Методичні вказівки

до виконання лабораторних робіт з курсу

«Технології розподілених систем і паралельних обчислень»

для студентів спеціальності 122 – Комп’ютерні науки

Міністерство освіти і науки України
Вінницький національний технічний університет

Методичні вказівки

до виконання лабораторних робіт з курсу
«Технології розподілених систем і паралельних обчислень»
для студентів спеціальності 122 – Комп’ютерні науки

Вінниця
ВНТУ
2019

Рекомендовано до друку Методичною радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 9 від 24.04.2019 р.)

Рецензенти:

Т. Б. Мартинюк, д.т.н., проф., зав. кафедри ОТ

В. П. Майданюк, к.т.н., доцент кафедри ПЗ

Методичні вказівки до виконання лабораторних робіт з курсу «Технології розподілених систем і паралельних обчислень» для студентів спеціальності 122 – Комп'ютерні науки / Уклад. А. А. Яровий, С. В. Барабан, В. С. Озеранський, Є. О. Шемет. – Вінниця : ВНТУ, 2019. – 56 с.

У методичних вказівках розглянуто теоретичні відомості та практичні рекомендації, які дають можливість майбутнім фахівцям раціонально використовувати паралельні та розподілені комп'ютерні системи та технології; наведено завдання для лабораторних робіт та рекомендовану літературу. Методичні вказівки розроблено відповідно до плану кафедри та робочої програми дисципліни «Технології розподілених систем і паралельних обчислень».

Зміст

Лабораторна робота № 1

Тема роботи: GRID-системи та організація розподілених обчислень за принципом «volunteer computing».

Мета роботи: метою цієї роботи є ознайомлення з сучасним станом розвитку GRID-технологій, типовою структурою GRID-систем та організацією розподілених обчислень за принципом «volunteer computing» на прикладі проекту Folding@Home.

1 Теоретична частина

1.1 GRID-системи

Наразі не існує універсального визначення технології GRID. Одні з засновників GRID, Ян Фостер (із Аргонської Національної лабораторії і Чиказького університету, США) і Карл Кессельман (із Інституту інформаційних наук Університету Південної Каліфорнії, США) дають таке означення: «*GRID* – узгоджене, відкрите і стандартизоване середовище, що забезпечує гнучкий, безпечний, скоординований розподіл (загальний доступ) ресурсів у рамках віртуальної організації». Під *віртуальною організацією* (ВО) в цьому випадку розуміють динамічне співтовариство людей, які спільно використовують Grid-ресурси відповідно до узгоджених між ними і власниками ресурсних центрів правил. Ці правила регулюють доступ до всіх типів засобів, включно й комп'ютерні системи, програмне забезпечення, а також дані.

В загальному випадку належність розподіленої системи до GRID можливо перевірити на основі виконання трьох критеріїв:

– система координує використання ресурсів за відсутності централізованого керування цими ресурсами;

– система використовує стандартні, відкриті, універсальні протоколи та інтерфейси;

– система забезпечує високоякісне обслуговування, з точки таких характеристик, як, наприклад, пропускна здатність, доступність, надійність тощо.

Основними загальними завданнями GRID є:

– створення із устаткування, що серійно випускається, широкомасштабних розподілених обчислювальних систем і систем обробки, комплексного аналізу і моніторингу даних, джерела яких можуть бути (глобально) розподілені;

– підвищення ефективності обчислювальної техніки шляхом надання GRID-системі ресурсів, що тимчасово простоюють.

Створення GRID-середовища передбачає розподіл обчислювальних ресурсів по територіально розподілених сайтах, на яких встановлено

спеціалізоване програмне забезпечення. Це дозволяє розподіляти завдання по сайтах і приймати їх, повертати результати користувачеві, контролювати права користувачів на доступ до певних ресурсів, здійснювати моніторинг ресурсів і так далі (рис. 1).

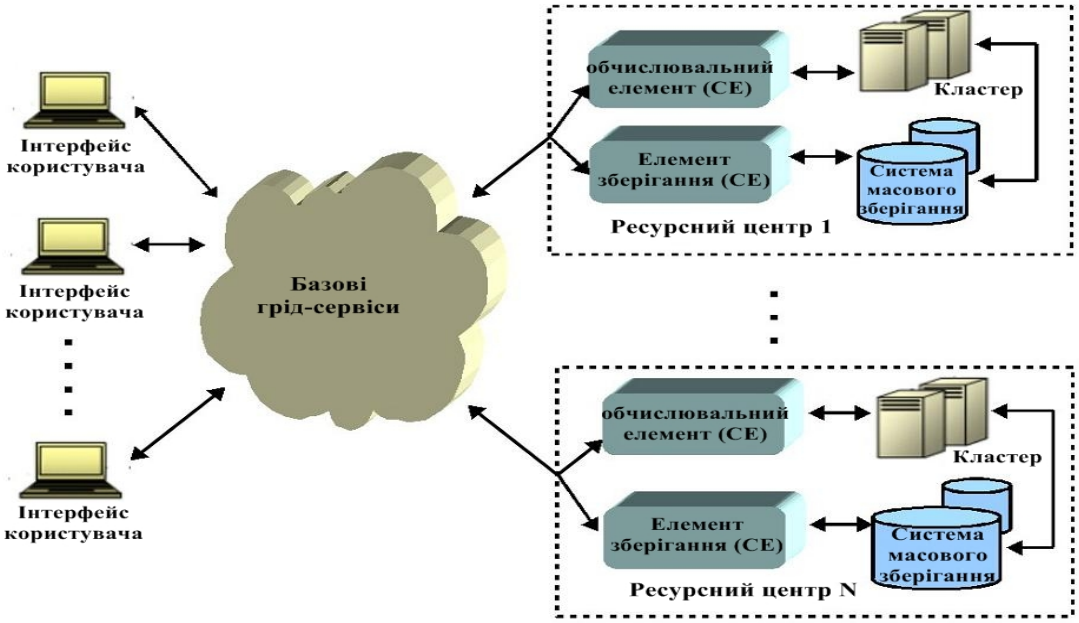


Рисунок 1 – Узагальнена схема структури GRID

Варто відзначити, що в контексті GRID-систем слово «ресурс» розуміється в широкому сенсі, тобто ресурсом може бути апаратура (жорсткі диски, процесори), а також системне і прикладне програмне забезпечення (бібліотеки, програми). Таким чином, GRID претендує на роль універсальної інфраструктури для обробки даних, в якій функціонує множина сервісів, що дозволяють надати нову якість вирішення різних класів задач. Особливо таких, які неможливо вирішити у адекватні строки локально на одному, навіть найпотужнішому, комп'ютері. Типовими прикладами таких задач є:

– масова обробка потоків даних надвеликого обсягу;

– реалістична візуалізація великих наборів даних;

– складні бізнес-додатки з великими обсягами обчислень.

Однак не всі задачі доцільно розв'язувати за допомогою GRID. Суперкомп'ютери до сих пір незамінні для вирішення наукових проблем, які пов'язані з необхідністю організації постійної взаємодії між окремими процесорами, як, наприклад, прогнозування погоди. Очевидним є те, що таку взаємодію неможливо забезпечити в умовах географічно розподілених та апаратно-неоднорідних ресурсів GRID-середовища. Іншими словами, GRID не є оптимальним рішенням для реалізації паралельних обчислень з інтенсивним міжпроцесорним обміном.

Сфера застосування GRID-технологій не обмежується лише вирішенням складних наукових та інженерних задач. Із розвитком GRID

проникає в промисловість і бізнес. GRID-технології вже активно застосовуються у світі як державними організаціями управління, оборони, сфери комунальних послуг, так і приватними компаніями, наприклад, фінансовими і енергетичними, зокрема й в Україні.

Однією з головних переваг GRID є те, що при всьому розмаїтті архітектур, будь-яка обчислювальна система може розглядатися як потенційний обчислювальний ресурс GRID-системи. Необхідною умовою для цього є наявність проміжного програмного забезпечення (далі ППЗ), що реалізує стандартний зовнішній інтерфейс з ресурсом і дозволяє зробити ресурс доступним для GRID-системи.

ППЗ для GRID являє собою взаємоузгоджений набір програмних засобів, який з досить великою повнотою і універсальністю підтримує розгортання та обслуговування GRID, містить засоби розробки систем для GRID і мінімальний набір необхідних служб, а також організовує функціонування GRID як єдиного операційного середовища.

Нині існує багато реалізацій проміжного програмного забезпечення, які продовжують розроблятися як в рамках різних науково-дослідних проектів, так і великими IT компаніями, такими як Sub, IBM, Intel, Sun та ін., причому більшість з цих реалізацій несумісні між собою. Очевидно, що така ситуація негативно позначається на перенесенні GRID-додатків (які розробляються під конкретну GRID-платформу) і встановлює розробникам та кінцевим користувачам досить жорсткі рамки.

Через це одним з найбільш актуальних питань щодо подальшого розвитку GRID-систем стає пошук рішення, яке б дозволило об'єднати функціональні можливості різних реалізацій ППЗ і забезпечувало б сумісність між ними.

1.2 Концепція «волонтерських обчислень»

Волонтерські обчислення (volunteer computing) – це домовленість, згідно з якою власники комп’ютерів (*волонтери*) надають свої обчислювальні ресурси *проектам*, які використовують їх для проведення розподілених обчислень та/або зберігання даних.

Волонтерами зазвичай стають звичайні люди, що мають персональний комп’ютер з доступом до мережі Internet. Однак, організації на кшталт університетів та фірм також можуть надавати доступ до своїх комп’ютерів. Проекти є, як правило, академічними (на базі університетів) та проводять наукові дослідження, хоча існують і винятки.

Актуальність та доцільність таких обчислень зумовлена дуже великою кількістю персональних комп’ютерів в сучасному світі (більше мільярда), що теоретично дозволяє отримати більше обчислювальних ресурсів для потреб науки ніж при застосуванні будь-якого іншого підходу. Це стає можливим за рахунок того, що сучасні комп’ютери використовують, в середньому, не більше 15% від їх максимальної обчислювальної потужності, що й створює велику кількість вільних

ресурсів. Більш того, переваги від застосування волонтерських обчислень постійно збільшуються, оскільки збільшується як кількість комп'ютерів, так і їх потужність.

Окрім того, перевагою волонтерських обчислень є те, що обчислювальні ресурси в них не купуються, а залучаються на добровільній основі.

Це дозволяє отримати необхідні ресурси науковим проектам, які мають широку громадську підтримку, але обмежене фінансування, що є особливо актуальним, зважаючи на дуже високу вартість оренди або купівлі сучасних суперкомп'ютерів.

Концепція волонтерських обчислень дозволяє підвищити інтерес громадськості до науки і її розвитку, а також дає можливість людям безпосередньо підтримати проекти, які вони вважають необхідними та актуальними.

Волонтерські обчислення мають певні характерні риси, які відрізняють їх від інших видів розподілених обчислень, таких як GRID.

Так, волонтерство є фактично анонімним. Хоча для участі в деяких проектах і потрібно створити акаунт, вказавши певну інформацію, він не є безпосередньо пов'язаним з конкретною людиною. Через фактичну анонімність волонтери не є підзвітними відносно проекту. Якщо волонтер почне заважати роботі проекту (наприклад, навмисно відправляти некоректні результати обчислень), то керівництво проекту не зможе вжити суттєвих штрафних або дисциплінарних заходів до нього.

Проте й волонтери вимушені довіряти проекту в багатьох аспектах:

– в тому, що програмне забезпечення проекту не завдасть шкоди їхнім комп'ютерам та не буде збирати приватні дані,

– в тому, що проект дійсно працює над заявленими задачами і не порушує при цьому права на інтелектуальну власність,

– в тому, що проект є належним чином захищеним від втручання ззовні і хакери не зможуть використати його для своїх цілей.

Такі відносини між учасниками обчислень значно відрізняються від прийнятих при організації інших типів розподілених обчислень.

Так, при реалізації GRID-обчислень ресурси використовують в межах віртуальної організації відповідно до погоджених правил. Ці правила регулюють доступ до всіх типів засобів, включно й комп'ютерні системи, програмне забезпечення, а також дані. Через це, у випадку порушення правил користування одним з учасників, інші учасники можуть вжити адекватних заходів, навіть й припинити надання ресурсів порушнику.

Також варто відзначити, що кожен учасник ВО може бути як постачальником, так і споживачем обчислювальних ресурсів, що теж є характерною особливістю GRID-систем.

1.3 Проект Folding@Home

1.3.1 Загальні характеристики проекту Folding@Home

Одним з найбільш успішних прикладів реалізації «волонтерських обчислень» є проект Folding@Home. Він спрямований на дослідження хвороб шляхом моделювання згортання білків з метою виявлення можливих проблем згортання, які призводять до хвороб Альцгеймера, Паркінсона, діабету типу II, коров'ячого сказу, склерозу та деяких типів раку.

Окрім того, проект також займається прогнозуванням кінцевої структури білків та визначенням того, як інші молекули взаємодіють з ними, що знаходить застосування при розробці нових ліків. Folding@Home був створений та працює при Стенфордському університеті, США й активно співпрацює з різноманітними науковими організаціями та дослідницькими лабораторіями по всьому світу.

Проект працює на основі використання незадіяних обчислювальних потужностей тисяч персональних комп'ютерів, які надаються їхніми власниками шляхом встановлення спеціального програмного забезпечення.

Як частина мережної архітектури «клієнт-сервер» кожен окремий комп'ютер отримує свій фрагмент симуляції (робочу одиницю), виконує її та повертає на сервера баз даних проекту, де вони об'єднуються в загальну симуляцію. Волонтери можуть відслідковувати свій внесок в обчислення на сайті проекту, що надає участі у проекті спортивного відтінку і підтримує зацікавленість учасників.

Folding@Home є одною з найбільш швидких обчислювальних систем в світі з приблизною швидкістю обчислень близько 100 петафлопс. Така обчислювальна потужність дозволяє моделювати в тисячі разів довші послідовності згортання білків, ніж це було можливо раніше. З моменту запуску проекту 1 жовтня 2000 року на основі отриманих результатів було опубліковано понад 120 наукових праць. Причому варто відзначити, що результати моделювання узгоджуються з результатами прикладних експериментів.

Такої результативності було досягнуто за рахунок інноваційного використання графічних процесорів (GPU), PlayStation 3s, MPI (для обчислень за допомогою багатоядерних процесорів) та деяких смартфонів Sony Xperia для проведення розподілених обчислень.

Особливо варто відзначити використання технології GPGPU (general purpose GPU) – застосування графічних процесорів для проведення неграфічних обчислень. GPU наразі є однією з найбільш потужних обчислювальних платформ і має значні перспективи розвитку, однак значними перепонами для її широкого використання є висока складність застосування для виконання неграфічних обчислень і часто виникаюча необхідність значної модифікації алгоритмів. Однак, за умов належного використання GPU дозволяє виконувати надзвичайно велику кількість операцій з плаваючою комою на секунду (FLOPS). Так, модифікація

алгоритму моделювання динаміки молекул під використання GPGPU дозволила проекту Folding@Home досягти підвищення швидкодії в середньому в 30–40 разів порівняно зі звичайними алгоритмами на основі CPU. Наразі графічні процесори забезпечують більше половини від загальної обчислювальної потужності проекту.

Також значний внесок у розвиток проекту внесло повноцінне використання можливостей багатоядерних процесорів. Так, чотирьохядерний процесор може виконати фрагмент симуляції майже в чотири рази швидше, ніж одноядерний. Наразі проект Folding@Home є орієнтований на роботу з архітектурою SMP (*Symmetric Multiprocessing*) як найбільш поширеною серед багатопроцесорних систем.

За означенням, SMP є архітектурою багатопроцесорних комп'ютерів, в якій два або більше однакових процесори підключаються до загальної пам'яті і виконують одні й ті самі функції. Саме загальна фізична пам'ять і є головною особливість цієї архітектури. Перевагами SMP-систем є простота та універсальність при програмуванні, простота експлуатації та відносна дешевизна. До недоліків варто віднести погану масштабованість у зв'язку з обмеженістю можливостей системної шини. Відповідно до класифікації Флінна SMP-системи відносяться до класу SM-MIMD.

1.3.2 Класифікація Флінна

Класифікація (таксономія) Флінна є однією з перших та найбільш відомих класифікацій архітектур обчислювальних систем. Вона була запропонована Майклом Флінном в 1966 р. В основу класифікації покладено поняття потоку. Потік – це послідовність, під якою розуміють послідовність даних або команд, що обробляються процесором. Розглядаючи кількість потоків команд та потоків даних, Флінн запропонував такі класи архітектур: MIMD, SIMD, SISD та MISD (рис. 2).

	Single Data Stream	Multiple Data Stream
Single Instruction Stream	SISD Single Instruction Single Data	SIMD Single Instruction Multiple Data
Multiple Instruction Stream	MISD Multiple Instruction Single Data	MIMD Multiple Instruction Multiple Data

Рисунок 2 – Класифікація Флінна

Архітектура *SISD* – це традиційний комп’ютер фон-Неймановської архітектури з одним процесором, який послідовно виконує одну інструкцію за іншою, працюючи з одним потоком даних. В такому класі фактично відсутній паралелізм.

Архітектура *SIMD* – це паралельна комп’ютерна система, в якій декілька процесорів виконують одну й ту саму інструкцію над декількома потоками даних одночасно. SIMD-системи поділяють на два підтипи:

– SM-SIMD (shared memory SIMD). Їх характерною рисою є спільна пам'ять, до якої має доступ кожен з процесорів. Прикладами таких систем є векторні процесори;

– DM-SIMD (distributed memory SIMD). Їх характерною рисою є розподілена пам'ять, окрема для кожного з процесорів. Прикладами таких систем є так звані матричні процесори.

Архітектура *MISD* – це архітектура, яка передбачає подання одного потоку даних на набір процесорів, кожен з яких виконує свою інструкцію щодо їх обробки. До цього класу найчастіше відносять конвеєрні ЄОМ, хоча такий підхід не є загальноприйнятим. Ряд експертів стверджує, що така архітектура ще не була повноцінно реалізована.

Архітектура *MIMD* – це паралельна комп’ютерна система, в якій декілька процесорів незалежно один від одного виконують декілька різних наборів команд, що обробляють різні набори даних. До класу MIMD відносять мультипроцесорні машини, багатоядерні та багатопоточні процесори, а також комп’ютерні кластери. За принципами роботи з пам’яттю MIMD-системи поділяють на два підтипи:

– SM-MIMD (shared memory MIMD). Їх характерною рисою є спільна пам'ять, до якої має доступ кожен з процесорів. Прикладами таких систем є мультипроцесорні машини та багатоядерні процесори з загальною пам’яттю. Головною їх перевагою є зручність роботи та підтримки, простота програмування. Головним їх недоліком є низька масштабованість;

– DM-MIMD (distributed memory MIMD). Їх характерною рисою є розподілена пам'ять, окрема для кожного з процесорів. Якщо процесору потрібні дані з пам’яті іншого процесора, для цього використовуються сторонні засоби на основі PVM та MPI. Прикладами таких систем є багатопроцесорні машини з розподіленою пам’яттю. Головною перевагою систем такого плану є висока масштабованість.

1.3.3 Налаштування та додаткові можливості Folding@Home

Для того, щоб взяти участь у проекті Folding@Home, спочатку потрібно встановити спеціальне програмне забезпечення, яке можливо завантажити на сайті проекту (http://folding.stanford.edu/). Процедура встановлення є тривіальною. Експрес-встановлення не потребує від користувача ніяких дій, окрім згоди з ліцензійною угодою користувача. Розширене встановлення також дозволяє самостійно обрати місце

встановлення (за замовчуванням – диск С), параметри запуску клієнта (при вмиканні комп'ютера, після входу у систему або вручну) та обліковий запис Windows, для якого будуть дійсні обрані налаштування.

В будь-якому випадку після встановлення потрібно запустити клієнта. Він відкриється у фоновому режимі, і одночасно в браузері відкриється сторінка (рис. 3), що дозволяє в зручному вигляді керувати роботою клієнта та відслідковувати прогрес і результати роботи.

Брати участь у проекті можливо як анонімно, так і вказавши ім'я користувача та номер команди, до якої є намір приєднатися. Номер найбільшої української команди – 2164.

Клієнт потребує підключення до Інтернету для отримання завдань, відправлення результатів та для налаштування і моніторингу роботи програми за допомогою веб-сторінки (рис. 3). В решті випадків підключення не потрібне. Веб-сторінка, що відкривається після запуску клієнта, дозволяє змінити ім'я користувача та групу. Окрім того, вона дозволяє обрати режим використання обчислюваних потужностей системи (power):

– Light. Клієнт буде використовувати лише 50% від потужності CPU. GPU буде використовуватися лише в період простою;

– Medium. Клієнт буде використовувати 80% від потужності CPU. Повноцінне використання GPU;

– Full. Клієнт буде використовувати всі наявні ресурси CPU та GPU.

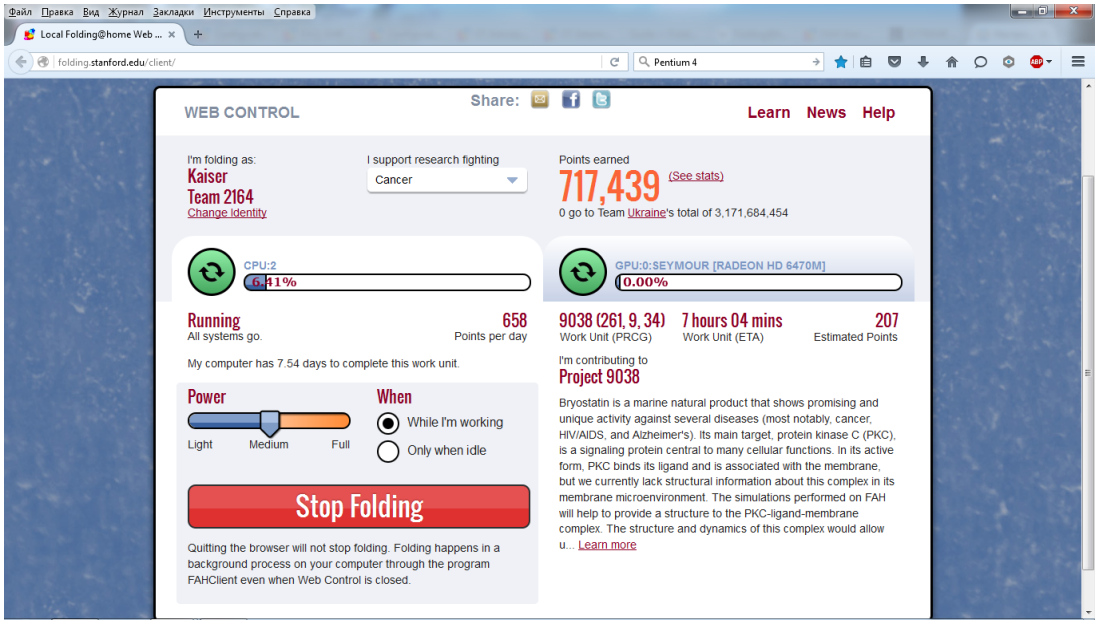


Рисунок 3 – Налаштування роботи клієнта через веб-сторінку

Також можливо визначити, коли буде працювати клієнт (меню «when»):

увесь час (варіант «when I'm working») або лише під час простою («Only when idle»). Всі ці налаштування, а також ряд додаткових можливо

виконати й безпосередньо в клієнті. Для цього потрібно натиснути на піктограму клієнта в області повідомлень панелі задач Windows та обрати пункт «Advanced Control», що відкриє меню налаштування (рис. 4).

Час виконання одного завдання займає, в середньому, від одного до декількох днів залежно від потужності системи. Мінімальні системні вимоги – процесор серії Pentium 4, оскільки більш старі моделі просто не зможуть виконати завдання вчасно навіть при максимальному завантаженні в режимі 24/7.

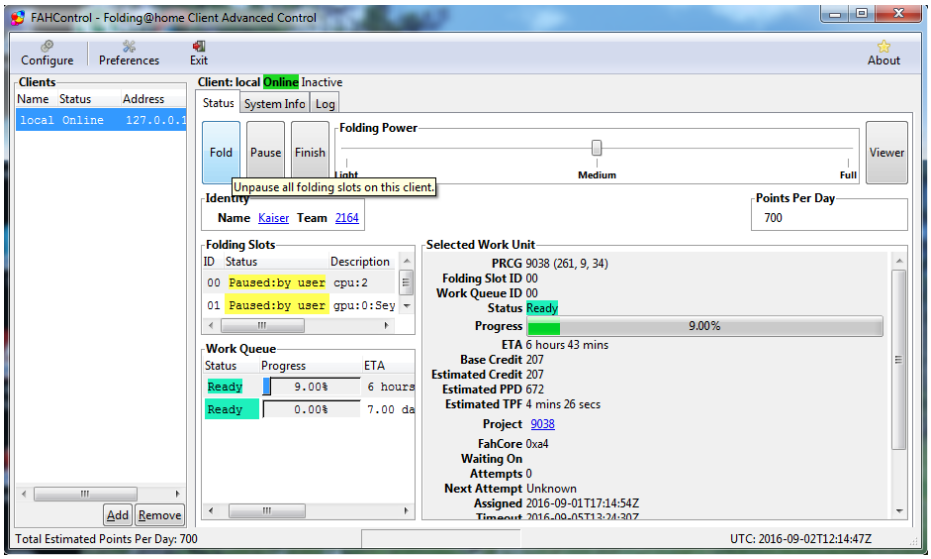


Рисунок 4 – Меню налаштування клієнта Folding@Home

2 Практична частина

2.1 Правила та рекомендації щодо виконання лабораторної роботи

Процес виконання лабораторної роботи передбачає ознайомлення з теоретичним матеріалом, виконання практичного завдання та оформлення звіту. Структура звіту має бути такою:

1. Титульний лист.
2. Мета роботи.
3. Короткі теоретичні відомості.
4. Покроковий алгоритм виконання ходу роботи.
5. Отримані результати.
7. Висновки.
8. Додатки.

Оцінка за лабораторну роботу є комплексною і визначається на основі результатів практичного завдання та усної співбесіди, яка проводиться у форматі «питання-відповідь».

Практичні завдання наведено у наступному підрозділі. Їх виконання є обов’язковим для захисту роботи. *Окрім того, отримані результати мають бути відображені в звіті.*

Перелік тем, питання з яких будуть використовуватися під час усної співбесіди, наведено у підрозділі 2.3.

Результату «задовільно» відповідають теми базового рівня, результату «добре» відповідають теми базового та розширеного рівня, результату «відмінно» відповідають теми базового, розширеного та поглибленого рівня.

Окрім того, в розділі «Перелік додаткових джерел» наведено список літератури, яка є корисною для підготовки до захисту та для більш глибокого ознайомлення з темою лабораторної роботи.

2.2 Практичне завдання

1. Дослідити сучасний стан розвитку української Grid-інфраструктури (наприклад, проект Ukrainian National Grid (Ugrid)).
2. Завантажити та встановити клієнт Folding@Home. Взяти участь у роботі проекту. Результати подати у вигляді скриншотів.

2.3 Перелік тем

Базовий рівень. Визначення GRID та ППЗ. Головні задачі GRID. Визначення волонтерських обчислень. Проект Folding@Home: мета та принцип роботи. Класифікація Флінна: головні класи.

Розширений рівень. Сучасний стан розвитку ППЗ. SMP-системи: визначення та місце в класифікації Флінна. Технологія GPGPU: означення і характерні особливості. Можливості налаштування клієнта Folding@Home.

Поглиблений рівень. Задачі GRID та суперкомп'ютерів: схожість та відмінності. Переваги та недоліки волонтерських обчислень. Характерні особливості систем з загальною (shared) та розподіленою (distributed) пам'яттю. Принципові відмінності між волонтерськими обчисленнями та GRID-обчисленнями.

2.4 Перелік додаткових джерел

1. Демичев А. П. Введение в грид-технологии / А. П. Демичев, В. А. Ильин, А. П. Крюков. – М. : НИИЯФ МГУ, 2007. – 87 с. (Робота містить велику кількість інформації про GRID-системи та є рекомендується до ознайомлення)
2. Ukrainian National Grid [Електронний ресурс] – Режим доступу: http://www.grid.ntu-kpi.kiev.ua. (Офіційний сайт проекту Ukrainian National Grid (Ugrid))
3. Folding@Home [Електронний ресурс] – Режим доступу: http://folding.stanford.edu/ (Офіційний сайт проекту Folding@Home)
4. Ukraine – Distributed Computing Team [Електронний ресурс] – Режим доступу: https://distributed.org.ua/index.php?newlang=ua (Сайт, присвячений розподіленим обчислення в Україні та світі)

Лабораторна робота № 2

Тема роботи: програмна платформа BOINC
Мета роботи: ознайомлення з організацією розподілених обчислень на основі програмної платформи BOINC.

1 Теоретична частина

1.1 Загальна характеристика програмної платформи BOINC

BOINC (Berkeley Open Infrastructure for Network Computing – відкрита інфраструктура університету Берклі для мережних обчислень) можливо визначити як відкрите проміжне програмне забезпечення для організації GRID та волонтерських обчислень. Спочатку BOINC розроблявся для підтримки проекту SETI@home, але потім був розширений до програмної платформи для реалізації проектів розподілених обчислень у широкому спектрі галузей: математика, медицина, лінгвістика, молекулярна біологія, кліматологія тощо. Станом на початок 2017 року BOINC загалом об'єднує понад 300 000 активних учасників та понад 800 000 комп'ютерів по всьому світу, що дозволяє досягти середньої обчислювальної потужності у 19 петафлопс.

В загальному випадку BOINC-обчислення мають такий вигляд:

- Комп'ютери учасників певного проекту отримують набір завдань з сервера планування (scheduling server) проекту. Обчислювана складність завдань, які отримує кожен окремий комп'ютер, залежить від його потужності та налаштування клієнта BOINC.
- Комп'ютери учасників завантажують (як правило, автоматично) потрібні для роботи файли з сервера даних (data server) проекту.
- Комп'ютери учасників проводять необхідні обчислення та отримують кінцеві результати.
- Результати роботи завантажуються назад на сервер даних проекту.
- Комп'ютери учасників повідомляють сервер планування проекту про виконання поточного завдання та отримують від нього нове.

За виконання завдань учасники отримують спеціальні бали, які дозволяють відслідковувати їхній внесок у проведення обчислень. Окрім того, користувачі можуть об'єднуватися в групи з тією самою метою.

Для попередження зловживань та зменшення вірогідності технічних помилок одне й те саме завдання зазвичай відсилається на два або більше комп'ютерів і лише якщо отримані результати збігатимуться, користувачі отримують свої бали, а результати будуть використані в подальшій роботі.

З програмної точки зору BOINC складається з декількох підпрограм (рис. 1). Програми для керування серверами (shedulers – сервера планування, data servers – сервера даних) встановлюються й обслуговуються безпосередньо на комп'ютерах, які належать організаторам проекту. На комп'ютерах учасників встановлюються такі програми:

- Ядро клієнта (core client) взаємодіє з зовнішніми серверами за допомогою протоколу HTTP для отримання та відправлення завдань. Ядро клієнта викликає та контролює усі інші додатки.
- Додатки (applications) – це програми, які безпосередньо виконують потрібні наукові обчислення.
- GUI (BOINC manager) реалізує графічний інтерфейс користувача, який дозволяє контролювати ядро клієнта. GUI взаємодіє з ядром клієнта за допомогою протоколу TCP.
- Скрінсейвер (screensaver) реалізовує заставку, яка може містити інформацію про поточний стан BOINC та активується в режимі очікування комп'ютера. Ця програма наявна не в усіх проектах.

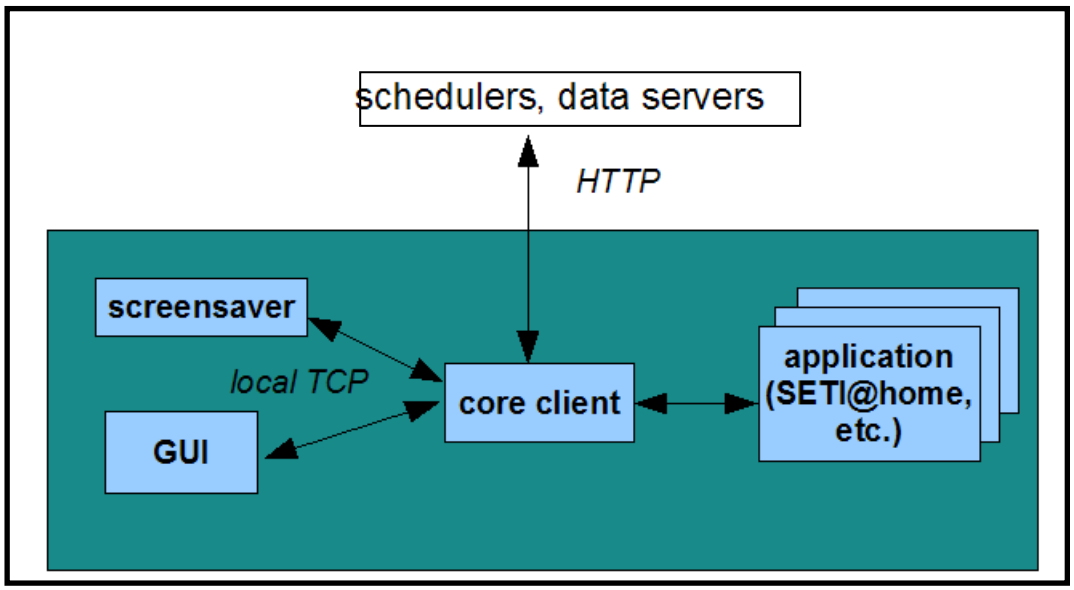


Рисунок 1 – Програмна структура BOINC

Клієнт BOINC підтримує використання технологій багатопотокових та гетерогенних обчислень (GPGPU). Застосування GPGPU-технологій (OpenCL, NVIDIA CUDA тощо) дозволяє в багатьох випадках отримати приріст швидкодії в 10 – 200 разів порівняно з послідовною реалізацією на основі CPU. Проте модифікація існуючих додатків під використання цих технологій є складною та трудомісткою задачею, реалізація якої безпосередньо залежить від організаторів кожного окремого проекту. Окрім того, не усі задачі та алгоритми можуть бути ефективно розпаралелені. Тому наразі лише частина проектів на основі BOINC підтримує гетерогенні та багатопотокові обчислення.

Значною перевагою BOINC є можливість виконання на різних програмно-апаратних платформах. Так, BOINC підтримує операційні системи Windows (починаючи з 2000 SP5), Linux, MAC та Android. У випадку платформи Android BOINC виконує обчислення лише за умови підключення пристрою до джерела живлення. Передавання ж даних проводиться лише за наявності доступу до мережі Wi-Fi. Варто відзначити, що в загальному випадку BOINC потребує підключення до мережі Інтернет лише для отримання завдань та відправлення результатів роботи.

1.2 Проекти на основі BOINC

Наразі існує близько 50 активних проектів на основі програмної платформи BOINC. Найбільш відомими з них є проекти SETI@home та Rosetta@home.

SETI@home. SETI (Search for Extraterrestrial Intelligence – пошук позаземного розуму) – це науково-практичний напрям, метою якого є пошук позаземних розумних форм життя. На першому етапі (radio SETI) використовуються радіотелескопи для спостереження за вузькополосними радіосигналами з космосу. Природні джерела таких сигналів не є відомими, тому їх виявлення могло б бути підтвердженням існування позаземних цивілізацій.

Попередні проекти radio SETI використовували спеціалізовані суперкомп'ютери для обробки головного масиву даних з телескопів. В 1995 р. було запропоновано проводити обробку даних на віртуальному суперкомп'ютері, який би складався з багатьох окремих комп'ютерів, підключених до мережі Інтернет. Так в 1999 р. був запущений проект SETI@home (SETI вдома). Проект мав на меті дві головні цілі:

– аналіз результатів спостережень, отриманих за допомогою радіотелескопів, з метою пошуку розумних позаземних форм життя (рис. 2);

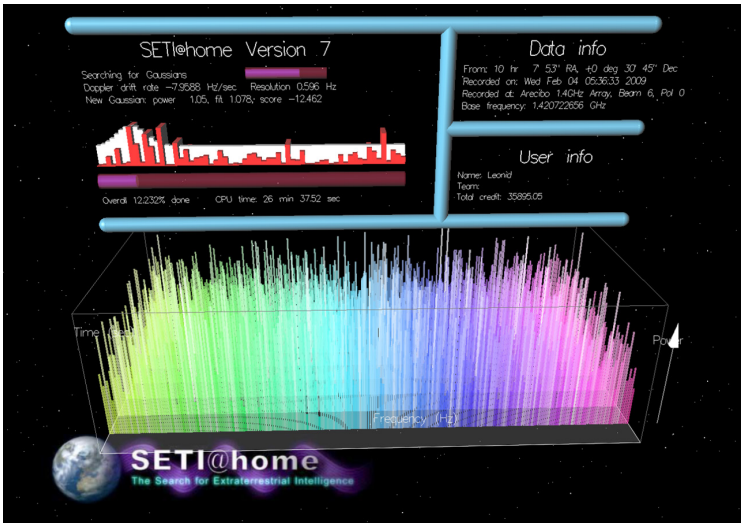


Рисунок 2 – Схематичне зображення процесу аналізу радіосигналу

– доведення доцільності та перспективності концепції «волонтерських обчислень».

Другу мету можна вважати досягнутою, оскільки на основі SETI@home було розроблено програмну платформу для організації розподілених обчислень BOINC, на основі якої наразі функціонує велика кількість проектів. І хоча під час досліджень не було отримано доказів існування позаземних форм життя, проект SETI@home продовжує свою роботу, базуючись на тому, що наразі досліджено лише дуже малу частину усього небесного простору і тому подальший аналіз результатів спостережень є доцільним.

Rosetta@home. Rosetta@home – це проект, спрямований на вирішення однієї зі значних задач молекулярної біології, а саме знаходження тривимірної структури білків на основі їх амінокислотних послідовностей (рис. 3). Функції білків та їх взаємодія є критичними для хімічної та біологічної структури й процесів усіх живих організмів. Функції білка і те, як він взаємодіє з іншими молекулами, значною мірою визначаються його формою (тривимірною структурою). Білки спочатку синтезуються як довгі ланцюги амінокислот, але вони зазвичай не можуть нормально функціонувати, поки не згорнуться у заплутані шароподібні структури. Розуміння та передбачення правил, які керують процесом згортання, є однією з центральних проблем біології. Знаючи, як білки згортаються та взаємодіють з іншими молекулами, та визначивши їх функції, можливо прийти до відкриття нових препаратів та лікування людських хвороб, таких як рак, хвороба Альцгеймера, малярія та інші.

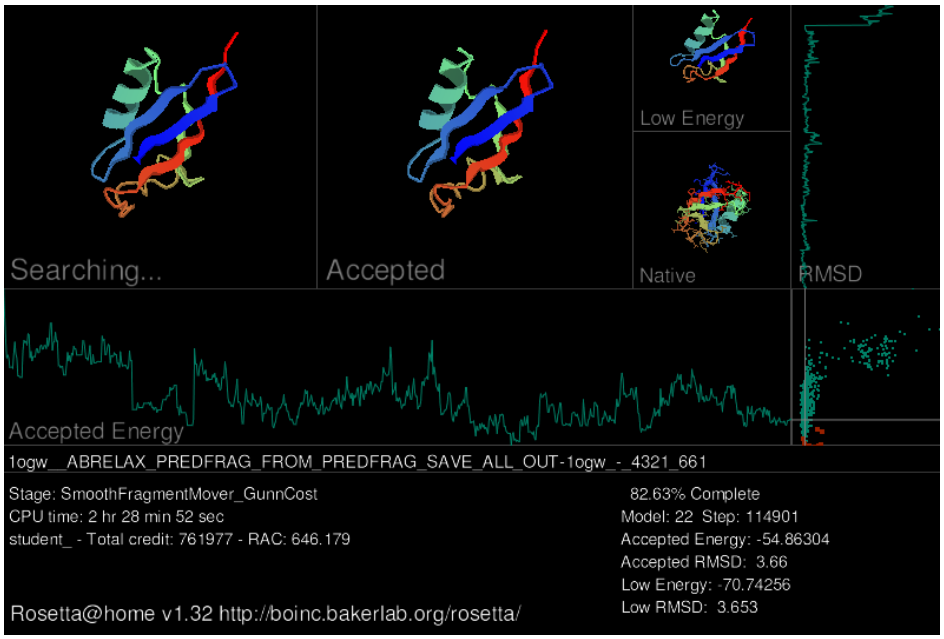


Рисунок 3 – Схематичне зображення процесу моделювання структури білка

1.3 Встановлення та налаштування BOINC

Для того, щоб взяти участь в обчисленнях, достатньо лише встановити клієнт BOINC та обрати потрібний проект. Клієнт може бути безкоштовно завантажений з офіційного сайту проекту BOINC (https://boinc.berkeley.edu/). Для завантаження доступні дві версії клієнта – BOINC та BOINC + VirtualBox. Друга версія, окрім звичайного функціонала, містить програмне забезпечення для створення віртуальних систем, що дозволяє працювати на Windows над проектами, які потребують для свого виконання специфічних операційних систем, наприклад Linux.

Процес встановлення клієнта є тривіальним. За бажанням користувач може обрати місце встановлення клієнта, місце зберігання результатів роботи, користувачів системи, які можуть змінювати налаштування BOINC (за замовчуванням усі користувачі, в іншому випадку лише адміністратор та користувач, який встановив програму) та, за бажанням, встановити клієнт у режимі «сервісу», в якому BOINC буде викликатися у фоновому режимі відразу після запуску системи та матиме доступ лише до своїх файлів, що підвищує безпечність роботи. Проте в цьому режими неможливе використання GPU для проведення обчислень.

Наступним кроком після встановлення програми є запуск клієнта BOINC Manager, який дозволяє керувати роботою проектів. Для того, щоб приєднатися до нового проекту, потрібно натиснути кнопку «додати проект» та обрати потрібний проект зі списку (рис. 4). Потім потрібно вказати свою електронну пошту та пароль (окремо для кожного проекту). Після цього відкриється Інтернет-сторінка обраного проекту, на якій можливо буде вказати або змінити свої персональні дані (ім'я, країну тощо) та за бажанням вступити в існуючу групу або створити свою власну.

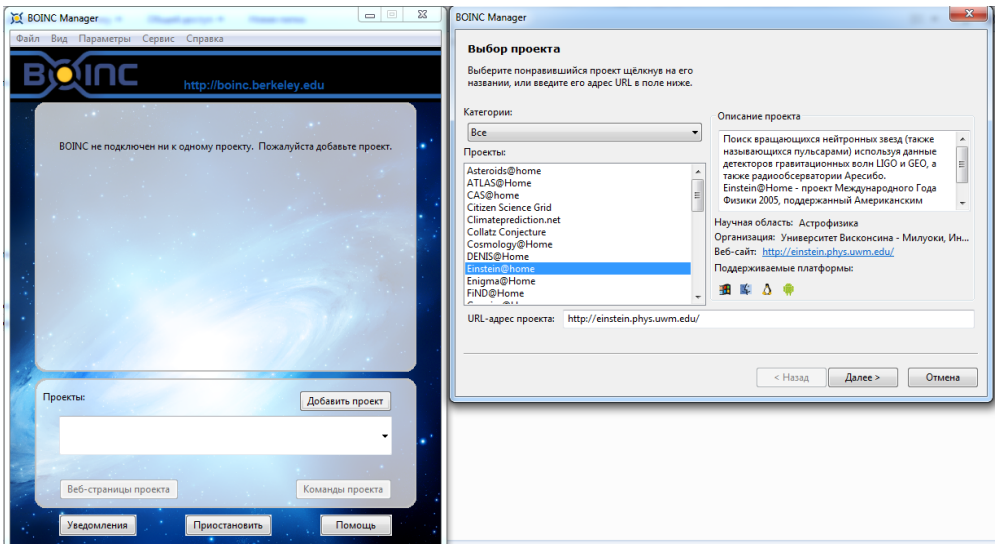


Рисунок 4 – Підключення до нового проекту обчислень

Одночасно можливо брати участь у декількох проектах. Для цього достатньо повторно натиснути на кнопку «додати проект», обрати потрібний проект та повторити вищенаведені дії. Варто відзначити, що розробники платформи BOINC не мають прямого контролю над створенням проектів на основі BOINC та в цілому не відповідають за їх перевірку та схвалення. Тому перед тим, як приєднатися до певного проекту, є доцільним отримати додаткову інформацію про нього. Зокрема, важливими є такі питання:

– чи викликає довіру сам проект та його розробники;

– чи мають сервери проекту належний захист від втручання;

– чи описані цілі проекту належним чином;

– кому будуть належати отримані результати обчислень?

Відповідей на ці питання зазвичай достатньо для того, щоб скласти своє враження про проект та прийняти рішення щодо участі у ньому.

Контролювати роботу клієнта BOINC можливо в двох режимах: спрощеному (рис. 4) та розширеному (рис. 5). Спрощений режим дозволяє додавати нові проекти, призупиняти та переривати їх роботу, проводити базове налаштування роботи клієнта. До базових налаштувань відносяться такі можливості: призупинення клієнта під час виконання іншої роботи на комп'ютері, встановлення періодів часу для проведення обчислень та відправлення результатів, встановлення обмежень на максимальний розмір файлів, вибір мови інтерфейсу, налаштування підключення до Інтернет тощо.

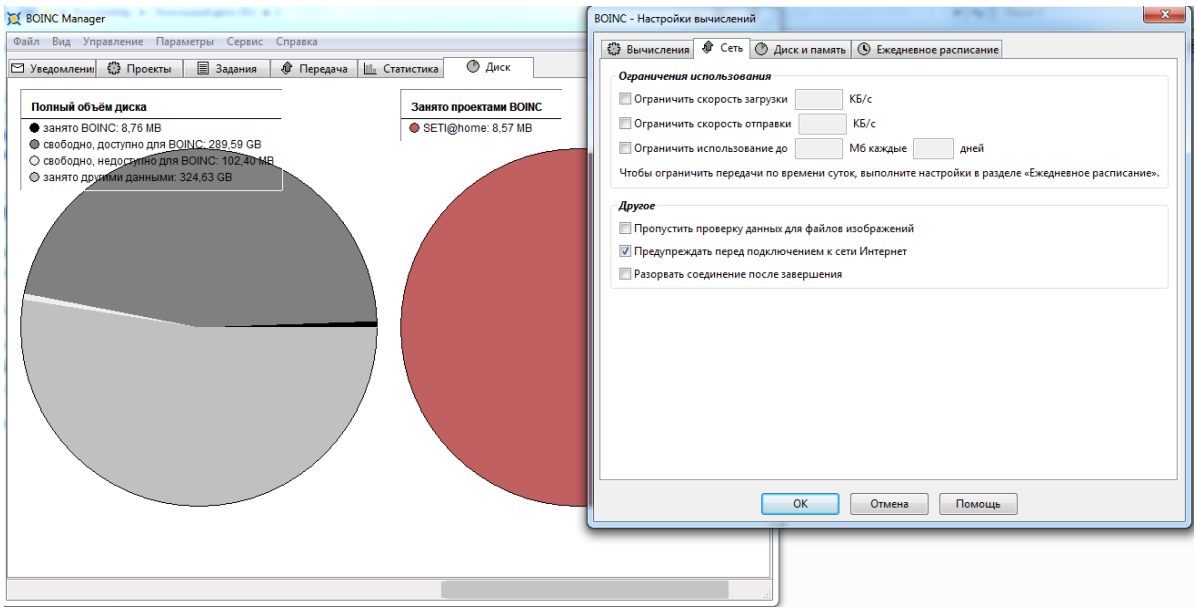


Рисунок 5 – Розширена версія BOINC Manager

Розширений режим містить в собі увесь функціонал спрощеного режиму та додає ряд додаткових можливостей: відслідковування прогресу у виконанні окремих завдань; відслідковування місця на жорсткому диску,

яке займають проекти BOINC; створення детального розкладу проведення обчислень; детальне налаштування отримання завдань та відправлення результатів через Інтернет (обмеження на швидкість завантаження/відправлення, максимальний обсяг трафіка тощо), налаштування обмежень на максимальне завантаження CPU та інші. Змінити режим можливо у вкладці «вигляд» клієнта BOINC.

Окрім того, BOINC можливо налаштовувати шляхом внесення змін у файл конфігурації. Як правило, це потрібно для більш індивідуального налаштування використання ресурсів або при застосуванні спеціальних технологій обчислень, таких як GPGPU. Так, в проектах, що підтримують GPGPU, за замовчуванням графічна карта використовується лише в режимі очікування. Окрім того, якщо графічних карт декілька, то для обчислень автоматично обирається найпотужніша з них. Для того, щоб змінити ці параметри, потрібно вручну внести зміни до файлу конфігурації. Більш детальна інформація про таке налаштування знаходиться на сайті проекту BOINC.

Практична частина

2.1 Правила та рекомендації до виконання лабораторної роботи

Процес виконання лабораторної роботи передбачає ознайомлення з теоретичним матеріалом, виконання практичного завдання та оформлення звіту. Структура звіту така:

1. Титульний лист.
2. Мета роботи.
3. Короткі теоретичні відомості.
4. Покроковий алгоритм виконання ходу роботи.
5. Отримані результати.
7. Висновки.
8. Додатки.

Оцінка за лабораторну роботу є комплексною і визначається на основі результатів практичного завдання та усної співбесіди, яка проводиться у форматі «питання–відповідь».

Практичні завдання наведено у наступному підрозділі. Їх виконання є обов’язковим для захисту роботи. *Окрім того, отримані результати мають бути відображені в звіті.*

Перелік тем, питання до яких будуть використовуватися під час усної співбесіди, наведено у підрозділі 2.3. Результату «задовільно» відповідають теми базового рівня, результату «добре» відповідають теми базового та розширеного рівня, результату «відмінно» відповідають теми базового, розширеного та поглибленого рівня.

Окрім того, в розділі «Перелік додаткових джерел» наведено список літератури, яка є корисною для підготовки до захисту та для більш глибокого ознайомлення з темою лабораторної роботи.

2.2 Практичне завдання

1. Завантажити та встановити клієнт BOINC. Ознайомитися з двома проектами на основі BOINC (на свій вибір) та взяти в них участь. Результати навести у вигляді скріншотів.

2.3 Перелік тем

Базовий рівень: Визначення проекту BOINC. Принцип роботи проекту BOINC. Клієнт BOINC (BOINC Manager).

Розширений рівень: Програмна структура проекту BOINC. Проекти на основі BOINC. Можливості налаштування BOINC Manager.

Поглиблений рівень: Принципові відмінності між проектами BOINC та Folding@Home. Співвідношення GRID-обчислень та обчислень на основі BOINC. Застосування технології GPGPU у проектах на основі BOINC.

2.4 Перелік додаткових джерел

1. BOINC [Електронний ресурс] – Режим доступу: https://boinc.berkeley.edu/ (Офіційний сайт проекту BOINC)

2. SETI@home [Електронний ресурс] – Режим доступу: https://setiathome.berkeley.edu/ (Офіційний сайт проекту SETI@home)

3. Rosetta@home [Електронний ресурс] – Режим доступу: http://boinc.bakerlab.org/rosetta/ (Офіційний сайт проекту Rosetta@home)

4. BOINC stats [Електронний ресурс] – Режим доступу: https://boincstats.com/en/stats/projectStatsInfo (Перелік проектів на основі BOINC)

Лабораторна робота № 3

Тема роботи: технологія GPGPU: загальні визначення та принципи роботи

Мета роботи: метою цієї роботи є ознайомлення з сучасними підходами до організації паралельних обчислень та програмною моделлю GPGPU на прикладі технології NVIDIA CUDA.

1 Теоретична частина

1.1 Технологія GPGPU

GPGPU (англ. General-purpose graphics processing units – «GPU загального призначення») – технологія використання графічного процесора для проведення неграфічних розрахунків.

Графічні процесори мають велику кількість ядер (до ~2600 ядер у сучасних моделях), що дозволяє програмісту писати ефективні програми, які використовують ядра одночасно для вирішення різних частин однієї задачі. Таким чином досягається значне прискорення у часі виконання. Але не всі алгоритми і задачі можна реалізувати паралельно.

При розробці паралельних програм для CPU (англ. central processing unit – «центральний процесор») і GPU (англ. graphics processing unit — «графічний процесор») необхідно пам’ятати те, що велика кількість ядер графічного процесора потребує, щоб кожне ядро виконувало відносно невелику кількість простих математичних операцій, в той час як ядра центрального процесора створені для вирішення складних задач, але їх зазвичай значно менше. Тобто, при використанні графічного процесора досягається потужніша паралелізація.

Існує декілька шляхів і мов програмування для організації паралельних обчислень на відеоадаптері (відеокарті, графічній карті), наприклад OpenCL, DirectCompute, AMD FireStream та багато інших. В цьому курсі розглядатиметься технологія CUDA, яка з’явилася в 2007 році і працює лише з відеоадаптерами компанії NVIDIA.

Технологія CUDA (англ. Compute Unified Device Architecture) – це середовище розробки на C/C++, що дозволяє програмістам і розробникам писати програмне забезпечення для вирішення складних обчислювальних завдань за менший час завдяки багатоядерній обчислювальній потужності графічних процесорів. Тобто, графічна підсистема комп'ютера з підтримкою CUDA може бути використана як обчислювальна.

CUDA дає розробникові можливість на свій розсуд організовувати доступ до набору інструкцій графічного прискорювача (відеоадаптера) і управляти його пам'яттю, організовувати на ньому складні паралельні обчислення. Графічний процесор з підтримкою CUDA стає потужною

програмованою відкритою архітектурою подібно до сьогоднішніх центральних процесорів.

Комп'ютери, які працюють і з центральним, і з графічним процесором, називають гетерогенними. Звичайна послідовна програма зазвичай використовуєте лише центральний процесор. CUDA ж також дозволяє використати обидва процесори в одній програмі. Частина програми, написана на C/C++, буде виконуватись на центральному процесорі, якому CUDA дає ім'я ХОСТ (HOST). Інша частина програми, написана на C/C++ з розширеннями, буде паралельно виконуватись на графічному процесорі, якому CUDA дає ім'я ДЕВАЙС (DEVICE). При цьому ХОСТ є керівним у союзі двох процесорів, і кожен процесор має свою окрему фізичну пам'ять (рис. 1).

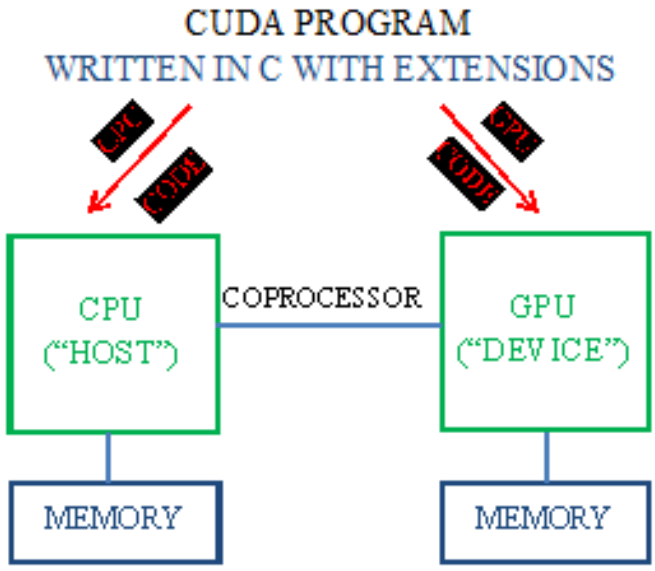


Рисунок 1 – Схема взаємодії між CPU та GPU

ХОСТ є головним, він виконує програму і надсилає команди графічному процесору. Він відповідає за таке:

1. Копіювання інформації з пам'яті CPU у пам'ять GPU;
2. Копіювання обробленої інформації з пам'яті GPU у пам'ять CPU;
3. Виділення місця у пам'яті GPU;
4. Запуск паралельних підпрограм на GPU, така підпрограма має назву – KERNEL.

З цього можна зробити висновок, що алгоритм типової GPU програми має таку структуру:

1. CPU виділяє місце в пам'яті GPU для подальшого копіювання інформації;
2. CPU копіює інформацію в пам'ять GPU;
3. CPU запускає паралельні підпрограми на GPU для обробки інформації;
4. CPU копіює оброблені дані назад із GPU.

1.2 Паралельне програмування з застосування NVIDIA CUDA

Розглянемо таку задачу і реалізацію її простого (послідовного) та паралельного вирішення. Нехай ми маємо масив чисел від 0 до 63 включно. І на виході необхідно отримати масив квадратів цих чисел (0 1 4 9…).

Звичайне вирішення може мати такий вигляд:

```
#include <stdio.h>

int main(int argc, char ** argv) {
	const int ARRAY_SIZE = 64;
	const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

	// генеруємо вхідний масив [0 1 2 3 . . .]
	float h_in[ARRAY_SIZE];
	for (int i = 0; i < ARRAY_SIZE; i++) {
		h_in[i] = float(i);
	}
	float h_out[ARRAY_SIZE];

	// розраховуємо вихідний масив квадратів
	for (int i = 0; i < ARRAY_SIZE; i++) {
	 h_out[i] = h_in[i] * h_in[i];
	}

	// виводимо результат на екран
	for (int i = 0; i < ARRAY_SIZE; i++) {
		printf("%.1f\n", h_out[i]);
	}

	return 0;
}
```

Паралельне вирішення з використанням CUDA має вигляд:

```
#include <stdio.h>

	// ця функція виконується на GPU
__global__ void square(float * d_out, float * d_in){
	int index = threadIdx.x; // індекс потока
	float f = d_in[index];
	d_out[index] = f * f;   // відповідній комірці вихідного масива присвоємо
значення квадрата числа
}

int main(int argc, char ** argv) {
	const int ARRAY_SIZE = 64;
	const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

	// генеруємо вхідний масив [0 1 2 3 . . .] на хості
	float h_in[ARRAY_SIZE];
	for (int i = 0; i < ARRAY_SIZE; i++) {
		h_in[i] = float(i);
	}
	float h_out[ARRAY_SIZE];

	// покажчики на пам’ять GPU
	float * d_in;
	float * d_out;

	// виділяємо пам’ять на GPU
	cudaMalloc((void**) &d_in, ARRAY_BYTES);
```

```
	cudaMalloc((void**) &d_out, ARRAY_BYTES);

	// копіюємо дані на GPU
	cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

	// запускаємо обробку
	square<<<1, ARRAY_SIZE>>>(d_out, d_in);

	// копіюємо результат назад на CPU
	cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

	// виводимо результат на екран і звільнюємо память на CPU
	for (int i = 0; i < ARRAY_SIZE; i++) {
		printf("%.1f\n", h_out[i]);
	}

	cudaFree(d_in);
	cudaFree(d_out);

	return 0;
}
```

Головним елементом такої програми є функція:

```
__global__ void square(float * d_out, float * d_in).
```

Це і є kernel, який виконується на GPU. Функція приймає покажчики на вхідний і вихідний масиви на GPU. Кількість параметрів і самі параметри можуть бути будь-якими. Ідентифікатор `__global__` показує компілятору, що цей код відноситься до GPU.

Функція виконується одночасно для всіх потоків, які були запущені. В заданому випадку було запущено 64 потоки, для кожної комірки масиву. Для кожного потоку в цьому випадку можна знайти його номер, використовуючи такий код: `int index = threadIdx.x;` Знаючи номер потоку, можливо визначити відповідну комірку вхідних даних, і комірку, в яку необхідно записати результат. Таким чином, отримується відповідь і копіюється назад на CPU.

Розглянемо детальніше процес створення потоку. Перша дія – це декларація покажчиків на пам’ять GPU. Програмісти CUDA часто називають змінні таким чином, щоб було зрозуміло, до якого процесора відноситься змінна: d_* – device (GPU), h_* – host (CPU). Звертання до даних, які знаходяться в пам'яті іншого процесора, може викликати збій або неправильну роботу програми.

```
float * d_in;
```

Друга дія – виділення пам’яті на GPU. Таким чином покажчики прив’язуються до конкретного місця в пам’яті GPU. Це відбувається за допомогою команди `cudaMalloc`. Вона приймає **2 аргументи** – це покажчик на пам’ять і кількість байтів, які необхідно виділити. Для 64 комірок дійсних чисел ми обрахували цю кількість раніше як `const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);`

```
cudaMalloc((void**) &d_in, ARRAY_BYTES);
```

Третя дія – копіювання інформації з пам’яті CPU у виділену пам'ять GPU. Це відбувається за допомогою функції `cudaMemcpy`. Функція повертає значення помилки у випадку її виникнення. Функція приймає **4 аргументи –** місце призначення даних, джерело даних, кількість байтів для копіювання і напрямок копіювання. Напрямок може бути: `cudaMemcpyHostToDevice (CPU -> GPU), cudaMemcpyDeviceToHost (GPU -> CPU), cudaMemcpyDeviceToDevice (GPU -> GPU).`

```
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
```

Четверта дія – виклик оператора запуску функції для виконання на GPU. Ці функції носять назву **kernels** (або ядра). Вони декларуються з використанням специфікатора **__global__.** Аргументами можуть слугувати лише змінні, які виділені в пам’яті GPU. Виклик відбувається за допомогою використання <<<…>>>. Перед дужками знаходиться назва функції, після – аргументи, які ця функція прийматиме, а в самих дужках – **конфігурація кількості блоків і потоків у кожному блоці.**

```
square<<<1, ARRAY_SIZE>>>(d_out, d_in);
```

1.3 Grid-модель.

CUDA використовує grid-модель організації потоків. Це означає, що може бути створена grid – сітка блоків, кожен з яких містить однакову кількість потоків. Блоки можуть бути організовані у одно-, дво- або тривимірній сітці. Загальна кількість блоків, що можуть бути створені, обмежена відеоадаптером. Подібно до блоків у сітці, організовуються і потоки у кожному блоці. Вони також можуть бути одно-, дво- або тривимірними.

Наступна діаграма показує сітку з двовимірною організацією блоків і двовимірною організацією потоків у кожному блоці (рис. 2).

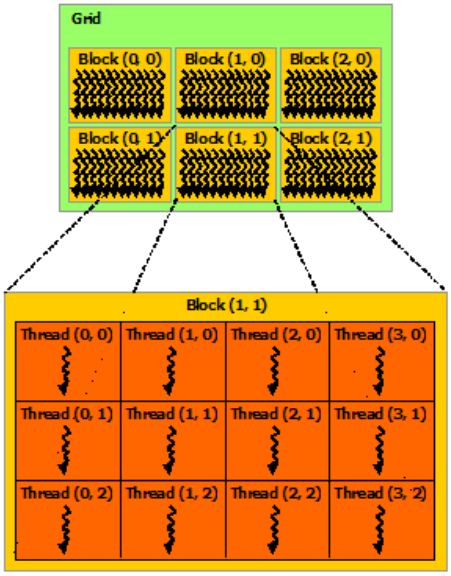


Рисунок 2 – Grid-модель організації потоків

Перший аргумент конструкції <<< >>> – це конфігурація блоків у сітці, другий – конфігурація потоків у блоках. Аргументи можуть бути типу int або dim3. В попередній задачі використовувався один блок, в якому було `ARRAY_SIZE` потоків. Для організації багатовимірних структур необхідно використовувати dim3:

```
dim3 threadsPerBlock(3, 3);  //(Dx, Dy, Dz), тут третя змінна автоматично дорівнює 1
dim3 numBlocks(2, 2);
something<<<numBlocks, threadsPerBlock>>>();
```

Це означає, що сітка (grid) має 3 блоки по x, 3 блоки по y (всього 9 блоків), а кожен блок – 2×2 потоків.

Номер потоку в блоці можна взнати за допомогою вбудованої змінної `threadIdx.` Це вектор, який має три компоненти – x, y, z. Аналогічно для блоків існує змінна `blockIdx.` `blockDim i gridDim` – аналогічні вектори, які дозволяють дізнатися встановлені розміри блоків і сітки відповідно по x, y, z безпосередньо з кернелів.

Приклад нумерації блоків і потоків та принцип знаходження загального номера потоку: for a three-dimensional block of size *(Dx, Dy, Dz)*, the thread ID of a thread of index *(x, y, z)* is *(x + y Dx + z Dx Dy)*.

Блок 4×4, вказані threadIdx.x, threadIdx.y для кожного потоку.

```
(0, 0)   (1, 0)   (2, 0)   (3, 0)

(0, 1)   (1, 1)   (2, 1)   (3, 1)

(0, 2)   (1, 2)   (2, 2)   (3, 2)

(0, 3)   (1, 3)   (2, 3)   (3, 3)
```

Знайти номер потоку всередині блока можливо таким чином:

```
threadId = threadIdx.x + (threadIdx.y * blockDim.x)
```

```
[ 0,  1,  2,  3]
[ 4,  5,  6,  7]
[ 8,  9, 10, 11]
[12, 13, 14, 15]
```

Так виглядають блоки у сітці 3×3:

```
            [0]                    [1]                    [2]


     0,0  1,0  2,0  3,0   |   0,0   1,0  2,0  3,0   |   0,0  1,0  2,0  3,0
[0]  0,1  1,1  2,1  3,1   |   0,1   1,1  2,1  3,1   |   0,1  1,1  2,1  3,1
     0,2  1,2  2,2  3,2   |   0,2   1,2  2,2  3,2   |   0,2  1,2  2,2  3,2
     0,3  1,3  2,3  3,3   |   0,3   1,3  2,3  3,3   |   0,3  1,3  2,3  3,3
                          |                        |
     ------------------------------------------------------------------------
                          |                        |
     0,0  1,0  2,0  3,0   |   0,0   1,0  2,0  3,0   |   0,0  1,0  2,0  3,0
[1]  0,1  1,1  2,1  3,1   |   0,1   1,1  2,1  3,1   |   0,1  1,1  2,1  3,1
     0,2  1,2  2,2  3,2   |   0,2   1,2  2,2  3,2   |   0,2  1,2  2,2  3,2
     0,3  1,3  2,3  3,3   |   0,3   1,3  2,3  3,3   |   0,3  1,3  2,3  3,3
                          |                        |
     ------------------------------------------------------------------------
                          |                        |
     0,0  1,0  2,0  3,0   |   0,0   1,0  2,0  3,0   |   0,0  1,0  2,0  3,0
[2]  0,1  1,1  2,1  3,1   |   0,1   1,1  2,1  3,1   |   0,1  1,1  2,1  3,1
     0,2  1,2  2,2  3,2   |   0,2   1,2  2,2  3,2   |   0,2  1,2  2,2  3,2
     0,3  1,3  2,3  3,3   |   0,3   1,3  2,3  3,3   |   0,3  1,3  2,3  3,3
```

У кожному блоці 16 потоків, всього 9 блоків, тобто потоків у сітці = 144.

Грід 3×3 з вказаними номерами потоків всередині кожного блока

```
          [0]               [1]               [2]


       [ 0,  1,  2,  3]   [ 0,  1,  2,  3]   [ 0,  1,  2,  3]
[0]    [ 4,  5,  6,  7]   [ 4,  5,  6,  7]   [ 4,  5,  6,  7]
       [ 8,  9, 10, 11]   [ 8,  9, 10, 11]   [ 8,  9, 10, 11]
       [12, 13, 14, 15]   [12, 13, 14, 15]   [12, 13, 14, 15]
```

```
       [ 0,  1,  2,  3]    [ 0,  1,  2,  3]    [ 0,  1,  2,  3]
[1]    [ 4,  5,  6,  7]    [ 4,  5,  6,  7]    [ 4,  5,  6,  7]
       [ 8,  9, 10, 11]    [ 8,  9, 10, 11]    [ 8,  9, 10, 11]
       [12, 13, 14, 15]    [12, 13, 14, 15]    [12, 13, 14, 15]

       [ 0,  1,  2,  3]    [ 0,  1,  2,  3]    [ 0,  1,  2,  3]
[2]    [ 4,  5,  6,  7]    [ 4,  5,  6,  7]    [ 4,  5,  6,  7]
       [ 8,  9, 10, 11]    [ 8,  9, 10, 11]    [ 8,  9, 10, 11]
       [12, 13, 14, 15]    [12, 13, 14, 15]    [12, 13, 14, 15]
```

Відповідні формули:

blockId = blockIdx.x + (blockIdx.y * gridDim.x)
threadId = threadIdx.x + (threadIdx.y * blockDim.x)

Щоб знайти загальний номер потоку, можна скористатися такими формулами:

threadsPerBlock = blockDim.x * blockDim.y

blockId = blockIdx.x + (blockIdx.y * gridDim.x)

threadId = threadIdx.x + (threadIdx.y * blockDim.x)

globalIdx = (blockIdx * threadsPerBlock) + threadId

globalIdx = (blockId * threadsPerBlock) + threadIdx.x + (threadIdx.y * blockDim.x)

```
             [0]                          [1]                          [2]

             [ 0,  1,  2,  3]             [ 0,  1,  2,  3]             [ 0,  1,  2,  3]
[0]    [0] + [ 4,  5,  6,  7]   [16] + [ 4,  5,  6,  7]      [32] + [ 4,  5,  6,  7]
             [ 8,  9, 10, 11]             [ 8,  9, 10, 11]             [ 8,  9, 10, 11]
             [12, 13, 14, 15]             [12, 13, 14, 15]             [12, 13, 14, 15]

             [ 0,  1,  2,  3]             [ 0,  1,  2,  3]             [ 0,  1,  2,  3]
[1]   [48] + [ 4,  5,  6,  7]   [64] + [ 4,  5,  6,  7]      [80] + [ 4,  5,  6,  7]
             [ 8,  9, 10, 11]             [ 8,  9, 10, 11]             [ 8,  9, 10, 11]
             [12, 13, 14, 15]             [12, 13, 14, 15]             [12, 13, 14, 15]
```

```
             [ 0,  1,  2,  3]          [ 0,  1,  2,  3]             [ 0,  1,  2,  3]
[2]   [96] + [ 4,  5,  6,  7]  [112] + [ 4,  5,  6,  7]     [128] + [ 4,  5,  6,  7]
             [ 8,  9, 10, 11]          [ 8,  9, 10, 11]             [ 8,  9, 10, 11]
             [12, 13, 14, 15]          [12, 13, 14, 15]             [12, 13, 14, 15]
```

Результат виглядатиме так:

```
          [  0,  1,   2,   3]    [ 16,  17,  18, 19]    [ 32,  33,  34, 35]
          [  4,  5,   6,   7]    [ 20,  21,  22, 23]    [ 36,  37,  38, 39]
          [  8,  9,  10,  11]    [ 24,  25,  26, 27]    [ 40,  41,  42, 43]
          [ 12, 13,  14,  15]    [ 28,  29,  30, 31]    [ 44,  45,  46, 47]


          [ 48,  49,  50, 51]    [ 64,  65,  66, 67]    [ 80,  81,  82, 83]
          [ 52,  53,  54, 55]    [ 68,  69,  70, 71]    [ 84,  85,  86, 87]
          [ 56,  57,  58, 59]    [ 72,  73,  74, 75]    [ 88,  89,  90, 91]
          [ 60,  61,  62, 63]    [ 76,  77,  78, 79]    [ 92,  93,  94, 95]


          [ 96,  97,  98,  99]   [112, 113, 114, 115]   [128, 129, 130, 131]
          [100, 101, 102, 103]   [116, 117, 118, 119]   [132, 133, 134, 135]
          [104, 105, 106, 107]   [120, 121, 122, 123]   [136, 137, 138, 139]
          [108, 109, 110, 111]   [124, 125, 126, 127]   [140, 141, 142, 143]
```

Практична частина

2.1 Правила та рекомендації щодо виконання лабораторної роботи

Процес виконання лабораторної роботи передбачає ознайомлення з теоретичним матеріалом, виконання практичного завдання та оформлення звіту. Структура звіту така:

1. Титульний лист.
2. Мета роботи.
3. Короткі теоретичні відомості.
4. Покроковий алгоритм виконання ходу роботи.
5. Отримані результати.
7. Висновки.
8. Додатки.

Оцінка за лабораторну роботу є комплексною і визначається на основі результатів практичного завдання та усної співбесіди, яка проводиться у форматі «питання-відповідь».

Практичні завдання наведено у наступному підрозділі. Їх виконання є обов'язковим для захисту роботи. *Окрім того, отримані результати мають бути відображені в звіті.*

Перелік тем, питання з яких будуть використовуватися під час усної співбесіди, наведено у підрозділі 2.3. Результату «задовільно» відповідають теми базового рівня, результату «добре» відповідають теми базового та розширеного рівня, результату «відмінно» відповідають теми базового, розширеного та поглибленого рівня.

Окрім того, в розділі «Перелік додаткових джерел» наведено список літератури, яка є корисною для підготовки до захисту та для більш глибокого ознайомлення з темою лабораторної роботи.

2.2 Практичне завдання

1. Зареєструватися на сайті https://cuda-on-line.parallel-computing.pro, ознайомитися з лекцією 1 «Введение в CUDA» та виконати відповідний тест.

2. Зобразити вручну сітку блоків різних розмірностей (з індексами «x» та «y»), як показано у роботі, і блоки з різними розмірностями потоків, а також визначити номери потоків усередині блоків і загальні номери потоків у сітці. Розмірності блоків у сітці і розмірності потоків вказано у наведеній нижче таблиці 1.

Таблиця 1 – Індивідуальні завдання

Варіант	Розмірності блоків у сітці	Розмірності потоків
1	3×3	4×2
2	3×3	2×3
3	3×3	2×4
4	3×3	5×2
5	2×3	2×2
6	2×3	4×2
7	2×3	3×2
8	2×3	2×4
9	2×4	5×2
10	2×4	2×2
11	2×4	5×2
12	2×4	3×2
13	3×4	2×3
14	3×4	3×3
15	3×4	5×2

2.3 Перелік тем

Базовий рівень. Головні означення (CPU, GPU, GPGPU, CUDA, kernel-функція). Типовий алгоритм GPGPU-програми. Grid-модель потоків.

Розширений рівень. Головні функції CUDA-програми, їх призначення та аргументи (cudaMalloc. cudaMemCpy, kernel-функції). Організація потоків у grid та функції для роботи з ними.

Поглиблений рівень. Переваги та недоліки гетерогенних обчислень на основі GPU. Порівняльний аналіз звичайних (CPU) та гетерогенних (CPU+GPU) програм з погляду їх структури.

2.4 Перелік додаткових джерел

За наявності відеоадаптера, що підтримує NVIDIA CUDA, Ви можете встановити середовище на свій комп'ютер, керуючись такими інструкціями: http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-microsoft-windows/index.html

Лабораторна робота № 4

Тема роботи: технологія GPGPU: робота з пам’яттю та синхронізація потоків

Мета роботи: метою цієї роботи є ознайомлення з структурою пам’яті GPU, особливостями роботи з нею та синхронізацією потоків за допомогою бар’єрів.

Теоретична частина

1.1 Структура пам’яті GPU

Звертання до пам’яті GPU при програмуванні паралельних програм займає набагато більше часу порівняно із звертанням звичайних програм до пам’яті CPU. При виконанні обчислень GPU використовує потоки, блоки потоків і сітку блоків (grid). Кожен потік має доступ до своєї локальної пам’яті (local memory), що є приватною для кожного потоку, тобто лише він може записувати або зчитувати дані в цю пам’ять. Блок потоків має доступ до так званої колективної пам’яті (shared memory). Кожен потік в одному блоці має доступ до неї. Також існує глобальна пам’ять (global memory). Будь-який потік в сітці (grid) має доступ до неї і може зчитувати/записувати дані до неї. Таким чином у розпорядженні одного потока є три типи пам’яті – local, shared, global memory (рис. 1).

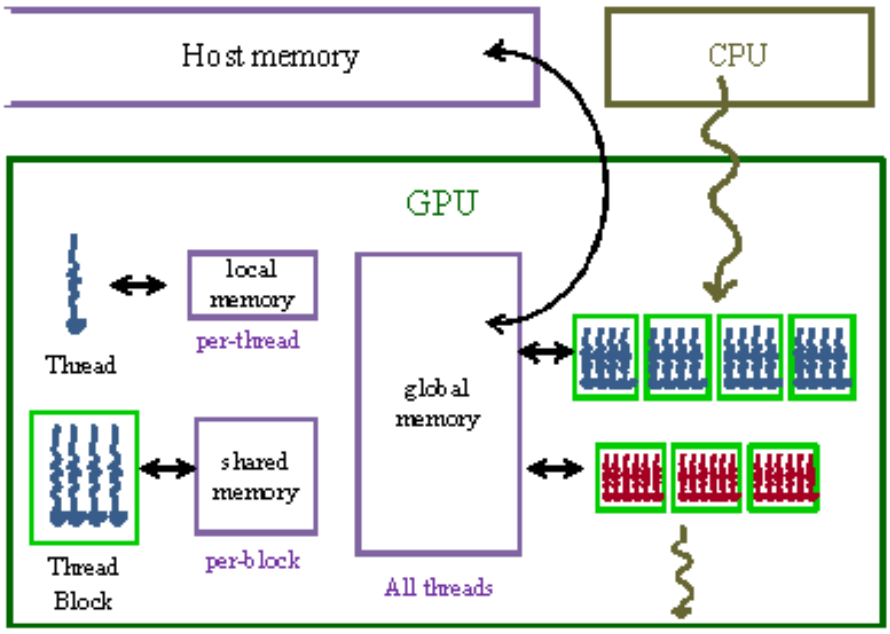


Рисунок 1 – Структура пам’яті GPU

Копіювання даних з CPU на GPU відбувається у глобальну пам’ять, яка є найбільшою за розміром, але найповільнішою для зчитування/записування. Найшвидшою ж у випадку коректного використання є колективна пам'ять (shared memory). Тому дані, які будуть зчитані або будуть перезаписуватися багато разів, краще перенести з глобальної пам’яті в колективну, обробити їх і перенести назад в глобальну пам’ять. В попередніх роботах використовувалась лише локальна та глобальна пам’яті. Розглянемо на прикладі використання різних типів пам’яті.

```
// використання різних типів пам’яті в CUDA
#include <stdio.h>
/*********************************
 * використання локальної пам’яті *
 *********************************/
// функція з __device__ або __global__ виконується на GPU
__global__ void use_local_memory_GPU(float in)
{
    float f;    // змінна "f" розташована в локальній пам’яті і є приватною для потока
    f = in;     // параметр "in" розташований в локальній пам’яті і є приватним для потока
}
/***********************
 * використання глобальної пам’яті *
 ***********************/

__global__ void use_global_memory_GPU(float *array)
{
    // "array" – покажчик на глобальну пам'ять GPU
    array[threadIdx.x] = 2.0f * (float) threadIdx.x;
}

/***********************
 * використання колективної пам’яті *
 ***********************/

__global__ void use_shared_memory_GPU(float *array)
{
    // локальні змінні, приватні для кожного потоку
    int i, index = threadIdx.x;
    float average, sum = 0.0f;

    // __shared__ змінні доступні для всіх потоків одного блока і
    // живуть доти, доки живе блок потоків
    __shared__ float sh_arr[128];

    // копіювання з "array" з глобальної пам’яті у sh_arr колективної.
    // кожен потік відповідає за один елемент.
    sh_arr[index] = array[index];

    // подальші дії з колективною памяттю.
}
```

```
int main(int argc, char **argv)
{
    /*
     * First, call a kernel that shows using local memory
     */
    use_local_memory_GPU<<<1, 128>>>(2.0f);

    /*
     * Next, call a kernel that shows using global memory
     */
    float h_arr[128];   // convention: h_ variables live on host
    float *d_arr;       // convention: d_ variables live on device (GPU global mem)

    // allocate global memory on the device, place result in "d_arr"
    cudaMalloc((void **) &d_arr, sizeof(float) * 128);
    // now copy data from host memory "h_arr" to device memory "d_arr"
    cudaMemcpy((void *)d_arr, (void *)h_arr, sizeof(float) * 128, cudaMemcpyHostToDevice);
    // launch the kernel (1 block of 128 threads)
    use_global_memory_GPU<<<1, 128>>>(d_arr);  // modifies the contents of array at d_arr
    // copy the modified array back to the host, overwriting contents of h_arr
    cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128, cudaMemcpyDeviceToHost);
    // ... do other stuff ...
    /*
     * Next, call a kernel that shows using shared memory
     */
    // as before, pass in a pointer to data in global memory
    use_shared_memory_GPU<<<1, 128>>>(d_arr);
    // copy the modified array back to the host
    cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128, cudaMemcpyHostToDevice);
    // ... do other stuff ...
    return 0;
}
```

1.2 Атомарні операції. Розглянемо тепер наступну задачу: нехай у нас є 10 000 потоків, кожен з яких намагається збільшити на 1 значення однієї з 100 комірок глобальної пам’яті. Що при цьому відбудеться?

При наївній реалізації відбудеться конфлікт і результат буде неправильним. Чому так відбувається? Оскільки комірок порівняно з потоками набагато менше, то з однією коміркою можуть одночасно працювати багато потоків. Кожен потік зчитує старе значення і перезаписує те саме значення + 1 декілька разів, що є неправильним. Наприклад, комірка має значення 2 і 100 потоків виконують рядок `g[i] = g[i] + 1;` Всі вони збільшать значення до 3 і перезапишуть його 100 разів, замість сподіваного значення 102.

Вирішити цю проблему дозволяють атомарні операції з пам’яттю. Існує багато типів таких операцій – додавання, знаходження мінімуму тощо. Для цієї задачі використовується атомічне додавання (`atomicAdd(&g[i], 1)`). Атомічні операції запобігають виникненню проблеми, описаної вище. Проте, ціною цього є більший час виконання програми. Нижче

наведено код з реалізацією наївного методу додавання і методу атомічного додавання.

```
  // increment_naive<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);
  // increment_atomic<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);
#include <stdio.h>
#include "gputimer.h"
#define NUM_THREADS 1000000
#define ARRAY_SIZE  100
#define BLOCK_WIDTH 1000
void print_array(int *array, int size)
{
    printf("{ ");
    for (int i = 0; i < size; i++)  { printf("%d ", array[i]); }
    printf("}\n");
}
__global__ void increment_naive(int *g)
{
       // which thread is this?
       int i = blockIdx.x * blockDim.x + threadIdx.x;

       // each thread to increment consecutive elements, wrapping at ARRAY_SIZE
       i = i % ARRAY_SIZE;
       g[i] = g[i] + 1;
}
__global__ void increment_atomic(int *g)
{
       // which thread is this?
       int i = blockIdx.x * blockDim.x + threadIdx.x;

       // each thread to increment consecutive elements, wrapping at ARRAY_SIZE
       i = i % ARRAY_SIZE;
       atomicAdd(& g[i], 1);
}
int main(int argc,char **argv)
{
    GpuTimer timer;
    printf("%d total threads in %d blocks writing into %d array elements\n",
           NUM_THREADS, NUM_THREADS / BLOCK_WIDTH, ARRAY_SIZE);
    // declare and allocate host memory
    int h_array[ARRAY_SIZE];
    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(int);
     // declare, allocate, and zero out GPU memory
    int * d_array;
    cudaMalloc((void **) &d_array, ARRAY_BYTES);
    cudaMemset((void *) d_array, 0, ARRAY_BYTES);
    // launch the kernel - comment out one of these
    timer.Start();
            // increment_naive<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);
    // increment_atomic<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);
    timer.Stop();
    // copy back the array of sums from GPU and print
    cudaMemcpy(h_array, d_array, ARRAY_BYTES, cudaMemcpyDeviceToHost);
    print_array(h_array, ARRAY_SIZE);
    printf("Time elapsed = %g ms\n", timer.Elapsed());
    // free GPU memory allocation and exit
    cudaFree(d_array);
    return 0;
}
```

1.3 Синхронізація потоків. Також часто виникає ситуація, коли один потік має доступ до результатів іншого потоку до того, як цей потік

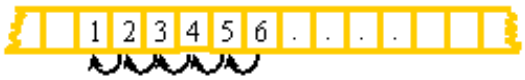
необхідні результати записав. Цю проблему може вирішити синхронізація потоків.

Синхронізація потоків – це одна з головних проблем у паралельному програмуванні. Найпростіша форма синхронізації – бар'єр (barrier). Бар'єр – це точка у програмі, де потоки призупиняють свою роботу і чекають, поки всі інші потоки дійдуть до цієї точки. Після цього вони продовжують роботу.

В цій роботі будуть розглянуті бар'єри, що працюють лише з блоком потоків, тобто для кожного блока існує свій бар'єр і всі потоки всередині певного блока продовжать роботу лише після того, як всі потоки відповідного блока зупиняться в точці бар'єра. Для розміщення описаного бар'єра необхідно використати таку вбудовану CUDA команду:

__syncthreads();

Розглянемо невеликий приклад, який показує необхідність використання бар'єрів. Нехай задано масив в пам'яті зі 128 елементів, який має вигляд [0, 1, 2, 3, 4, 5, 6 …] і задача полягає в тому, щоб змістити всі елементи на одну позицію вліво:



Подивимося на частини коду, що вирішує цю задачу, і подумаємо про можливі місця розташування бар'єрів.

```
int idx = threadIdx.x;
__shared__ int array[128];
array[idx] = threadIdx.x;
if (idx < 127) {
      array[idx] = array[idx + 1];
}
d_out[idx] = array[idx];
```

Скільки бар'єрів необхідно використати тут? Відповідь – 2. Розглянемо де і чому. Перший бар'єр необхідно розмістити після рядка

```
array[idx] = threadIdx.x;
```

для того, щоб впевнитись в тому, що всі потоки записали відповідне значення в комірку до того, як з цих комірок буде відбуватись зчитування.

Наступний бар’єр буде відноситись до рядка `array[idx] = array[idx + 1];`. Цей рядок поєднує операцію зчитування і операцію записування даних в розподілену пам'ять. Для розташування бар’єра і правильності роботи ці операції необхідно розділити на дві:

```
if (idx < 127) {
      int tmp = array[idx + 1];
      __syncthreads();
      array[idx] = tmp;
}
```

Розташовуючи бар’єр після ініціалізації локальної змінної, ми запобігаємо ситуації, що допускає запис в цю змінну вже перезаписаного значення. Кінцевий код виглядатиме таким чином:

```
int idx = threadIdx.x;

__shared__ int array[128];

array[idx] = threadIdx.x;
__syncthreads();

if (idx < 127) {
      int tmp = array[idx + 1];
      __syncthreads();
      array[idx] = tmp;
}

d_out[idx] = array[idx];
```

1.4 Паралелізм у CUDA-обчисленнях

Паралелізація при створенні CUDA-програм не обмежується лише розподіленням навантаження між ядрами графічної карти. Сучасні технології дозволяють реалізовувати концепцію паралелізму відразу на багатьох рівнях: одночасний виклик декількох kernel-функцій на одній відеокарті, одночасне використання декількох відеокарт і так далі.

Важливу роль при цьому відіграє такий параметр, як синхронність/асинхронність функцій. Асинхронні команди повертають контроль CPU ще до закінчення їх виконання на GPU й дають можливість відразу викликати наступну функцію (наприклад, на CPU або іншій графічній карті). Синхронні функції навпаки – блокують всі наступні операції до закінчення своєї роботи. Так, kernel-функції та cudaMemcpuAsync є асинхронними, а cudaMemcpu – синхронною.

Це дає можливість одночасно викликати декілька kernel-функцій на одній відеокарті за допомогою потоків (streams). Поток (stream) – це

послідовність операцій, які реалізуються на графічній карті в заданому при написанні коду порядку. Потоки можуть виконуватися одночасно, послідовно або накладатися один відносно одного. Нижче наведено приклад коду, що створює чотири потоки та одночасно викликає в кожному з них одну й ту саму kernel-функцію. Варто відзначити, що для kernel-функції номер потоку, в якому її потрібно виконати, є четвертим (опціональним) параметром. За замовчуванням всі функції виконуються в так званому «нульовому» потоці.

```
#define NSTREAM  4
cudaStream_t stream_array [NSTREAM] ; // декларація потоків

for (int i = 0; i <NSTREAM; i++){
    cudaStreamCreate(&stream_array [ i ] ) ; // створення потоків

    mykernel <<< nblocks , nthreads ,0 , stream_array [ i ] >>>  ( . . . ) ;
                                    }
```

Більш того, існують технології, що націлені на паралельне використання декількох графічних карт, наприклад, Multi-GPU Programming. Існує декілька шляхів реалізації такої концепції. Найпростіший з них оснований на виборі поточної відеокарти, до якої будуть застосовуватися усі команди.

Практична частина

2.1 Правила та рекомендації до виконання лабораторної роботи

Процес виконання лабораторної роботи передбачає ознайомлення з теоретичним матеріалом, виконання практичного завдання та оформлення звіту. Структура звіту така:

1. Титульний лист.
2. Мета роботи.
3. Короткі теоретичні відомості.
4. Покроковий алгоритм виконання ходу роботи.
5. Отримані результати.
7. Висновки.
8. Додатки.

Оцінка за лабораторну роботу є комплексною і визначається на основі результатів практичного завдання та усної співбесіди, яка проводиться у форматі «питання-відповідь».

Практичні завдання наведено у наступному підрозділі. Їх виконання є обов’язковим для захисту роботи. *Окрім того, отримані результати мають бути відображені в звіті.*

Перелік тем, питання до яких будуть використовуватися під час усної співбесіди, наведено у підрозділі 2.3. Результату «задовільно» відповідають теми базового рівня, результату «добре» відповідають теми базового та розширеного рівня, результату «відмінно» відповідають теми базового, розширеного та поглибленого рівня.

Окрім того, в розділі «Перелік додаткових джерел» наведено список літератури, яка є корисною для підготовки до захисту та для більш глибокого ознайомлення з темою лабораторної роботи.

2.2 Практичне завдання

1. Самостійно дослідити інші атомічні операції, що можуть бути використані в CUDA-програмах та навести їх у звіті.

2. Використовуючи бар'єри та shared memory, модифікувати CUDA-код з підрозділу 1.3 або написати відповідний псевдокод згідно з власним варіантом:

1. Кожен елемент, крім першого і останнього, замінити на максимальний з двох сусідніх, наприклад: вхід [0, 1, 2, 2, 4, 2, 1] – вихід [0, 2, 2, 4, 2, 4, 1].
2. У кожну комірку записати суму трьох значень справа, якщо справа значень немає, то вважати, що там знаходяться нулі,
   наприклад: вхід [0, 1, 2, 2, 4, 2, 1] – вихід [5, 8, 8, 7, 3, 1, 0].
3. Кожен елемент, крім першого і останнього, замінити на мінімальний з двох сусідніх.
4. В кожну комірку записати середнє значення усього масиву.
5. У кожну комірку записати середнє значення її двох сусідніх комірок. Якщо комірка має лише один сусідній елемент – то він і буде цим середнім значенням.
6. У кожну комірку записати середнє значення цієї комірки разом з двома її сусідніми комірками. Якщо однієї сусідньої комірки немає – вважати, що вона набуває нульового значення.
7. У кожну комірку записати сумарне значення цієї комірки разом з двома її сусідніми комірками. Якщо однієї сусідньої комірки немає – вважати, що вона набуває нульового значення.
8. У кожну комірку записати значення, що дорівнює значенню цієї комірки мінус сума двох її сусідніх комірок. Якщо однієї сусідньої комірки немає – вважати, що вона набуває нульового значення.
9. У кожну комірку записати добуток її сусідніх елементів. Якщо однієї сусідньої комірки немає – вважати, що вона набуває одиничного значення.

10. У кожну комірку записати результат ділення значення її правого сусіда на значення її лівого сусіда. Якщо при цьому потрібно ділити на нуль – запишіть у комірку нульове значення.

2.3 Перелік тем

Базовий рівень: Типи пам’яті GPU. Атомарні операцій: означення. Бар’єри: означення.

Розширений рівень: Атомарні операції: переваги та недоліки. Паралелізм в CUDA-програмах. Бар’єри: переваги та недоліки.

Поглиблений рівень: Доцільність використання shared memory. Актуальність проблеми синхронізації потоків.

2.4 Перелік додаткових джерел

Додаткові посилання що паралелізму у CUDA-програмах англійською мовою:

1. http://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/
2. http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/
3. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

Лабораторна робота № 5

Тема роботи: технологія MPI
Мета роботи: метою цієї роботи є ознайомлення з технологією MPI та головними принципами організації обчислень на її основі.

Теоретична частина

1.1 Загальна характеристика технології MPI

MPI є однією з найбільш поширених технологій для паралельних обчислень на основі розподілених систем (систем з розподіленою пам'яттю). Головним засобом взаємодії між паралельними процесами у таких системах є обмін повідомленнями, на що й вказує назва технології (MPI – Message Passing Interface / інтерфейс передачі повідомлень). Варто відзначити, що сам MPI є лише специфікацією обчислень, на основі якої було створено велику кількість програмних реалізацій цієї технології, таких як MPICH, Open MPI, WMPI та інші.

Інтерфейс MPI підтримує створення паралельних програм за принципом MIMD (Multiple Instruction Multiple Data), тобто об'єднання процесів з різним вихідним кодом. Але розробка та тестування таких програм є надзвичайно складною задачею, тому на практиці найбільш часто замість концепції MIMD використовується концепція SIMD (Single Instruction Multiple Data), в межах якої усі паралельні процеси виконують один й той самий код.

Виконання MPI-програм потребує встановлення відповідного програмного забезпечення, яке відповідає за створення паралельних процесів та взаємодію між ними. Як правило, таке програмне забезпечення має бути встановленим на усіх комп'ютерах, які беруть участь у MPI-обчисленнях, але можливі й винятки, коли достатньо лише налаштованого головного комп'ютера.

MPI-програма за своєю структурою є множиною паралельних взаємодіючих процесів. Ці процеси можуть виконуватися як на окремих комп'ютерах, так і в межах одного комп'ютера на основі багатоядерного процесора. Кожен процес працює в своєму адресному просторі без використання будь-яких загальних даних та взаємодіє з іншими процесами шляхом явного відправлення повідомлень. Для локалізації взаємодії окремих паралельних процесів можливо створювати групи процесів шляхом надання їм власного середовища спілкування – комунікатора. Склад таких груп є довільним. Вони можуть збігатися, входити одна в одну, не перетинатися або перетинатися частково. Процеси можуть взаємодіяти між собою лише в межах певного комунікатора. На початку роботи MPI-програми усі процеси автоматично відносяться до загального

комунікатора з назвою MPI_COMM_WORLD. Кожен процес MPI-програми має в кожній групі, в яку він входить, свій унікальний параметр – номер процесу. Але оскільки кожен процес може одночасно входити в декілька груп, процес в процесі взаємодії визначається двома параметрами: комунікатором та номером процесу в межах цього комунікатора.

Ці параметри відіграють значну роль при відправленні та отриманні повідомлень. Під повідомленнями розуміють набір даних певного типу. Кожне повідомлення має декілька атрибутів: номер процесу-відправника, номер процесу-отримувача, номер самого повідомлення, назва поточного комунікатора та інші.

Існує два головних типи обміну повідомленнями:

1. Обмін повідомленнями між двома процесами. Такі операції називають індивідуальними або типу «точка-точка». В них можуть брати участь лише два процеси: відправник та отримувач, при чому відправник має явно ініціювати відправлення повідомлення, а отримувач – отримання. Такі операції поділяються на операції з блокування (з синхронізацією) та без блокування (асинхронні);

2. Колективний обмін повідомленнями між усіма процесами у межах комунікатора. Такі операції використовуються для відправлення однакового набору даних усім процесам, для збирання даних з усіх процесів на одному та для інших, більш складних видів взаємодії, таких як колективні обчислювальні операції. Колективний обмін повідомленнями завжди синхронний.

1.2 Створення програм на основі MPI

Розглянемо структуру MPI-програм на основі тестового прикладу. В нижченаведеному прикладі реалізується обробка та додавання чотирьох масивів чисел. Спочатку створюються паралельні процеси, кожен з яких отримує свій масив та виконує над ним арифметичні дії. Потім результати обчислень з допоміжних процесів передаються до головного процесу, яких й виконує процедуру додавання. Програмний код виглядає таким чином:

```
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"

using namespace std;

int main (int argc, char* argv[]){

	MPI_Init( &argc, &argv);

	int nRanks;
	MPI_Comm_size(MPI_COMM_WORLD, &nRanks);

	int rank;
	MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

```
	int arr[10];

	for (int i = 0; i < 10; i++){

	arr[i] = rank+1;
	}

	for (int i = 0; i < 10; i++){
		arr[i] *= arr[i];
	}

	if (rank != 0){

	   MPI_Send ( arr, _countof(arr), MPI_INT, 0, 0, MPI_COMM_WORLD);

	} else {
	  int buff[10];
	  for (int i = 1; i < nRanks; i++){
	   MPI_Recv (buff, _countof(buff), MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUSES_IGNORE);
	   for (int i = 0; i < 10; i++){
	       arr[i] += buff[i];
                                                              }
                                                                    }
	printf ("First element =  %d \n", arr[0]);
	printf ("Last element =  %d \n", arr[9]);
	      }

	MPI_Finalize();
	return 0;
}
```

Спочатку потрібно підключити бібліотеку MPI – «mpi.h». Паралельна ділянка програми створюється за допомогою функцій MPI_Init(&argc, &argv) та MPI_Finalize(), які позначать її початок та кінець. Кількість процесів визначається при запуску програми (детальніше в розділі 1.3).

Весь код в межах паралельної ділянки одночасно виконується усіма процесами. Функція MPI_Comm_size(MPI_COMM_WORLD, &nRanks) дозволяє дізнатись про кількість процесів у обраному комунікаторі. Першим аргументом функції є комунікатор, другим – змінна типу int, в яку буде записаний результат.

Функція MPI_Comm_rank (MPI_COMM_WORLD, &rank) дозволяє отримати номер процесу, який викликає цю функцію, в обраному комунікаторі. Номер процесу відіграє дуже важливу роль у організації роботи MPI-програми. Він використовується в процесі обміну повідомленнями, а також для визначення ділянок коду, які мають виконувати лише певні процеси (наприклад, за допомогою оператора if).

Далі в кожному процесі створюється своя версія масиву чисел і заповнюється значеннями згідно з номером процесу. Після цього кожен процес виконує операції над своїм масивом (піднесення до квадрата). Після виконання обчислень всі процесі, окрім головного (№ 0), відправляють результати обчислень на головний процес за допомогою

функції MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm). Ця функція має такі аргументи: buf – адреса буфера пам’яті, в якому знаходяться потрібні дані, count – кількість елементів даних у повідомленні, type – тип елементів даних, dest – номер процесу-отримувача, tag – ідентифікатор повідомлення (потрібен, коли здійснюється обмін декількома повідомленнями між двома процесами), comm – комунікатор. Варто зазначити, що функція MPI_Send лише починає процес відправлення повідомлення, і її закінчення не свідчить про завершення обміну та отримання повідомлення.

Потім в процесі № 0 створюється буферний масив для отримання результатів.

За допомогою функції MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status) дані, що містяться в відправленому повідомленні записуються у буферний масив. Функція MPI_Recv має такі аргументи: buf – адреса буфера пам’яті, в який потрібно записати дані, count – кількість елементів даних у повідомленні, type – тип елементів даних, source – номер процесу-відправника, tag – ідентифікатор повідомлення (потрібен, коли здійснюється обмін декількома повідомленнями між двома процесами), comm – комунікатор, status – інформаційна структура даних.

Варто зазначити, що буфер пам'яті має бути достатнього розміру, щоб прийняти повідомлення, а також мати той самий тип елементів. Функція MPI_Recv є синхронною і блокує подальшу роботу процесу до свого завершення.

Після запису даних у буферний масив вони додаються до значень головного масиву процесу. Ці операції поетапно виконуються для всіх процесів, окрім головного. Після їх завершення результати виводяться на екран і функція MPI_Finalize() завершує паралельну ділянку.

1.3 Встановлення та налаштування програмних засобів MPI

Для виконання MPI-програм потрібно встановити спеціальне програмне забезпечення.

Прикладом такого забезпечення може бути MS-MPI SDK, який можливо завантажити за таким посиланням: https://msdn.microsoft.com/en-us/library/bb524831.aspx#.

Процес його встановлення є тривіальним.

Після встановлення можливо перевірити коректність створених системних змінних, набравши в командному рядку (cmd) команду «set MSMPI» (рис. 1).

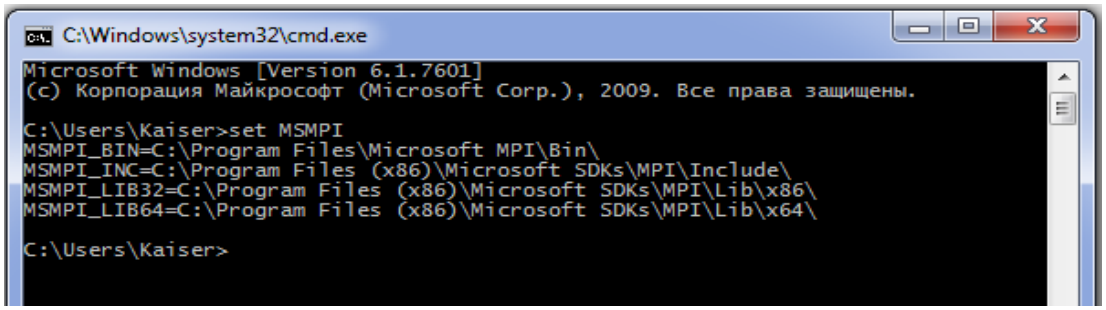


Рисунок 1 – Системні змінні MSMPI

Після цього потрібно відкрити MS Visual Studio та створити новий проект (C++ Win32 Console Application). Потім в налаштуваннях проекту (PROJECT -> «Project Name» Properties -> Configuration Properties) потрібно вказати шлях до відповідних файлів MPI, щоб компілятор був в змозі їх використовувати. В розділі «C/C++ -> General -> Additional Include Directories» потрібно вказати значення «$(MSMPI_INC)» та «$(MSMPI_INC)\x64» або «$(MSMPI_INC)\x86» залежно від обраної розрядності проекту. В розділі «Linker-> General-> Additional Include Directories» потрібно вказати значення «$(MSMPI_LIB64)» або «$(MSMPI_LIB32)». В розділі «Linker-> Input -> Additional Dependencies» потрібно додати значення «msmpi.lib» (рис. 2).

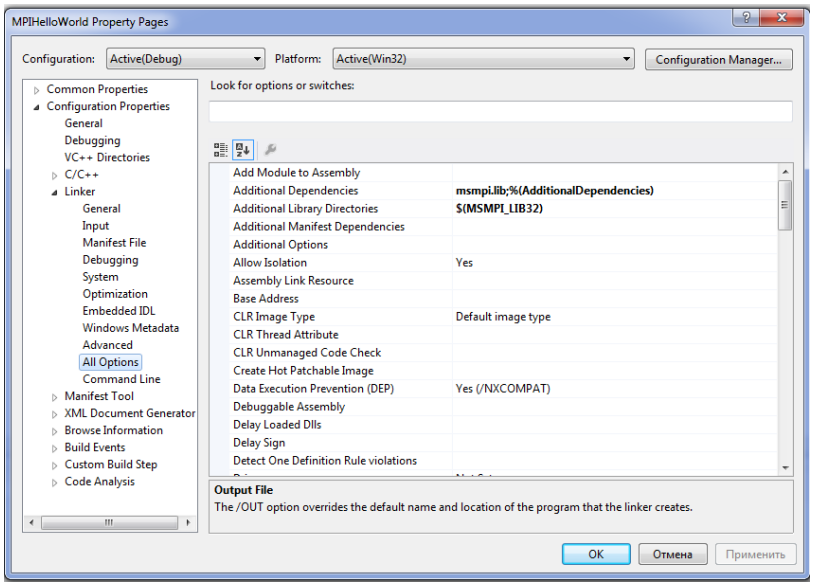


Рисунок 2 – Налаштування проекту Visual Studio

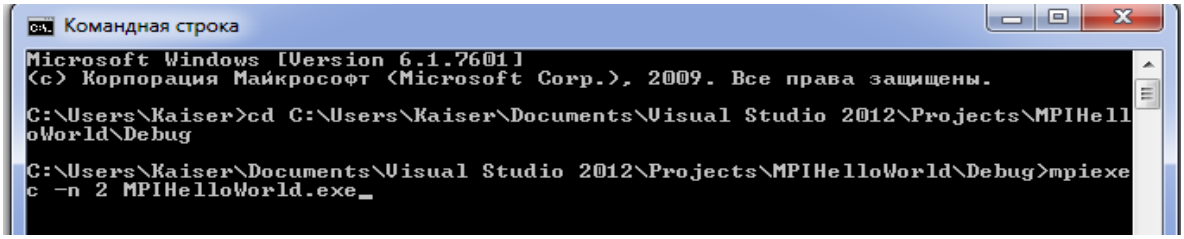


Рисунок 3 – Виклик MPI-програми

Наступним кроком є створення та компілювання MPI-програми у згідно з даними, наведеними в розділі 1.2 (рис. 3). Для тестового запуску створеної програми потрібно відкрити командний рядок (cmd) та обрати папку, в якій знаходиться .exe-файл програми, за допомогою команди зміни каталогу «cd» (наприклад, cd C:\Users\Kaiser\Documents\Visual Studio 2012\Projects \MPIHelloWorld\Debug). Потім потрібно виконати команду «mpiexec –n nThreads ProjectName.exe», де замість nThreads потрібно вказати потрібну кількість паралельних процесів (4 у випадку тестової програми з підрозділу 1.2), а замість ProjectName.exe потрібно вказати назву .exe-файлу програми.

Практична частина

2.1 Правила та рекомендації щодо виконання лабораторної роботи

Процес виконання лабораторної роботи передбачає ознайомлення з теоретичним матеріалом, виконання практичного завдання та оформлення звіту. Структура звіту така:

1. Титульний лист.
2. Мета роботи.
3. Короткі теоретичні відомості.
4. Покроковий алгоритм виконання ходу роботи.
5. Отримані результати.
7. Висновки.
8. Додатки.

Оцінка за лабораторну роботу є комплексною і визначається на основі результатів практичного завдання та усної співбесіди, яка проводиться у форматі «питання–відповідь».

Практичні завдання наведено у наступному підрозділі. Їх виконання є обов’язковим для захисту роботи. *Окрім того, отримані результати мають бути відображені в звіті.*

Перелік тем, питання до яких будуть використовуватися під час усної співбесіди, наведено у підрозділі 2.3. Результату «задовільно» відповідають теми базового рівня, результату «добре» відповідають теми базового та розширеного рівня, результату «відмінно» відповідають теми базового, розширеного та поглибленого рівня.

Окрім того, в розділі «Перелік додаткових джерел» наведено список літератури, яка є корисною для підготовки до захисту та для більш глибокого ознайомлення з темою лабораторної роботи.

2.2 Практичне завдання

1. Завантажити та встановити MS-MPI SDK. Запустити тестовий приклад з розділу 1.2 та навести результати його роботи у вигляді скріншотів.

2. Модифікувати тестовий приклад таким чином, щоб:

1) результати обчислень відправлялися на процес № 1, а не на № 0;

2) результати на головний процес відправляли лише процеси з парним номером;

3) результати на головний процес відправляли лише процеси з непарним номером;

4) обчислення виконувались лише при кількості процесів не менше двох;

5) в обчислення брали участь не більше чотирьох перших процесів, навіть якщо їх кількість буде більшою.

Виконати потрібно лише одне завдання з п’яти залежно від номера у списку групи. Лістинг програми навести в звіті

2.3 Перелік тем

Базовий рівень. Визначення MPI. Головні поняття MPI-обчислень (процес, комунікатор, повідомлення). Загальна структура MPI-програми.

Розширений рівень. Програмне забезпечення для MPI-обчислень (MS-MPI SDK). Типи обміну повідомлення. Структура MPI-програми з програмної точки зору.

Поглиблений рівень. Особливості колективного обміну повідомленнями. Операції обміну повідомленнями з блокуванням та без нього. Детальне пояснення практичного завдання №2.

2.4 Перелік додаткових джерел

1. Michael J. Quinn Parallel Programming in C with MPI and OpenMP / Michael J. Quinn – McGraw-Hill Higher Education, 2004. – 544c. (Класична англомовна робота, присвячена паралельному програмуванню на основі MPI та OpenMP)

2. А. С. Антонов Паралельне програмування з використанням технології MPI / А. С. Антонов. – М. : МГУ, 2004. – 71 с. (Посібник, що містить базову інформацію про технологію MPI)

3. Введение в технологии параллельного программирования (MPI) [Електронний ресурс] – Режим доступу: http://www.intuit.ru/studies/courses/4447/983/lecture/14927 (набір лекцій, присвячених основам програмування за допомогою MPI)

4. MPI Tutorial [Електронний ресурс] – Режим доступу: http://mpitutorial.com/ (англомовний сайт, що містить базову інформацію про програмування на основі MPI)

Лабораторна робота № 6

Тема роботи: технологія OpenMP.
Мета роботи: метою цієї роботи є ознайомлення з технологією OpenMP та головними принципами організації багатопотокових обчислень на її основі.

Теоретична частина

1.1 Багатопотоковість (Multithreading)

Спрощено багатопотоковість можна визначити як концепцію паралельних обчислень, яка передбачає розподіл задачі на декілька підзадач, що виконуються одночасно. З більш технічної точки зору багатопотоковість передбачає створення декількох *потоків* (*threads*) – одиниць виконання паралельної програми – в межах одного процесу, які можуть виконуватися одночасно та за потреби взаємодіяти між собою.

Кожен процес починає свою роботу з один потоком, який ще називають головним. Потім, за потреби, з нього можуть бути створені додаткові потоки (рис. 1) . Характерними відмінностями потоків від процесів є те, що вони мають спільний адресний простір та спільний доступ до пам'яті, що полегшує взаємодію між ними.

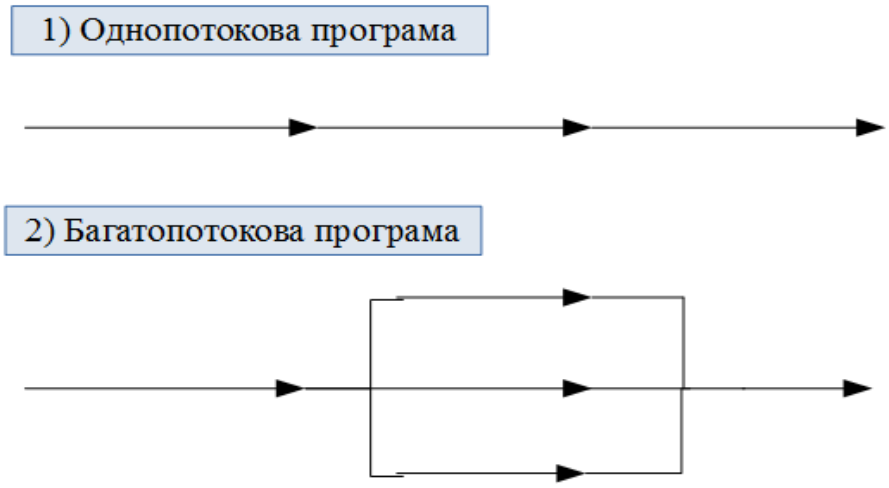


Рисунок 1 – Спрощена схема багатопотокової програми

З цього й випливає велика кількість переваг багатопотоковості: потоки потребують менше часу на створення, зміну та видалення; спільна пам'ять та адресний простір спрощують та пришвидшують взаємодію; також паралельне виконання потоків дозволяє повноцінно використовувати усі переваги паралельних обчислень у системах з багатоядерними CPU, що приводить до більш ефективного використання

апаратних ресурсів і, як наслідок, до збільшення швидкодії програми. Подібна багатопотокова модель знаходить застосування і в концепції GPU-обчислень, таких як CUDA та OpenCL.

До недоліків такого підходу можна віднести необхідність синхронізації роботи потоків та загальне ускладнення структури програми.

1.2 Стандарт OpenMP

OpenMP є одним з найбільш поширених відкритих стандартів для організації багатопотокових обчислень. Перша версія OpenMP з'явилася в 1997 році для Fortran, в наступному році з'явилася версія для C/C++. Наразі стандарт постійно оновлюється, підтримує розробку програм на C, C++ та Fortrun на більшості платформ та операційних систем, включно з Windows, Linux, macOS та іншими.

З програмної точки зору OpenMP складається здебільшого з директив компілятора та змінних середовища, які дозволяють створювати паралельні програми згідно з концепцією багатопотоковості, тобто спочатку створюється головний потік, який також, за потреби, породжує додаткові потоки, що виконуються паралельно, а після завершення роботи передають отримані результати знову на головний потік.

Варто відзначити, що OpenMP може застосовуватися у поєднанні з іншими технологіями паралельних обчислень. Яскравим прикладом є комбіноване застосування OpenMP та MPI, яке є доцільним через те, що MPI є орієнтованим на роботу з розподіленою пам'яттю, має місце в комп'ютерному кластері в цілому; а OpenMP більш орієнтований на роботу зі спільною пам'яттю, що має місце на окремих нодах кластера.

1.3 Створення програм з застосуванням OpenMP

З точки зору програмного коду, для написання базової паралельної програми за допомогою OpenMP достатньо ознайомитися лише з декількома функціями, більшість з яких займає не більше 1–2 рядків коду. Проте, як і випадку інших паралельних технологій, головна складність полягає в тому, щоб визначити, де та які саме функції буде доцільно застосовувати, оскільки не усі алгоритми можуть бути ефективно розпаралелені. Також важливим фактором є розмір та тип даних, які потрібно обробити.

В загальному випадку спочатку потрібно створити паралельну ділянку безпосередньо в коді програми, в якій буде створено декілька паралельних потоків. Це робиться за допомогою команди *#pragma omp parallel {...}* , яка створює декілька потоків, кожен з яких виконує код, вказаний в фігурних дужках. Кількість потоків визначається автоматично, зазвичай, залежно від кількості ядер процесора. Проте можливо явно встановити кількість потоків таким чином: *#pragma omp parallel num_threads(n) {...}* , де *n* – потрібна кількість потоків.

Для автоматичного розпаралелювання циклу *for* існує спеціальна команда *#pragma omp for*, яку потрібно застосовувати всередині паралельної ділянки, тобто остаточний код виглядатиме таким чином:

```
#pragma omp parallel
{
  #pragma omp for
    for (int i = 0; i < 100; i++) {...}
                    }
```

В цьому прикладі ітерації циклу будуть рівномірно розподілені між створеними потоками, кожен з яких одночасно виконає свою частину обчислень. Звичайно, у випадку коли цикл обчислює рекурентну формулу (формула, в якій кожен наступний елемент знаходиться на основі попереднього), результат роботи програми може бути некоректним. Взагалі, в загальному випадку потоки виконуються неупорядковано, тому за потреби для їх упорядкування та синхронізації використовуються спеціальні додаткові функції.

Якщо ж потрібно виконувати різні функції одночасно, використовується команда *#pragma omp sections*, яка теж має викликатися з паралельної ділянки. Загальний синтаксис виглядатиме таким чином:

```
#pragma omp parallel
{
   ...
  #pragma omp sections
    {

      { Function1(); }

      #pragma omp section
      { Function2(); }

      #pragma omp section
      { Function3(); }

}}
```

У вищенаведеному прикладі кожна з функцій № 1, 2 та 3 отримує власний потік і виконуються одночасно.

1.4 Налаштування OpenMP

В загальному випадку підключення OpenMP є досить тривіальним. У MS Visual Studio достатньо активувати OpenMP в налаштуваннях проекту (Project→Properties→C/C++→Language: змінити OpenMP Support на yes(/openmp)), а потім підключити бібліотеку у коді програми (#include <omp.h>).

1.5 Приклад OpenMP програми

Нижче наведено приклад тривіальної OpenMP-програми, що паралельно збільшує значення кожного елемента у масиві на 2.

```
using namespace std;
#include <iostream>
#include <omp.h>
int main(){

       int A[100];
       for (int i = 0; i < 100; i++)
       { A[i] = i;}

#pragma omp parallel
       {
     #pragma omp for
              for (int i = 0; i < 100; i++){
               A[i] += 2;
              }
       }

return 0;
}
```

Практична частина

2.1 Правила та рекомендації щодо виконання лабораторної роботи

Процес виконання лабораторної роботи передбачає ознайомлення з теоретичним матеріалом, виконання практичного завдання та оформлення звіту. Структура звіту така:

1. Титульний лист.
2. Мета роботи.
3. Короткі теоретичні відомості.
4. Покроковий алгоритм виконання ходу роботи.
5. Отримані результати.
7. Висновки.
8. Додатки.

Оцінка за лабораторну роботу є комплексною і визначається на основі результатів практичного завдання та усної співбесіди, яка проводиться у форматі «питання–відповідь».

Практичні завдання наведено у наступному підрозділі. Їх виконання є обов’язковим для захисту роботи. *Окрім того, отримані результати мають бути відображені в звіті.*

Перелік тем, питання до яких будуть використовуватися під час усної співбесіди, наведено у підрозділі 2.3. Результату «задовільно» відповідають теми базового рівня, результату «добре» відповідають теми базового та розширеного рівня, результату «відмінно» відповідають теми базового, розширеного та поглибленого рівня.

2.2 Практичне завдання

1. Налаштувати OpenMP та написати з його допомогою паралельну програму, яка буде виконувати таке:

1) знаходити суму двох матриць, заповнених довільними значеннями;

2) одночасно ділити усі елементи першого числового масиву даних на 5 та множити усі елементи другого масиву даних на 10;

3) підносити до квадрата елементи числового масиву. (Студент повинен дослідити як змінюється швидкодія при зміні розміру масиву від малого до надвеликого);

4) паралельно та послідовно підносити до квадрата елементи числового масиву. (Студент повинен дослідити різницю у часі виконання);

5) знаходити добуток двох матриць, заповнених довільними значеннями;

6) одночасно заповнювати три порожні (заповнені нулями) числові масиви довільними даними;

7) додавати відповідні елементи двох числових масивів. (Студент повинен дослідити як змінюється швидкодія при зміні розмірів масивів від малого до надвеликого);

8) паралельно та послідовно додавати відповідні елементи двох числових масивів. (Студент повинен дослідити різницю у часі виконання);

9) знаходити різницю двох матриць, заповнених довільними значеннями;

10) одночасно збільшувати першу половину числового масиву на 3 та зменшувати другу половину на 5;

11) множити відповідні елементи двох числових масивів. (Студент повинен дослідити як змінюється швидкодія при зміні розмірів масивів від малого до надвеликого);

12) паралельно та послідовно множити відповідні елементи двох числових масивів. (Студент повинен дослідити різницю у часі виконання).

Виконати потрібно лише одне завдання залежно від номера у списку групи. Лістинг програми навести в звіті.

2.3 Перелік тем

Базовий рівень. Означення OpenMP. Поняття багатопотоковості. Поняття потоку і його відмінність від процесу.

Розширений рівень. Головні команди OpenMP (*pragma omp sections, pragma omp for, ...)* та їх застосування. Структура OpenMP-програми з програмної точки зору.

Поглиблений рівень. Переваги та недоліки багатопотоковості. Багатопотоковість та інші технології паралельних обчислень.

Список використаної літератури

Базова

1. Кузьменко Б. В. Технологія розподілених систем та паралельних обчислень : навчальний посібник / Б. В. Кузьменко, Чайковська О. А. – К. : Видавничий центр КНУКІМ, 2011. – 126 с.
2. Яровий А. А. Методи та засоби організації високопродуктивних паралельно-ієрархічних обчислювальних систем із рекурсивною архітектурою : монографія / Яровий А. А. – Вінниця : ВНТУ, 2016. – 363 с.
3. Кожем'яко В. П. Паралельно-ієрархічні мережі як структурно-функціональний базис для побудови спеціалізованих моделей образного комп'ютера : монографія / Кожем'яко В. П., Тимченко Л. І., Яровий А. А. – Вінниця : УНІВЕРСУМ-Вінниця, 2005. – 161 с.
4. Аксак Н. Г. Паралельні та розподілені обчислення : підручник / Аксак Н. Г., Руденко О. Г., Гуржій А. М. – Х. : Компанія СМІТ, 2009. – 480 с.
5. Воеводин В. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. – СПб. : БХВ-Петербург, 2002. – 608 с.
6. Хьюз К. Параллельное и распределенное программирование на C++. / Хьюз К., Хьюз Т.– М. : ИД «Вильямс», 2004. – 672 с.
7. Засоби паралельного програмування / [Стіренко С. Г., Грибенко Д. В., Зіненко А. І., Михайленко А. В.]. – К. : НТУУ «КПІ», 2011. – 181 с.
8. Лупин С. А. Технологии параллельного программирования / С. А. Лупин, М. А. Посыпкин. – М. : ИД «ФОРУМ»; ИНФРА-М, 2011. – 208 с.
9. Таненбаум Э. Распределенные системы. Принципы и парадигмы. / Э. Таненбаум, М. ван Стеен. – СПб. : Питер, 2003. – 877 с.
10. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования / Эндрюс Г. Р. – М. : ИД «Вильямс», 2003. – 512 с.
11. Черноруцкий И. Г. Методы принятия решений / Черноруцкий И. Г. – СПб. : БХВ-Петербург, 2005. – 416 с.
12. Ларичев О. И. Теория и методы принятия решений / Ларичев О. И. – М. : Логос, 2000.– 296 с.
13. Саати Т. Принятие решений. Метод анализа иерархий / Саати Т. – М. : Радио и связь, 1993. – 320 с.

Допоміжна

1. Бройнль Т. Паралельне програмування : навч. посібник / Бройнль Т. – К. : Вища школа, 1997. – 358 с.

2. Михайлов Б. М. Классификация и организация вычислительных систем : учебное пособие / Б. М. Михайлов, Р. Ф. Халабия. – М. : МГУПИ, 2010. – 144 с.

3. Гергель В. П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем / Гергель В. П. – Нижний Новгород : ННГУ им. Н. И. Лобачевского, 2010. – 421 с.

4. Эхтер Ш. Многоядерное программирование / Ш. Эхтер, Д. Робертс – СПб. : «Питер», 2010. – 316 с.

5. Барский А. Б. Параллельные информационные технологии / Барский А. Б. – М. : ИнТУИТ; Бином, 2007. – 503 с.

6. Немнюгин С. Параллельное программирование для многопроцессорных вычислительных систем / С.Немнюгин, О. Стесик. – СПб. : БХВ-Петербург, 2002. – 400 с.

7. Биллиг В. А. Параллельные вычисления и многопоточное программирование / Биллиг В. А. – М. : НОУ «Интуит», 2016. – 310 с.

Інформаційні ресурси

1. GPGPU: General Purpose computations on Graphic Processing Unit [Електронний ресурс]. – Режим доступу: http://www.gpgpu.org

2. Parallel Programming and Computing Platform – CUDA – NVIDIA [Електронний ресурс] – Режим доступу: http://www.nvidia.com/object/cuda_home_new.html

3. Електронний ресурс інформаційно-методичної підтримки навчально-виховного процесу кафедри комп'ютерних наук НУВГП : [сайт]. Режим доступу: https://sites.google.com/site/emonitorlab/disceplina

Навчальне видання

Методичні вказівки до виконання лабораторних робіт з курсу «Технології розподілених систем і паралельних обчислень» для студентів спеціальності 122 – Комп'ютерні науки

Укладачі: Андрій Анатолійович Яровий
Сергій Володимирович Барабан
Володимир Сергійович Озеранський
Євген Олександрович Шемет

Рукопис оформив А. Яровий

Редактор Т. Старічек

Оригінал-макет підготував О. Ткачук

Підписано до друку 08.07.2019.
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. 3,36.
Наклад 40 (1-й запуск 1–21) пр. Зам. № 2019-094.

Видавець та виготовлювач
Вінницький національний технічний університет,
інформаційний редакційно-видавничий центр.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
Тел. (0432) 65-18-06.
press.vntu.edu.ua;
E-mail: kivc.vntu@gmail.com.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.