КИЇВСЬКИЙ НАЦIОНАЛЬНИЙ УНIВЕРСИТЕТ IМЕНI ТАРАСА ШЕВЧЕНКА Факультет комп’ютерних наук та кібернетики Кафедра теорії та технології програмування

Звіт до лабораторної роботи

з дисциплiни

Розподілене та паралельне

програмування

Виконала студентка 3-го курсу,

групи ТТП-3  
Скоробагатько Карина

Київ - 2020

Змiст

|  |  |  |
| --- | --- | --- |
| Завдання | | 2 |
| Теоретична частина | | 3 |
| Практична частина | | 4 |
| а) | Реалiзацiя алгоритму . . . . . . . . . . . . . . . . . . . . . | 5 |
| б) | Послідовний запуск . . . . . . . . . . . . . . . . . . . . . . | 6 |
| в) | Використання MPI . . . . . . . . . . . . . . . . . . . . | 7 |
| г) | Використання openMP . . . . . . . . . . . . . . . . . . . . . | 10 |
| Висновки | | 12 |
| Перелiк джерел посилання | | 13 |

Завдання

Метою лабораторної роботи є виконання наступних завдань:

* Поглибити знання паралельного програмування;
* Створити програму для нормалізації вектора; Використати технологiї MPI та openMP;
* Розглянути використання MPI та openMP на мовi Python; Порівняти результати та зробити висновки.

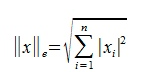
Теоретична частина

Нормалізація - приведення до одиничного розміру; нормалізація в тривимірному просторі - по суті є масштабуванням в куб одиничного розміру

Нормалізація вектора - це перетворення заданого вектора в вектор в тому ж напрямку, але довжиною в один символ.

Для нормалізації вектора потрібно кожну компоненту поділити на довжину вектора (норму), або помножити інверсію довжини на даний вектор.

Нор́ма — це функція, що задана на лінійному просторі і є узагальненням поняття довжини вектора.

Я використовую Евклідову норму:

Практична частина

У проекті є два способи нормалізації вектора: послідовно та паралельно.

У послідовній частині проекту вектор нормалізуємо послідовним алгоритмом.

У паралельній частині проекту за допомогою MPI створюється основний процес (master) та декілька підпорядкованих йому (slave). Основний процес роздає задачі та отримує результати з підпорядкованих йому. Кожному з процесів передаються на обчислення рядки, а на виході отримуються результати їх обчислень. Коли всі рядки отримано, головний процес завершується та повертає результати обчислень у програму, з якої він був викликаний.

Використання MPI було продемонстроване на c++, а оскільки в мене операційна система Ubuntu, то реалізація openMP була виконана на мові Python за допомогою відповідного аналога.

а) Реалiзацiя алгоритму

Алгоритм реалізований за допомогою бібліотеки math[2]. Для проведення тестування був використаний наступний модуль для пошуку:

import time

from math import gcd

def norm(num1, num2):

return gcd(num1, num2) == 1

def normalize(chunk, input\_list):

start = time.time()

res = []

for number in chunk:

for s\_number in input\_list:

if norm(number, s\_number):

if number > s\_number:

res.extend([number, s\_number])

else:

res.extend([s\_number, number])

return time.time() - start, res

б) Послідовний запуск

Для послідовного запуску встановлюємо параметр в кількості процесів на 1 ( pymp.Parallel(1) ) та викликаємо нашу функцiю з уже знайомого модуля (normalize() ).

from utiles import normalize

import time

import pymp

input\_list = list(range(1, 5000))

input\_list = list(set(input\_list))

start = time.time()

results = pymp.shared.list()

with pymp.Parallel(1) as p:

for iter\_item in p.iterate(input\_list):

\_, r = normalize([iter\_item], input\_list)

results.append(r)

print("Serial work", time.time() - start, "seconds")

Результат на 100:

****

Результат на 2500:



Результат на 5000:



в) Використання MPI

Для написання програми використано середовище CLion, мову програмування C++. Для створення паралельного алгоритму використано MPI (Message Parsing Interface), власне, його реалізацію у C++.

Інтерфейс може працювати в послідовному і в паралельному режимі. Я запускаю одразу обидва, щоб порівняти результати наглядно.

MPI\_Init(&argc, &argv);// ініціалізація бібліотеки

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); //Дві інформаційних функції: повідомляють розмір групи (тобто загальна кількість завдань, приєднаних до її галузі зв'язку) і порядковий номер викликає завдання

MPI\_Send(&norm, 1, MPI\_INT, dest, i, MPI\_COMM\_WORLD);  
MPI\_Recv(&a, 1, MPI\_INT, source, i, MPI\_COMM\_WORLD, &status); //Це найпростіший тип зв'язку між завданнями: одна гілка викликає функцію передачі даних, а інша - функцію прийому

Аргументи функцій:

1. Адреса буфера, з якого в задачі 1 беруться, а в завданні 2 поміщаються дані. Пам'ятайте, що набори даних у кожного завдання свої, тому, наприклад, використовуючи один і той же ім'я масиву в декількох завданнях, Ви вказуєте не одну і ту ж область пам'яті, а різні, ніяк один з одним не пов'язані.
2. Розмір буфера. Здається не в байтах, а в кількості осередків. Для MPI\_Send вказує, скільки клітинок потрібно передати (в прикладі передаються 5 чисел). У MPI\_Recv означає максимальну ємність приймального буфера. Якщо фактична довжина прийшов повідомлення менше - останні осередки буфера залишаться недоторканими, якщо більше - відбудеться помилка часу виконання.
3. Тип осередку буфера. MPI\_Send і MPI\_Recv оперують масивами однотипних даних. Для опису базових типів Сі в MPI визначені константи MPI\_INT, MPI\_CHAR, MPI\_DOUBLE і так далі, які мають тип MPI\_Datatype. Їх назви утворюються префіксом "MPI\_" і ім'ям відповідного типу (int, char, double, ...), записаним великими літерами. Користувач може "реєструвати" в MPI свої власні типи даних, наприклад, структури, після чого MPI зможе обробляти їх нарівні з базовими. Процес реєстрації описується в розділі "Типи даних".
4. Номер завдання, з якою відбувається обмін даними. Всі завдання всередині створеної MPI групи автоматично нумеруються від 0 до (розмір групи-1). У прикладі завдання 0 передає завданню 1, завдання 1 приймає від завдання 0.
5. Ідентифікатор повідомлення. Це ціле число від 0 до 32767, яке користувач вибирає сам. Воно служить тієї ж мети, що і, наприклад, розширення файлу - завдання-приймач:
   1. за ідентифікатором визначає сенс прийнятої інформації;
   2. повідомлення, що прийшли в невідомому порядку, може отримувати від загального вхідного потоку в потрібному алгоритму порядку. Хорошим тоном є позначення ідентифікаторів символьними іменами за допомогою операторів "#define" або "const int".
6. Описувач галузі зв'язку (комунікатор). Зобов'язаний бути однаковим для MPI\_Send і MPI\_Recv.
7. Статус завершення прийому. Містить інформацію про прийняте повідомленні: його ідентифікатор, номер завдання-передавача, код завершення і кількість фактично прийшли даних.

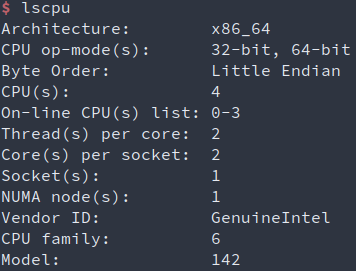
MPI\_Scatter(&norm, 1, MPI\_INT, &norm, 1, MPI\_INT, size - 1, MPI\_COMM\_WORLD); // частини буфера, що передається з завдання root розподіляються по прийомним буферам всіх завдань

MPI\_Gather(&v1[0], nT, MPI\_INT, &vT[0], nT, MPI\_INT, 0, MPI\_COMM\_WORLD); // збирає в приймальний буфер завдання root, що передають буфера інших завдань

MPI\_Finalize(); // нормальне закриття бібліотеки

# **Результати тестів**

Програму протестовано на комп’ютері з наступними характеристиками:



1. const int N = 1000; 4 процеси



1. const int N = 2500; 4 процеси



3. const int N = 5000; 4 процеси



4. const int N = 5000; 8 процесів



Інші результати будуть продемонстровані в порівняльній таблиці нижче у висновках.

г) Використання openMP

Для реалiзацiї openMP використано iнтерфейс pyMP[3]. Для збереження результатів використовується shared.list(), що є абстракцією для використання shared memory. Відповідний параметр для кількості процесів встановлено на 4 (pymp.Parallel(4)). Зараз нижче будуть показані порівняльні результати для 2, 3 та 4 процесорів, потім у висновках буде порівняльна табличка.

from utiles import normalize

import time

import pymp

input\_list = list(range(1, 5000))

input\_list = list(set(input\_list))

start = time.time()

results = pymp.shared.list()

with pymp.Parallel(4) as p:

for iter\_item in p.iterate(input\_list):

\_, r = normalize([iter\_item], input\_list)

results.append(r)

print("Paralel work", time.time() - start, "seconds")

Результат на 100 (2 процесори):



Результат на 2500 (2 процесори):



Результат на 5000 (2 процесори):

****

Результат на 100 (3 процесори):

****

Результат на 2500 (3 процесори):

****

Результат на 5000 (3 процесори):

****

Результат на 100 (4 процесори):



Результат на 2500 (4 процесори):



Результат на 5000 (4 процесори):



Висновки

Далі наведено результати запуску на 2, 4, 8 потоках i 1000, 2500, 5000 елементів відповідно.

**Синхронний запуск**

|  |  |  |  |
| --- | --- | --- | --- |
| N | 1000 | 2500 | 5000 |
|  |  |  |  |
| 1 | 2.0271 | 3.3242 | 14.9416 |
|  |  |  |  |
|  |  | **MPI** |  |
|  |  |  |  |
| N | 1000 | 2500 | 5000 |
|  |  |  | 25 |
| 2 | 0.5214 | 3.0929 | 12.5593 |
|  |  |  |  |
| 4 | 0.2755 | 2.6065 | 6.8017 |
|  |  |  |  |
| 8 | 0.3640 | 2.2386 | 6.0903 |
|  |  |  |  |

**openMP**

|  |  |  |  |
| --- | --- | --- | --- |
| N | 1000 | 2500 | 5000 |
|  |  |  |  |
| 2 | 1.8572 | 4.3269 | 14.4537 |
|  |  |  |  |
| 4 | 1.8032 | 2.8028 | 7.93392 |
|  |  |  |  |
| 8 | 1.0334 | 3.0248 | 7.02621 |
|  |  |  |  |

З результатiв очевидно, що MPI працює швидше за синхронний, а openMP повільніше. Причиною уповільнення є використання спільної пам'яті openMP i обробка кожного елементу, як окремого списку. Отже, використання MPI вигiдно, коли потоки не мають великих проміжних результатів, в іншому випадку варто надати перевагу openMP. Паралельний алгоритм стає значно швидший за послідовний на великій кількості розрахунків (довжина вектора) при нормалізації вектора.

Перелiк джерел посилання

1. Евклідів вектор Вікіпедія [Електронний ресурс] : Режим до-

ступу: <https://uk.wikipedia.org/wiki/%D0%95%D0%B2%D0%BA%D0%BB%D1%96%D0%B4%D1%96%D0%B2_%D0%B2%D0%B5%D0%BA%D1%82%D0%BE%D1%80>

1. math Mathematical functions; Python 3.7.3 documentation [Електронний ресурс] : Режим доступу: <https://docs.python.org/3/library/math.html>
2. MPI for Python ; MPI for Python 3.0.1 documentation [Електронний ресурс] : Режим доступу: <https://mpi4py.readthedocs.io/en/stable/>
3. classner/pymp: Easy, OpenMP style multiprocessing for Python on Unix. [Електронний ресурс] : Режим доступу: <https://github.com/classner/pymp>