**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 1 з дисципліни

«Проектування алгоритмів»

„ **Проектування і аналіз алгоритмів зовнішнього сортування**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-35 Адаменко А. Б.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2024

Зміст

[1 Мета лабораторної роботи 3](#_Toc109342184)

[2 ЗаВдання 4](#_Toc109342185)

[3 Виконання 6](#_Toc109342186)

[3.1 Псевдокод алгоритму 6](#_Toc109342187)

[3.2 Програмна реалізація алгоритму 6](#_Toc109342188)

[3.2.1 Вихідний код 6](#_Toc109342189)

[Висновок 7](#_Toc109342190)

[Критерії оцінювання 8](#_Toc109342191)

# Мета лабораторної роботи

Мета роботи – вивчити основні алгоритми зовнішнього сортування та способи їх модифікації, оцінити поріг їх ефективності.

# Завдання

Згідно варіанту (таблиця 2.1), розробити та записати алгоритм зовнішнього сортування за допомогою псевдокоду (чи іншого способу за вибором).

Виконати програмну реалізацію алгоритму на будь-якій мові програмування та відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі (розмір файлу має бути не менше 10 Мб, можна значно більше).

Здійснити модифікацію програми і відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі розміром не менше ніж двократний обсяг ОП вашого ПК. Досягти швидкості сортування з розрахунку 1Гб на 3хв. або менше. Достатньо штучно обмежити доступну ОП, для уникнення багатогодинних сортувань (наприклад використовуючи віртуальну машину).

Рекомендується попередньо впорядкувати серії елементів довжиною, що займає не менше 100Мб або використати інші підходи для пришвидшення процесу сортування.

Зробити узагальнений висновок з лабораторної роботи, у якому порівняти базову та модифіковану програми. У висновку деталізувати, які саме модифікації було виконано і який ефект вони дали.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Алгоритм сортування** |
| 1 | Пряме злиття |
| 2 | Природне (адаптивне) злиття |
| 3 | Збалансоване багатошляхове злиття |
| 4 | Багатофазне сортування |
| 5 | Пряме злиття |
| 6 | Природне (адаптивне) злиття |
| 7 | Збалансоване багатошляхове злиття |
| 8 | Багатофазне сортування |
| 9 | Пряме злиття |
| 10 | Природне (адаптивне) злиття |
| 11 | Збалансоване багатошляхове злиття |
| 12 | Багатофазне сортування |
| 13 | Пряме злиття |
| 14 | Природне (адаптивне) злиття |
| 15 | Збалансоване багатошляхове злиття |
| 16 | Багатофазне сортування |
| 17 | Пряме злиття |
| 18 | Природне (адаптивне) злиття |
| 19 | Збалансоване багатошляхове злиття |
| 20 | Багатофазне сортування |
| 21 | Пряме злиття |
| 22 | Природне (адаптивне) злиття |
| 23 | Збалансоване багатошляхове злиття |
| 24 | Багатофазне сортування |
| 25 | Пряме злиття |
| 26 | Природне (адаптивне) злиття |
| 27 | Збалансоване багатошляхове злиття |
| 28 | Багатофазне сортування |
| 29 | Пряме злиття |
| 30 | Природне (адаптивне) злиття |
| 31 | Збалансоване багатошляхове злиття |
| 32 | Багатофазне сортування |
| 33 | Пряме злиття |
| 34 | Природне (адаптивне) злиття |
| 35 | Збалансоване багатошляхове злиття |

# Виконання

## Псевдокод алгоритму

Function MergeAt(m, l, r, chk, m\_idx, l\_beg, l\_sz, r\_beg, r\_sz)

l\_idx = l\_beg

r\_idx = r\_beg

While (l\_idx < l\_sz AND l\_idx - l\_beg < chk) AND (r\_idx < r\_sz AND r\_idx - r\_beg < chk)

l\_val = l.GetInt(l\_idx)

r\_val = r.GetInt(r\_idx)

If l\_val <= r\_val

m.SetInt(m\_idx, l\_val)

l\_idx = l\_idx + 1

Else

m.SetInt(m\_idx, r\_val)

r\_idx = r\_idx + 1

m\_idx = m\_idx + 1

While l\_idx < l\_sz AND l\_idx - l\_beg < chk

m.SetInt(m\_idx, l.GetInt(l\_idx))

l\_idx = l\_idx + 1

m\_idx = m\_idx + 1

While r\_idx < r\_sz AND r\_idx - r\_beg < chk

m.SetInt(m\_idx, r.GetInt(r\_idx))

r\_idx = r\_idx + 1

m\_idx = m\_idx + 1

Function SortAtScale(m, l, r, c)

s = m.Length()

ls = 0

rs = 0

For i from 0 to s - 1

lrChoice = (i // c) AND 1

blockOffset = (i // (2 \* c)) \* c

index = i % c

[l, r][lrChoice].SetInt(

blockOffset + index

, m.GetInt(i)

)

If lrChoice == 0

ls = ls + 1

Else

rs = rs + 1

m\_idx = 0

lr\_idx = 0

While m\_idx < s

MergeAt(

m, l, r,

c,

m\_idx,

lr\_idx, ls,

lr\_idx, rs

)

m\_idx = m\_idx + 2 \* c

lr\_idx = lr\_idx + c

Function DirectMergeSort(m, l, r)

s = m.Length()

c = 1

While c < s \* 2

SortAtScale(m, l, r, c)

c = c \* 2

## Програмна реалізація алгоритму

### Вихідний код

3.1.1.1 Звичайна реалізація алгоритму

import math

import os

def IntToRaw(value, intSize):

overInt = 2 \*\* (8 \* intSize)

if not (-overInt <= value < overInt):

raise ValueError('IntToRaw: too big number or too small number')

if value < 0:

value = overInt + value

res = [0] \* intSize

for i in range(intSize):

if value == 0:

break

res[i] = value & 0xff

value >>= 8

return res

def RawToInt(raw):

overInt = 2 \*\* (8 \* len(raw))

res = 0

for i, v in enumerate(raw):

res += v \* 2 \*\* (8 \* i)

if not (raw[-1] < 0x7f):

res = res - overInt

return res

def GetFileSize(fileName):

return os.path.getsize(fileName)

def CreateIntArrayFile(fileName, length, intSize):

rawZeroBytes = bytes(IntToRaw(0, intSize))

with open(fileName, 'wb') as file:

for i in range(length):

file.write(rawZeroBytes)

class RWIntArrayFile:

def \_\_init\_\_(self, fileName, length, intSize):

self.file = open(fileName, 'r+b')

self.length = length

self.intSize = intSize

def GetInt(self, index):

file = self.file

length = self.length

if not (0 <= index < length):

raise IndexError('GetInt: invalid index')

file.seek(index \* self.intSize)

rawInt = list(file.read(self.intSize))

return RawToInt(rawInt)

def SetInt(self, index, value):

file = self.file

length = self.length

intSize = self.intSize

if not (0 <= index < length):

raise IndexError('SetInt: invalid index')

file.seek(index \* intSize)

try:

rawIntBytes = bytes(IntToRaw(value, intSize))

except:

raise ValueError('SetInt: value of input integer is not in bounds')

file.write(rawIntBytes)

def Length(self):

return self.length

def IntSize(self):

return self.intSize

def \_\_del\_\_(self):

self.file.close()

def MergeAt(m, l, r, chk, m\_idx, l\_beg, l\_sz, r\_beg, r\_sz):

l\_idx = l\_beg

r\_idx = r\_beg

while (l\_idx < l\_sz and l\_idx - l\_beg < chk) and (r\_idx < r\_sz and r\_idx - r\_beg < chk):

l\_val = l.GetInt(l\_idx)

r\_val = r.GetInt(r\_idx)

if l\_val <= r\_val:

m.SetInt(m\_idx, l\_val)

l\_idx += 1

else:

m.SetInt(m\_idx, r\_val)

r\_idx += 1

m\_idx += 1

while l\_idx < l\_sz and l\_idx - l\_beg < chk:

m.SetInt(m\_idx, l.GetInt(l\_idx))

l\_idx += 1

m\_idx += 1

while r\_idx < r\_sz and r\_idx - r\_beg < chk:

m.SetInt(m\_idx, r.GetInt(r\_idx))

r\_idx += 1

m\_idx += 1

def SortAtScale(m, l, r, c):

s = m.Length()

ls = 0

rs = 0

for i in range(s):

lrChoice = (i // c) & 1

blockOffset = i // (2 \* c) \* c

index = i % c

[l, r][lrChoice].SetInt(

blockOffset + index

, m.GetInt(i)

)

if lrChoice == 0:

ls += 1

else:

rs += 1

m\_idx = 0

lr\_idx = 0

while m\_idx < s:

MergeAt(

m, l, r,

c,

m\_idx,

lr\_idx, ls,

lr\_idx, rs

)

m\_idx += 2 \* c

lr\_idx += c

def DirectMergeSort(m, l, r):

s = m.Length()

c = 1

while c < s \* 2:

SortAtScale(m, l, r, c)

c \*= 2

def main():

BlockSize = 2 \*\* 20

IntSize = 8

Length = GetFileSize('A.qwa') // IntSize

CreateIntArrayFile('B.qwa', Length, IntSize)

CreateIntArrayFile('C.qwa', Length, IntSize)

mainFile = RWIntArrayFile('A.qwa', Length, IntSize, blockSize = BlockSize)

leftFile = RWIntArrayFile('B.qwa', Length, IntSize, blockSize = BlockSize)

rightFile = RWIntArrayFile('C.qwa', Length, IntSize, blockSize = BlockSize)

DirectMergeSort(mainFile, leftFile, rightFile)

mainFile.FlushCache()

leftFile.FlushCache()

rightFile.FlushCache()

if \_\_name\_\_ == '\_\_main\_\_':

main()

3.2.1.2 Модифікована реалізація алгоритму на мові Python 3

import os

def IntToRaw(value, intSize):

return value.to\_bytes(intSize, byteorder='little', signed=True)

def RawToInt(raw):

return int.from\_bytes(raw, byteorder='little', signed=True)

def GetFileSize(fileName):

return os.path.getsize(fileName)

def CreateIntArrayFile(fileName, length, intSize):

with open(fileName, 'wb') as file:

for i in range(length):

file.write(IntToRaw(0, intSize))

class RWIntArrayFile:

def \_\_init\_\_(self, fileName, length, intSize, \*, blockSize=4096):

if not (0 <= blockSize and (blockSize & (blockSize - 1) == 0)):

raise ValueError('a block size has to be a power of 2')

self.fileName = fileName

self.length = length

self.intSize = intSize

self.blockSize = blockSize

self.blockSizeMask = blockSize - 1

self.blockSizeBitShift = blockSize.bit\_length() - 1

self.blockSizeMulIntSize = self.intSize << self.blockSizeBitShift

self.cache = bytearray(blockSize \* intSize)

self.cache\_start\_index = -1

self.cache\_size = 0

self.file = None

self.\_open\_file()

def \_open\_file(self):

try:

self.file = open(self.fileName, 'r+b')

except IOError as e:

print(f"Error opening file: {e}")

raise

def \_load\_block(self, block\_index):

try:

self.file.seek(block\_index \* self.blockSizeMulIntSize)

read\_size = self.blockSizeMulIntSize

bytes\_read = self.file.read(read\_size)

self.cache\_size = len(bytes\_read)

self.cache[:self.cache\_size] = bytes\_read

self.cache\_start\_index = block\_index

except IOError as e:

print(f"Error loading block: {e}")

raise

def \_flush\_cache(self):

if self.cache\_start\_index == -1:

return

try:

self.file.seek(self.cache\_start\_index \* self.blockSizeMulIntSize)

self.file.write(self.cache[:self.cache\_size])

self.file.flush()

self.cache\_start\_index = -1

self.cache\_size = 0

except IOError as e:

print(f"Error flushing cache: {e}")

raise

def \_get\_cache\_offset(self, index):

return (index & self.blockSizeMask) \* self.intSize

def GetInt(self, index):

block\_index = index >> self.blockSizeBitShift

offset = self.\_get\_cache\_offset(index)

if self.cache\_start\_index != block\_index:

self.\_flush\_cache()

self.\_load\_block(block\_index)

rawInt = self.cache[offset:offset + self.intSize]

return RawToInt(rawInt)

def SetInt(self, index, value):

block\_index = index >> self.blockSizeBitShift

offset = self.\_get\_cache\_offset(index)

if self.cache\_start\_index != block\_index:

self.\_flush\_cache()

self.\_load\_block(block\_index)

rawIntBytes = bytes(IntToRaw(value, self.intSize))

self.cache[offset:offset + self.intSize] = rawIntBytes

def Length(self):

return self.length

def IntSize(self):

return self.intSize

def FlushCache(self):

self.\_flush\_cache()

def \_\_enter\_\_(self):

self.\_open\_file()

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

self.\_flush\_cache()

self.file.close()

def \_\_del\_\_(self):

if self.file:

self.\_flush\_cache()

self.file.close()

def MergeAt(m, l, r, chk, m\_idx, l\_beg, l\_sz, r\_beg, r\_sz):

l\_idx = l\_beg

r\_idx = r\_beg

while (l\_idx < l\_sz and l\_idx - l\_beg < chk) and (r\_idx < r\_sz and r\_idx - r\_beg < chk):

l\_val = l.GetInt(l\_idx)

r\_val = r.GetInt(r\_idx)

if l\_val <= r\_val:

m.SetInt(m\_idx, l\_val)

l\_idx += 1

else:

m.SetInt(m\_idx, r\_val)

r\_idx += 1

m\_idx += 1

while l\_idx < l\_sz and l\_idx - l\_beg < chk:

m.SetInt(m\_idx, l.GetInt(l\_idx))

l\_idx += 1

m\_idx += 1

while r\_idx < r\_sz and r\_idx - r\_beg < chk:

m.SetInt(m\_idx, r.GetInt(r\_idx))

r\_idx += 1

m\_idx += 1

def SortAtScale(m, l, r, c):

s = m.Length()

ls = 0

rs = 0

for i in range(s):

lrChoice = (i // c) & 1

blockOffset = i // (2 \* c) \* c

index = i % c

[l, r][lrChoice].SetInt(

blockOffset + index

, m.GetInt(i)

)

if lrChoice == 0:

ls += 1

else:

rs += 1

m\_idx = 0

lr\_idx = 0

while m\_idx < s:

MergeAt(

m, l, r,

c,

m\_idx,

lr\_idx, ls,

lr\_idx, rs

)

m\_idx += 2 \* c

lr\_idx += c

def DirectMergeSort(m, l, r):

s = m.Length()

c = 1

while c < s:

SortAtScale(m, l, r, c)

c \*= 2

def main():

BlockSize = 1024 \* 2 \*\* 10

IntSize = 8

Length = GetFileSize('A.qwa') // IntSize

CreateIntArrayFile('B.qwa', Length, IntSize)

CreateIntArrayFile('C.qwa', Length, IntSize)

mainFile = RWIntArrayFile('A.qwa', Length, IntSize, blockSize = BlockSize)

leftFile = RWIntArrayFile('B.qwa', Length, IntSize, blockSize = BlockSize)

rightFile = RWIntArrayFile('C.qwa', Length, IntSize, blockSize = BlockSize)

DirectMergeSort(mainFile, leftFile, rightFile)

mainFile.FlushCache()

leftFile.FlushCache()

rightFile.FlushCache()

if \_\_name\_\_ == '\_\_main\_\_':

main()

3.2.1.3 Модифікована реалізація алгоритму на мові C++

#include <iostream>

#include <fstream>

#include <vector>

#include <cstring>

#include <stdexcept>

#include <cstdint>

#include <filesystem>

#include <cmath>

std::vector<char> IntToRaw(std::int64\_t value, std::size\_t intSize)

{

std::vector<char> raw(intSize);

for (std::size\_t i = 0; i < intSize; ++i)

{

raw[i] = (value >> (i \* 8)) & 0xFF;

}

return raw;

}

std::int64\_t RawToInt(const std::vector<char>& raw)

{

std::int64\_t value = 0;

for (std::size\_t i = 0; i < raw.size(); ++i)

{

value |= (static\_cast<int64\_t>(static\_cast<unsigned char>(raw[i])) << (i \* 8));

}

return value;

}

std::size\_t GetFileSize(const std::string& fileName)

{

return std::filesystem::file\_size(fileName);

}

void CreateIntArrayFile(const std::string& fileName, std::size\_t length, std::size\_t intSize)

{

std::ofstream file(fileName, std::ios::binary);

std::vector<char> zeroBytes(intSize, 0);

for (std::size\_t i = 0; i < length; ++i)

{

file.write(zeroBytes.data(), intSize);

}

}

class RWIntArrayFile

{

private:

std::string fileName;

std::size\_t length;

std::size\_t intSize;

std::size\_t blockSize;

std::size\_t blockSizeMask;

std::size\_t blockSizeBitShift;

std::size\_t blockSizeMulIntSize;

std::vector<char> cache;

int64\_t cache\_start\_index;

std::size\_t cache\_size;

std::ifstream readFile;

std::ofstream writeFile;

void CloseReadFile()

{

if (readFile.is\_open())

{

readFile.close();

}

}

void CloseWriteFile()

{

if (writeFile.is\_open())

{

writeFile.close();

}

}

void OpenReadFile() {

if (!readFile.is\_open())

{

CloseWriteFile();

readFile.open(fileName, std::ios::in | std::ios::binary);

if (!readFile.is\_open())

{

throw std::runtime\_error("Failed to open file for reading");

}

}

}

void OpenWriteFile() {

if (!writeFile.is\_open())

{

CloseReadFile();

writeFile.open(fileName, std::ios::in | std::ios::out | std::ios::binary);

if (!writeFile.is\_open())

{

throw std::runtime\_error("Failed to open file for writing");

}

}

}

public:

RWIntArrayFile(

const std::string& fileName

, std::size\_t length

, std::size\_t intSize

, std::size\_t blockSize = 4096

)

:

fileName(fileName)

, length(length)

, intSize(intSize)

, blockSize(blockSize)

, cache(blockSize \* intSize)

, cache\_start\_index(-1)

, cache\_size(0)

{

if (!(blockSize > 0 && (blockSize & (blockSize - 1)) == 0))

{

throw std::invalid\_argument("Block size must be a power of 2");

}

if (! (blockSize / intSize \* intSize == blockSize))

{

throw std::invalid\_argument("Block size has to be divided by the size of integers of an array in a file");

}

blockSizeMask = blockSize - 1;

blockSizeBitShift = static\_cast<std::size\_t>(std::log2(blockSize));

blockSizeMulIntSize = intSize << blockSizeBitShift;

OpenReadFile();

}

~RWIntArrayFile()

{

FlushCache();

CloseReadFile();

CloseWriteFile();

}

void FlushCache()

{

if (cache\_start\_index == -1) return;

OpenWriteFile();

writeFile.seekp(cache\_start\_index \* blockSizeMulIntSize, std::ios::beg);

writeFile.write(cache.data(), cache\_size);

writeFile.flush();

cache\_start\_index = -1;

cache\_size = 0;

CloseWriteFile();

}

void LoadBlock(std::size\_t block\_index)

{

OpenReadFile();

readFile.seekg(block\_index \* blockSizeMulIntSize, std::ios::beg);

readFile.read(cache.data(), blockSizeMulIntSize);

cache\_size = readFile.gcount();

cache\_start\_index = block\_index;

CloseReadFile();

}

std::size\_t GetCacheOffset(std::size\_t index)

{

return (index & blockSizeMask) \* intSize;

}

std::int64\_t GetInt(std::size\_t index)

{

std::size\_t block\_index = index >> blockSizeBitShift;

std::size\_t offset = GetCacheOffset(index);

if (cache\_start\_index != static\_cast<int64\_t>(block\_index))

{

FlushCache();

LoadBlock(block\_index);

}

std::vector<char> rawInt(cache.begin() + offset, cache.begin() + offset + intSize);

return RawToInt(rawInt);

}

void SetInt(std::size\_t index, int64\_t value)

{

std::size\_t block\_index = index >> blockSizeBitShift;

std::size\_t offset = GetCacheOffset(index);

if (cache\_start\_index != static\_cast<int64\_t>(block\_index))

{

FlushCache();

LoadBlock(block\_index);

}

auto rawIntBytes = IntToRaw(value, intSize);

std::copy(rawIntBytes.begin(), rawIntBytes.end(), cache.begin() + offset);

cache\_start\_index = block\_index;

}

std::size\_t Length() const

{

return length;

}

std::size\_t IntSize() const

{

return intSize;

}

};

void MergeAt(RWIntArrayFile& m, RWIntArrayFile& l, RWIntArrayFile& r, std::size\_t chk, std::size\_t m\_idx, std::size\_t l\_beg, std::size\_t l\_sz, std::size\_t r\_beg, std::size\_t r\_sz)

{

std::size\_t l\_idx = l\_beg;

std::size\_t r\_idx = r\_beg;

while ((l\_idx < l\_sz && l\_idx - l\_beg < chk) && (r\_idx < r\_sz && r\_idx - r\_beg < chk))

{

int64\_t l\_val = l.GetInt(l\_idx);

int64\_t r\_val = r.GetInt(r\_idx);

if (l\_val <= r\_val)

{

m.SetInt(m\_idx++, l\_val);

l\_idx++;

}

else

{

m.SetInt(m\_idx++, r\_val);

r\_idx++;

}

}

while (l\_idx < l\_sz && l\_idx - l\_beg < chk)

{

m.SetInt(m\_idx++, l.GetInt(l\_idx++));

}

while (r\_idx < r\_sz && r\_idx - r\_beg < chk)

{

m.SetInt(m\_idx++, r.GetInt(r\_idx++));

}

}

void SortAtScale(RWIntArrayFile& m, RWIntArrayFile& l, RWIntArrayFile& r, std::size\_t c)

{

std::size\_t s = m.Length();

std::size\_t ls = 0;

std::size\_t rs = 0;

for (std::size\_t i = 0; i < s; ++i)

{

std::size\_t lrChoice = (i / c) & 1;

std::size\_t blockOffset = i / (2 \* c) \* c;

std::size\_t index = i % c;

(lrChoice == 0 ? l : r).SetInt(blockOffset + index, m.GetInt(i));

if (lrChoice == 0)

{

ls++;

}

else

{

rs++;

}

}

std::size\_t m\_idx = 0;

std::size\_t lr\_idx = 0;

while (m\_idx < s)

{

MergeAt(m, l, r, c, m\_idx, lr\_idx, ls, lr\_idx, rs);

m\_idx += 2 \* c;

lr\_idx += c;

}

}

void DirectMergeSort(RWIntArrayFile& m, RWIntArrayFile& l, RWIntArrayFile& r)

{

std::size\_t s = m.Length();

std::size\_t c = 1;

while (c < s)

{

SortAtScale(m, l, r, c);

c \*= 2;

}

}

int main()

{

constexpr std::size\_t BlockSize = 4096 \* (1 << 10);

constexpr std::size\_t IntSize = 8;

std::size\_t Length = GetFileSize("A.qwa") / IntSize;

CreateIntArrayFile("B.qwa", Length, IntSize);

CreateIntArrayFile("C.qwa", Length, IntSize);

RWIntArrayFile mainFile("A.qwa", Length, IntSize, BlockSize);

RWIntArrayFile leftFile("B.qwa", Length, IntSize, BlockSize);

RWIntArrayFile rightFile("C.qwa", Length, IntSize, BlockSize);

DirectMergeSort(mainFile, leftFile, rightFile);

mainFile.FlushCache();

leftFile.FlushCache();

rightFile.FlushCache();

return 0;

}

Програмна реалізація на мові «Python 3» та C++ є коректними [1] та швидкими після проведення пробних запусків завчасно. Для виконання програмного забезпечення було використано JIT компілятор PyPy3 з метою пришвидшення виконання програмного забезпечення. Шел та середовище виконання програмного забезпечення:

1. OS: «Linux sudo-su-sie 6.9.9-zen1-1-zen #1 ZEN SMP PREEMPT\_DYNAMIC Fri, 12 Jul 2024 00:06:19 +0000 x86\_64 GNU/Linux»;
2. Python: «Python 3.12.4»;
3. PyPy3: «Python 3.10.14 (39dc8d3c85a7, Aug 30 2024, 08:27:45) [PyPy 7.3.17 with GCC 14.2.1 20240805]»;
4. ROM: SSD.
5. RAM: 16 Gb.
6. CPU: 1.6 GHz, 8 ccores.
7. G++: «g++ (GCC) 14.1.1 20240522»
8. C++: «g++ -O3 main.cpp -o main.exe»

Під час проведення внутрішнього профайлінгу програмного забезпечення було виявлено кілька вузьких місць в програмі як операції зчитування та запису до файлів, які додавали значну кількість часу практичного виконання програмногої реалізації.

Під час модифіцкації алгоритму було прийнято рішення використовувати агресивне кешування файлів, чий буфер займає достатньо багато місця (десь 1Мб на файл) для нівелювання затрат на велику кількість системних викликів операційної системи, також щоб пришвидшення таких основних операції алгоритму, як запис до файлів на жорсткому диску. Під час вимірювання практичного часу виконання програми було встановлено, що використання кешування збільшило швидкодію програмного забезпечення на вхідних даних малого розміру (з 56Кб).

Також при профайлінгу було помічено, що абстраговані операції зчитування та запису до файлу через клас-обертку є досить неоптимізованими з точки зору арифметичних та математичних дій, тому було замінено всі операції ділення на модуль на побітове «і» використовуючи маски. Повторюючі операції, які включали подалі незмінювані зміні, були заздалегіть обрахованими під час створення класу-обертку. Єдиним недоліком є те, що розмір кеш-блоку має бути ступінню 2.

Нижче (таблиця 3.3.1) наведено приклад виконання програмного забезпечення без та з «PyPy3» замість «CPython» та з та без використання кешування файлів.

Таблиця 3.3.1 — Практичний час виконання програмного забезпечення при вхідних умовах різного типу

| Номер тесту | З PyPy3? | З кешування файлів (десь 1Мб на файл)? | Розмір вхідних даних (Кб) | Практичний час виконання (с.) |
| --- | --- | --- | --- | --- |
| 1 | — | — | 56 | 1.721 |
| 2 | + | — | 56 | 1.678 |
| 3 | — | + | 56 | 0.341 |
| 4 | + | + | 56 | 0.163 |
| 5 | + | + | 200 | 0.293 |
| 6 | + | + | 640 | 0.791 |
| 7 | + | + | 2364 | 2.658 |
| 8 | + | + | 7088 | 8.603 |

Так як теоретична часова складність алгоритму [1] є , то пратична часова складність програмної реалізації при найкращих умовах (таблиця 3.3) та середовиз після підрахунків методом виведення константи швидкодії, тому приблизним результатом буде: , час при вхідних даних розміром в мегабайти: . Нижче (таблиця 3.3.2) наведено час виконання програмної реалізації використовуючи апроксимовані формули практичного часу виконання при найкращих умовах та модифікацій.

Таблиця 3.3.2 — Практичний час виконання використовуючи теоретичні методи та апроксимації

| Номер практичного тесту | Розмір файлу (Мб) | Практичний час виконання за теоретично-практичними формулами |
| --- | --- | --- |
| 1 | 100 | 2.3 хв. |
| 2 | 300 | 7.5 хв. |
| 3 | 1 000 | 26 хв. |
| 4 | 3 000 | 84 хв. |
| 5 | 10 000 | 4.9 г. |
| 6 | 30 000 | 15.5 г. |
| 7 | 100 000 | 2.5 д. |
| 8 | 300 000 | 7 д. |

Швидкість виконання програмної реалізації на мові Python є незадовільними та відстають по ліміту практичного часу викоання в 4.33 разів. Так як мова Python є динамічно типізованою, то можливостей для оптимізації PyPy3 не так і багато. Єдиним можливим кроком є перехід на мову зі статичною типізацією як C++.

Практичний час виконання на мові C++ (таблиця 3.3.3) використовуючы усы практики оптимізації та пришвидшення коду наведено нижче.

Таблиця 3.3.3 — практичний час виконання на мові C++ на даних різного розміру

|  |  |  |
| --- | --- | --- |
| Номер практичного тесту | Розмір файлу (Мб) | Практичний час виконання (с.) |
| 1 | 1 | 0.366 |
| 2 | 3 | 1.146 |
| 3 | 10 | 4.029 |
| 4 | 30 | 13.104 |
| 5 | 100 | 48.389 |

Як і минулого разу, практичний час викоання програмної реалізації на мові C++ вже буде становити: [1].

Таблиця практично-теоретичного часу викоанння програмної реалізації на мові C++ наведено в таблиці 3.3.4 по виведеній формулі часу виконання за практичними .

Таблиця 3.3.4 — практично-теоретичний час виконання програмної реалізації на мові C++ по формулі вище

|  |  |  |
| --- | --- | --- |
| Номер теоретичного тесту | Розмір файлу (Мб) | Практичний час виконання |
| 1 | 100 | 48.389 с. |
| 2 | 300 | 2.563 хв. |
| 3 | 1000 | 9.070 хв. |
| 4 | 3000 | 28.650 хв. |
| 5 | 10000 | 1.679 г. |
| 6 | 16384 | 2.810 г. |
| 7 | 32768 | 5.785 г. |
| 8 | 65536 | 11.902 г. |
| 9 | 131072 | 24.465 г. |
| 10 |  |  |

Висновок

Під час перевірки коректності виконання алгоритму було виявлено, що результат сорутвання в масиві чисел в файлі є коректним. Псевдокод алгоритму зовнішнього сортування є достатньо компактним, тому виникнення помилок є більш малоймовірним. Програмна реалізація була створена мовами Python 3 та C++.

При виконанні даної лабораторної роботи не вийшло реалізувати достатньо швидкодійну програмну реалізацію на мові «Python 3» при використанні найкращих модифікацій та програмного забезпечення. Але на C++ результат є практично задовільним.

Було виявлено, що такі операції з файлами, як зчитування та запис є дуже повільними, мається на увазі, що операції значно знижують швидкодію програмного забезпечення. Також створено та впроваджено кешування файлів для збільшення швидкодії програмного забезпечення. За результатами цієї впровадженої модифікації швидкодія була значно збільшена. Використання JIT компілятора «PyPy3» дало значне покращення швидкодії. Також було створено нову програмну реалізацію на мові C++ для покращення швидкодії.

Окремо кажучи, багаточисленні та часті операції над змінними були опитмізовані на більш швидкі та більш оптимізовані альтернативи. На жаль розмір кеш-блоку має бути ступінню 2.

Час виконання алгоритму є практично задовільним та відстає від попередньо встановленого часу лаборантом в 300 секунд (6 хвилин) в 1.814 разів при використанні програмної реалізації на мові C++.

Результатом викоанння цієї лабораторної роботи є практично задовільно. Відставання від встановленого ліміту часу лаборантом цієї практичної роботи є достатньо незначним.

Посилання на джерела

[1] Головченко Максим Миколайович (2023). Проектування алгоритмів. Факультет інформатики та обчислювальної техніки, Кафедра інформатики та програмної інженерії.

Критерії оцінювання

Максимальний бал за виконання лабораторної роботи дорівнює – 5.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 15%;
* програмна реалізація алгоритму – 30%;
* програмна реалізація модифікацій – 40%;
* робота з git – 10%;
* висновок – 5%.