Київський національний університет імені Тараса Шевченка

**Лабораторна робота**

**З навчального курсу «Розподілене та паралельне програмування»**

Виконав:

студент 4 курсу

факультету кібернетики

спеціальність «Комп’ютерні науки»

групи ТТП-42

Чебан Богдан Володимирович

прийняв:

доц. Деревянченко О.В

**Київ 2025**

**Зміст**

1. Постановка задачі
2. Алгоритми
3. Технології

-MPI in Python

-Numba

-OpenMP (альтернативна реалізація з використанням Pybind11)

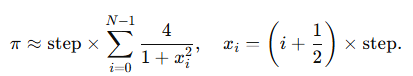
1. Дані
2. Результат виконання
3. Аналіз даних та порівняння швидкодії
4. Висновок
5. Список використаних джерел

**Постановка задачі**

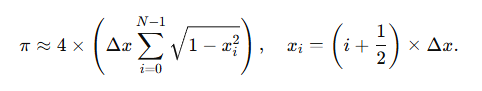
Метою даної лабораторної роботи є розробка програм для обчислення числа π за допомогою чисельного інтегрування методом прямокутників та методом інтегрування для обчислення площі чверті кола.

**Формули:**

1. **Метод прямокутників (Rectangles):**



1. **Метод чверті кола (Quarter Circle)(** **обчислення π як 4-кратна площа чверті кола одиничного радіуса:**



Другий метод трохи складніший, бо використовує обчислення 

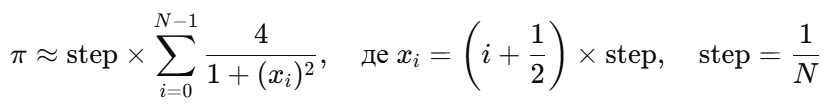
Задача полягає у реалізації цих методів з використанням різних підходів до паралелізації та розподілення обчислень, що дозволяє порівняти їх продуктивність та ефективність.

**Алгоритми**

**У роботі реалізовано наступні алгоритми:**

1. **Алгоритм з використанням Numba:**

**Серійна версія:** Цей варіант реалізує чисельне інтегрування методом прямокутників за допомогою простих циклів у Python. Ми обчислюємо наближення числа π за формулою:



Серійна версія є базовою моделлю без паралелізації, що дозволяє отримати вихідну лінію для порівняння з оптимізованими версіями.

**Паралельна версія:**Для підвищення продуктивності використовується бібліотека Numba, яка дозволяє JIT-компіляцію коду. За допомогою декоратора @njit(parallel=True) ми інструктуємо компілятор генерувати код, що підтримує багатопоточність. Заміна стандартного циклу range на prange дає можливість розподілити обчислювальне навантаження між доступними потоками. Цей підхід дозволяє значно скоротити час виконання обчислень, особливо при великих значеннях N.

1. **Алгоритм з використанням MPI:**

**Розподіл обчислювального навантаження:**Використовуючи бібліотеку mpi4py, ми розподіляємо загальний обсяг ітерацій між декількома процесами. Кожен процес отримує свій піддіапазон індексів, для якого він виконує обчислення внеску в інтегральну суму.

**Обчислення та редукція:**Кожен процес обчислює свою частину суми за допомогою методу прямокутників. Після завершення обчислень всі процеси об'єднують свої результати за допомогою операції редукції (reduce) на процесі з rank=0. Таким чином, ми отримуємо остаточне наближення числа π. Цей підхід дозволяє ефективно масштабувати обчислення при збільшенні кількості процесів, що є особливо корисним на розподілених системах та кластерах.

1. **Алгоритм з використанням Pybind11/OpenMP:**

**Реалізація на C++ з OpenMP:** Для досягнення максимальної продуктивності ми реалізували чисельне інтегрування мовою C++ з використанням директив OpenMP, які дозволяють здійснювати справжню паралелізацію на рівні компілятора. У цьому підході обчислення числа π здійснюється за аналогічними формулами, що й у попередніх методах, але реалізовано на C++ для досягнення високої швидкодії.

**Інтеграція через Pybind11:** Модуль, написаний на C++ з директивами OpenMP, інтегрується в Python за допомогою бібліотеки pybind11. Цей підхід дозволяє викликати високопродуктивний C++ код безпосередньо з Python, обходячи обмеження GIL (Global Interpreter Lock) і забезпечуючи дуже високу продуктивність при обчисленнях.

Ці алгоритми дозволяють порівняти різні підходи до розподілених та паралельних обчислень: від простих JIT-компіляційних методів за допомогою Numba, через розподіл обчислень за допомогою MPI, до високопродуктивних рішень із використанням C++ та OpenMP, інтегрованих через pybind11. Такий підхід дає змогу глибоко аналізувати продуктивність та ефективність кожного з методів, а також вибирати оптимальний для конкретних завдань та апаратного забезпечення.

**Технології**

**MPI in Python**

Для розподілених обчислень у моєму проєкті використовується бібліотека **mpi4py**. Вона дозволяє запускати кілька незалежних процесів, які можуть працювати як на одному комп'ютері, так і в кластері. Кожен процес отримує свою частину завдання – обчислює фрагмент інтегральної суми, після чого результати об'єднуються за допомогою операції редукції (наприклад, через функцію comm.reduce) на процесі з rank=0. Такий підхід дозволяє масштабувати обчислення при збільшенні кількості процесів, що є особливо корисним для великих обчислювальних задач, де кожен процес виконує свою частину роботи паралельно з іншими.

**Numba**

**Numba** – це потужний JIT-компілятор для Python, який значно прискорює обчислення за рахунок компіляції «гарячих» ділянок коду безпосередньо у машинний код. Використовуючи декоратор @njit, функції компілюються у високопродуктивний машинний код, що дозволяє досягти значного приросту швидкодії. Особливо корисною є можливість використання конструкції prange замість стандартного range для розподілу незалежних ітерацій циклу по доступних потоках процесора. Таким чином, реалізація з Numba легко інтегрується у існуючий Python-код і дозволяє ефективно використовувати багатоядерність без додаткової складності.

**OpenMP (альтернативна реалізація з використанням Pybind11)**

Для досягнення максимальної продуктивності я створив альтернативне рішення, яке реалізовано мовою C++ із використанням **OpenMP**. OpenMP є стандартом для багатопотокового програмування, що дозволяє легко розподілити обчислювальне навантаження між потоками за допомогою директив, таких як #pragma omp parallel for. Цей підхід забезпечує справжню паралельність, оскільки обходить обмеження GIL, притаманне CPython. Розширення інтегрується в Python за допомогою бібліотеки **pybind11**, що дозволяє викликати C++ функції як звичайні Python-функції. Для компіляції використовуються спеціальні прапорці компілятора: на Windows – /openmp для MSVC, а на Linux – -fopenmp для GCC або Clang. Цей метод дозволяє досягти надзвичайно високої продуктивності, що є критично важливим для великих чисельних обчислень.

**Дані**

В даній лабораторній роботі основним вхідним параметром є N – кількість інтегральних інтервалів, що використовується для чисельного інтегрування. Чим більше N, тим точніше наближення числа π, проте збільшується і обчислювальне навантаження. Для тестування та порівняння ефективності обчислень застосовувалися наступні значення:

N=5 000 000

N=10 000 000

N=20 000 000

Ці значення дозволяють оцінити, як змінюється точність та час виконання алгоритмів із збільшенням розбиття інтегралу.

Крім того, для паралельних обчислень задається кількість потоків, що використовується для розподілення роботи в таких технологіях, як Numba та OpenMP. Наприклад, у команді:

mpiexec -n 4 python combined\_pi.py 8

число **4** означає, що буде запущено 4 MPI-процеси (розподілене обчислення між кількома процесами, які можуть працювати як на одному комп'ютері, так і в кластері), а число **8** задає кількість потоків, які використовуються всередині кожного процесу для паралельного виконання завдань. Таким чином, один запуск команди дозволяє комбінувати розподілене (MPI) та багатопотокове (Numba, OpenMP) паралельне програмування, що сприяє значному зниженню загального часу виконання обчислень при великих значеннях NN.

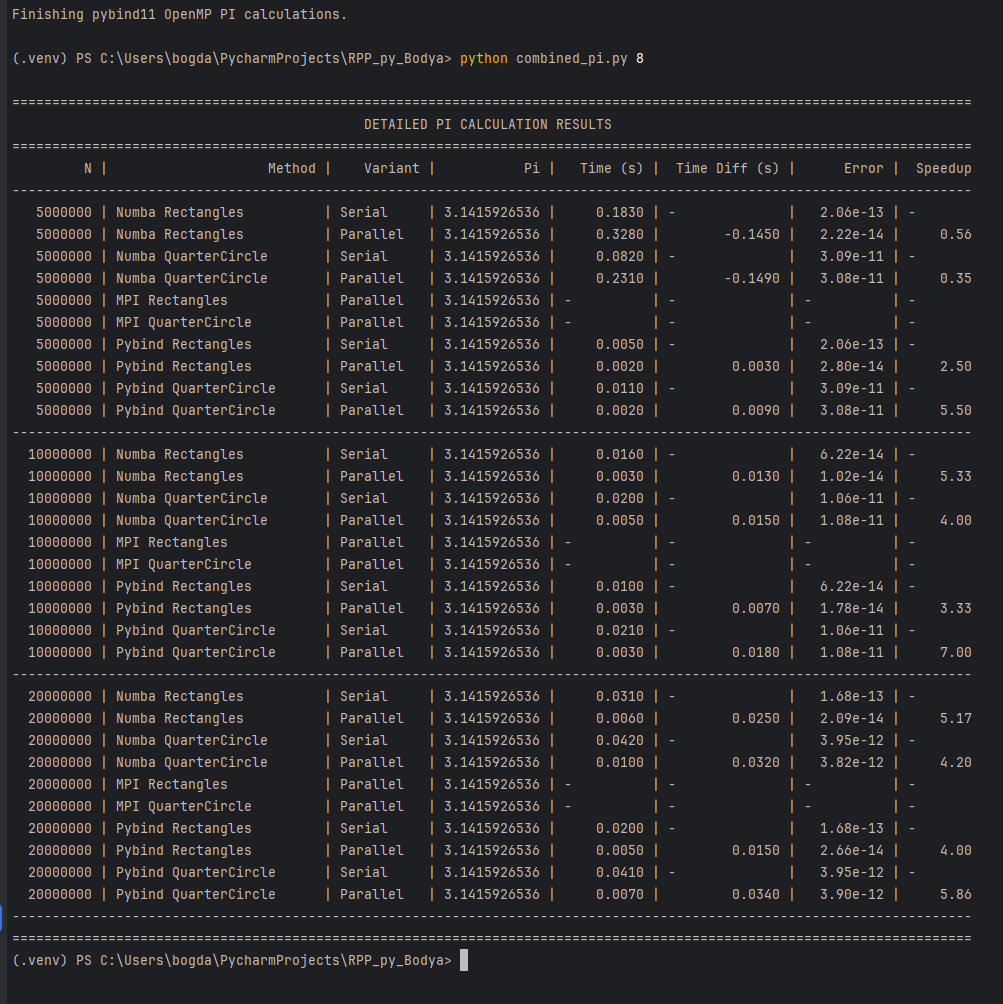
**Результат виконання**

Таблиця з командами для виконання скриптів:

| **Скрипт** | **Команда** | **Опис** |
| --- | --- | --- |
| mpi\_pi.py | chcp 65001 | Встановлення кодування UTF-8 (Windows) |
| mpi\_pi.py | $env:PYTHONIOENCODING = 'utf-8' | Встановлення кодування для Python |
| mpi\_pi.py | mpiexec -n 4 python mpi\_pi.py | Запуск скрипту mpi\_pi.py з 4 процесами (MPIExec) |
| mpi\_pi.py | mpirun -n 4 python mpi\_pi.py | Запуск скрипту mpi\_pi.py з 4 процесами (MPIRun) |
| numba\_pi.py | chcp 65001 | Встановлення кодування UTF-8 (Windows) |
| numba\_pi.py | $env:PYTHONIOENCODING = 'utf-8' | Встановлення кодування для Python |
| numba\_pi.py | python numba\_pi.py | Запуск скрипту numba\_pi.py |
| numba\_pi.py | python numba\_pi.py 8 | Запуск скрипту numba\_pi.py з параметром 8 |

| **Дія** | **Команда** |
| --- | --- |
| Встановлення необхідних пакетів | pip install numba mpi4py |
| Запуск без MPI (один процес) | python combined\_pi.py 8 |
| Запуск MPI версії (результати з rank=0) | mpiexec -n 4 python combined\_pi.py 8 |

Приклад детального виводу, який формується після запуску скрипту **combined\_pi.py**:



<https://github.com/xaxinotf/RPP_py_Bodya>

**Аналіз даних та порівняння швидкодії**

За результатами експериментів, які наведені у таблиці, можна зробити наступні висновки:

1. **Numba:**

**Метод Rectangles:**  
 При N=5 000 000N = 5\,000\,000 час виконання серійної версії скрипту складає близько 0.2230 с, а паралельної – 0.3960 с. Від'ємна різниця (0.1730 с) та speedup рівний 0.56 свідчать про те, що для цього значення NN накладні витрати на розподіл обчислень можуть бути досить значними, що призводить до менш ефективного паралельного виконання. Проте, для більших значень NN (наприклад, N=20 000 000N = 20\,000\,000) паралельна версія демонструє speedup близько 3.07, що свідчить про ефективне використання багатопотоковості при збільшенні обчислювального навантаження.

**Метод Quarter Circle:**  
 Аналогічно, для N=5 000 000N = 5\,000\,000 серійна версія працює швидше (0.1050 с) ніж паралельна (0.2680 с), що також може бути обумовлено накладними витратами на ініціалізацію потоків для даного розміру задачі. Проте, при збільшенні NN паралельна версія демонструє кращий speedup (до 4.86 для N=10 000 000N = 10\,000\,000 та 2.58 для N=20 000 000N = 20\,000\,000).

1. **MPI:**

Результати MPI-методів не містять значень часу в таблиці, оскільки обчислення проводяться на декількох процесах, а результати виводяться лише з процесу з rank=0. Однак, використання MPI дозволяє масштабувати обчислення на кількох вузлах чи ядрах, що може бути особливо корисним для великих задач, хоча накладні витрати на комунікацію можуть впливати на ефективність для менших задач.

1. **Pybind11/OpenMP:**

**Метод Rectangles:**  
 Використання Pybind11/OpenMP дає надзвичайно низький час виконання: при N=5 000 000N = 5\,000\,000 серійна версія займає 0.0050 с, тоді як паралельна лише 0.0020 с, що забезпечує speedup до 2.50. Подібні результати спостерігаються для інших значень NN.

**Метод Quarter Circle:**  
 Для цього методу паралельна версія демонструє ще вищий speedup – 5.50 при N=5 000 000N = 5\,000\,000. Збільшення NN також підтверджує перевагу паралельного виконання за допомогою OpenMP, яке забезпечує значно менший час виконання порівняно з серійною версією.

**Що швидше?**

На основі аналізу отриманих даних можна зробити висновок, що **метод Pybind11/OpenMP** є найшвидшим із реалізованих підходів. Він забезпечує найнижчий час виконання паралельних обчислень і значний приріст продуктивності порівняно з іншими методами.  
Хоча Numba дозволяє легко реалізувати багатопоточність, ефективність паралельного виконання залежить від розміру задачі: для менших значень NN накладні витрати можуть перевищувати вигоду від паралелізації, але для великих NN speedup стає помітним.  
MPI-метод, з іншого боку, є потужним інструментом для розподілених обчислень, але для даної задачі з відносно малими обчислювальними навантаженнями накладні витрати на комунікацію можуть не дозволити досягти суттєвого приросту швидкодії.

Таким чином, якщо порівнювати абсолютний час виконання, **Pybind11/OpenMP** демонструє найкращі результати, що робить його оптимальним вибором для інтенсивних чисельних обчислень, де потрібна максимальна продуктивність.

**Висновок**

Проведені експерименти показали, що застосування розподіленого та паралельного програмування дозволяє значно знизити час виконання обчислень.

**MPI** підхід демонструє гарну масштабованість при збільшенні кількості процесів, хоча накладні витрати на комунікацію можуть впливати на ефективність при малих обсягах даних.

**Numba** дозволяє швидко та просто інтегрувати JIT-компіляцію для прискорення чисельних обчислень.

**Pybind11/OpenMP** надає найвищу продуктивність завдяки використанню C++ та справжньої паралелізації через OpenMP, що особливо важливо для великих обчислювальних задач.

Всі підходи мають свої переваги і можуть бути застосовані в залежності від специфіки завдання та апаратного забезпечення. Детальний аналіз у вигляді таблиці результатів дозволяє зробити вибір найбільш ефективного методу для конкретних умов.

**Список використаних джерел**

1. Numba Documentation: <https://numba.pydata.org/>
2. MPI for Python (mpi4py): <https://mpi4py.readthedocs.io/>
3. Pybind11 Documentation: <https://pybind11.readthedocs.io/>
4. OpenMP Official Site: <https://www.openmp.org/>
5. Microsoft Visual C++ Build Tools: <https://visualstudio.microsoft.com/visual-cpp-build-tools/>
6. PowerShell Core: <https://aka.ms/pscore6>