

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Методичні вказівки
до виконання лабораторних робіт з дисципліни
"Високопродуктивні обчислення"
для студентів магістратури напряму підготовки
121 „Інженерія програмного забезпечення”
за освітньою програмою
„Інженерія програмного забезпечення ”
усіх форм навчання



2016

Методичні вказівки до виконання лабораторних робіт з дисципліни "Високопродуктивні обчислення" для студентів магістратури напряму підготовки 121 „Інженерія програмного забезпечення” за освітньою програмою „Інженерія програмного забезпечення” усіх форм навчання / Укл. Неласа Г.В., Кудерметов Р.К., Верещак М.І. – Запоріжжя: ЗНТУ, 2016. – 75 с.

Укладачі: Г.В. Неласа, к.т.н., доцент.
Р.К. Кудерметов, к.т.н., професор
Верещак М.І., асистент

Рецензент: В.І. Дубровін, к.т.н., професор

Відповідальний
за випуск: Г.В. Неласа, к.т.н., доцент

Затверджено:
на засіданні кафедри
програмних засобів

протокол № 1
від 16.08.2016

ЗМІСТ

Вступ.....	5
Лабораторна робота № 1. Основи програмування та засоби синхронізації в багатозадачній операційній системі WINDOWS... ..	6
1.1 Теоретичні відомості.....	6
1.2 Опис програми ThreadWar.....	12
1.3 Хід роботи.....	15
1.4 Зміст звіту.....	16
1.5 Контрольні питання.....	16
Лабораторна робота № 2. Побудова паралельних алгоритмів.....	17
2.1 Теоретичні відомості.....	17
2.2 Хід роботи.....	21
2.3 Зміст звіту.....	21
2.4 Контрольні питання.....	21
Лабораторна робота № 3. Знайомство з бібліотекою MPI.....	22
3.1 Теоретичні відомості.....	22
3.1.1 Ініціалізація бібліотеки та MPI-середовища.....	22
3.1.2 Аварійне завершення роботи MPI-середовища.....	22
3.1.3 Нормальне закриття бібліотеки.....	23
3.1.4 Інформаційні функції.....	23
3.1.5 Функції пересилки даних.....	23
3.2 Завдання до лабораторної роботи.....	24
3.2.1 Завдання 1.....	24
3.2.2 Завдання 2.....	25
3.2.3 Завдання 3.....	27
3.2.4 Завдання 4.....	27
3.3 Зміст звіту.....	27
3.4 Контрольні питання.....	28

Лабораторна робота № 4. Функції обміну в MPI	29
4.1 Теоретичні відомості.....	29
4.2 Завдання до лабораторної роботи.....	31
4.2.1 Завдання 1.....	31
4.2.2 Завдання 2.....	35
4.3 Зміст звіту.....	39
4.4 Контрольні питання.....	40
Лабораторна робота № 5. Паралельні методи інтегрування. Використання функцій колективного обміну MPI.....	41
5.1 Паралельний метод обчислення визначених інтегралів.....	41
5.1.1 Теоретичні відомості.....	41
5.1.2 Завдання.....	43
5.2 Паралельна реалізація метода Монте-Карло.....	43
5.2.1 Теоретичні відомості.....	43
5.2.2 Завдання 1.....	49
5.2.3 Завдання 2.....	50
5.3 Зміст звіту.....	51
5.4 Контрольні питання.....	51
Лабораторна робота № 6. Використання технології CUDA.....	53
6.1 Теоретичні відомості.....	53
6.2 Завдання.....	66
6.3 Зміст звіту.....	66
6.4 Контрольні питання.....	66
Література	67
Додаток А. Текст програми Thread War.....	68

ВСТУП

Лабораторні роботи спрямовані на засвоєння базових знань з паралельного програмування при використанні високопродуктивних обчислювальних систем. Вивчаються основні концепції організації паралельних обчислювальних процесів на високопродуктивних системах з застосуванням середовищ Microsoft Visual C++, CUDA та MPI/C на під управлінням операційних систем (ОС) Windows та Linux.

У лабораторних роботах розглядається створення програм для реалізації паралельних алгоритмів; вивчаються питання створення консольних багатопоточних програм для операційної системи Windows, створення потоків на мові C/C++, створення програм з використанням CUDA в рамках парадигми GPGPU, створення програм з використанням інтерфейса передачі повідомлень MPI/MPI на використанні обчислювального кластера на базі комп'ютерної мережі, засоби та варіанти передачі й отримання параметрів у потоках, передачі параметрів та збір результатів з комп'ютерів кластера.

ЛАБОРАТОРНА РОБОТА № 1

ОСНОВИ ПРОГРАМУВАННЯ ТА ЗАСОБИ СИНХРОНІЗАЦІЇ В БАГАТОЗАДАЧНІЙ ОПЕРАЦІЙНІЙ СИСТЕМІ WINDOWS.

Мета роботи: ознайомитися із застосуванням різних методів синхронізації при створенні багатопоточних додатків.

1.1 Теоретичні відомості

Мультипрограмування або **багатозадачність** (multitasking) – це спосіб організації обчислювального процесу, при якому на одному процесорі поперемінно виконуються відразу кілька програм. Ці програми спільно використовують не тільки процесор, але й інші ресурси комп'ютера: оперативну й зовнішню пам'ять, пристрої вводу-виводу, дані.

Мультипроцесорна обробка – це спосіб організації обчислювального процесу в системах з декількома процесорами, при якому декілька задач (процесів, потоків) можуть виконуватися на різних процесорах системи.

Щоб програмний код міг бути виконаний, його необхідно завантажити в оперативну пам'ять, можливо, виділити місце на диску для зберігання даних, надати доступ до пристроїв вводу-виводу. У ході виконання програми може також знадобитися доступ до інформаційних ресурсів, наприклад, файлам, портам TCP/UDP, семафорам. І, звичайно ж, неможливе виконання програми без надання їй процесорного часу, тобто часу, протягом якого процесор виконує коди даної програми.

Процес розглядається операційною системою як заявка на споживання всіх видів ресурсів, крім одного – процесорного часу. Цей останній найважливіший ресурс розподіляється операційною системою між іншими одиницями роботи – **потокami**, які й одержали свою назву завдяки тому, що вони являють собою послідовності (потоки виконання) команд.

Щоб процес що-небудь виконав, у ньому потрібно створити потік. Потоки відповідають за виконання коду, що знаходиться в адресному просторі процесу. Один процес може володіти декількома

потоками й тоді вони “одночасно” виконують код в адресному просторі процесу. Кожний потік має у своєму розпорядженні власний набір регістрів процесу й власний стек. У кожному процесі є мінімум один потік. У таблиці 1.1 наведені функції для роботи з потоками.

Таблиця 1.1 - Функції для роботи з потоками

Найменування	Опис
<code>_BeginThread</code>	Запускає потік.
<code>_beginthredex</code>	Запускає ініціалізований потік.
<code>AttachThreadInput</code>	Зв'язує обробку вводу потоку з іншим потоком, дозволяє передавати фокус вводу вікнам іншого потоку, а також використовувати загальний стан вводу.
<code>CreateRemoteThread</code>	Створює потік в іншому процесі.
<code>CreateThread</code>	Створює порожній потік у поточному процесі
<code>ExitThread</code>	Завершує роботу поточного потоку.
<code>TerminateThread</code>	Припиняє роботу потоку без виконання необхідних підготовчих процедур.
<code>GetCurrentThread</code>	Повертає дескриптор поточного потоку.
<code>GetCurrentThreadID</code>	Повертає ідентифікатор поточного потоку.
<code>GetExitCodeThread</code>	Повертає код завершення потоку.
<code>SetThreadPriority</code>	Установлює рівень пріоритету потоку.
<code>GetThreadPriority</code>	Повертає рівень пріоритету потоку.
<code>SetTheadPriorityBoost</code>	Дозволяє або забороняє динамічні зміни рівня пріоритету даного потоку.

Продовження таблиці 1.1

Найменування	Опис
GetThreadPriorityBoost	Повертає статус динамічних змін пріоритету для даного потоку.
GetThreadTimes	Повертає час, коли був створений або знищений потік, та яка кількість процесорного часу була їм використана.
SuspendThread	Тимчасово припиняє виконання потоку.
ResumeThread	Продовжує роботу потоку, роботу якого було припинено в результаті звертання до SuspendThread.
SetThreadAffinityMask	Установлює, які процесори можуть використовуватися для виконання даного потоку.
SetThreadIdealProcessor	Установлює який процесор переважно використовується для виконання даного потоку.
SwichToThread	Переключає процесор на виконання іншого потоку (невідомо, якого саме).

Для створення багатопоточного додатка в Visual C++6.0 необхідно у властивостях проекту в категорії Code Generation установити тип проекту Debug Multithreaded як показано на рис 1.1.

Забезпечення спільної роботи декількох потоків - важлива задача, яку доводиться вирішувати при розробці багатопоточних додатків. У різних ситуаціях для синхронізації потоків зручніше використовувати різні механізми операційної системи. До них належать такі об'єкти синхронізації як семафори, події, м'ютекси, можливість виконувати блоковані виклики й визначати критичні секції в тілі програм.

Якщо необхідно звернутися до змінної без побоювань, що в момент звернення операційна система передасть керування іншому

поток, можна скористатися **блокованими викликами** (interlocked call), які є частиною Windows API. Функції для роботи із блокованими викликами представлені в таблиці 1.2.

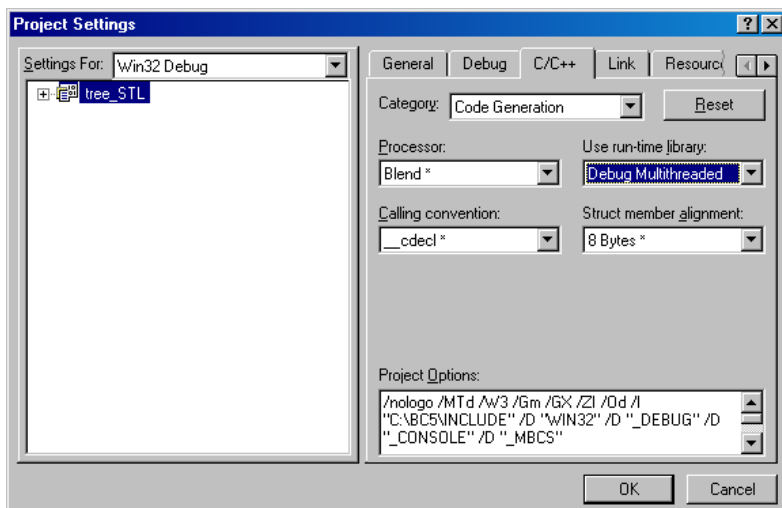


Рисунок 1.1 - Установка типу проекту Debug Multithreaded

Таблиця 1.2 - Блоковані виклики Windows API

Найменування	Опис
InterlockedDecrement	Віднімає одиницю із блокованої змінної.
InterlockedIncrement	Додає одиницю до блокованої змінної.
InterlockedExchange	Міняє місцями значення блокованої змінної й значення іншої змінної.
InterlockedExchangeAdd	Додає значення до блокованої змінної.
InterlockedCompareExchange	Порівнює значення блокованої змінної з деяким значенням, і у випадку, якщо значення рівні, міняє місцями значення блокованої змінної й значення іншої змінної.

Критична секція аналізує значення спеціальної змінної процесу, що використовується як прапорець, що запобігає виконанню деякої ділянки коду декількома потоками одночасно. Функції для роботи із критичними секціями представлені в таблиці 1.3.

Таблиця 1.3 - Функції для роботи із критичними секціями

Найменування	Опис
InitializeCriticalSection	Ініціалізує змінну типу CRITICAL_SECTION.
InitializeCriticalSectionAndSpinCount	Ініціалізує змінні типу CRITICAL_SECTION і лічильник її очікування.
EnterCriticalSection	Водить у критичну секцію й блокує інші потоки, що намагаються увійти в критичну секцію.
LeaveCriticalSection	Злишає критичну секцію, дозволяючи іншим потокам увійти до неї.
TryEnterCriticalSection	Намагається увійти в критичну секцію, повертаючи помилку у випадку якщо критична секція вже виконується будь-яким потоком.
SetCriticalSectionSpinCount	Установлює значення лічильника очікування критичної секції.
DeleteCriticalSection	Знищує змінні CRITICAL_SECTION.

Основні об'єкти, призначені для синхронізації (події, м'ютекси й семафори), багато в чому схожі. Виклик, що створює об'єкт, повертає дескриптор цього об'єкта. Потоки, що належать одному процесу, можуть використовувати один і той самий дескриптор. Якщо ж необхідно забезпечити доступ до об'єкта з декількох різних процесів, необхідно привласнити цьому об'єкту деяке ім'я.

Якщо дескриптор уже отриманий, можна визначити, чи перебуває об'єкт у сигнальному стані. Подія перебуває в сигнальному стані у випадку, якщо відповідний прапорець установлений. М'ютекс перебуває в сигнальному стані у випадку, якщо він нікому не належить. Семафор перебуває в сигнальному стані, якщо його лічильник більше нуля.

Події можуть бути мануальними й одиничними. Тип події вказується при її створенні. Подія може бути встановлена або скинута. Функції для роботи з подіями представлені в таблиці 1.4.

Таблиця 1.4 - Функції для роботи з подіями

Найменування	Опис
CreateEvent	Створити подію або відкрити існуючу подію.
OpenEvent	Відкрити подію.
SetEvent	Установити одиничну подію, перевести її в сигнальний стан. На неї реагує тільки один потік. Всі інші потоки продовжують чекати.
ResetEvent	Скидає подію.
PulseEvent	Переводить мануальну подію в сигнальний стан на період часу, поки на цю подію не прореагують всі потоки, що її очікують, потім вона скидається.
WaitForSingleObject	Очікує, поки подія не перейде в сигнальний стан.
WaitForMultiplyObject	Стежить за станом декількох подій.

Якщо потік є власником **м'ютекса**, він має право ексклюзивного використання ресурсу, що захищається цим м'ютексом. Жоден інший потік не може заволодіти м'ютексом, що вже належить одному з потоків. Разом з тим потік, що є власником м'ютекса, може спробувати стати власником м'ютекса повторно. Якщо потік

привласнював м'ютекс собі кілька разів, він повинен звільнити його таку ж кількість разів. Функції для роботи з м'ютексами представлені в таблиці 1.5.

Таблиця 1.5 - Функції для роботи з м'ютексами

Найменування	Опис
CreateMutex	Створює або відкриває м'ютекс.
OpenMutex	Відкриває м'ютекс.
ReleaseMutex	Звільняє й робить м'ютекс доступним для інших потоків.

Щораз, коли програма звертається до **семафора**, значення лічильника семафора зменшується на одиницю. Коли значення лічильника стає рівним нулю, семафор стає недоступним. Якщо потік звертається до семафора двічі, значення лічильника зменшується на дві одиниці. У цьому семафори відрізняються від м'ютексів, тому що, якщо до м'ютексу звертається кілька разів той самий потік, це вважається одним звертанням. Функції для роботи із семафорами представлені в таблиці 1.6.

Таблиця 1.6 - Функції для роботи із семафорами

Найменування	Опис
CreateSemaphore	Створює новий семафор або відкриває існуючий.
OpenSemaphore	Відкриває існуючий семафор.
ReleaseSemaphore	Додає деяке значення до лічильника семафора, роблячи його доступним для більшої кількості потоків.

1.2 Опис програми ThreadWar

Для прикладу застосування перелічених засобів синхронізації розглянемо програму «Війна програмних потоків». Текст програми ThreadWar.cpp приведений в додатку А.

Ви управляєте гарматою, що переміщається в нижній частині екрана, і стріляєте по ворогах, які літають по всьму екрану. Нічого складного, однак використання потоків і об'єктів синхронізації (подій, семафорів, м'ютексів і критичних секцій) робить цю гру незвичайною. Крім іншого в грі також використовується виклик **InterlockedIncrement**.

Мета гри проста - необхідно знищити якнайбільше супротивників. Керування переміщенням пушки здійснюється за допомогою клавіш «уліво» (←) і «вправо» (→). Щоб вистрілити, необхідно натиснути «пробіл». Вистрілити можна не більше трьох разів одночасно. Щосекунди може з'явитися новий супротивник. Згодом імовірність виникнення нового супротивника збільшується. Крім того, згодом супротивники починають рухатися швидше.

За кожного знищеного супротивника нараховується одне очко (**hit**). Якщо супротивник тікає за край екрана, вважається, що ви промахнулися (**miss**). Якщо ви упустили 30 ворогів, вважається, що гра програна. Кількість промахів і влучень відображається в заголовку вікна гри.

Коли гра починає працювати, необхідно максимізувати консольне вікно. Для цього ви можете зробити подвійне клацання на заголовку вікна. Вороги не з'являються доти, поки ви не натиснете на одну із кнопок керування курсором («уліво» або «вправо»). Якщо ви не натиснете на кнопки протягом 15 секунд, вороги не будуть більше чекати й з'являться на екрані.

Програма значно простіша, ніж можна припустити. Усе через те, що для реалізації різних об'єктів активно використовуються потоки. Потік **main** займається відображенням гармати й обробкою команд клавіатури. Цей потік також запускає інший потік, що займається створенням нових ворогів. Кожний супротивник - це теж потік. Коли ви стріляєте з гармати, ваш снаряд також стає новим потоком.

Багатопоточний підхід дозволяє зробити кожну з підпрограм надзвичайно простою. Наприклад, кожна куля стежить тільки за власним положенням і переміщенням. Той самий код управляє всіма супротивниками незалежно від того, скільки їх і де вони розташовані.

Програма використовує декілька об'єктів синхронізації. М'ютекс **screenlock** запобігає одночасному доступу декількох потоків до екрана консолі. Семафор **bulletsem** обмежує кількість куль, що одночасно перебувають на екрані трьома штуками. Подія **startevt**

блокує генерацію ворогів доти, поки користувач не натисне на клавішу або не мине 15 секунд. Нарешті, критична секція **gameover** запобігає багаторазовому виводу на екран повідомлення про закінчення гри.

На початку роботи функція **main** робить ініціалізацію множини об'єктів, включаючи об'єкти синхронізації. Дескриптор потоку зберігається таким чином, що коли гра завершується, підпрограма **score** блокує обробку вводу. Зверніть увагу, що дескриптор, що повертається функцією **GetCurrentThreadHandle**, насправді не є реальним дескриптором. Це спеціальне значення, що позначає діючий потік. Наприклад, якщо ви збережете це значення, а потім передасте його виклику **SuspendThread**, ви автоматично призупините виконання потоку, що виконав цей виклик. Щоб одержати реальний дескриптор, функція **main** звертається до виклику **DuplicateHandle**, щоб зробити копію дескриптора поточного потоку.

Перш ніж функція **main** увійде в цикл обробки вводу, вона запускає функцію **badguys** у новому потоці. Функція **badguys** очікує подію **startevt** протягом 15 секунд. Якщо подія відбувається або 15 секунд минають, починається генерація ворогів. Очікування реалізується за допомогою функції **WaitForSingleObject**.

Виконавши початкову ініціалізацію, функція **main** входить у цикл очікування клавіатурного вводу й виводу на екран консолі зображення пушки. Натискання на клавіші «уліво» (←) і «вправо» (→) реалізуються досить легко, а при натисканні на клавішу «пробіл» просто створюється новий програмний потік, у якому запускається функція **bullet**.

Потік **badguys** щосекунди з деякою ймовірністю створює нового супротивника. Для цього він перевіряє випадково отримане значення, після чого чекає протягом секунди й знову перевіряє випадкове значення. Якщо він вирішує, що необхідно створити нового супротивника, він випадковим образом визначає **у-координату** нового ворога й запускає потік **badguy**.

Якщо порівнювати з іншими потоками програми, потік **badguy** виконує трохи більш складні дії. Насамперед він визначає, чи буде супротивник рухатися зправа на ліво або зліва на право, а потім починає переміщення супротивника. У міру того як він переміщає супротивника, він постійно перевіряє, чи відбулося зіткнення з кулею. Якщо зіткнення відбулося, потік відзначає влучення й звертається до

функції **score** для того, щоб оновити інформацію про очки. Якщо супротивник ховається за кордоном екрана, потік також відзначає це й знову звертається до **score**. Функція **score** не тільки оновлює заголовок вікна, але також перевіряє умову закінчення гри.

Для відновлення змінних **hit** і **miss**, у яких зберігається кількість влучень і промахів, використовується виклик **InterlockedIncrement**. При цьому, якщо два вороги одночасно спробують змінити значення цих змінних, все спрацює нормально.

Якщо спрацює умова завершення гри, функція **score** виконує критичну секцію **gameover**. Таким чином, повідомлення про закінчення гри з'явиться на екрані тільки один раз. При цьому також відбувається завершення роботи потоку **main** і звертання до **exit** для того, щоб завершити гру. Для цього користувач повинен підтвердити завершення гри.

Потік **bullet** перевіряє, чи можна здійснити постріл. Цей потік негайно завершує роботу у випадку, якщо в позиції, куди повинна переміститися куля, уже перебуває інша куля. Це може відбутися у випадку, якщо користувач стріляє занадто швидко. Потік **bullet** також робить перевірку семафора **bulletsem**. Щоб постріл відбувся, значення семафора повинне бути відмінним від нуля. Завдяки семафору на екрані не можуть одночасно перебувати більше трьох куль. Якщо потік **bullet** не може заволодіти семафором, він завершує роботу. Таким чином, створити четверту кулю не можна - гра просто ігнорує запит.

Звичайно, написати подібну гру можна зовсім по-іншому, однак не можна не погодитися з тим, що використання багатопоточного підходу надає програмі певну елегантність. Кожна з підпрограм виразна й легка для розуміння. Разом з тим реалізувати подібну програму без використання стандартних механізмів синхронізації операційної системи було б значно складніше.

1.3 Хід роботи

1.3.1 Відкомпілювати й запустити запропонований приклад простий консольної багатопоточної комп'ютерної гри «Війна програмних потоків».

1.3.2 Розібрати текст програми ThreadWar.cpp. Знайти й підкреслити в тексті програми функції WinAPI для роботи з об'єктами

синхронізації. Пояснити їх застосування значення аргументів кожної функції.

1.3.3 Використовуючи набуті навички написати та пояснити код для вирішення задачі, запропонованої Е. Дейкстрою «про філософів, що обідають», яка формулюється так:

За круглим столом сидять п'ять філософів, що проводять час у міркуваннях та споживанні їжі. У центрі столу стоїть тарілка рису, на столі лежать п'ять паличок для їжі так, що кожен філософ опиняється між парою паличок (рис. 1.2). Коли філософ розмірковує, він не думає про їжу. Коли ж він зголоднів, то намагається взяти дві палички — ліворуч і праворуч від себе. Якщо хоча б одна паличка зайнята, філософ змушений чекати. Коли філософ взяв обидві палички, він починає їсти. Поївши, він кладе обидві палички на стіл.

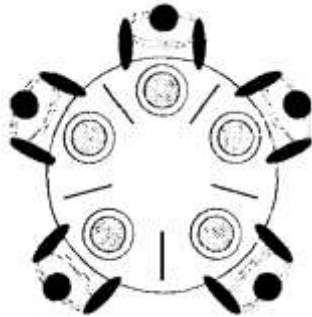


Рисунок 1.2. - Ілюстрація до задачі філософів, що обідають

Розробіть програму моделювання цієї ситуації. Зверніть увагу на те, що необхідно уникати взаємних блокувань і голодування.

1.4 Зміст звіту

1.4.1 Мета лабораторної роботи.

1.4.2 Текст програми ThreadWar.cpp.

1.4.3 Схема взаємодії потоків.

1.4.4 Опис функцій Windows API, що зустрічаються в програмі (призначення; параметри; значення, що повертається).

1.4.5 Код програми моделювання задачі «про філософів, що обідають».

1.4.6 Відповіді на контрольні питання.

1.5 Контрольні питання

1.5.1 Поняття процесу і потоку. Стани процесу.

1.5.2 Багатозадачність. Багатопроекторність.

1.5.3 Моделі взаємодії процесів. Конкуренція.

1.5.4 Моделі взаємодії процесів. Співробітництво з використанням поділу.

1.5.5 Моделі взаємодії процесів. Співробітництво з використанням зв'язку.

1.5.6 Проблеми паралельних обчислень. Взаємне блокування. Голодування. Вимоги до взаємних виключень.

1.5.7 Механізми синхронізації. Блоковані виклики.

1.5.8 Механізми синхронізації. Критичні секції.

1.5.9 Механізми синхронізації. Мьютекси.

1.5.10 Механізми синхронізації. Події.

1.5.11 Механізми синхронізації. Семафори.

ЛАБОРАТОРНА РОБОТА № 2 ПОБУДОВА ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

Мета роботи: одержати навички програмування багатопоточних додатків в WIN32 на прикладі алгоритмів модульного піднесення до степені.

2.1 Теоретичні відомості

Два цілих числа a й b називаються порівняними за модулем m (m – натуральне), якщо їх різниця $a - b$ ділиться на m без залишку. Число m називають модулем порівняння. Це записується так:

$$a \equiv b \pmod{m} . \quad (2.1)$$

Приклади:

$35 \equiv 2 \pmod{11}$, тому що $35 - 2 = 33$ ділиться на 11 ;

$25 \equiv -11 \pmod{9}$, тому що $25 - (-11) = 36$ ділиться на 9 .

Запис $a \equiv 0 \pmod{m}$ означає, що саме число ділиться на m , тобто $a = k \cdot m$.

Якщо зафіксувати деякий модуль порівняння m , то всяке натуральне число c можна єдиним образом представити у вигляді

$$c = k \cdot m + r , \quad (2.2)$$

де k – частка від ділення на m , а r – залишок, що збігається з одним із чисел $\{0, 1, 2, \dots, m - 1\}$.

Залишок r називають лишком числа c за модулем m . Запис виду (2.2), де $0 \leq r \leq m - 1$, допускає не тільки натуральні, але й будь-які цілі числа. З рівності (2.2) випливає, що $c \equiv r \pmod{m}$, тобто всяке число порівнянне зі своїм лишком за модулем m .

Обчислення $y = a^n \pmod{m}$ називається модульним піднесенням до степені.

Нехай n представлено в двійковому вигляді $n = (b_{N-1}b_{N-2}b_{N-3}\dots b_2b_1b_0)_2$, де $N = \lfloor \log_2 n \rfloor + 1$, тобто $n = (b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \dots + b_22^2 + b_12 + b_0)_2$. Тоді

$$y = a^n = \underbrace{((((a * a) \bmod m) * a) * \dots * a) \bmod m}_{n \text{ разів}}. \quad (2.3)$$

Однак обчислення по формулі (2.3) не є ефективним, особливо для великих чисел, які використовуються, наприклад, у криптографії. Існує велика кількість більш ефективних алгоритмів, у тому числі паралельних, для виконання цієї операції. Деякі з них наведені нижче.

Алгоритм 2.1 Бінарний метод

Вхід: показник степені $n \neq 0$ довжиною N біт, основа a , модуль m .

Вихід: $y = a^n \pmod{m}$.

1. Якщо $n = 1$, то $y := a \pmod{m}$; закінчити роботу алгоритму.
2. $k := N - 2$; $y := a \pmod{m}$.
3. Для i , що приймає значення від k до 0 , виконати кроки 4-5.
4. $y := y^2 \pmod{m}$.
5. Якщо i -й біт n дорівнює 1 , то $y := y \cdot a \pmod{m}$.
6. Повернення y .
7. Закінчити роботу алгоритму.

Алгоритм 2.2 Спуск Монтгомері

Вхід: показник степені $n \neq 0$ довжиною N біт, основа a , модуль m .

Вихід: $y = a^n \pmod{m}$.

1. $y1 = a \pmod{m}$, $y2 = a^2 \pmod{m}$.

2. Для i від $N-2$ до 0

Якщо i -й біт n дорівнює 1 то

$$y1 = y1 \cdot y2 \pmod{m}$$

$$y2 = y2^2 \pmod{m}$$

Інакше

$$y2 = y1 \cdot y2 \pmod{m}$$

$$y1 = y1^2 \pmod{m}$$

3. Повернення у 1.
4. Закінчити роботу алгоритму.

Приклад паралельної реалізації алгоритму Монтгомери на двухпроцесорному комп'ютері.

Таблиця 2.1 – Паралельне обчислення $Y=a^{741}$ ($741_{10}=10111001012$)

№	Процесор I	Процесор II
1	$Y1=a$	$Y2=a^2$
2	$Y1=a^2$	$Y2=a^3$
3	$Y1=a^5$	$Y2=a^6$
4	$Y1=a^{11}$	$Y2=a^{12}$
5	$Y1=a^{23}$	$Y2=a^{24}$
6	$Y1=a^{46}$	$Y2=a^{47}$
7	$Y1=a^{92}$	$Y2=a^{93}$
8	$Y1=a^{185}$	$Y2=a^{186}$
9	$Y1=a^{370}$	$Y2=a^{371}$
10	$Y1=a^{741}$	

Алгоритм 2.3 Метод гребеня

Нехай у нашій розпорядженні є P процесорів. Розділимо двійковий вигляд показника степені на $\left\lceil \frac{n}{P} \right\rceil$ блоків по P двійкових цифр кожний таким чином, що для i -го блоку встановлюються в 0 всі цифри, крім i -ой. При цьому $n = \lceil N/w \rceil$ - кількість двійкових цифр показника степені. А i -а цифра приймає значення відповідної цифри у вхідному вигляді множника. Мовою формул це виглядає в такий спосіб:

$$\left. \begin{aligned}
 s_0 &= b_0 + b_p 2^p + b_{2p} 2^{2p} + \dots + b_{\left\lfloor \frac{n}{p} \right\rfloor - 1} 2^{\left(\left\lfloor \frac{n}{p} \right\rfloor - 1\right)p} \\
 s_1 &= b_1 2 + b_{p+1} 2^{(p+1)} + b_{2p+1} 2^{(2p+1)} + \dots + b_{\left\lfloor \frac{n}{p} \right\rfloor - 1} 2^{\left(\left\lfloor \frac{n}{p} \right\rfloor - 1\right)(p+1)} \\
 s_2 &= b_2 2^2 + b_{p+2} 2^{(p+2)} + b_{2p+2} 2^{(2p+2)} + \dots + b_{\left\lfloor \frac{n}{p} \right\rfloor - 1} 2^{\left(\left\lfloor \frac{n}{p} \right\rfloor - 1\right)(p+2)} \\
 &\vdots \\
 s_{p-1} &= b_{p-1} 2^{(p-1)} + b_{2p-1} 2^{(2p-1)} + b_{3p-1} 2^{(3p-1)} + \dots + b_{\left\lfloor \frac{n}{p} \right\rfloor - 1} 2^{\left(\left\lfloor \frac{n}{p} \right\rfloor - 1\right)p}
 \end{aligned} \right\}$$

або, у загальному виді:

$$s_i = \sum_{j=0}^{\left\lfloor \frac{n}{p} \right\rfloor - 1} b_{jp+i} 2^{(jp+i)},$$

де n – кількість двійкових цифр;

$\left\lfloor \frac{n}{p} \right\rfloor$ – кількість блоків;

P – кількість процесорів, рівна кількості двійкових цифр у кожному блоці;

$s_i, 0 \leq i \leq p-1$ – часткові множники, число яких дорівнює кількості процесорів;

$$n = s_0 + s_1 + \dots + s_{p-1}.$$

Значення s_0, s_1, \dots, s_{p-1} можуть бути легко отримані з n шляхом накладення по XOR відповідної маски. Тоді $y = a^n \pmod m = a^{s_0+s_1+\dots+s_{p-1}} = a^{s_0} \times a^{s_1} \times \dots \times a^{s_{p-1}} \pmod m$, де всі a^{s_i} обчислюються кожна на своєму процесорі. Збільшення швидкості досягається за рахунок того, що часткові показники s_i мають ваги Хеммінга (кількість одиничних бітів у числі) менші, ніж вага Хеммінга вхідного показника n .

2.2 Хід роботи

2.2.1 Реалізувати бінарний метод модульного піднесення до степені.

2.2.2 Реалізувати двухпоточний варіант алгоритму Монтгомері.

2.2.3 Реалізувати метод гребеня модульного піднесення до степені. Для методу гребеня число потоків повинне вводитися із клавіатури під час виконання програми.

2.2.4 Порівняти час виконання операції модульного піднесення до степені трьома реалізованими методами.

2.3 Зміст звіту

2.3.1 Мета лабораторної роботи.

2.3.2 Графи алгоритмів.

2.3.3 Тексти розроблених програм.

2.3.4 Результати роботи програм.

2.3.5 Гістограма порівняння часу виконання розроблених програм.

2.3.6 Відповіді на контрольні питання.

2.4 Контрольні питання

2.4.1 Методи побудови паралельних алгоритмів. Послідовна та паралельна моделі програмування.

2.4.2 Парадигми паралельного програмування. Паралелізм даних. Паралелізм задач.

2.4.3 Як побудувати граф алгоритму?

2.4.4 З яких кроків складається розробка паралельного алгоритму? В чому сутність кожного кроку?

ЛАБОРАТОРНА РОБОТА № 3

ЗНАЙОМСТВО З БІБЛІОТЕКОЮ MPI

Мета роботи: вивчити каркас паралельних програм на базі функцій бібліотеки стандарту MPI, навчитися створювати паралельні програми та запускати їх на виконання

3.1 Теоретичні відомості

Існує декілька функцій, які використовуються в будь-якому додатку MPI та призначені для ініціалізації, коректного завершення та виконання необхідних дій після виявлення помилки за логікою роботи програми.

3.1.1 Ініціалізація бібліотеки та MPI-середовища

Одна з перших функцій в тілі головної програми `main()` – це функція ініціалізації MPI- середовища:

```
MPI_Init(&argc, &argv);
```

До неї передаються адреси аргументів, які стандартно отримуються самою `main()` від операційної системи та зберігають параметри командного рядка. В кінець командного рядка програми MPI-завантажувач `mpirun` додає ряд інформаційних параметрів, які необхідні `MPI_Init()`.

3.1.2 Аварійне завершення роботи MPI-середовища

Викликається, якщо програма користувача повинна завершитися через помилки часу виконання, що пов'язані з MPI:

```
MPI_Abort(опис_області_зв'язку, код_помилки_MPI);
```

Виклик `MPI_Abort()` з будь-якої задачі примусово завершує роботу всіх задач, що під'єднані до заданої області зв'язку. Якщо вказано опис `MPI_COMM_WORLD`, то буде завершено весь додаток. Використовуйте код помилки `MPI_ERR_OTHER`, якщо не знаєте, як охарактеризувати помилку в класифікації MPI.

3.1.3 Нормальне закриття бібліотеки

Останньою функцією будь-якого MPI-додатку є функція:

```
MPI_Finalize();
```

Дану функцію потрібно викликати перед поверненням з програми, тобто перед кожним оператором `return` в функції `main()`, який може бути викликаний після `MPI_Init()`.

3.1.4 Інформаційні функції

Повідомляють розмір групи (тобто загальну кількість задач, під'єднаних до її області зв'язку) та порядковий номер задачі (процесу), що її викликає:

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

3.1.5 Функції пересилки даних

Для організації простої пересилки даних між процесами використовуються функції:

```
int MPI_Send(buf, count, datatype, dest, tag, comm);
int MPI_Recv(buf, count, datatype, source, tag, comm,
status);
```

де

`void *buf` – адреса буфера, тобто початкова адреса буфера прийому (передачі). Кожний процес має власні набори даних та власний буфер прийому (передачі), тому адреси буферів кожного з процесів відрізняються одна від одної;

`int count` – розмір буфера в кількості комірок (не в байтах) типу `datatype`. Для функції передачі `MPI_Send()` вказується, скільки комірок потрібно передати, а для функції прийому `MPI_Recv()` – максимальна ємність приймального буфера. Якщо фактична довжина

повідомлення, що надійшло, є меншою – останні комірки буфера не заповнюються, якщо більшою – виникне помилка часу виконання;

`MPI_Datatype datatype` – тип комірок буфера. Функції `MPI_Send()` та `MPI_Recv()` оперують масивами однотипних даних. Для опису базових типів мови C в MPI визначені константи `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE` та інші, які мають тип `MPI_Datatype`. Їх назви утворюються префіксом `MPI_` та ім'ям відповідного типу (`int`, `char`, `double`,...), що записуються великими літерами. Користувач може зареєструвати (визначити) в MPI-додатку свої власні типи даних, наприклад структури, після чого функції MPI зможуть обробляти їх таким же чином, як і базові типи;

`int dest (source)` – номер процесу призначення (приймача), з яким відбувається обмін даними;

`int tag` – ідентифікатор повідомлення, за допомогою якого одне повідомлення відрізняється від іншого. Ідентифікатор повідомлення – ціле число від 0 до 32767, яке призначається користувачем. Важливо, щоб відправлене повідомлення з призначеним номером, було прийнято з таким же номером;

`MPI_Comm comm` – опис області зв'язку (комунікатор);

`MPI_Status *status` – статус завершення прийому. За адресою `status` міститься інформація про прийняте повідомлення, зокрема, його ідентифікатор, номер процесу-передавача, код завершення та кількість фактично прийнятих даних.

3.2 Завдання до лабораторної роботи

3.2.1 Завдання 1

Програма демонструє використання функцій MPI для ініціалізації та завершення роботи паралельної програми, а також наводить звичайний приклад використання інформаційних функцій `MPI_Comm_size()` та `MPI_Comm_rank()`.

Програма 3.1

```

#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int size, rank, i;
    MPI_Init(&argc, &argv); //Ініціалізація бібліотеки
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* Кількість
                                           процесів в додатку */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* Власний
                                           номер процесу */
    if(rank==0)
        printf("Amount of tasks=%d\n", size);
    printf("My number in MPI_COMM_WORLD=%d\n", rank);
    /* Точка синхронізації, після неї процес 0 друкує
    * аргументи командного рядка. В командному рядку
    * можуть бути параметри, що додаються
    * завантажувачем MPIRUN.
    */
    MPI_Barrier(MPI_COMM_WORLD);
    if(rank==0)
        for(puts("CommandLine for task 0:"), i=0;
            i<argc; i++)
            printf("%d: \"%s\"\n", i, argv[i]);
    MPI_Finalize(); //Всі процеси завершують виконання
    return 0;
}

```

Завдання для самостійного програмування

Варіант 1. Змінити вихідну програму так, щоб кожний процес виводив інформацію щодо парності свого номера.

Варіант 2. Змінити вихідну програму так, щоб процес з номером, що дорівнює номеру робочого місця, виводив прізвище студента, а інші процеси виводили свій порядковий номер.

3.2.2 Завдання 2

Програма демонструє використання функцій прийому та передачі, а також використання функції `MPI_Abort()`.

Програма 3.2

```

#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int size, rank, count;
    double doubleData[20];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (size != 2) // запустити тільки два процеси
    {if (rank == 0)
        printf("Only 2 tasks required instead of %d,
                abort\n", size);
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        return -1;
    }
    if (rank == 0) MPI_Send(doubleData,
        5, MPI_DOUBLE, 1, 100, MPI_COMM_WORLD);
    else
    { MPI_Recv(doubleData, 5, MPI_DOUBLE, 0, 100,
        MPI_COMM_WORLD, &status);
    // Визначаємо фактично прийняту кількість даних
    MPI_Get_count(&status, MPI_DOUBLE, &count);
    printf("Received %d elements\n", count);
    }
    MPI_Finalize();
    return 0;
}

```

Завдання для самостійного програмування

Варіант 1. Доповніть вихідну програму так, щоб процес "1" виконував посилку, а процес "0" – прийом елементів масиву цілих чисел типу long, причому, розміри буферів передачі та прийому мають дорівнювати номеру робочого місця, помноженому на 10, а кількість елементів, що передаються – номеру робочого місця, збільшеного на одиницю. В процесі "0" реалізувати виведення кількості фактично прийнятих елементів.

Варіант 2. Доповніть вихідну програму так, щоб процес "1" виконував посилку, а процес "0" – прийом елементів масиву чисел типу `float`, причому розміри буферів передачі та прийому мають дорівнювати номеру робочого місця, помноженому на 11, а кількість елементів, що передаються, дорівнює номеру робочого місця, збільшеному на одиницю. В процесі "0" вивести кількість фактично прийнятих елементів.

3.2.3 Завдання 3

Переробити програму завдання 2 так, щоб в якості буферів прийому та передачі використовувались масиви динамічної пам'яті.

3.2.4 Завдання 4

Напишіть програму, яка складається з чотирьох процесів. Процес "0" передає до процесів "1", "2" та "3" рядок з прізвищем студента. Процес "1" конкатенує прийнятий рядок з рядком, відповідним імені студента, і відсилає отриманий рядок назад. Процес "2" визначає кількість символів в прийнятому рядку та відсилає це число до нульового процесу. Процес "3" множить число, що дорівнює сумі кодів символів прийнятого рядка на число π , та відсилає отримане значення до процесу "0".

Після завершення обмінів процес "0" виводить на друк отримані від інших процесів значення.

3.3 Зміст звіту

- 3.3.1 Мета лабораторної роботи.
- 3.3.2 Опис порядку створення та запуску паралельної MPI-програми.
- 3.3.3 Текст програми, розробленої в завданні 4.
- 3.3.4 Відповіді на контрольні питання.

3.4 Контрольні питання

- 3.4.1 Який стандарт MPI та яку назву має бібліотека (реалізація) функцій MPI, які використовуються в даній лабораторній роботі?
- 3.4.2 Які функції вихідної програми завдання 1 виконуються у всіх процесах?
- 3.4.3 Що означає ідентифікатор MPI_COMM_WORLD?
- 3.4.4 Для чого використовується власний номер процесу в комунікаторі?
- 3.4.5 Напишіть фрагмент паралельної програми, який використовує значення кількості процесів в області зв'язку.
- 3.4.6 Напишіть фрагмент паралельної програми, в кожному з процесів якої створюється масив динамічної пам'яті, розмір якого дорівнює добутку номера процесу на загальну кількість процесів.
- 3.4.7 До якого класу належать програми даної лабораторної роботи: SIMD чи MIMD?

ЛАБОРАТОРНА РОБОТА № 4

ФУНКЦІЇ ОБМІНУ В MPI

Мета роботи: продовжити вивчення функцій обміну бібліотеки MPI, зокрема, функцій колективного обміну. Освоїти деякі прийоми їх використання для розподілу даних та обчислень між паралельними процесорами.

4.1 Теоретичні відомості

Для обміну даними між процесами всередині заданої області взаємодії можуть використовуватись функції колективного обміну. Ці функції повинні викликатись у всіх процесах області взаємодії.

Для розсилання одних й тих же даних від одного процесу до всіх інших використовується функція широкомовного розсилання:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type,
             int root, MPI_Comm comm);
```

де *buf* – адреса буфера;

count – кількість елементів даних у повідомленні;

type – тип даних;

root – ранг процесу, який виконує розсилання;

comm – комунікатор.

Для розподілу та збору даних використовуються, відповідно, наступні функції, вони мають однакові аргументи:

```
int MPI_Scatter(void *sendbuf, int sendcount,
               MPI_Datatype sendtype, void *rcvbuf, int rcvcount,
               MPI_Datatype rcvtype, int root, MPI_Comm comm);
```

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype sendtype, void *rcvbuf, int rcvcount,
               MPI_Datatype rcvtype, int root, MPI_Comm comm);
```

Функція розподілу `MPI_Scatter()` розсилає рівні частини буфера `sendbuf` процесу `root` всім процесам. При цьому зміст буфера процесу `root` розбивається на рівні частини за кількістю

процесів, які беруть участь в обміні, кожна з яких складається з `sendcount` елементів. Перша частина поміщається до буфера `rcvbuf` процесу, ранг якого дорівнює нулю, друга – до буфера `rcvbuf` процесу, ранг якого дорівнює одиниці і т.д. Аргументи, що належать до тієї частини списку аргументів функції, яка передається, мають силу тільки для процесу `root`.

Функція `MPI_Gather()` має зворотню дію у порівнянні з функцією `MPI_Scatter()`, тобто вона приймає та розташовує за порядком прийняті дані з процесів, які передають. При цьому параметри прийому дійсні тільки для процесу, що приймає.

Наступна функція є векторною версією функції `MPI_Scatter()` та призначена для розсилання різним процесам різної кількості елементів даних.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts,
    int *displs, MPI_Datatype sendtype, void *rcvbuf,
    int rcvcount, MPI_Datatype rcvtype, int root,
    MPI_Comm comm);
```

В `MPI_Scatter()` функції параметр `sendcounts` – масив цілих чисел, який містить кількість елементів, що передаються кожному процесу (індекс дорівнює рангу процесу). Параметр `displs` – масив цілих чисел, кожний з елементів якого задає зсув відносно початку буфера передачі. Таким чином, `displs[i]` – номер елемента буфера передачі, починаючи з якого будуть передані дані в i -й процес в кількості `sendcounts[i]` елементів.

Функція зведення `MPI_Reduce()` оброблює елементи масиву даних наступним чином. Функція бере один елемент від кожного процесу, виконує над ними задану операцію і розміщує результат у вказаному процесі. Синтаксис цієї функції:

```
MPI_Reduce(void *buf, void *result, int count,
    MPI_Datatype datatype, MPI_Op op, int root,
    MPI_Comm comm);
```

де `op` – операція зведення, яка може мати значення, що визначені попередньо, такі як `MPI_SUM`, `MPI_PROD`, `MPI_LAND`, `MPI_BAND`,

MPI_MAX і т.д. Всього 12 операцій. Крім того можна визначити свої власні операції зведення за допомогою функції `MPI_Op_create()`.

4.2 Завдання до лабораторної роботи

4.2.1 Завдання 1

Програма виконує паралельне обчислення суми ряду $\sum_{i=0}^{N-1} a_i x_i$.

Для цього до кожного з P процесів за допомогою функції `MPI_Scatter()` передається відповідна частина масивів чисел a_i та x_i , кожна з яких складається з $M = \frac{N}{P}$ елементів. Процеси обчислюють часткові суми ряду (змінна `sum`), які далі за допомогою функції `MPI_Reduce()` приводяться до загальної суми (змінна `total`).

Програма 4.1

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define N 400000
int main(int argc, char *argv[])
{int P, M;
 double x[N], a[N];
 int myrank; long i, j;
 double sum=0.0, total=0.0;
 double start_time, use_time;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &P);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

 if (myrank==0)
 {srand((unsigned)time(NULL));
  for(i=0; i<N; i++)
  { x[i]=-1.073741824+rand()*1E-9;
    a[i]=(-1.073741824+rand()*1E-9)*0.1;
  }
 }
```

```

    M=N/P;
}

if (myrank==0)
{
    start_time=MPI_Wtime();
}
MPI_Bcast (&M,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Scatter (x,M,MPI_DOUBLE,x,M,MPI_DOUBLE,0,
    MPI_COMM_WORLD);
MPI_Scatter (a,M,MPI_DOUBLE,a,M,MPI_DOUBLE,0,
    MPI_COMM_WORLD);

for (i=0;i<M;i++)
    sum+=a[i]*x[i];

MPI_Barrier (MPI_COMM_WORLD);
MPI_Reduce (&sum,&total,1,MPI_DOUBLE,MPI_SUM,0,
    MPI_COMM_WORLD);
if (myrank==0)
{
    use_time=MPI_Wtime()-start_time;
    printf ("t=%lf sec.\n",use_time);
    printf ("sum in %d procs is %.5f\n",P,total);
}
MPI_Finalize();
return 0;
}

```

Завдання для самостійного програмування

4.2.1.1 Створити програму обчислення суми ряду.

Перший варіант: $\sum_i a_i \cdot \sin(x_i)$, де $i = 0, 1, \dots, N$,

$N = 240000$, $x_i = 0.0001 \cdot i$, a_i – випадкові дійсні числа в діапазоні $(-1, 1)$. Функцію $\sin(x_i)$ обчислювати за допомогою розкладання її

в ряд Тейлора: $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \dots$

Обчислення кожного значення функції $\sin(x_i)$ можна організувати за наступним алгоритмом:

$$\text{а) } y = x; s = y; k = 1;$$

$$\text{б) } y = -\frac{x^2}{k+1 \cdot k+2} y;$$

$$\text{в) } s = s + y; k = k + 2;$$

г) якщо $k \leq K$, то перехід на п. б), інакше – кінець.

Для обчислення синуса в програмі створити окрему функцію, K нехай дорівнює 500.

Другий варіант: $\sum_i a_i \cdot \cos(x_i)$, де $i = 0, 1, \dots, N$,

$N = 240000$, $x_i = 0.0001 \cdot i$, a_i – випадкові дійсні числа в діапазоні $(-1, 1]$. Функцію $\cos(x_i)$ обчислювати за допомогою розкладання її

в ряд Тейлора: $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots + (-1)^n \frac{x^{2n}}{2 \cdot n!} + \dots$

Обчислення кожного значення функції $\cos(x_i)$ можна організувати за наступним алгоритмом:

$$\text{а) } y = 1; c = y; k = 0;$$

$$\text{б) } y = -\frac{x^2}{k+1 \cdot k+2} y;$$

$$\text{в) } c = c + y; k = k + 2;$$

г) якщо $k \leq K$, то перехід на п. б), інакше – кінець.

Для обчислення косинуса в програмі створіть окрему функцію, K нехай дорівнює 500.

4.2.1.2 Виміряйте час виконання розробленої програми на одному, двох, трьох та чотирьох процесорах для $N=240000$ та визначте прискорення обчислень для цих випадків. Дані занесіть до таблиці 4.1 (форми таблиць наведені нижче). Намалюйте графік залежності прискорення від кількості процесорів.

4.2.1.3 Дослідити та намалювати графіки залежності прискорення від кількості членів ряду при обчисленнях на чотирьох

процесорах, змінюючи кількість членів ряду від 20000 до 240000 з дискретністю 20000. Вимірювання виконуйте 3 рази з інтервалом між вимірюваннями 15 хвилин, результати вимірювань занесіть до таблиці 4.2.

4.2.1.4 Додайте до програми код для вимірювання часу, який витрачається на виконання двох функцій `MPI_Scatter()`, дослідіть та намалюйте графіки залежності цього часу від довжини масиву, що розсилається при виконанні задачі на чотирьох процесорах. Результати занесіть до таблиці 4.3.

Таблиця 4.1 – Залежність прискорення від кількості процесорів ($N=240000$, номер робочого місця №)

Дата та час	Число проц.Р	T_1	T_p	S

Таблиця 4.2 – Залежність прискорення від кількості членів ряду при $P=4$

Дата та час	N	T_1	T_4	S

Таблиця 4.3 – Залежність швидкості розсилання від довжини масиву при $P=4$

Дата та час	N	$T_{Scatter}$	S (байт/с)

4.2.2 Завдання 2

Постановка задачі. Нехай необхідно обчислити суму ряду $\sum_{i=1}^N a_i x_i^i$, де N – досить велике число та обчислення слід розподілити між P процесорами, причому, степінь чисел x_i повинна обчислюватись шляхом їх перемноження. Якщо в кожному процесорі обчислювати частину суми, яка складається з $\frac{N}{P}$ елементів вихідного ряду, то це приведе до нерівномірного навантаження процесорів. Наприклад, якщо $N=30$, $P=3$, то розбивання ряду на три частини по $\frac{N}{P}$ елементів буде: $S = \sum_{i=1}^{10} a_i x_i^i + \sum_{i=11}^{20} a_i x_i^i + \sum_{i=21}^{30} a_i x_i^i$. В результаті перший процесор виконає 55, другий – 155, а третій – 255 множень. В програмі реалізовано рівномірний розподіл навантаження обчислень цієї модельної задачі наступним чином.

В другому елементі ряду, що розглядається, необхідно виконати одне множення, в третьому – два і т.д. Загальна кількість множень будь-якого "відрізка" ряду з M елементів утворюють арифметичну прогресію, отже:

$$S_M = n_1 \cdot M + \frac{M-1 \cdot M}{2},$$

де n_1 – номер першого елемента ділянки.

Тоді кількість операцій множення, яку повинний виконати кожний процесор, можна визначити за формулою:

$$S_N = \frac{1}{P} \left(1 \cdot N + \frac{N-1 \cdot N}{2} \right) = \frac{N(N+1)}{2P}.$$

Для знаходження кількості операцій множення, яку повинний виконати перший процесор, складемо рівняння:

$$n_1 \cdot M_1 + \frac{M_1-1 \cdot M_1}{2} = \frac{N(N+1)}{2P},$$

де $n_1 = 1$.

Розв'язуючи це квадратне рівняння, знайдемо значення M_1 – кількість елементів першого відрізка. Початком другого відрізка, очевидно, буде елемент з номером $n_2 = n_1 + M_1$, тоді кількість елементів другого відрізка, призначеного для обчислення на другому процесорі, можна знайти аналогічно, розв'язуючи рівняння:

$$n_2 \cdot M_2 + \frac{M_2 - 1}{2} \cdot M_2 = \frac{N - M_1}{P},$$

і т.д.

З тим, щоб врахувати незначну неточність у визначенні довжин відрізків, пов'язану з цілочисленним розподіленням, кількість елементів відрізка для останнього процесора знайдемо із співвідношення:

$$M_p = N - M_1 - M_2 - \dots - M_{p-1}.$$

Реалізація програми. Для розподілення різної кількості елементів між процесами використовується функція `MPI_Scatterv()`, яку описано вище.

Програма 4.2

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define N 10000
int main(int argc, char *argv[])
{
    int P; // Number of processes
    double *x, *a;
    int myrank;
    long i, j;
    double q, an, nn;
    int *pn, *poffset; // pointers to numbers array and
                       // to offsets array
    int n, offset; // numbers multiples and offset
    double C;
    double mul=1.0, sum=0.0, total=0.0;
    double start_time, use_time;
```

```

MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&P);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank==0)
{
  pn=(int *)malloc(P*sizeof(int));
  if (pn==NULL)
    {printf("Not memory for n\n"); return -1;}
  poffset=(int *)malloc(P*sizeof(int));
  if (poffset==NULL) {printf("Not memory for
    offset\n"); return -1;}
  C=N*(1.0+N)/P;
  an=1;
  pn[P-1]=N;
  poffset[0]=0;
  for (i=0;i<P-1;i++)
  {
    q=(2*an-1)/2.0;
    nn=-q+sqrt(q*q+C);
    an+=nn;
    pn[i]=(int) floor(nn);
    pn[P-1]-=pn[i];
  }
  poffset[0]=0;
  for (i=1;i<P;i++)
    poffset[i]=poffset[i-1]+pn[i-1];
  for (i=0;i<P;i++)
    printf("process %d will be do %d
      multiples\n",i,pn[i]);
}

MPI_Scatter(pn,1,MPI_INT,&n,1,MPI_INT,0,
  MPI_COMM_WORLD);
MPI_Scatter(poffset,1,MPI_INT,&offset,1,MPI_INT,0,
  MPI_COMM_WORLD);

if (myrank==0)
{x=(double *)malloc(N*sizeof(*x));
  if (x==NULL) {printf("Not memory\n"); return -1;}
  a=(double *)malloc(N*sizeof(*a));
}

```

```

if(a==NULL)
    {printf("Not memory\n"); free(x); return -1;}
srand((unsigned)time(NULL));
for(i=0;i<N;i++)
    {x[i]=(-1.073741824+rand()*1E-9)*0.8;
    a[i]=(-1.073741824+rand()*1E-9)*0.01;
    }
}
if(myrank!=0)
{ x=(double *)malloc(n*sizeof(*x));
  if(x==NULL) {printf("Not memory\n"); return -1;}
  a=(double *)malloc(n*sizeof(*a));
  if(a==NULL)
    {printf("Not memory\n");free(x); return -1;}
}

if(myrank==0)
  start_time= MPI_Wtime();

MPI_Scatterv(x,pn,poffset,MPI_DOUBLE,x,n,MPI_DOUBLE,0,
MPI_COMM_WORLD);
MPI_Scatterv(a,pn,poffset,MPI_DOUBLE,a,n,MPI_DOUBLE,0,
MPI_COMM_WORLD);

for(i=0;i<n;i++)
{
  for(j=0;j<i+1+offset;j++)
    mul*=x[i];
  sum+=a[i]*mul;
  mul=1.0;
}

MPI_Barrier(MPI_COMM_WORLD);
MPI_Reduce(&sum,&total,1,MPI_DOUBLE,MPI_SUM,0,
MPI_COMM_WORLD);
if(myrank==0)
{ use_time = MPI_Wtime()-start_time;
  printf("total=%.12e used=%f sec.\n",
    total,use_time);
  free(pn);
  free(poffset);
}

```

```

free(x);
free(a);
MPI_Finalize();
return 0;
}

```

Завдання для самостійного програмування

4.2.2.1 Для перевірки результату паралельного обчислення додайте до програми код, який обчислює суму даного ряду тільки в нульовому процесі та використовує для обчислення степеня бібліотечну функцію.

4.2.2.2 Використовуючи наведені вище програми, розробіть власну програму, в якій між паралельними процесорами

розподіляється однакова кількість членів ряду $\sum_{i=1}^N a_i x_i^i$.

4.2.2.3 Виміряйте час роботи програми для трьох, досить великих значень кількості членів ряду і порівняйте його з часом вирішення цієї задачі за допомогою вихідної програми даного завдання для двох, трьох та чотирьох процесорів. Результати наведіть у вигляді таблиці.

4.3 Зміст звіту

- 4.3.1 Мета лабораторної роботи.
- 4.3.2 Тексти програм.
- 4.3.3 Графіки, таблиці та висновки, отримані при дослідженнях.
- 4.3.4 Відповіді на контрольні питання.

4.4 Контрольні питання

- 4.4.1 Чи можна оператор $M=N/P$ в програмі першого завдання виконати у всіх процесорах? Якщо так, покажіть, що треба змінити в програмі.
- 4.4.2 Використовуючи довідкові матеріали на комп'ютері, самостійно вивчіть функцію `MPI_Gatherv()`.
- 4.4.3 Що таке функція обміну з блокуванням, які функції її виконують в MPI?
- 4.4.4 Для чого використовується функція `MPI_Barrier()`?
- 4.4.5 Як в програмі, що використовує функції MPI, створити приймальний буфер, який у точності відповідає повідомленню, що передається?
- 4.4.6 Чим відрізняються алгоритми логарифмічного здовоення та рекурсивного подвоєння?
- 4.4.7 Опишіть роботу функції `MPI_Scan()`.
- 4.4.8 Як можна розподілити між процесорами обчислення множення матриць?

ЛАБОРАТОРНА РОБОТА № 5

ПАРАЛЕЛЬНІ МЕТОДИ ІНТЕГРУВАННЯ. ВИКОРИСТАННЯ ФУНКЦІЙ КОЛЕКТИВНОГО ОБМІНУ MPI

Мета роботи: продовжити вивчення функцій обміну бібліотеки MPI, зокрема, функцій колективного обміну. Вивчити деякі прийоми їх використання для розподілення даних та обчислень між паралельними процесорами.

5.1 Паралельний метод обчислення визначених інтегралів

5.1.1 Теоретичні відомості

Для тестування паралельних систем часто використовують задачу наближеного обчислення числа P (π) за допомогою відомої формули:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi. \quad (5.1)$$

Даний інтеграл можна обчислити, наприклад, за допомогою методу прямокутників:

$$\int_a^b f(x) dx \approx h \sum_{i=1}^n f\left(x_{i-\frac{1}{2}}\right), \quad (5.2)$$

де $x_i = a + ih$, $i = 1, 2, \dots, n$;

$h = \frac{b-a}{n}$ – крок дискретизації;

n – кількість підінтервалів.

Підінтегральний вираз (функція $f(x)$) в наведеній формулі методу прямокутників може обчислюватись в кожній i -й точці дискретизації незалежно, тому задача добре розпаралелюється (має внутрішній паралелізм). Можлива реалізація алгоритму у вигляді паралельної програми наводиться нижче.

Програма 5.1

```

#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int done=0, n, myid, numprocs, i;
    double PI25DT=3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    while (!done)
    {
        if(myid==0)
        {
            printf("Enter the number of intervals: (0=quit)");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if(n==0)
            break;
        h=1.0/(double)n;
        sum=0.0;
        for(i=myid+1; i<=n; i+=numprocs)
        {
            x=h*((double)i-0.5);
            sum+=4.0/(1.0+x*x);
        }
        mypi=h*sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);
        if(myid==0)
            printf("pi is approximately %.16f,
                Error is %.16f\n", pi, fabs(pi-PI25DT));
    }
    MPI_Finalize();
    return 0;
}

```

5.1.2 Завдання

5.1.2.1 Розробити послідовні алгоритм та програму обчислення числа P з можливістю вимірювання часу роботи програми для заданого n та визначення помилки апроксимації інтеграла. Для цього використовуйте функцію вимірювання часу бібліотеки MPI, як еталонне значення числа P прийміть 3.141592653589793238462643.

Дослідіть точність та швидкодію програми обчислення P для значень кількості інтервалів $10^3, 10^4, 10^5, 10^6, 10^7, 10^9$. Результати подайте у звіті у вигляді таблиці.

5.1.2.2 Реалізуйте власну паралельну програму обчислення інтеграла (1) з використанням наведеної програми 1, додайте до неї функцію обчислення часу її виконання. Визначте прискорення паралельної програми для чотирьох процесорів.

5.1.2.3 Використовуючи опановану техніку інтегрування, розробіть послідовні алгоритм та програму обчислення інтеграла у відповідності до наведених нижче *варіантів*. В програмі реалізуйте введення кількості інтервалів з клавіатури. Дослідіть точність та швидкодію програми для значень кількості інтервалів $10^3, 10^4, 10^5, 10^6, 10^7, 10^9$. Результати подайте у звіті у вигляді таблиці.

Перший варіант. Обчислити інтеграл $\int_0^{\frac{\pi}{2}} \sin^2(x) dx$.

Другий варіант. Обчислити інтеграл $\int_0^{\pi} \cos^2(x) dx$.

5.2 Паралельна реалізація метода Монте-Карло

5.2.1 Теоретичні відомості

При інтегруванні складних функцій та складній формі області інтегрування можна використовувати метод Монте-Карло, для якого в деяких випадках необхідно менше обчислень підінтегральної функції.

Пояснимо суть методу на простому прикладі.

Площа кола S_C та площа описаного навколо неї квадрата S_R обчислюються за формулами:

$$S_C = \pi r^2, S_R = 4r^2.$$

За допомогою генератора випадкових чисел будемо генерувати координати точок, що рівномірно розподілені усередині квадрата R , одночасно підраховуючи кількість точок, які опинились в межах кола C (рис.5.1). Припустимо, кількість згенерованих точок дорівнює n , а кількість точок, які опинились в межах кола дорівнює m ($m \leq n$). Тоді, при досить великому n відношення m до n буде пропорційним відношенню площі кола S_C до площі квадрата S_R (внаслідок рівномірності розподілу), тобто:

$$\lim_{n \rightarrow \infty} \frac{m}{n} \approx \frac{S_C}{S_R} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}, \quad (5.3)$$

отже:

$$\pi = 4 \lim_{n \rightarrow \infty} \frac{m}{n} \quad (5.4)$$

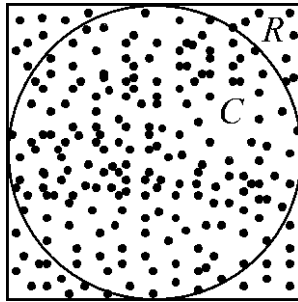


Рисунок 5.1 – Обчислення числа π методом Монте-Карло

Отримане значення числа π є випадковим. Виконаємо серію N незалежних реалізацій обчислення числа π таким способом. Позначимо отримані значення числа π як $\xi_1, \xi_2, \dots, \xi_N$, а їх середнє

$$\bar{\xi}_N = \frac{\xi_1 + \xi_2 + \dots + \xi_N}{N}.$$

Тоді при досить великому N для похибки середнього обчисленого значення числа π внаслідок центральної граничної теореми і згідно з відомим правилом "трьох сигм" можна записати

$$P\left(|\pi - \bar{\xi}_N| < 3 \frac{\sigma}{\sqrt{N}}\right) \approx 0.997, \quad (5.5)$$

тобто нерівняння $|\pi - \bar{\xi}_N| < 3 \frac{\sigma}{\sqrt{N}}$ виконується з ймовірністю, досить близькою до одиниці ($P \approx 0.997$), де σ^2 – дисперсія випадкової величини ξ_i , $i = \overline{1, N}$.

Розглянемо інтеграл

$$I = \int_0^1 f(x) dx. \quad (5.6)$$

Нехай η – рівномірно розподілена на відрізку $[0, 1]$ випадкова величина, тобто, її щільність розподілу ймовірностей:

$$p_\eta = \begin{cases} 1, & x \in [0, 1] \\ 0, & x \notin [0, 1] \end{cases}$$

Тоді $\xi = f(\eta)$ також буде деякою випадковою величиною, до того ж її математичне сподівання

$$M \xi = \int_0^1 f(x) p_\eta(x) dx = \int_0^1 f(x) dx = I$$

Таким чином,

$$I = \int_0^1 f(x) dx \approx \bar{\xi}_N = \frac{1}{N} \sum_{i=1}^N f(\eta_i), \quad (5.7)$$

де η_i – незалежні реалізації рівномірно розподіленої на відрізку $[0, 1]$ випадкової величини η .

Якщо ε – задана припустима похибка обчислення інтеграла, то відповідно до (5.5) з ймовірністю, близькою до одиниці, має бути

$$\varepsilon \approx 3 \frac{\sigma}{\sqrt{N}}, \quad (5.8)$$

звідки випливає, що для досягнення похибки порядку ε необхідно обчислити

$$N \approx c\varepsilon^{-2} \approx O(\varepsilon^{-2}) \quad (5.9)$$

значень функції f .

Кратні інтеграли обчислюють аналогічно:

$$I = \int_0^1 \int_0^1 f(x, y) dx dy \approx \xi_N = \frac{1}{N} \sum_{i=1}^N f(\eta_i, \theta_i), \quad (5.10)$$

де η_i , θ_i – незалежні реалізації рівномірно розподілених на $[0, 1]$ випадкових величин η , θ .

Інтегрування методом Монте-Карло також зручно використовувати, якщо необхідно обчислити кратний інтеграл по деякій області Ω , яка має складну форму.

В цьому випадку припускають

$$I = \iint_{\Omega} f(P) dP = \iint_R f^*(P) dP, \quad (5.11)$$

де R – квадрат, який містить задану область Ω (рис. 5.2),

$$f^*(P) = \begin{cases} f(P), & P \in \Omega, \\ 0, & P \notin \Omega. \end{cases}$$

Далі використовують формулу (5.10).

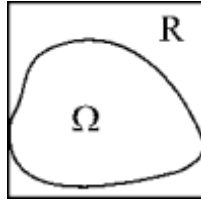


Рисунок 5.2 – Ω – область інтегрування.

Розв'язання задач методом Монте-Карло добре розпаралелюється, оскільки генерувати випадкові числа та обчислювати від них функції можна незалежно в різних процесорах. При цьому, як правило, загальна кількість необхідних обчислень (генерування випадкових чисел, обчислення функцій та їх підсумовування) рівномірно розподіляють між процесорами.

Нижче наведена програма для обчислення числа π методом Монте-Карло за формулою (5.4).

Програма 5.2

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <time.h>

#define n_in_proc 1000000
#define RAND_MAX 2147483647

int main(int argc, char *argv[])
{
    time_t t;
    int rank, numprocs, i, N_in_proc;
    long j, in_circle=0L, total_in_circle, total;
    double x, y, R=0.5, approx,
           pi=3.141592653589793238462643;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==0) N_in_proc=atoi(argv[1]);
    MPI_Bcast(&N_in_proc, 1, MPI_INT, 0, MPI_COMM_WORLD);
    for(i=0; i<N_in_proc; i++)
```

```

{
  srand((unsigned)time(&t));
  for (j=0L;j<n_in_proc;j++)
  {
    x=rand()/(double)RAND_MAX-0.5;
    y=rand()/(double)RAND_MAX-0.5;
    if(x*x+y*y<R*R)
      in_circle++;
  }
  MPI_Reduce(&in_circle,&total_in_circle,1,MPI_INT,
            MPI_SUM,0,MPI_COMM_WORLD);
  if(rank==0)
  {
    total=n_in_proc*(i+1)*numprocs;
    approx=4.0*((double)total_in_circle/total);
    printf("pi=%.16f; error=%.16f, points=%ld\n",
          approx, fabs(pi-approx), total);
  }
}
MPI_Finalize();
return 0;
}

```

В даній програмі кожний процес виконує N_{in_proc} реалізацій обчислення числа π , кількість реалізацій визначається з командного рядку. В кожній реалізації n_{in_proc} раз генеруються "випадкові" значення координат x та y точки всередині квадрата $[-0.5, 0.5] \times [-0.5, 0.5]$ та визначається кількість точок (змінна in_circle), які опинились в межах кола з радіусом 0.5.

5.2.2 Завдання 1

5.2.2.1 Програму 5.2 доповніть функцією обчислення часу її виконання та функцією обчислення дисперсії σ^2 випадкової величини значення π

$$\sigma^2 \approx \frac{1}{N-1} \sum_{i=1}^N \xi_i - \bar{\xi}_N^2, \quad (5.12)$$

причому, оскільки середнє значення $\bar{\xi}_n$ заздалегідь не відомо, зручніше використати формулу

$$\sigma^2 \approx \frac{1}{N+1} \left[\sum_{i=1}^N \xi_i^2 - \frac{1}{N} \left(\sum_{i=1}^N \xi_i \right)^2 \right]. \quad (5.13)$$

5.2.2.2 Використовуючи формули (5.9) визначити N таке, щоб похибка ϵ обчислення числа π наближено дорівнювала 10^{-4} .

5.2.2.3 Підібрати для знайденого N параметри `N_in_proc` та `n_in_proc` для випадку використання двох та чотирьох процесів і за допомогою доповненої програми обчислити число π , показати, що виконуться відношення (5.8).

5.2.2.4 Визначити прискорення паралельної програми для двох та чотирьох процесорів.

5.2.2.5 Проведені розрахунки та експериментальні результати навести у звіті.

Програма 5.3 реалізує послідовний алгоритм обчислення інтеграла (5.1) методом Монте-Карло.

Програма 5.3

```
#include <stdio.h>
#include <math.h>
#include <time.h>

#define N 1000000000L
#define RAND_MAX 2147483647L
```

```

int main( int argc, char *argv[] )
{
    time_t t;
    long i;
    double x, mc_pi=0.0,
           pi=3.141592653589793238462643;
    srand( (unsigned) time(&t));
    mc_pi=0.0;
    for(i=0L;i<N;i++)
    {
        x=rand()/(double)RAND_MAX;
        mc_pi+=(4.0/(1.0+x*x))/N;
    }
    printf("mc_pi=%.16lf  err=%.15lf
           points=%ld\n", mc_pi, fabs(pi-mc_pi), i);
    return 0;
}

```

5.2.3 Завдання 2

5.2.3.1 Розробити паралельну програму інтегрування методом Монте-Карло, використовуючи програму 5.3.

5.2.3.2 Допрацювати програму та провести дослідження аналогічно завданню за п. 5.2.2.

5.2.3.3 Виконати наступні варіанти завдання.

Варіант 1: Розробити паралельну програму для обчислення кратного інтеграла $\iint_S f(x, y) \, dx dy$, де $f(x, y) = xy$, S – область, яка міститься між $x_1 = 0$, $x_2 = 1$ та $y_1(x) = x^2$, $y_2(x) = \sqrt{x}$ (точне значення інтеграла дорівнює $\frac{1}{12}$).

Варіант 2: Розробити паралельну програму для обчислення кратного інтеграла $\iint_S f(x, y) \, dx dy$, де $f(x, y) = xy$, S – чверть кола

$x^2 + y^2 \leq R^2$, $x \geq 0$, $y \geq 0$ (точне значення інтеграла дорівнює $\frac{1}{8}R^4$).

5.3 Зміст звіту

- 5.3.1 Мета лабораторної роботи.
- 5.3.2 Тексти програм.
- 5.3.3 Результати розрахунків.
- 5.3.4 Відповіді на контрольні питання.

5.4 Контрольні питання

- 3.1 Проаналізуйте програму 5.1. Нехай інтервал інтегрування складається з 10 підінтервалів (0, 1, 2 і т.д.), а кількість процесорів дорівнює чотирьом. Назвіть номери підінтервалів, в яких обчислює підінтегральну функцію кожний з процесорів.
- 3.2 Чи залежить прискорення обчислення числа π за допомогою програми 1 від кількості підінтервалів?
- 3.3 Порівняйте точності обчислень числа π за допомогою програм 5.1 та 5.3. Які Ваші висновки щодо точності реалізованих в програмах 5.1 та 5.3 методів?
- 3.4 Як в MPI визначити розмір буфера, необхідного для прийому повідомлень?
- 3.5 Що таке групи процесів в MPI? За допомогою яких функцій бібліотеки MPI їх можна створити?
- 3.6 Як здійснити обмін даними між процесами групи?
- 3.7 Поясніть призначення і роботу функції
`MPI_Comm_split(MPI_Comm comm, int split, int key, MPI_Comm *newcomm);`
- 3.8 Чи можуть декілька комунікаторів в одній паралельній програмі мати однакові ідентифікатори?

ЛАБОРАТОРНА РОБОТА № 6

ВИКОРИСТАННЯ ТЕХНОЛОГІЇ CUDA

Мета роботи: отримати навички програмування під технологію CUDA.

6.1 Теоретичні відомості

Основні поняття технології CUDA

CUDA будується на концепції, що GPU (званий пристроєм, device) виступає в ролі масивно-паралельного співпроцесора до CPU (званому host). Програма на CUDA використовує як CPU, так і GPU. При цьому звичайний (послідовний, тобто не паралельний) код виконується на CPU, а для масивно-паралельних обчислень відповідний код виконується на GPU як набір одночасно виконуються ниток (потоків, threads). Таким чином, GPU розглядається як спеціалізований обчислювальний пристрій, який:

- є співпроцесором до CPU;
- володіє власною пам'яттю;
- має можливість паралельного виконання великої кількості окремих ниток;

При цьому дуже важливо розуміти, що між нитками на CPU і нитками на GPU є принципові відмінності:

- нитки на GPU мають вкрай невисоку ціну створення, управління і знищення (контекст - нитки мінімальний, все регістри розподілені заздалегідь);
- для ефективної загрузки GPU необхідно використовувати багато тисяч окремих ниток, в той час як для CPU зазвичай досить 10-20 ниток.

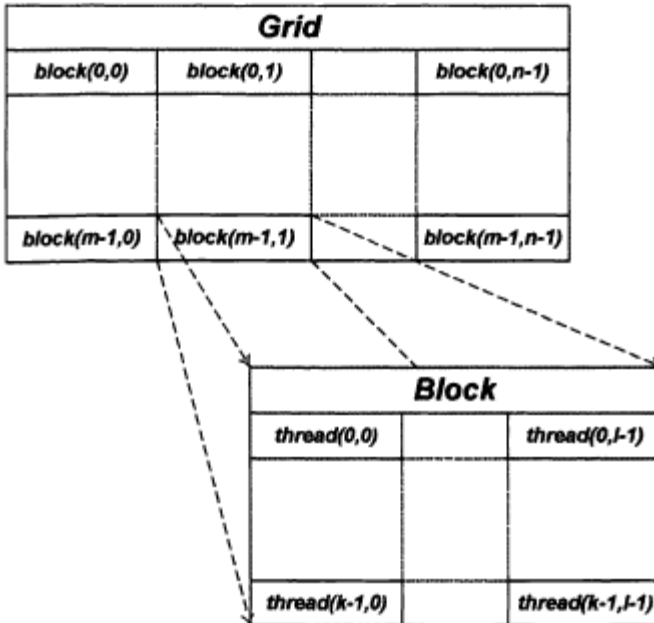
Особливості паралельної моделі CUDA

Ключовим поняттям програм, написаних на CUDA є ядро. Ядро - це функція, код якої буде паралельно виконуватися множиною потоків (званих нитками) на графічному процесорі. При цьому кожна

нитка, яка виконує дане ядро, отримує в своє розпорядження свій блок оброблюваних даних.

Важливим моментом є те, що хоча подібний підхід дуже схожий на роботу з SIMD моделлю, є і принципові відмінності (компанія Nvidia використовує термін SIMT Single Instruction, Multiple Thread). Нитки розбиваються на групи по 32 нитки, звані warp'ами. Тільки нитки в межах одного warp'a виконуються фізично одночасно. Нитки з різних warp'ів можуть перебувати на різних стадіях виконання програми. При цьому управління warp'ми прозора здійснює сам GPU. Для вирішення завдань CUDA використовує дуже велику кількість паралельно виконуваних ниток, при цьому зазвичай кожній нитці відповідає один елемент обчислюваних даних. Всі запущені на виконання нитки організовані в наступну ієрархію.

Верхній рівень ієрархії - сітка (grid) - відповідає всім ниткам, які виконують дане ядро. Верхній рівень представляє з себе одновимірний або двовимірний масив блоків (block).



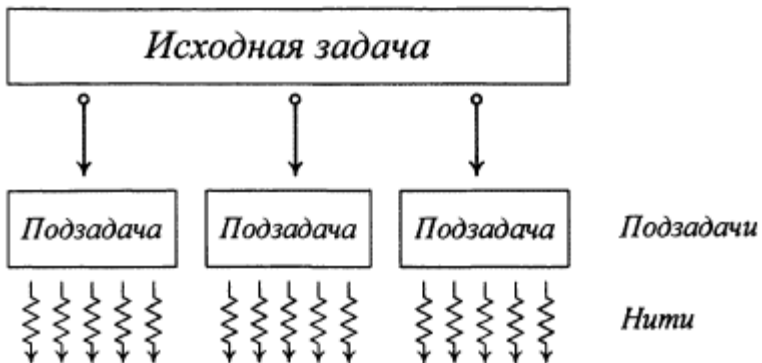
Кожен блок - це одновимірний, двовимірний або тривимірний масив ниток (thread). При цьому всі блоки, що утворюють сітку, мають однакову розмірність і розмір.

Кожен блок в сітці має свою адресу, що складається з одного або двох невід'ємних цілих чисел (індекс блоку в сітці). Аналогічно кожна нитка всередині блоку також має свою адресу - одне, два або три невід'ємних цілих числа, які задають індекс нитки всередині блоку.

Оскільки одне й те ж ядро виконується одночасно дуже великим числом ниток, то для того, щоб ядро мало можливість однозначно визначити номер нитки (тобто і елемент даних, який потрібно обробляти), використовуються вбудовані змінні **threadIdx** і **blockIdx**. Кожна з цих змінних є тривимірним цілочисельним вектором. Зверніть увагу, що вони доступні тільки для функцій, які виконуються на GPU, для функцій, що виконуються на CPU, вони не мають сенсу.

Також ядро може отримати розміри сітки і блоку через вбудовані змінні **gridDim** і **blockDim**.

Подібне розділення всіх ниток є ще одним загальним прийомом використання CUDA - вихідне завдання розбивається на набір окремих підзадач, що вирішуються незалежно одна від одної. Кожній такій підзадачі відповідає свій блок ниток.



При цьому кожна підзадача спільно вирішується всіма нитками свого блоку. Розбиття ниток на warp'i відбувається окремо для кожного блоку; таким чином, всі нитки одного warp'a завжди

належать одному блоку. При цьому нитки можуть взаємодіяти між собою тільки в межах блоку. Нитки різних блоків взаємодіяти між собою не можуть.

Подібний підхід є вдалим компромісом між необхідністю забезпечити взаємодію ниток між собою і вартістю забезпечення такої взаємодії - забезпечити можливість взаємодії кожної нитки з кожної було б занадто складно і дорого. Існують лише два механізми, за допомогою яких нитки всередині блоку можуть взаємодіяти одна з одною:

- колективна (shared) пам'ять;
- бар'єрна синхронізація.

Кожен блок отримує в своє розпорядження певний обсяг швидкої пам'яті, що розділяється (shared), яку можуть спільно використовувати всі нитки блоку. Оскільки нитки блоку необов'язково виконуються фізично паралельно (тобто ми маємо справу не з чистою SIMD архітектурою, а має місце прозоре управління нитками), то для того, щоб не виникало проблем з одночасною роботою з shared пам'яттю, необхідний деякий механізм синхронізації ниток блоку.

CUDA пропонує досить простий спосіб синхронізації - так звана бар'єрна синхронізація. Для її здійснення використовується виклик вбудованої функції `_syncthreads ()`, яка блокує нитки блоку до тих пір, поки всі нитки блоку не ввійдуть в цю функцію. Таким чином, за допомогою `_syncthreads ()` ми можемо організувати «бар'єри» всередині ядра, які гарантують, що якщо хоча б одна нитка пройшла такий бар'єр, то не залишилося жодної за бар'єром (хто не пройшов його).

Ієрархія пам'яті CUDA

Однією з серйозних відмінностей між GPU і CPU є організація пам'яті і робота з нею. Зазвичай більшу частину CPU займають кеші різних рівнів. Основна частина пам'яті GPU відведена на обчислення. Як наслідок, на відміну від CPU, де є всього один тип пам'яті з декількома рівнями кешування в самому CPU, GPU володіє складнішою, але в той же час набагато більш документованою структурою пам'яті. Чисто фізично пам'ять GPU можна розділити на DRAM і на пам'ять, розміщену безпосередньо на GPU (точніше, в

потоків мультіпроцесорах). Однак класифікація пам'яті в CUDA не обмежується її чисто фізичним розташуванням (Табл. 6.1).

Таблиця 6.1 — Характеристики пам'яті GPU

Тип пам'яті	Розташування	Кешується	Доступ	Рівень доступу	Час життя
Регістри	MS	Ні	R/w	Per-thread	Нитка
Локальна	DRAM	Ні	R/w	Per-thread	Нитка
Що розділяється	MS	Ні	R/w	Нитки блоку	Блок
Глобальна	DRAM	Ні	R/w	Все нитки и CPU	Виділяється CPU
Константна	DRAM	Так	R/o	Все нитки и CPU	Виділяється CPU
Текстурна	DRAM	Так	R/o	Все нитки и CPU	Виділяється CPU

Як видно з наведеної таблиці, частина пам'яті розташована безпосередньо в кожному з потоків мультіпроцесорів (регістри і пам'ять, що розділяється), частина пам'яті розміщена в DRAM.

Найбільш простим видом пам'яті є регістрова пам'ять (регістр). Кожен потіковий мультіпроцесор містить 8192 або 16384 32-бітових регістрів (для позначення всіх регістрів потікового мультіпроцесора використовується термін register file).

Наявні регістри розподіляються між нитками блоку на етапі компіляції (і, відповідно, впливають на кількість блоків, які може виконувати один мультіпроцесор). Кожна нитка отримує в своє монопольне користування кілька регістрів, які доступні як на читання, так і на запис (read / write). Нитка не має доступу до регістрів інших ниток, але свої регістри доступні їй протягом виконання даного ядра. Оскільки регістри розташовані безпосередньо в потіковому мультіпроцесорі, то вони мають максимальну швидкість доступу. Якщо наявних регістрів не вистачає, то для розміщення локальних даних (змінних) нитка використовується так звана локальна пам'ять, розміщена в DRAM. Тому доступ до локальної пам'яті характеризується дуже високою латентністю - від 400 до 600 тактів.

Наступним типом пам'яті в CUDA є так звана колективна (shared) пам'ять. Ця пам'ять розташована безпосередньо в потіковому

мультипроцесорі, але вона виділяється на рівні блоків - кожен блок отримує в своє розпорядження одну і ту ж кількість пам'яті, що розділяється. Всього кожен мультипроцесор містить 16 Кбайт пам'яті, що розділяється, і від того, скільки такої пам'яті потрібно блоку, залежить кількість блоків, що можуть бути запущені на одному мультипроцесорі. Пам'ять, що розділяється має дуже невелику латентність (доступ до неї може бути настільки ж швидким, як і доступ до регістрів), доступна всім ниткам блоку, як на читання, так і на запис.

Глобальна пам'ять - це звичайна DRAM-пам'ять, яка виділяється за допомогою спеціальних функцій на CPU. Всі нитки сітки можуть читати і писати в глобальну пам'ять. Оскільки глобальна пам'ять розташована поза GPU, то природно, що вона має високу латентність (від 400 до 600 тактів). Правильне використання глобальної пам'яті є одним з краєугольних каменів оптимізації в CUDA.

Константна і текстурна пам'ять, хоч і розташовані в DRAM, проте кешуються, тому швидкість доступу до них може бути дуже високою (набагато вище швидкості доступу до глобальної пам'яті). Обидва цих типу пам'яті доступні відразу всім ниткам сітки, але тільки на читання, запис в них може здійснювати тільки CPU за допомогою спеціальних функцій. Загальний обсяг константної пам'яті обмежений 64 Кбайтами.

Текстурна пам'ять дозволяє отримати доступ до можливостей GPU по роботі з текстурами.

Введення в CUDA API. Розширення мови C

Програми для CUDA (відповідні файли зазвичай мають розширення .cu) пишуться на «розширеному» C і компілюються за допомогою команди `nvcc`. Введені в CUDA розширення мови C складаються з:

- специфікаторів функцій, що показують, де буде виконуватися функція і звідки вона може бути викликана;
- специфікаторів змінних, які задають тип пам'яті, що використовується для цих змінних;
- директиви, що служить для запуску ядра, яка задає як дані, так і ієрархію ниток;

- вбудованих змінних, що містять інформацію про поточну нитку;
- runtime, що включає в себе додаткові типи даних. (Табл. 6.2)

Таблиця 6.2 — Специфікатори функцій та змінних

Специфікатор	Функція виконується на	Функція може бути викликана з
<code>__device__</code>	Device (GPU)	Device (GPU)
<code>__global__</code>	Device (GPU)	Host (CPU)
<code>__host__</code>	Host (CPU)	Host (CPU)

Специфікатори `__host__` і `__device__` можуть бути використані разом (це означає, що відповідна функція може виконуватися як на GPU, так і на CPU - відповідний код для обох платформ буде автоматично згенерований компілятором). Специфікатори `__global__` і `__host__` не можуть бути використані разом.

Специфікатор `__global__` позначає ядро, і відповідна функція повинна повертати значення типу `void`.

На функції, що виконуються на GPU (`__device__` і `__global__`), накладаються такі обмеження:

- не можна брати їх адресу (за винятком `__global__` функцій);
- не підтримується рекурсія;
- не підтримуються `static`-змінні всередині функції;
- не підтримується змінне число вхідних аргументів.

Для завдання розміщення в пам'яті GPU змінних використовуються наступні специфікатори `__device__`, `__constant__` і `__shared__`. На їх використання також накладається ряд обмежень:

- ці специфікатор не можуть бути застосовані до полів структури (`struct` або `union`);
- відповідні змінні можуть використовуватися тільки в межах одного файлу, їх не можна оголошувати як `extern`;
- запис в змінні типу `__constant__` може здійснюватися тільки CPU за допомогою спеціальних функцій;
- `__shared__` змінні не можуть бути ініціалізовані при оголошенні.

Додані типи

У мову додані 1/2/3/4-міни вектори з базових типів (`char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long`

long, float и double): char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, short4, ushort1, ushort2, ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4, long1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, float1, float2, float3, float4, longlong1, longlong2, double1 double2.

Звернення до компонентів вектора йде поіменно: x, y, z і w. Для створення значень-векторів заданого типу служить конструкція виду `make_<typeName>`.

```
int2 a = make_int2(1, 7);
```

Для цих типів не підтримуються векторні покомпонентні операції, тобто не можна просто скласти два вектора за допомогою оператора «+» - це необхідно явно робити для кожної компоненти.

Також доданий тип `dim3`, що використовується для завдання розмірності. Цей тип заснований на типі `uint3`, але має при цьому нормальний конструктор, що ініціалізує всі не задані компоненти одиницями.

```
dim3 blocks(16, 16); // Эквивалентно blocks(16, 16, 1).
```

```
dim3 grid(256); // Эквивалентно grid(256, 1, 1).
```

Додані змінні

У мову додані наступні спеціальні змінні:

- `gridDim` - розмір сітки (має тип `dim3`);
- `blockDim` - розмір блоку (має тип `dim3`);
- `blockIdx` - індекс поточного блоку в сітці (має тип `uint3`);
- `threadIdx` - індекс поточної нитки в блоці (має тип `uint3`);
- `warpSize` - розмір warp'a (має тип `int`).

Директива виклику ядра

Для запуску ядра на GPU використовується наступна конструкція:

```
kernelname<<<Dg,Db,Ns,S>>> (args);
```

Тут:

- kernelname - це ім'я відповідної `__global__` функції;
- через `Dg` позначена змінна (або значення) типу `dim3`, що задає розмірність і розмір сітки (в блоках);
- змінна (або значення) `Db` - типу `dim3` задає розмірність і розмір блоку (в нитках);
- необов'язкова змінна (або значення) `Ns` типу `size_t` задає додатковий обсяг пам'яті в байтах, яка повинна бути динамічно виділена кожному блоку (до вже статично виділеної пам'яті, що розділяється), якщо не задано, то використовується значення 0;
- змінна (або значення) `S` типу `cudaStream_t` задає потік (CUDA stream), в якому повинен відбутися виклик, за замовчуванням використовується потік 0;
- через `args` позначені аргументи виклику функції `kernelname` (їх може бути кілька).

Наступний приклад запускає ядро з ім'ям `myKernel` паралельно на `n` нитках, використовуючи одномірний масив з двовимірних (16x16) блоків ниток, і передає на вхід ядру два параметри: `a` та `n`. При цьому кожному блоку додатково виділяється 512 байт пам'яті. Запуск проводиться в потоці `myStream`:

```
myKernel<<<dim3(n/256), dim3(16,16), 512, myStream>>> (a, n);
```

Додані функції

CUDA підтримує всі математичні функції зі стандартної бібліотеки мови C. Однак при цьому слід мати на увазі, що більшість стандартних математичних функцій використовують числа з подвійною точністю (`double`). Однак, оскільки для сучасних GPU операції з `double`-числами виконуються повільніше, ніж операції з `float`-числами, то краще там, де це можливо, використовувати `float`-аналоги стандартних функцій. Так, `float`-аналогом функції `sin` є функція `sinf`.

Крім того, CUDA надає також спеціальний набір функцій зниженою точності, які забезпечують ще більшу швидкодію. Таким аналогом для функції обчислення синуса також є функція `sinf`.

Для ряду функцій можна задати необхідний спосіб округлення, який задається за допомогою одного з наступних суфіксів:

- gn - округлення до найближчого;
- gz - округлення до нуля;
- gu - округлення вгору;
- rd - округлення вниз.

Крім ряду оптимізованих функцій для роботи з числами з плаваючою точкою, також є ряд «швидких» функцій для роботи з цілими числами.

Основи CUDA host API

CUDA надає в розпорядження програміста ряд функцій, які можуть бути використані тільки CPU (так званий CUDA host API). Ці функції відповідають за:

- управління GPU;
- роботу з контекстом;
- роботу з пам'яттю;
- роботу з модулями;
- управління виконанням коду;
- роботу з текстурами;
- взаємодія з OpenGL і Direct3D.

CUDA host API виступає в двох формах:

- низькорівневий CUDA driver API;
- високорівнева CUDA runtime API (реалізований через CUDA driver API).

Ці API є взаємовиключними. У своїй програмі ви можете працювати тільки з одним з них.

CUDA driver API

Низькорівневий API дає більше можливостей програмісту, але й вимагає більшого обсягу коду. Даний API реалізований в динамічній бібліотеці `nvcuda`, і всі імена в ньому починаються із префікса `cu`.

Слід мати на увазі, що у кожної функції CUDA runtime API є прямиий аналог в CUDA driver API, тобто перехід з CUDA runtime API на CUDA driver API не дуже складний, зворотнє в загальному випадку невірно.

CUDA driver API має зворотну сумісність з більш ранніми версіями. До числа недоліків цього API відносяться більший обсяг коду і необхідність явних налаштувань, вимога явної ініціалізації.

CUDA runtime API

Це високорівневий API, до того ж CUDA runtime API не вимагає явної ініціалізації - вона відбувається автоматично при першому виклику будь-якої його функції. Даний API реалізований в динамічній бібліотеці cudart, все імена починаються із префікса cuda. Одним з плюсів даного API є можливість використання додаткових бібліотек (CUFFT, CUBLAS, CUDPP та інших).

Ми будемо використовувати саме цей API як більш простий. При необхідності код, заснований на ньому, може бути переписаний на CUDA driver API, оскільки останній містить повні аналоги всіх функцій CUDA runtime API.

Розглянемо основні елементи CUDA runtime API на прикладі наступної програми, яка виконує операцію складання матриць:

```
// Я д р о
__global__ void vectorAdd(float a*, float *b, float *c) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    c [index] = a [index] + b[index];
}

int main() {

    const unsigned int blockSize = 512;
    const unsigned int numBlocks = 3;
    const unsigned int numItems = numBlocks * blockSize;

    cudaSetDevice(0);

    float *a = new float[numItems];
    float *b = new float[numItems];
```

```

float *c = new float[numItems];

// Инициализация массивов
// ...

float *aDev, *bDev, *cDev;

cudaMalloc((void**)&aDev, numItems * sizeof(float));
cudaMalloc((void**)&bDev, numItems * sizeof(float));
cudaMalloc((void**)&cDev, numItems * sizeof(float));

cudaMemcpy(aDev, a , numItems * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(bDev, b, numItems * sizeof(float), cudaMemcpyHostToDevice);

vectorAdd<<<nurnBlocks,blockSize>>> (aDev, bDev, cDev);

cudaMemcpy((void*)c , cDev, numItems * sizeof(float),
cudaMemcpyDeviceToHost);

delete[] a;
delete[] b;
delete[] c;

cudaFree(aDev);
cudaFree(bDev);
cudaFree(cDev);
}

```

Кожна функція CUDA runtime API (крім запуску ядра) повертає значення типу `cudaError_t`. При успішному виконанні функції

повертається значення `cudaSuccess`, в іншому випадку повертається код помилки.

Отримати опис помилки у вигляді рядка по її коду можна за допомогою функції `cudaGetErrorString`:

```
char* cudaGetErrorString(cudaError_t code);
```

Також можна отримати код останньої помилки за допомогою функції `cudaGetLastError`:

```
cudaError_t cudaGetLastError();
```

Важливим моментом роботи з CUDA, на який слід відразу ж звернути увагу, є те, що багато функцій API - асинхронні, тобто управління повертається ще до реального завершення необхідної операції.

До числа асинхронних операцій відносяться:

- запуск ядра;
- функції копіювання пам'яті, імена яких закінчуються на `Async`;
- функції копіювання пам'яті `device ↔ device`
- функції ініціалізації пам'яті.

Для синхронізації поточної нитки на CPU з GPU використовується функція `cudaThreadSynchronize`, яка чекає завершення виконання всіх операцій CUDA, раніше викликаних з поточного потоку на CPU.

```
cudaError_t cudaThreadSynchronize(void);
```

CUDA підтримує синхронізацію через потоки (`streams`) - кожен потік задає послідовність операцій, що виконуються в строго певному

порядку. При цьому порядок виконання операцій між різними потоками не є строго певним і може змінюватися.

6.2 Завдання

6.2.1 Розробити паралельну програму інтегрування методом Монте-Карло, використовуючи технологію CUDA Дослідити своє рішення аналогічно п. 5.2.3 Завдання 2.

6.2.2 Отримати індивідуальне завдання у викладача.

6.3 Зміст звіту

- 6.3.1 Мета лабораторної роботи.
- 6.3.2 Тексти програм.
- 6.3.3 Результати розрахунків.
- 6.3.4 Відповіді на контрольні питання.

6.4 Контрольні питання

- 6.4.1 В чому полягає сутність технології CUDA?
- 6.4.2 Види пам'яті в CUDA.
- 6.4.3 Поясніть використання вбудованих змінних `threadIdx`, `blockIdx`, `gridDim` і `blockDim`.
- 6.4.4 Поясніть синтаксис директиви виклику ядра.
- 6.4.5 Які Ви знаєте додані типи, змінні та функції?
- 6.4.6 Поясніть різницю між CUDA driver API та CUDA runtime API.
- 6.4.7 Як отримати опис останньої помилки у вигляді рядка?

ЛІТЕРАТУРА

1. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ – Санкт-Петербург, 2002. – 400 с.
2. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования.: Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 512 с.
3. Шеховцов В.А. Операційні системи. – К.: Видавнича група ВНУ, 2005. - 576 С.
4. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие – Нижний Новгород; Изд-во ННГУ им. Н.И. Лобачевского, 2000. –176 с.
5. Parallel.ru – информационно-аналитический центр по параллельным вычислениям – <http://www.parallel.ru>.
6. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 608 с.
7. Корнеев В.Д. Параллельное программирование в MPI. – 2-е изд., испр. – Новосибирск: Изд-во ИВМиМГ СО РАН, 2002. – 215 с.
8. Бройнль Т. Паралельне програмування: Початковий курс: Навч. посібник / Вступ. Слово А.Ройтера; Пер. з нім. В.А. Святого. – К.: Вища школа, 1997. – 358 с.
9. Букатов А.А., Дацок В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. – Ростов-на-Дону: Изд-во ООО "ЦВВР", 2003. – 208 с.
10. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем. – М.: Мир, 1991. – 367 с.
11. Корнеев В.В. Параллельные вычислительные системы. – М.: Нолидж, 1999. – 320 с.
12. Параллельные вычисления на GPU. Архитектура и программная модель CUDA[Текст]: Учеб. пособие / А.В. Боресков и др. Предисл.: В.А. Садовничий. – М.: Издательство Московского университета, 2012. – 336 с.

Додаток А

Текст програми Thread War

```
// Проста комп'ютерна гра Thread War
// Використовуйте клавіші "уліво" і "вправо", щоб переміщати пушку
// клавіша "пробіл" робить постріл,
// Якщо 30 ворогів підуть із екрана не знищеними, ви програли
// Очки даються за кожного вбитого супротивника
```

```
#include <windows.h>
#include <process.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
```

```
// Об'єкти синхронізації
HANDLE screenlock; // зміною екрана займається тільки один потік
HANDLE bulletsem; // можна вистрілити тільки три рази підряд
HANDLE startevt; // гра починається з натисканням клавіші "уліво"
або "вправо"
HANDLE conin, conout; // дескриптори консолі
HANDLE mainthread; // основний потік main
CRITICAL_SECTION gameover;
```

```
CONSOLE_SCREEN_BUFFER_INFO info; // інформація про консоль
// кількість влучень і промахів
long hit=0;
long miss=0;
long delayfactor=7; // фактор затримки для ворогів
```

```
// Створення випадкового числа від n0 до n1
int random(int n0, int n1)
{
if (n0==0 && n1==1) return rand()%2; // спеціальний випадок
return rand()%(n1-n0)+n0;
}
// Очищення екрана консолі
void cls()
```

```

{
COORD org={0,0};
DWORD res;
FillConsoleOutputCharacter(conout,
',', info.dwSize.X*info.dwSize.Y, org, &res);
}

// вивести на екран символ в позицію x и y
void writeat(int x, int y, char c)
{
// Блокувати вивід на екран за допомогою м'ютекса
WaitForSingleObject(screenlock, INFINITE);
COORD pos={x,y};
DWORD res;
WriteConsoleOutputCharacter(conout, &c, 1, pos, &res);
ReleaseMutex(screenlock);
}

// Одержати натискання на клавішу (лічильник повторень в ct)
int getakey(int &ct)
{
INPUT_RECORD input;
DWORD res;
while (1)
{
ReadConsoleInput(conin,&input, 1, &res);

// ігнорувати інші події
if (input.EventType!=KEY_EVENT) continue;

// ігнорувати події відпускання клавіш
// нас цікавлять тільки натискання
if (!input.Event.KeyEvent.bKeyDown) continue;
ct=input.Event.KeyEvent.wRepeatCount;
return input.Event.KeyEvent.wVirtualKeyCode;
}
}

```

```

// Обробка комбінацій ^C, ^Break, і т.і.
BOOL WINAPI ctrl(DWORD type)
{
    exit(0);
    return TRUE;
// не досяжна ділянка коду
}

// Визначити символ в заданій позиції екрана
int getat(int x, int y)
{
    char c;
    DWORD res;
    COORD org={x,y};

// Блокувати доступ до консолі доти, поки процедура не буде виконана
WaitForSingleObject(screenlock,INFINITE);
ReadConsoleOutputCharacter(conout, &c, 1, org, &res);
ReleaseMutex(screenlock); // unlock
return c;
}

// Відобразити очки в заголовку вікна й перевірити умову завершення
гри
void score(void)
{
    char s[128];
    sprintf(s, "Thread War! Hit: %d Miss : %d", hit, miss);
    SetConsoleTitle(s);
    if (miss>=30)
    {
        EnterCriticalSection(&gameover);
        SuspendThread(mainthread); // призупинити головний потік
        MessageBox(NULL, "Game Over!", "Thread War",
        MB_OK|MB_SETFOREGROUND);
        exit(0); // не виходить із критичної секції
    }
}

```

```

if ((hit+miss)%20==0)
InterlockedDecrement(&delayfactor);           // повинен бути ilock
}

```

```

char badchar[]="-\\|/";
// це потік супротивника
void badguy(void *_y)
{
int y=(int) _y; // випадкова координата у
int dir;
int x;
// непарні у з'являються ліворуч, парні у з'являються праворуч
x=y%2?0:info.dwSize.X;
// установити напрямок залежно від початкової позиції
dir=x?-1:1;
//поки супротивник перебуває в межах екрана
while ((dir==1&&x!=info.dwSize.X)||((dir==-1&&x!=0))
{
int dly;
BOOL hitme=FALSE;
// перевірка на влучення (куля?)
if (getat(x,y)=='*') hitme=TRUE;

// вивід символу на екран
writeat(x,y,badchar[x%4]);

// ще одна перевірка на влучення
if (getat(x,y)=='*') hitme=TRUE;
// перевірка на влучення через невеликі
// проміжки часу
if (delayfactor<3) dly=3;
else dly=delayfactor+3;
for (int i=0; i<dly; i++)
{
Sleep(40);
if (getat(x,y)=='*')
{
hitme=TRUE;

```

```

    break;
}
}
writeat(x,y,' ');
// ще одна перевірка на влучення
if (getat(x,y)=='*') hitme=TRUE;
if (hitme)
{
// у супротивника влучили!
MessageBeep(-1);
InterlockedIncrement(&hit);
score();
_endthread();
}
x+=dir;
}

//супротивник утік!
InterlockedIncrement(&miss);
score();
}

// цей потік займається створенням потоків супротивників
void badguys(void *)
{
// чекаємо сигналу до початку гри протягом 15 секунд
WaitForSingleObject(startevt, 15000);
// створюємо випадкового ворога
// кожні 5 секунд з'являється шанс створити
//супротивника з координатами від 1 до 10
while (1)
{
if (random(0,100)<(hit+miss)/25+20)
// згодом імовірність збільшується
    _beginthread(badguy,0, (void *) (random(1,10)));
    Sleep(1000); // шосекунди
}
}
}

```

```

// Це потік кулі
// кожна куля - це окремий потік
void bullet(void *_xy_)
{
COORD xy=*(COORD *)_xy_;
if (getat(xy.X, xy.Y)=='*') return; // тут уже є куля
// треба почекати
// перевірити семафор
// якщо семафор дорівнює 0, пострілу не відбувається
if (WaitForSingleObject(bulletsem,0)==WAIT_TIMEOUT) return;
while (-ixy.Y)
{
writeat(xy.X, xy.Y, '*'); // відобразити кулю
Sleep(100);
writeat(xy.X, xy.Y, ' '); // стерти кулю
}
// постріл зроблений - додати 1 до семафора
ReleaseSemaphore(bulletsem,1,NULL);
}

// Основна програма
void main()
{
HANDLE me;
// Налаштування глобальних змінних
conin=GetStdHandle(STD_INPUT_HANDLE);
conout=GetStdHandle(STD_OUTPUT_HANDLE);
SetConsoleCtrlHandler(ctrl,TRUE);
SetConsoleMode(conin,ENABLE_WINDOW_INPUT);
me=GetCurrentThread(); // не є реальним дескриптором

// змінити псевдодескриптор на реальний дескриптор поточного
потіку
DuplicateHandle(GetCurrentProcess(), me, GetCurrentProcess(),
&mainthread, 0, FALSE,
DUPLICATE_SAME_ACCESS);

```

```

startevt=CreateEvent(NULL,TRUE,FALSE,NULL);
screenlock=CreateMutex(NULL,FALSE,NULL);
InitializeCriticalSection(&gameover);
bulletsem=CreateSemaphore(NULL,3,3,NULL);
GetConsoleScreenBufferInfo(conout,&info);

// Ініціалізувати відображення інформації про очки
score();
// Настроїти генератор псевдовипадкових чисел
srand( (unsigned)time( NULL ) );
cls(); // насправді не потрібно
// установка початкової позиції пушки
int y=info.dwSize.Y-1;
int x=info.dwSize.X/2;
//запустити потік badguys; нічого не робити доти,
// поки не відбудеться подія або минуть 15 секунд
_beginthread(badguys,0,NULL);
// основний цикл гри
while (1)
{
int c,ct;
writeat(x,y,'|'); // намалювати пушку
c=getakey(ct); // одержати символ
switch (c)
{
case VK_SPACE: // вогонь!
{
static COORD xy;
xy.X=x;
xy.Y=y;

_beginthread(bullet, 0, (void *) &xy);
Sleep(100); // дати кулі час полетіти на деяку відстань
break;
}
case VK_LEFT: // команда "уліво!"
SetEvent(startevt); // потік badguys працює
writeat(x,y,' '); // зтерти з екрана пушку

```

```
while (ct--) // переміститися
    if (x) x--;
break;
case VK_RIGHT: // команда "вправо!"; логіка та ж
SetEvent(startevt);
writeat(x,y, ' ');
while (ct-i)
if (x!=info.dwSize.X-1) x++;
break;
}
}
}
```