc((n + s - r) \* sizeof(int));

for (i = 0; i < n; i++)

data[i] = rand();

if (r != 0)

{

for (i = n; i < n + s - r; i++)

data[i] = 0;

s = s + 1;

}

MPI\_Bcast(&s, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

chunk = (int\*)malloc(s \* sizeof(int));

MPI\_Scatter(data, s, MPI\_INT, chunk, s, MPI\_INT, 0, MPI\_COMM\_WORLD);

m\_sort(chunk, 0, s - 1);

}

else

{

MPI\_Bcast(&s, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

chunk = (int\*)malloc(s \* sizeof(int));

MPI\_Scatter(data, s, MPI\_INT, chunk, s, MPI\_INT, 0, MPI\_COMM\_WORLD);

m\_sort(chunk, 0, s - 1);

}

step = 1;

while (step < p)

{

if (id % (2 \* step) == 0)

{

if (id + step < p)

{

MPI\_Recv(&m, 1, MPI\_INT, id + step, 0, MPI\_COMM\_WORLD, &status);

other = (int\*)malloc(m \* sizeof(int));

MPI\_Recv(other, m, MPI\_INT, id + step, 0, MPI\_COMM\_WORLD, &status);

chunk = merge(chunk, s, other, m);

s = s + m;

}

}

else

{

int near = id - step;

MPI\_Send(&s, 1, MPI\_INT, near, 0, MPI\_COMM\_WORLD);

MPI\_Send(chunk, s, MPI\_INT, near, 0, MPI\_COMM\_WORLD);

break;

}

step = step \* 2;

}

stopT = clock();

MPI\_Finalize();

//printf("Input array: \n");

//for (i = 0; i < n; i++)

//{

// data[i] = rand();

// printf("%d ", data[i]);

//}

//m\_sort(data, 0, n);

non\_growth\_check(data);

//printf("\nOutput array: \n");

//for (i = 0; i < n; i++)

//{

// data[i] = rand();

// printf("%d ", data[i]);

//}

printf("\nTime spend: %f secs", (stopT - startT) / CLOCKS\_PER\_SEC);

scanf\_s("%d", & i);

}