**Синхронізація потоків засобами С++**

Ціль роботи

– вивчити основи багатопотокового програмування на мові С++ та його використання для реалізації паралельних алгоритмів і програм.

Теоретичні відомості

### Клас thread

У C ++ 11, робота з потокам здійснюється з використанням класу std::thread (ФЗ <thread>), який може працювати з регулярними функціями, лямбда і функторами.

Існує можливість передавати різну кількість аргументів у функцію потоку.

Об’єкт класу thread не можна копіювати, але його можна переміщати (std::move) і присвоювати. Проте присвоювати можна лише ті об'єкти, які не зв'язані з яким-небудь потоком. У цьому випадку тоді об'єкту буде присвоєно тільки стан.

При переміщенні об'єкту передається стан і право на управління потоком

Виклик join (аналог Wait-функції із SDK Windows) блокує викликаючий потік (наприклад main) до тих пір, поки функція потоку не виконає свою роботу (тобто потік не перейде у вільний стан). Якщо функція потоку повертає значення - воно буде проігноровано.

Метод detach дозволяє від'єднати потік від об'єкта, іншими словами, зробити його фоновим. До від'єднаного потоку більше не можна застосовувати join.

Кожен потік має свій ідентифікатор типу std::thread::id, який можна отримати викликом методу get\_id або викликом статичного методу std::thread::this\_thread::get\_id із функції самого потоку

Створення потоку

#include <iostream>

#include <thread>

#include <atomic>

#include <string>

#include <stdio.h>

#include <system\_error>

void say(string& s) {

while (getchar() != 'c')

std::cout << "It’s my first " << s << endl ;

}

void say\_all() {

std::cout << "It’s my second THREAD" << endl ;

}

void say\_one(int i, double d, const std::string s) {

std::cout << "It’s my third " << s << endl ;

}

void say\_error(int& i) {

std::cout << "It’s my error THREAD " << i << endl;

}

int main() {

try {

std::thread thr0;

string mes("THREAD");

std::thread thr1(say, std::ref(mes));

std::thread thr2(say\_all);

std::thread thr3(say\_one, 100, 3.14, "THREAD");

int n = -1;

std::thread thr4(say\_error, std::ref(n));

std::thread thr5(std::move(thr4));

thr1.join();

thr2.join();

thr3.join();

//thr4.join();

thr5.join();

} catch (const std::system\_error e) {}

return 0;

}

Кожен потік має свій ідентифікатор типу std::thread::id, який можна отримати викликом методу get\_id або викликом статичного методу std::thread::this\_thread::get\_id із функції самого потоку

### Базові методи

* get\_id: повертає id поточного потоку
* yield: вказує планувальнику виконувати інші потоки, може використовуватися при активному очікуванні
* sleep\_for: блокує виконання поточного потоку протягом встановленого періоду
* sleep\_until: блокує виконання поточного потоку, поки не буде досягнутий вказаний момент часу
* hardware\_concurrency - статичний метод, який повертає кількість потоків, які можуть бути виконані справді паралельно (стандарт дозволяє функції повертати 0, якщо на даній системі це значення не можна підрахувати або воно Не визначено)

### Об’єкти синхронізації

mutex - найпростіший бінарний семафор, який може перебувати в одному з двох станів (вільний-зайнятий або відкритий і закритий). Призначення захист об'єкта від доступу до нього інших потоків, відмінних від того, яким заволодів м'ютекс.

У кожен конкретний момент тільки один потік може володіти об'єктом, захищеним м'ютексом. Якщо іншому потоку буде потрібен доступ до об’єкта, захищеного м'ютексом, то цей потік “засинає” (вимикається із планування) доти, поки м'ютекс не стане вільним.

Стани визначаються викликами функцій lock () і unlock () і ідентифікуються методом try\_lock (). Приклад м’ютекса

#include <vector>

#include <mutex>

#include <thread>

std::vector<int> x;

std::mutex mutex;

void thread\_func1() {

mutex.lock();

x.push\_back(0);

mutex.unlock();

}

void thread\_func2() {

mutex.lock();

x.pop\_back();

mutex.unlock();

}

int main() {

std::thread th1(thread\_func1);

std::thread th2(thread\_func2);

th1.join();

th2.join();

return 0;

}

Види м’ютекса

* recursive\_mutex – м’ютекс для якого допускаються повторні захоплення тим же потоком, ведеться лічильник таких захоплень;
* timed\_mutex - немає контролю повторного захоплення тим же потоком, підтримується захоплення м’ютекса з тайм-аутом;
* recursive\_timed\_mutex - повторні захоплення тим же потоком допустимі, ведеться лічильник таких захоплень, підтримується захоплення м’ютекса з тайм-аутом.

### Шаблонный класс std::lock\_guard

Не рекомендується використовувати клас mutex безпосередньо, оскільки між викликами lock і unlock буде згенеровано виключення - відбудеться deadlock (тобто заблокований потік так і залишиться в стані очікування).

Проблема виникає, тоді конструктор викликає метод lock для заданого об'єкта, а деструктор викликає unlock. В загальному випадку deadlock можливий, коли між викликами lock і unlock згенеровано виключення, або потоки блокують більше одного м’ютекса, тобто у випадках:

Два м’ютекси A і B захищають два різних ресурсу, і, двом потокам, одночасно необхідний доступ до цих двох ресурсів.

Блокування одного м’ютекса є атомарним, але блокування двох м’ютексів - це дві окремі дії, і, якщо перший потік заблокує mutex A, в той час, як другий заблокує mutex B, то обидва потоки зависнуть чекаючи один одного.

Для вирішення проблеми рекомендується використання обгорток:

* Клас lock\_guard, якому можна передати аргумент adopt\_lock - індикатор, що означає, що mutex вже заблокований і блокувати його наново непотрібно. Обгортка lock\_guard не містить ніяких інших методів, його не можна копіювати, переносити або присвоювати
* Клас unique\_lock надає трохи більше можливостей, ніж lock\_guard, зокрема:
  + надає можливість ручного блокування і розблокування контрольованого м’ютекса за допомогою методів lock і unlock відповідно;
  + можна переміщати за допомогою виклику std::move;
  + об'єкт классa unique\_lock може не володіти правами на mutex, який він контролює;
  + при створенні об'єкта можна відкласти блокування м’ютекса передачею аргументу defer\_lock конструктору unique\_lock і вказати, що об'єкт не володіє правами на mutex і викликати unlock в деструкції не треба;
  + права на mutex можна отримати пізніше, викликавши метод lock для об'єкта;
  + функцією owns\_lock можна перевірити, чи володіє поточний об'єкт правами на mutex.

#include <vector>

#include <mutex>

#include <thread>

std::vector<int> x;

std::mutex mutex;

void thread\_func1() {

std::lock\_guard<std::mutex> lock(mutex);

x.push\_back(0);

}

void thread\_func2() {

std::lock\_guard<std::mutex> lock(mutex);

x.pop\_back();

}

int main() {

std::thread th1(thread\_func1);

std::thread th2(thread\_func2);

th1.join();

th2.join();

return 0;

}

std::call\_once - створений для того, щоб захищати загальні дані під час ініціалізації. Він дозволяє викликати окремі ділянку коду один раз, незалежно від кількості потоків, які намагаються виконати цю ділянку коду. Є швидким і зручним механізмом для створення потоко безпечних singleton-ів

#include <mutex>

#include <thread>

#include <iostream>

struct x {

x() {

std::cout << std::this\_thread::get\_id() <<

std::endl;

}

};

x\* instance;

void create\_x() {

instance = new x();

}

std::once\_flag instance\_flag;

void thread\_func() {

std::call\_once(instance\_flag, create\_x);

}

int main() {

std::thread th1(thread\_func);

std::thread th2(thread\_func);

th1.join();

th2.join();

return 0;

}

### Очікування

Потік очікує настання певного події. Варіанти реалізації:

* регулярно в циклі перевіряти умова настання події (не ефективно, так як потік, витрачає процесорний ресурс) .
* std::condition\_variable - це об'єкт синхронізації, призначений для блокування одного потоку, поки він не буде сповіщений про настання деякої події з іншого потоку. Працює тільки з блокуваннями типу std::unique\_lock.
* std::condition\_variable\_any – на відміну від std::condition\_variable працює з будь-якими блокуваннями.

Сповіщення події здійснюється функціями data\_cond.notify\_one(); data\_cond.notify\_all ()

#include <vector>

#include <mutex>

#include <thread>

#include <iostream>

#include <condition\_variable>

std::vector<int> data;

std::condition\_variable data\_cond;

std::mutex m;

void thread\_func1() {

std::unique\_lock<std::mutex> lock(m);

data.push\_back(10);

data\_cond.notify\_one();

}

void thread\_func2() {

std::unique\_lock<std::mutex> lock(m);

data\_cond.wait(lock, [] { return !data.empty(); });

std::cout << data.back() << std::endl;

}

int main() {

std::thread th1(thread\_func1);

std::thread th2(thread\_func2);

th1.join();

th2.join();

return 0;

}

### Повернення значення

У випадку коли необхідно викликати функцію в окремому потоці, і повернути значення створення нового потоку за допомогою thread, не дає прямої можливості це зробити.

Для вирішення цієї проблеми треба запустити потік викликом функції async і передаванням їй потокової функції/функтора.

Функція async повертає об'єкт типу future <T>, де T – тип значення, що повертається потоковою функцією

#include <iostream>

#include <future>

#include <thread>

int calculate() {

return 2 \* 2;

}

int main() {

std::future<int> result =

std::async(calculate);

std::cout << result.get() << std::endl;

}

Безпечне передавання значень у потоках можна здійснити такими способами:

* promise - дозволяє передавати значення між потоками. Кожен об'єкт promise пов'язаний з

об'єктом future. Разом вони утворюю пару, якій один (promise) відповідає за встановлення

значення, а інший future - за його отримання.

* Перший потік може очікувати установки значення за допомогою виклику методу future::wait aбо future::get. У той час, другий потік:
  + Встановить це значення за допомогою виклику методу std::promise::set\_value,
  + Або дасть першому виняток викликом методу std::promise::set\_exception.

#include <iostream>

#include <future>

#include <thread>

std::promise<int> promise;

void thread\_func1() { promise.set\_value(10); }

void thread\_func2() {

std::cout << promise.get\_future().get() << std::endl;

}

int main() {

std::thread th1(thread\_func1);

std::thread th2(thread\_func2);

th1.join();

th2.join();

return 0;

}

Для передавання винятків повинен викликатися метод promise::set\_exception, який приймає об'єкт типу exception\_ptr.

Отримати об'єкт цього типу можна, або викликавши current\_exception () з блоку catch, або створити об'єкт цього типу безпосередньо за допомогою виклику функції make\_exception\_ptr.

#include <iostream>

#include <future>

#include <thread>

std::promise<int> promise;

void thread\_func1() {

promise.set\_exception(std::make\_exception\_ptr

(std::runtime\_error("fatalerror")));

}

void thread\_func2() {

try {

std::cout << promise.get\_future().get() << std::endl;

} catch (const std::exception& e) {

std::cout << e.what() << std::endl;

}

}

int main() {

std::thread th1(thread\_func1);

std::thread th2(thread\_func2);

th1.join();

th2.join();

return 0;

}

Робоче завдання

1. Реалізувати індивідуальне завдання.
2. Задокументувати у звіті по виконаній роботі висновки за результатами досліджень і підготувати відповіді на контрольні питання.

Методичні вказівки

### Створення і запуск потоків

Найпростіша багатопоточна програма наведена у програмі 1. У цій програмі використовується клас thread, що призначений для створення потоків засобами мови С++. Його визначення знаходиться у однойменному заголовному файлі thread.h (#inсlude <thread>).

Конструктор цього класу приймає функцію та опційні параметри. Функція, що передається до конструктора, виконується у окремому потоці.

Кожний створений потік має власний ідентифікатор (ID), тип котрого – thread::id. Ідентифікатор потоку можна одержати шляхом виклику статичного методу класу this\_thread::get\_id() чи шляхом виклику методу get\_id() екземпляру класу thread, наприклад, th1.get\_id().

*Програма 1 – Проста багатопотокова програма на мові С++*

#include <iostream>

#include <sstream>

#include <thread>

using namespace std;

void hello(string name) {

cout << "Hello, " << name << " Thread ID = " <<

this\_thread::get\_id() << endl;}

void getid(int i) {

thread::id id = this\_thread::get\_id();

stringstream str;

str << id << i << endl;

str >> i;

cout << i << endl;

}

int main() {

thread th1(hello, "csn!");

thread th2(hello, "scs!");

th1.join();

th2.join();

thread th[3];

for(int i = 0; i < 3; i++)

th[i] = thread(getid,i+1); 28

for(int i = 0; i < 3; i++)

th[i].join();

return 0;

}

У результаті роботи програми одержимо на екрані повідомлення наступного змісту (не обов’язково у зазначеній послідовності рядків):

Hello, csn! Thread ID = 2

Hello, scs! Thread ID = 3

41

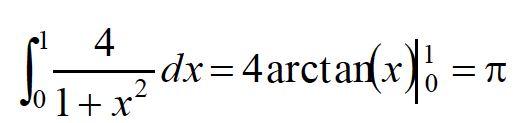
52

63

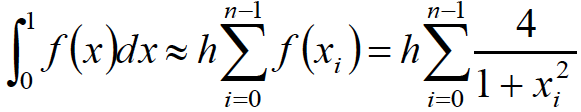
З наведеного результату роботи програми бачимо, що функція main() виконується у потоці з ID = 1. Для того, щоб потік, виконуючий функцію main(), не завершив свою роботу раніше за дочірні потоки th1 і th2, використовуються виклики функції join(), що змушує головний (батьківський) потік очікувати на завершення роботи дочірніх потоків.

### Розподілення обчислювального навантаження між потоками

У програмі 2 демонструється один із способів розпаралелювання алгоритмів – розпаралелювання за ітераціями. У програмі обчислюється наближене значення числа π за відомою формулою:

 (1.1)

Даний інтеграл можна обчислити методом прямокутників:

 (1.2)

де *h* = 1/ *n* – крок дискретизації; *xi* = *h*(*i* + 1/ 2) , *i* = 1, 2,..., *n* - 1; *n* – кількість підінтервалів.

Суть розпаралелювання обчислень для даної задачі полягає у тому, що підінтегральна функція *f*(*x*) може обчислюватися у кожній точці дискретизації незалежно. Таким чином, обчислення функції у першій точці можна виконати у першому програмному потоці, обчислення у другій точці – у другому програмному потоці і т.д.

Отже, за наявності у процесорі використовуваного комп’ютера декількох ядер, обчислення функції *f*(*x*) у різних точках дискретизації можуть виконуватися на різних ядрах процесора.

*Програма 2. Обчислення числа.*

У даній програмі (лістинг 1.2) головний потік, що виконує функцію main() (ID = 1), створює два дочірніх потоки з ID = 2 і ID = 3 (рядок 48). Обчислення на парних підінтервалах ( i = 0, 2,...) виконуються потоком з ID = 2, обчислення на непарних підінтервалах ( i = 1, 3,...) – потоком з ID = 3.

У даній реалізації потоки мають доступ до спільного ресурсу – змінної pi (рядок 28). Для унеможливлення одночасних звернень незалежних потоків до даного ресурсу (цей процес називається гонкою даних) у програмі використовується примітив синхронізації, що носить назву "м’ютекс" (mutex – mutual exclusion, взаємне виключення). Перед тим як звернутися до спільного ресурсу, потік захоплює (lock()) м’ютекс, а по завершенні роботи з ресурсом – вивільняє (unlock()) м’ютекс.

Бібліотека Thread Library гарантує, що, якщо один потік захопив певний м’ютекс, то всі інші потоки, які намагаються захопити той самий м’ютекс, будуть вимушені очікувати, доки м’ютекс не буде вивільнено [1]. Процес захоплення і вивільнення м’ютекса реалізовано у рядках 27 і 29, відповідно.

Відзначимо, що використання м’ютекса у даній програмі наведено виключно з навчальною метою, оскільки реалізувати дану програму можна було би і без м’ютекса, наприклад, накопичуючи суми результатів обчислень *f*(*x*) у кожному з потоків, а результуюче складання цих результатів виконати у головному потоці, після завершення роботи дочірніх потоків.

Дана програма містить також функцію computePi1()що призначена для обчислення значення π послідовним шляхом (в одному потоці) у функції main(), і демонструє один із можливих шляхів визначення часових витрат на проведення обчислення.

#include <iostream>

#include <mutex>

#include <thread>

#include <iomanip>

#include <cmath>

using namespace std ;

const double pi0 = 3.141592653589793238462643;

double pi = 0.0;

mutex m;

long intervals = 5000000;

int numThreads = 2;

long t1,t2;

void computePi(int num) {

double x, width, localSum = 0.0;

width = 1.0/intervals;

for(int i=num; i<intervals; i+=numThreads) {

x = (i +0.5)\*width;

localSum += 4.0/(1.0 + x\*x);

}

localSum \*= width;

m.lock();

pi += localSum;

m.unlock();

}

void computePi1() {

double x, width;

width = 1.0/intervals;

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

x = (i+0.5)\*width;

pi += 4.0/(1.0 + x\*x);

}

pi \*= width;

}

int main() {

thread th[numThreads];

t1 = clock();

for(int i = 0; i < numThreads; i++)

th[i] = thread(computePi,i);

for(int i = 0; i < numThreads; i++)

th[i].join();

t2 = clock();

cout << setprecision(16) << pi << endl;

cout << setprecision(4) << "err:" <<

fabs(pi0 - pi) << endl;

cout << "time: " << t2-t1 << endl;

pi = 0.0;

t1 = clock();

computePi1();

t2 = clock();

cout << setprecision(16) << pi << endl;

cout << setprecision(4) << "err: " <<

fabs(pi0 - pi) << endl;

out << "time: " << t2-t1 << endl;

return 0;

}

*Програма 3. Розкладання числа на прості співмножники*

У даній програмі розпаралелюється задача розкладання масиву чисел на прості співмножники. Нагадаємо, простими називаються числа, що діляться лише на одиницю і на самих себе. Ряд простих чисел має наступний вигляд: 2, 3, 5, 7, 11, 13, 17...

Перевірка, чи є число x простим, зводиться до перевірки того,чи існує на відрізку [ 2; ] число, на яке ділиться x . Реалізація алгоритму такої перевірки на мові С++ наведена нижче.

bool algorithm(int x) {

for(int i = 2; i <= sqrt(x); i++) {

if (x % i == 0) {

return false;

}

}

return true;

}

Великі числа у програмі мають тип unsigned long long, Для цього зроблено визначення макросу ullong. Числа можуть мати різну кількість простих співмножників, тому для кожного з них використовується тип vector. Тип vector – це клас, який призначений для збереження масивів даних змінного розміру у динамічній пам’яті. Цей клас містить набір функцій-членів для роботи з даними, які у ньому зберігаються: різноманітні функції доступу до елементів, визначення розміру масиву, додавання/видалення елементів і т.п.

Функцію обчислення розкладання (факторизації) наведено у прграмі 4. Для заповнення масиву чисел псевдовипадковими значеннями у програмі визначено функцію, що генерує великі псевдовипадкові числа.

У даній програмі для оцінки числа апаратних потоків (процесорів, ядер) використовується функція thread::hardware\_concurrency(). Обчислювальне навантаження полягає у розкладанні восьми великих псевдовипадкових чисел на прості співмножники, яке розподіляється між апаратними потоками порівну. Так, якщо у процесора є два ядра, то кожне з них буде обчислювати розкладання для чотирьох чисел. Для кожного числа визначається екземпляр класу vector, елементами якого є прості співмножники.

*Програма 4 – Програма для розкладання великих чисел на прості співмножники*

#include <iostream>

#include <cstdlib>

#include <thread>

#include <vector>

#include <cmath>

#include <climits>

#include <ctime>

#define ullong unsigned long long

using namespace std;

void factoring(ullong arr[], int size, vector <ullong> vec[]) {

ullong x, div;

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

x = arr[i];

for(int j = 2; j <= sqrt(x); j++) {

while (x % j == 0) {

vec[i].push\_back(j);

x /= j;

}

}

if (x != 1)

vec[i].push\_back(x);

}

}

ullong ullrand() {

ullong rnd = 0;

//ullong counter = ULLONG\_MAX;

//ULLONG = 18446744073709551615

ullong counter = 1E+13;

while(counter > 0) {

rnd = (rnd \* (RAND\_MAX + 1)) + rand();

counter /= (RAND\_MAX + 1);

}

return rnd;

}

int main() {

srand(time(NULL));

int n = 8;

long t;

int numThreads = thread::hardware\_concurrency();

if(n%numThreads != 0) {

cout << "The array size must be a multiple of the threads number!\n";

exit(0);

}

thread th[numThreads];

int size\_scatter = n/numThreads;

ullong\* arr[numThreads];

vector<ullong> \*\*vec = new vector<ullong>\*[numThreads];

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

arr[i] = new ullong[size\_scatter];

vec[i] = new vector<ullong>[size\_scatter];

for(int j = 0; j < size\_scatter; j++) {

arr[i][j] = ullrand();

vec[i][j].reserve(10);

}

t = clock();

th[i] = thread(factoring, arr[i], size\_scatter, vec[i]);

}

for(int i = 0; i < numThreads; i++)

th[i].join();

long time\_p = clock() - t;

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

for(int j = 0; j < size\_scatter; j++) {

cout << "arr[" << i << "][" << j << "] = ";

cout << arr[i][j] << " = 1";

for(int k = 0; k < vec[i][j].size(); k++)

cout << "\*" << vec[i][j][k];

cout << endl;

}

}

cout << "time\_p = " << time\_p << endl;

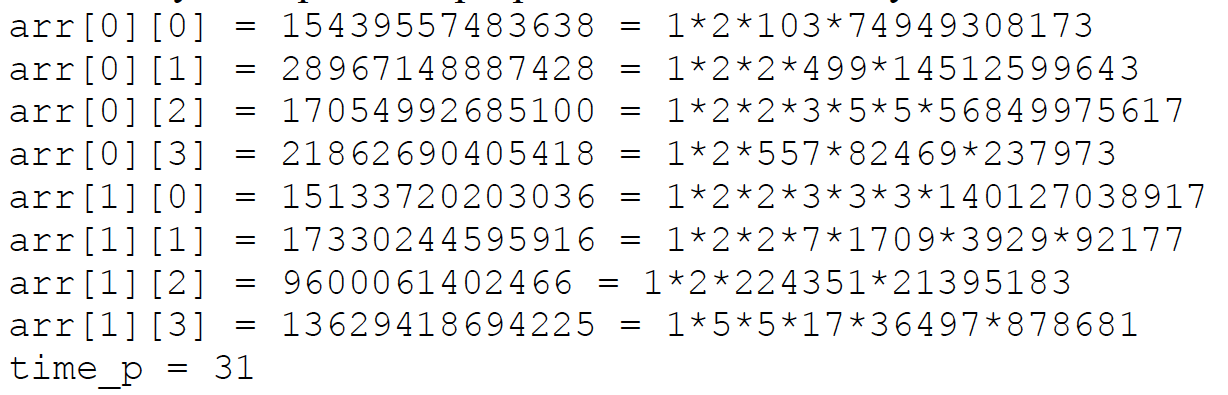
for(int i = 0; i < numThreads; i++)

delete [] arr[i];

return 0;

}

Результат роботи програми може мати наступний вигляд:



Індивідуальні завдання.

1. Розробити власну багатопоточну програму згідно наступних вимог:

– у програмі має створюватися масив з 10 потоків;

– до конструктора кожного потоку має передаватися функція із трьома аргументами: перший аргумент – ім’я, другий аргумент – прізвище, третій аргумент – випадкове ціле число у діапазоні від 0 до 9;

– функція має виводити на консоль лише ім’я, якщо третій аргумент парний, і лише прізвище, якщо третій аргумент непарний.

2. У програмі 2 визначити, на яких підінтервалах будуть виконувати обчислення потоки, у випадку, якщо їх чотири.

3. Провести достатню кількість експериментів з програмою 2 і визначити середнє значення прискорення для випадків двох та чотирьох потоків.

4. Допрацювати програму 2 таким чином, щоб вона коректно виконувалася без використання примітиву синхронізації mutex.

5. Розробити програму, у якій є функції для паралельного та послідовного обчислень числа π за формулою на основі ряду Лейбніца:

6. Експериментально визначити значення кількості членів ряду Лейбніца, при якому число π обчислюється із похибкою e < 10-7 .

7. У програмі 4 Визначити прискорення паралельного методу обчислення при розподіленні задач між двома та чотирма потоками.

Контрольні питання

1. Для чого призначений клас EventWaitHandle?
2. Які класи .NET Framework можна застосувати, щоб оцінити час виконання програми у багатопоточному режимі?
3. Наведіть паралельний алгоритм додавання двох векторів.
4. Наведіть паралельний алгоритм множення двох матриць.
5. У чому полягає сутність принципу організації параельного алгоритму «Producer/Consumer»?
6. Обгрунтуйте або спростуйте можливість створення двох різних паралельних алгоритмів, які одну й ту ж саму задачу вирішують за приблизно рівний час.
7. Які програмні бібілотеки використовуються для створення паралельних програм на основі передач повідомлень?
8. У чому відміннсть класів AutoResetEvent та ManualResetEvent?ми.