НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Кафедра обчислювальної техніки\_\_\_\_\_\_\_\_\_\_\_\_

(повна назва кафедри, циклової комісії)

**КУРСОВА РОБОТА**

з дисципліни «Паралельне програмування»

(назва дисципліни)

на тему: «Розробка програмного забезпечення для паралельних комп’ютерних систем»

Студента (ки) 3 курсу групи ІП-42

напряму підготовки 050103 «Програмна інженерія»

Дзюби В.В.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище та ініціали)

Керівник доцент Корочкін О.В.

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_

Оцінка: ECTS \_\_\_\_\_\_\_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали

Київ- 2017 р.

Національний технічний університет України

“Київський політехнічний інститут”

Факультет (інститут) інформатики та обчислювальної техніки

( повна назва )

Кафедра обчислювальної техніки

( повна назва )

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки 6.050103 «Програмна інженерія»

# (шифр і назва)

## З А В Д А Н Н Я

### НА КУРСОВУ РОБОТУ СТУДЕНТУ

Дзюби Влада Володимировича

(прізвище, ім’я, по батькові)

1. Тема роботи «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

керівник роботи Корочкін Олександр Володимирович к.т.н.**,** доцент

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи 11 травня 2017 р.

3. Вхідні дані до роботи

- засоби роботи з потоками в бібліотеці OpenMP

- математична задача

- структури ПКС ОП та ПКС ЛП

- мови і бібліотеки програмування: C++, OpenMP.

- засоби організації взаємодії процесів: цикли for.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- огляд засобів роботи з процесами в мові Ада

- розробка і тестування програми ПРГ1 для ПКС ОП

- розробка і тестування програми ПРГ2 для ПКС ЛП

5. Перелік графічного матеріалу

- структурна схема ПКС ОП

- структурна схема ПКС ЛП

- схеми алгоритмів процесів і головної програми для ПРГ1

- схеми алгоритмів процесів і головної програми для ПРГ2.

7. Дата видачі завдання \_\_\_\_\_\_\_\_хх\_\_\_\_\_

#### КАЛЕНДАРНИЙ ПЛАН

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання розділу 1 | 13.03.2017 |
| 2 | Виконання розділу 2 | 3.04.2017 |
| 3 | Виконання розділу 3 | 23.04.2016 |
| 4 | Оформлення КР | 8.05.2016 |
| 5 | Перевірка КР викладачем | 11.05.2016 |
| 6 | Захист КР | 18.05.2016 |

**Студент \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

( підпис ) (прізвище та ініціали)

**Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

( підпис ) (прізвище та ініціали)

Зміст

[Вступ 4](#__RefHeading___Toc2080_596635936)

[1. Основна частина 5](#__RefHeading___Toc2074_596635936)

[2. Основні результати та висновки 6](#__RefHeading___Toc2082_596635936)

[3. Перелік умовних позначень 7](#__RefHeading___Toc2084_596635936)

[4. Список використаних джерел 8](#__RefHeading___Toc2086_596635936)

[Додатки 9](#__RefHeading___Toc2088_596635936)

Вступ

Метою курсовою роботи є аналіз та використання засобів розробки параллельних програм з використання OpemMP. Прикладом використання є параллельне обчислення математичної задачі.

З ростом виробництва комп’ютерних процесорів вдосконалення засобів комунікації між багатьма процессорами стає важлиівшою за вдосконалення окремих процессорів. OpenMP є тим стандартом, який дозволяє писати параллельни програми максимально коротко та зрозуміло.

Не менш важливим є питання вибору системи пам’яті для обчислювальної техніки: спільна пам’ять або локальна пам’ять. Система зі спільною пам’яттю є більш швидкими. Системи ж з локальною пам’яттю є більш структурованими та простішими для сприйняття та написання програм.

У першому розділі розглянуто засоби роботи з потокми бібліотеки OpenMP. У другому розділі описан процес розробки програм для систем зі спільною пам’яттю. У третьому розділі описан процес розробки програм для систем з локальною пам’яттю.

РОЗДІЛ 1. ОГЛЯД БІБЛІОТЕКИ OPENMP

1.1. Загальний опис.

*OpenMP* — *API*, призначений для розробки багатопоточних додатків для багатопроцесорних систем з спільною пам’яттю. Розробку специфікації ведуть декілька великих виробників обчислювальної техніки та програмного забезпечення. *OpenMP* підтримується основними компіляторами *С*/*C++*/*Fortran*.

В *OpenMP* немає явного задання потоків у коді. Натомість є можливість вказати компілятору з допомогою директив *#pragma*, що блок кода може бути розпаралелен. Знаючи цю інформацію, компілятор в стані згенерувати додаток, яке складається з одного головного потока, який створює багато інших потоків для паралельного блоку коду. Ці потоки синхронізуються у кінці паралельного блоку кода, вертаючись к головному потоку.

Так як *OpenMP* контролюється прагамами, то код на *C++* коректно скомпілюється будь-яким компілятором *C++*, тому що прагми, які не підтримуються, повинні ігноруватись. Проте *OpenMP* *API* містить також декілька функцій, та, щоб їх використати, необхідно підключити заголовочний файл. Найпростіший спосіб визначити, чи підтримує компілятор *OpenMP* — спробувати підключити *omp.h*.

Якщо *OpenMP* підтримується, потрібно його ввімкнути за допомогою спеціального флагу компілятора: *g++ -fopenmp.*

1.2. Синтаксис

Директиви *OpenMP* починаються з *#pragma omp*.

Parallel

Директива *#pragma omp parallel*створює групу з *N* потоків. *N* визначається під час виконання, загалом це кількість ядер процесора, але також можно задать *N* вручну. Кожен з потоків у групі виконує наступну за директивою команду (або блок команд, визначений в фігурних дужках). Після виконання, потоки “зливаються” в один.

Такий прозорий спосіб розмітки паралельної програми дозволяє мінімізувати написання коду для породження *N* потоків, задання їм функції виконання та синхронізації закінчення їх виконання. В наслідок того, що така задача є поширеною, *OpenMP* пропонує простий синтаксис для її вирішення.

**Директива for**

Директива *for* розділяє цикл між поточною групою потоків, так що кожен потік у группі оброблює свою частину цикла. Таким чином, кожен потів оброблює свою частину цикла параллельно з усіма потоками. При цьому послідовність виконання ітерацій цикла довільна.

Важливо, що директива *#pragma omp* лише делегує порції цикла різним потокам у поточній групі потоків. В момент запуска програми група з одиничного (головного) потока. Щоб створити нову групу потоків, необхідно використати ключове слово *parallel*.

Для того, щоб задати кількість потоків у групі, можна скористатись параметром *num\_threads*.

В *OpenMP* 2.5, ітераційна змінна цикла повинна бути типа *signed*. В *OpenMP* 3.0 вона також може мати тип *unsigned integer,* може бути вказівником або *constant-time random access* ітератором. В останньому випадку, для визначення кількості ітерацій цикла буде використовуватись *std::distance()*.

Перевагою синтаксису *OpenMP* для паралельного проходу цикла є автоматичне розбиття ітерацій між наявною групою потоків. При цьому в коді модифікується тільки директиви, що призводить до того, що без *OpenMP*, отримуємо простий цикл, який працює таким самим чином проте без паралельності.

**1.2.3. Планування**

Програміст може контролювати то, яким чином потоки будуть завантажуватись роботою при обробці цикла. Існують декілька варіантів.

*Static* є варіантом за замовчуванням. Ще до входу в цикл кожний поток “знає”, які частини цикла від буде обробляти. Такий варіант підходить, коли варіація часу виконання частини ітерації одним потоком мінімальна. При цьому балансування, яке проводить *OpenMP*, повністю задоволняє необхідність в ній. Цей варіант поширен настільки, що розробники *OpenMP* зробили цю стратегію балансування за замовчуванням.

Використовуючи d*ynamic*, неможливо передбачити порядок, в якому ітерації цикла будуть призначені потокам. Кожен потік виконує вказану кількість ітерацій. Якщо це число не задане, за замовчуванням воно дорівнює 1. Після того, як потік завершить виконання заданих ітерацій, він переходить до наступного набору ітерацій. Так продовжується, поки не будуть пройдені всі ітерації. Останній набір ітерацій може бути меншим, ніж заданий на початку. Такий варіант дуже корисний, коли різні ітерації цикла обраховуються різний час. Можна також вказати кількість ітерацій, після виконання яких поток “попросить” у *OpenMP* наступні.

Існує також варіант *guided*. Він схожий на *dynamic*, проте розмір порції зменьшується експоненційно. Така стратегія упорядкування дозволяє боротись з проблемою, яка виникає, коли відношення потоків до ітерацій циклу велике. Тоді видатки на створення різних потоків перевищують дохід від розпаралелення. Така стратегія схожа на тип алгоритмів “жадібні алгоритми”. Вона не гарантує найкраший результат, проте є оптимальною в багатьох випадках.

Динамічне та кероване планування добре підходять, коли при кожній ітерації виконується різні об’єми роботи або якщо процесори більш виробничими, ніж інші. При статичному плануванні немає способу, який дозволяє сбалансувати нагрузку на різні потоки. При динамічному та керованому плануванні нагрузка розподіляється автоматично — такова сама суть цих підходів. Як правило, при керованому плануванні код виконується швидше, ніж при динамічному, в наслідок менших витрат на планування.

***Ordering* (впорядкування)**

Порядок, в якому будуть обрабляти ітерації цикла непердбачуваним. Проте можливо “змусити” *OpenMP* виконувати вирази в циклі в порядку. Для цього існує ключеве слово *ordered*[1].

Хоча більшість ітеративних задач є аккумулятивними, тобто такими, для яких немає різниці у порядку виконання ітерацій, існують і ті, що потребують порядку. Для задач таких, як сумування, множення, перетворення кожного елементу масиву таким чином, який не залежить від значень інших елементів масиву, видбір елементів масиву, що відповідають певному крітерію, немає різниці в порядку. Проте для копіювання масиву, створення нового масиву з зворотнім порядком, виводу в один і той же *stream* грає роль порядок. Саме для таких задач є можливість вказання порядку для паралелювання цикла в *OpenMP*.

**Директива *sections***

Неітеративна паралельна конструкція. Визначає набір незалежних секцій кода (“кінечний паралелізм”). Секції відділяються друг від друга директивой. На відміну від директиви *parallel* секції кода виконується не всіма потоками, а кожна секція виконується окрмемим потоком. Це гарна заміна використанню функції *OPM\_GET\_THREADS\_NUM* з конструкцією *switch*, яка теж дозволяє задати, що робить поток, основуючись на його номеру по порядку. Директива *sections* менш об’ємна, проте не залежить від кількості наявних потоків. Якщо *switch* приймає у клаузи *case* лише константні вирази, що говорить про те, що кількість потоків не можливо змінити динамічно, то *OpenMP* сам проводить розрахунок логіки, за якою розподіляються паралельні секція по потах. Це показує, що якщо потоків більше ніж секція або навпаки, *OpenMP* має готовий алгоритм для таких ситуацій.

**Директива *single***

Визначає блок кода, який виконається тільки одним потоком (першим, який дійде до цього блока). Така можливість дає змогу легко проводити ініціалізації різних структур або ресурсів. В деяких випадках повторна ініціалізація змінних може затерти результати, які вже були отримані іншим потоком. Ще більш неприйнятна ситуація трапляється, коли ініціалізованою структурою не проста зміна типу число, буква, рядок чи флаг, а структурою є багаторозмірний масив, дерево чи масив, який потребує початкового сортування. Тоді обчислювальноємкі задачі будуть повторно витрачати час процесора, що є дуже неефективно. Ця неефективність походить від багатьох непотрібних обчислень, та існує неефективність, пов’язана з витратою великої кількості пам’яті. Якщо робота з одним і тим же зовнішним ресурсом, такими як файл, база даних, віддалений сервер, спричинятиме відкриття цього ресурса в кожному потоці, це призвиде до великої кількості відкритих дискрипторів в операційній системи, кожен з яких через паралельність може не закриватись або закриватись двічі. Саме ці проблеми вирішує директива *single*.

**Явне керування розподіленням роботи**

За допомогою функції *OMP\_GET\_THREAD\_NUM* та *OMP\_GET\_NUM\_THREADS* потік може визначити свій номер та загальну кількість потоків, а далі виконати свою частину роботи в залежності від свого номеру (цей підхід широко виспользується у програмах на базі інтерфейса).

**Директиви синхронізації**

*Master —* визначає блок коду, який виконається тільки *master*-ом (нульовим потоком). Ця директива схоже на директиву *single*, проте несе інший логічний зміст. Ця директива виконується нульовим потоком, тоже дії в серединні блока коду повинні бути направленні на керування потоками, ніж на обчислення. До того ж на відміну від директиви *single* код блока виконається раніше, бо нульовий потік в загальному випадку має менш задач ніж будь-який інший.

*Critical* — визначає критичну секцію, блок кода, яикй не повинен виконуватись одночасно двумя або більшою кількістю потоків. Критична секція — розповсюджений паттерн паралельного програмування. ЇЇ реалізують різноманітними засобами: мьютексами, семафорами, моніторами. Розробники *OpenMP* винесли цей паттерн в окрему абстракцію, що призвело до зручності написання коду: не потрібно створювати додаткові об’єкти, *OpenMP* автоматизує цей процес.

*Barrier* — визначає точку бар’єрної синхронизації, в якій кожний потік чекає всіх інших. Ця директива відіграє роль функції *join* в моделі *fork/join*. При цьому *fork*-ом є початок паралельної секції. Такий елемент синхронізації є зручним для неідеально паралельних задач або поєднання двох паралельних задач так, щоб початок першого потоку другої задачі знаходився після кінця останього потоку першої задачі. Ця директива аналогічна послідовної об’яві двох паралельних секцій.

*TaskWait* — визначає очікування для закінчення задач потомків потоків, які були сгенеровані після початку поточної задачі.

Зважаючи на те, що конструкція *taskwait* немає виразу мови *C*, як чистини свого синтаксису, присутні деякі заборони для вибору місця використання у програмі. Директива *taskwait* може бути розміщена тілька в точці, де вирази базової мови дозволені. Директива *taskwait* не може бути використана в місці, яке слідує за виразами *if, while, do, switch* або *label*.

*Atomic* — визначає змінну в лівій частині оператора “атомарного” присвоювання, який повинен коректно оновлюватись декільками потоками. Ця директива дозволяє замінити мьютекс, який потрібен був би для зміни змінної декільками потоками. Потрібно зважати на те, що *atomic* змінні працюють повільніше за звичайні змінні. Це означає, що, якщо змінна у програмі не змінюється, то краще використовувати звичайні змінні.

*Flush* — явно визначає точку, в якій реалізації повинна забезпечувати однаковий вид пам’яті для всіх потоків. Неявно *FLUSH* присутній в наступних директивах: *BARRIER, CRITICAL, END CRITICAL, END DO, END PARALLEL, END SECTIONS, END SINGLE, ORDERED, END ORDERED.*

В цілях синхронизації можна також використовувати механізмом замків (*locks*).

**1.2.9. Класи змінних**

В *OpenMP* змінних у паралельних областях програми розділяються на два основних класа:

- *SHARED* (загальні; з ім’ям А всі потоки бачать одну змінну) та

- *PRIVATE* (приватні; з ім’ям А кожен потік бачить свою змінну).

Окремі правила визначають поведінку змінних при вході та виходів з паралельної області або паралельного цикла: *REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN*.

За замовчанням, всі *COMMON-*блоки, а також змінні, які породжені поза паралельної області, при вході в цю область залишаються спільними (*SHARED*). Виключенням є змінні — лічильники ітерацій в циклі. Змінні, які породженні у паралельній області є приватними (*PRIVATE)*. Явно назначити клас змінних за замовченням можна за допомогою клаузи *DEFAULT.*

*SHARED —* використовується до змінних, які необхідно зробити спільними.

*PRIVATE —* використовується до змінних, які необхідно зробити приватними. При вході в паралельну область для кожного потоку створюється окремий екземпляр змінної, який не має ніякого зв’язку з оригінаьною змінною поза паралельною областю.

*THREADPRIVATE* — використовується к *COMMON-*блокам, які необхідно зробити приватними. Директива повинна бути після кожної декларації *COMMON*-блока.

*FIRSTPRIVATE —* приватні копії змінної при вході в паралельну область ініціалізуються значенням оригінаьної змінної.

*LASTPRIVATE* — після кінця паралельного цикла або блока паралельних секцій, потік, яка виконала останню ітерацію цикла або останню сецкію блока, оновлює значення оригінальної змінної.

*REDUCATION(+ :A)* — означає змінну, з якою в циклі проводиться *reduction*-операція (наприклад, сумування). При виході з цикла, данна операція проводиться над копіями змінної в усіх потоках, та результат присвоються оригінальной змінній.

*COPYIN* — застосовується к *COMMON*-блокам, які помічені як *THREADPRIVATE*. При вході в паралельну область приватні копії цих данних ініціалізуються оригінальними значеннями.

***Runtime*-процедури та змінні середовища**

В цілях створення середовища запуска паралельних програм, які можна перенести, в *OpenMP* визначен ряд змінних середовища, які контролюють поведінку додатка.

В *OpenMP* передбачен також набір бібліотечних процедур, які дозволяють:

- під час виконання контролювати та запитувати різні параметри, які визначають поведінку додатку (такі як кількість потоків та процесорів, можливість вкладеного паралелізма); процедури назначення параметрів мають пріоритет над відповідними змінними середи.

- використовувати синхронізацію на базі замків (*locks*).

**Змінні середовища**

*OMP\_SCHEDULE* — визначає розподілення ітерацій в циклі, якщо в директиві *DO* використана клауза *SCHEDULE(RUNTIME)*.

*OMP\_NUM\_THREADS* — визначає кількість потоків для виконання паралельних областей додатка.

*OMP\_DYNAMIC* — дозволяє або забороняє динамічне зміну кількості потоків.

*OMP\_NESTED —* дозволяє або заюороняє вкладений паралелізм.

**Процедури для контроля/запроса параметрів середовища виконання**

*OMP\_SET\_NUM\_THREADS —* дозволяє назначити максимальну кількість потоків для використання в наступній паралельній області (якщо цю кількість дозволено динамічно змінювати). Визивається з послідовної області програми.

*OMP\_GET\_MAX\_THREADS* — повертає максимальну кількість потоків.

*OMP\_GET\_NUM\_THREAD* — повертає фактичне число потоків в паралельній області програми.

*OMP\_GET\_NUM\_PROCS* — повертає кількість процесорів, які доступні додатку.

*OMP\_IN\_PARALLEL* — повертає *TRUE*, якщо викликана з паралельної області програми.

*OMP\_SET\_SYNAMIC / OMP\_GET\_DYNAMIC* — встановлює / запитує стан флага, який дозволяє динамічно змінювати кількість потоків.

*OMP\_GET\_NESTED / OMP\_SET\_NESTED* — встановлює / запитує стан флага, який дозволяє вкладений паралелізм.

**Процедури синхронізації на базі замків**

В якості замків використовується загальні змінні типа *INTEGER* (розмір повинен бути достатнім для зберегання адреси). Данні змінні повині використовуватись тільки параметри примітивів синхронізації.

*OMP\_INIT\_LOCK(var) / OMP\_DESTROY\_LOCK(var) —* ініціалізує замок, який пов’язан зі змінною var.

OMP\_SET\_LOCK — змушує потік, який викликав цю процедуру, дочекатись замка, а далі захоплює його.

*OMP\_UNSET\_LOCK* — звільняє замок, якщо він був захоплений потіком, який визвав цю процедуру.

*OMP\_TEST\_LOCK —* пробує захопити вказаний замок. Якщо це неможливо, повертає *FALSE*.

**1.3. Специфікація *OpenMPI для мов C/C++***

Специфікація *OpenMP* для мов *C/C++*, яка випущена на рік пізніше фортраної, містить в основному аналогічну функціональність.

Необхідно лише відмітити наступні моменти:

1) Замість спецкоментарів використовуються директиви компілятора “#pragma omp”.

2) Компілятор з підтримкою *OpenMP* визначає макрос *“\_OPENMP”*, який може використовуватись для умовної компіляції окремих блоків, які характерні для паралельної версії програми.

3) Розпаралелення застосовується к *for*-циклам, для цього використовується директива *“#pragma omp for”*. В паралельних циклах забороняється використовувати оператор *break.*

4) Статичні (*static*) змінні, які визначенні в паралельній області програми є спільними (*shared*).

*5)* Пам’ять, яка виділена за допомогою *malloc()*, є спільною (проте вказівник на неї може бути, як спільним, так і приватним).

6) Типи та функції *OpenMP* визначені в файлі *<omp.h>.*

7) Крім звичайних, можливі також “вкладені” (*nested*) замки — замість логічних змінних використовуються цілі числа, та потік, яка вже захопила замок, при повторному захопленні може збільшити це число[2].

**1.4. Порівняння спільної та розподіленої пам’яті**

Машини, що можуть проводити паралельни обчислення, можуть мати різні типи пам’яті: розподілену та спільну.

**Розподілена пам’ять**

Цей тип характеризується такими ознаками:

- кожен процесор має свій адресний простір пам’яті.

- значення змінних незалжені. На одному процесорі *x* може дорівнювати 2, і водночас в іншому — 3.

- приклади: кластера лінукс, *Blue Gene/L*.

**Спільна пам’ять**

Цей тип характерезується такими ознаками:

- також називається *Symmetric Multiprocessing (SMP)*.

- один адресний простір для всіх процесорів. Якщо один процесор присвоює *х* значення 2, то *x* дорівнює 2 на інших процесорах.

- приклади: *IBM p-series*, багатоядерні персональні комп’ютери.

**Використання різних типів пам’яті**

Декілька процесів:

- кожен процесор виконує незалежну задачу з його власним адресним пространстов пам’яті.

Декілька потоків:

- процес створює додаткові задачі (потоки) з тим самим адресним простором пам’яті[3].

**1.5. Розмір стеку та прив’язка потоків**

**Розмір стеку**

Стандарт *OpenMP* не вказує об’єм пам’яті стеку, який має мати кожен потік. Саме тому різні реалізації можуть мати різний розмір стеку.

За замовчуванням розмір стеку є досить малим. Для прикладу приведено розміри стеку та розмріи доступних масивів типу *double* в табл. 1.5.1.1.

|  |  |  |
| --- | --- | --- |
| **Компілятор** | **Приблизний розмір стеку** | **Приблизний розмір масиву** |
| *Linux icc, ifort* | 4 *MB* | 700 x 700 |
| *Linux pgcc, pgf90* | 8 *MB* | 1000 x 1000 |
| *Linux gcc, gfortran* | 2 *MB* | 500 x 500 |

Табл. 1.5.1.1

Потоки, які використають повністю їх стек, мають непердбачену поведінку. Вони можуть завершитись з помилкою *seg fault*, а можуть і ні. Додаток загалом може продовжувати роботу навіть попри те, що данні спотворено.

Статично злінкований код може бути опорною причиною для подальших меж об’єму стека.

Також обмеження поточного користувача можуть зменшити розмір стека.

Якщо реалізація *OpenMP* підтримує стандарт *OpenMP* 3.0, можливо використати *OMP\_STACKSIZE* змінну середовища для того, щоб задати розмір стеку для кожного потоку перед виконанням програми. Наприклад:

- *setenv OMP\_STACKSIZE 2000500B*

*- setenv OMP\_STACKSIZE “300 k “*

*- setenv OMP\_STACKSIZE 10M*

*- setenv OMP\_STACKSIZE “ 10 M “*

*- setenv OMP\_STACKSIZE “20 m “*

*- setenv OMP\_STACKSIZE “ 1G”*

*- setenv OMP\_STACKSIZE 20000*

**Прив’язка потоків**

В деяких випадках програма працює краще, якщо потоки прив’язані до процсорів/ядер.

“Прив’язка” потоку до процесора означає, що потік буде запланован операційною системою завжди запускатись на тому ж самому процесорі. Інакше, потоки можуть бути заплановані виконуватись на будь-якому процесорі та “скакати” між процесорами з кожним інтервалом часу.

Також це називається “потокова близькість” або “процесорна близкість”.

Прив’зка потоків до процесорів може привести до кращою утилізації кешей, що призвиде до зменшення дорогого доступу до пам’яті. Це головна мотивація для прив’язки потоків до процесорів.

Зважаючи на платформу, операційну систему, компілятор та *OpenMP* реалізацію, прив’язка потоків до процесорів може бути зроблена декількома різними шляхами.

*OpenMP 3.1 API* впроваджує змінні середовища для ввімкнення або вимкнення процесорної прив’язки. Наприклад:

***setenv OMP\_PROC\_BIND TRUE***

***setenv OMP\_PROC\_BIND FALSE***

На більшом високому рівні абстракії до процесорів можуть прив’язуватись процеси замість потоків.

**1.6. Нагляд, налагодження та інструменти аналізу ефективності для *OpenMP***

**Нагяд та налогодження потоків**

Налагоджувачі різняться в їхній здібності оброблювати потоки. *TotalView debugger* є рекомендованим налагоджувачем для паралельних програм. Він добре підходить водночас і для наглядом та налагодженням поточних програм.

*TotalView* має такі елементи:

1. Панель st*ack trace* головного потоку.
2. Панель для розрізнення процесів/потоків.
3. Панель stack frameголовного потоку для показу спільних змінних.
4. Панель для показу *stack trace* для неголовних потоків.
5. Панель *stack frame* для неголовних потоків.
6. Головне вікно, яке показує всі потоки.
7. Панель потоків, яка показує всі потоки та обраний потік.

Команда *ps* в лінукс забезпечує деякі прапори для просмотру інформації потоку. Деякі приклади наведено нижче:

*% ps -Lf*

UID PID PPID LWP C NLWP STIME TTY TIME CMD

blaise 22529 28240 22529 0 5 11:31 pts/53 00:00:00 a.out

blaise 22529 28240 22530 99 5 11:31 pts/53 00:01:24 a.out

blaise 22529 28240 22531 99 5 11:31 pts/53 00:01:24 a.out

blaise 22529 28240 22532 99 5 11:31 pts/53 00:01:24 a.out

blaise 22529 28240 22533 99 5 11:31 pts/53 00:01:24 a.out

% *ps -T*

**PID SPID TTY TIME CMD**

22529 22529 pts/53 00:00:00 a.out

22529 22530 pts/53 00:01:49 a.out

22529 22531 pts/53 00:01:49 a.out

22529 22532 pts/53 00:01:49 a.out

22529 22533 pts/53 00:01:49 a.out

*% ps -Lm*

PID LWP TTY TIME CMD

22529 **-** pts/53 00:18:56 a.out

**- 22529 - 00:00:00 -**

**- 22530 - 00:04:44 -**

**- 22531 - 00:04:44 -**

**- 22532 - 00:04:44 -**

**- 22533 - 00:04:44 -**

Кластера лінукс також забезпечують команду *top* для нагяду за процесами на елементі кластера. Якщо використовувати флаг *-H*, потоки, які містяться у данному процесі.

**Інструмент для аналізу ефективності**

Є великиа кількість інструментів для аналізу ефективності, які можна використати з *OpenMP*. Приклади наведено нижче:

- *Open | SpeedShop*

*- TAU*

*- PAPI*

*- Intel Vtune Amplifier*

*- ThreadSpotter*[5]

**Висновки:**

1. Виконан аналіз засобів написання програм, які використовують паралельну систему обчислення, за допомогою библіотеки *OpenMP*. Показано, що паралельність програм досягається за рахунок використання директив *#pragma omp*, що дозволяє компілювати код програми, який написан для паралельного виконаня, з компілятором, який не підтримує *OpenMP*, проте з утратою паралельності.

2. Розглянуто директиви для розпаралелення циклів та директиви для балансування навантаження при цьому. Показано, що різні задачі потребують різних стратегій навантаження: статичної, динамочної або керованої.

3. Розглянуто типи змінних в залежності від їх поведінки у різних потоках. Показано, що зміна типу змінної за допомогою спеціальних директив може змістовно змінити роботу програми.

4. Проведено аналіз систем з спільною пам’яттю та розподіленою. Показано, що використання кожного типа на відповіному йому рівні краще за використання тільки одного типу пам’ятті.

5. Наведені приклади інструментів для роботи за паралельними програми. Показано, які засоби є можливи для аналізу роботи потоків. Наведено приклади використання цих інструментів.

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС ОП

**2.1. Розробка паралельного математичного алгоритму.**

Рис. 2.1. Структура ПКС ОП

Пристрої

вводу-виводу

ОП

Z, MO, A

1

E, S, MT

P

Пристрої

вводу-виводу

ОП

Z, MO, A

1

E, S, MT

P

2.2. Розроботка алгоритмів процесів.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Ввести Z, MO. 2. Сигнал про завершення введення до всіх потоків. 3. Чекати сигнала про завершення введення від потока . 4. Обрахувати .. 5. Обрахувати . 6. Сигнал про завершення обрахунку до всіх потоків. 7. Чекати сигнал про завершення обрахунку від всіх потоків. 8. Копіювати . 9. Обрахувати . 10. Сигнал про завершення обрахунку до всіх потоків. 11. Чекати сигнал про завершення обрахунку від всіх потоків. 12. Обрахувати 13. Сигнал про завершення обрахунку до всіх потоків. 14. Чекати сигнал про завершення обрахунку від всіх потоків. 15. Обрахувати . 16. Сигнал про завершення обрахунку до всіх потоків. 17. Чекати сигнал про завершення обрахунку від всіх потоків. 18. Обрахувати . 19. Чекати сигнал про завершення обрахунку від всіх потоків. 20. Вивести . |  | 1. Чекати сигнала про завершення введення від та . 2. Обрахувати. 3. Обрахувати . 4. Сигнал про завершення обрахунку до всіх потоків. 5. Чекати сигнал про завершення обрахунку від всіх потоків. 6. Копіювати . 7. Обрахувати . 8. Сигнал про завершення обрахунку до всіх потоків. 9. Чекати сигнал про завершення обрахунку від всіх потоків. 10. Обрахувати 11. Сигнал про завершення обрахунку до всіх потоків. 12. Чекати сигнал про завершення обрахунку від всіх потоків. 13. Обрахувати . 14. Сигнал про завершення обрахунку до всіх потоків. 15. Чекати сигнал про завершення обрахунку від всіх потоків. 16. Обрахувати . 17. Сигнал про завершення обрахунку до . |  |

|  |  |
| --- | --- |
| 1. Ввести E, S, MT. 2. Сигнал про завершення введення до всіх потоків. 3. Чекати сигнала про завершення введення від потока . 4. Обрахувати .. 5. Обрахувати . 6. Сигнал про завершення обрахунку до всіх потоків. 7. Чекати сигнал про завершення обрахунку від всіх потоків. 8. Копіювати . 9. Обрахувати . 10. Сигнал про завершення обрахунку до всіх потоків. 11. Чекати сигнал про завершення обрахунку від всіх потоків. 12. Обрахувати . 13. Сигнал про завершення обрахунку до всіх потоків. 14. Чекати сигнал про завершення обрахунку від всіх потоків. 15. Обрахувати . 16. Сигнал про завершення обрахунку до всіх потоків. 17. Чекати сигнал про завершення обрахунку від всіх потоків. 18. Обрахувати . 19. Чекати сигнал про завершення обрахунку від всіх потоків. 20. Вивести . |  |

2.3. Розробка схеми взаємодії процесів

Рис. 2.2. Схема взаємодії потоків

**T1**

**Ti**

**Tp**

zUpdate

dataCopy

**T1**

**Ti**

**Tp**

zUpdate

dataCopy

Критичні секції zUpdate та dataCopy використовуються для доступу к спільним ресурсам. При цьому zUpdate використовується, як для читання, так і для запису в змінну z. Критична секція dataCopy використовується лише для читання зі змінних z, E, MO.

Бар’єр, який зображено за допомогою S1 та W1, потрібен для синхронізації потоків після вводу данних.

Бар’єр, який зображено за допомогою S2 та W2, потрібен для синхронізації потоків після обрахунку максимального z.

Бар’єр, який зображено за допомогою S3 та W3, потрібен для синхронізації потоків після обрахунку множення матриць.

Бар’єр, який зображено за допомогою S4 та W4, потрібен для синхронізації потоків після сортування вектору S.

Кінець паралельної секції, який зображено за допомогою S5 та W5, потрібен для синхронізації потоків для виводу результату загальної задачі.

2.4. Розробка програми ПРГ1

Вхідні данні: Z, MO, E, S, MT.

Вихідні данні: A.

Тимчасові данні: z, B, C.

mval — максимальне значення серед .

z — максимальне значення серед Z.

MO1, E1, z1 — копії MO, E та z для обробки потоком відповідно.

X — елемент матриці, яка є результатом множення MO та MT.

C — відсортований S.

ind — масив індексів для сортування цільного масиву S.

mini — індекс частини S, яка має максимальний елемент у місці, вказаному індексом масиву ind.

ReadArr — функція для створення масиву заданої довжини, заповненого одиницями.

ReadVec — створення масиву довжиною N, заповненого одиницями.

ReadMat — створення масиву довжиною N\*N, заповненого одиницями.

Arrcpy — функція для копіювання значень з іншого масиву заданої довжини.

Veccpy — функція для копіювання масиву довжиною N.

Matcpy — функція для копіювання масиву довжиною N\*N.

get\_timestamp — функція для отримання поточного часу.

2.5. Тестування програми ПРГ1

2.6. Висновки до розділу 2

4. Основні результати та висновки

5. Перелік умовних позначень

6. Список використаних джерел

1. Kos66, Введение в OpenMP: параллельное программирование на C++ // Intel Developer Zone, 2011. [Електорнний ресурс] — Режим доступу: <https://software.intel.com/ru-ru/blogs/2011/11/21/openmp-c>
2. Что такое OpenMP? // Лаборатория Паралельніх Информационних Технологий, НИВЦ МГУ. [Електорнний ресурс] — Режим доступу: https://parallel.ru/tech/tech\_dev/openmp.html
3. Sondak D. Parallel Processing with OpenMP // Boston University, Scientific Computing and Visualization Office of Information Technology. [Електорнний ресурс] — Режим доступу: <http://www.compunity.org/training/tutorials/openmp_Boston.pdf>
4. Blaise B. OpenMP // Lawrence Livermore National Laboratory. [Електорнний ресурс] — Режим доступу: <https://computing.llnl.gov/tutorials/openMP/>

ДОДАТОК А. Лістінг ПРГ1

/\*

// main

// Author:

// Dzyuba Vlad, IP-42

\*/

#include <iostream>

#include <omp.h>

#include <string.h>

#include <stdlib.h>

#include <sys/time.h>

typedef unsigned long long timestamp\_t;

timestamp\_t get\_timestamp () {

struct timeval now;

gettimeofday (&now, NULL);

return now.tv\_usec + (timestamp\_t)now.tv\_sec \* 1000000;

}

using namespace std;

int N, P, H;

// functions for generating structures

int\* readVec();

int\* readMat();

// functions for copying structures

int\* veccpy(int \*src);

int\* matcpy(int \*src);

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

N = atoi(argv[1]);

P = atoi(argv[2]);

H = N / P;

// input data

int \*Z, \*MO, \*E, \*S, \*MT;

// output data

int \*A = new int[N];

// intermidiate data

int z = -65536, \*B = new int[N], \*C = new int[N];

timestamp\_t startTime = get\_timestamp();

#pragma omp parallel num\_threads(P)

{

int tid = omp\_get\_thread\_num();

// generate data if first or last thread

if (tid == 0) {

Z = readVec();

MO = readMat();

}

if (tid == P - 1) {

E = readVec();

S = readVec();

MT = readMat();

}

// finding maximum of Z

#pragma omp barrier

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

int mval = Z[i\*H];

for (int j = i\*H+1; j < (i+1)\*H; j++) {

mval = Z[j] > mval ? Z[j] : mval;

}

#pragma omp critical(zUpdate)

if (mval > z) {

z = mval;

}

}

#pragma omp barrier

// calculating B

int \*MO1, \*E1, z1;

#pragma omp critical(dataCopy)

{

MO1 = matcpy(MO);

E1 = veccpy(E);

z1 = z;

}

#pragma omp for

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

for (int j = i \* H; j < (i + 1) \* H; j++) {

B[j] = 0;

for (int k = 0; k < N; k++) {

int x = 0;

for (int l = 0; l < N; l++) {

x += MO1[k\*N+l] \* MT[l\*N+j];

}

B[j] += x \* E1[k];

}

B[j] \*= z1;

}

}

#pragma omp barrier

// sorting parts of S

#pragma omp for

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

for (int j = 1; j < H; j++) {

for (int k = i \* H; k < (i + 1) \* H - j; k++) {

if (S[k] > S[k + 1]) {

int x = S[k];

S[k] = S[k + 1];

S[k + 1] = x;

}

}

}

}

#pragma omp barrier

// sorting whole S and stores into C

#pragma omp single

{

int\* ind = new int[N];

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

ind[i] = 0;

}

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

int mini = -1;

for (int j = 0; j < N; j++) {

if (ind[j] < N) {

if (mini == -1 || S[ind[j]] < S[ind[mini]]) {

mini = j;

}

}

}

C[i] = S[ind[mini]];

ind[mini]++;

}

}

#pragma omp barrier

// calculating A

#pragma omp for

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

A[i] = B[i] + C[i];

}

}

// print A

if (N <= 20) {

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

cout << A[i] << " ";

}

cout << endl;

} else {

timestamp\_t endTime = get\_timestamp();

cout << (endTime - startTime) / 1000000.0 << endl;

}

delete[] A;

}

int\* readArr(int n) {

int\* res = new int[n];

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

res[i] = 1;

}

return res;

}

int\* readVec() { return readArr(N); }

int\* readMat() { return readArr(N\*N); }

int\* arrcpy(int \*src, int n) {

int \*res = new int[n];

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

res[i] = src[i];

}

return res;

}

int\* veccpy(int \*src) { return arrcpy(src, N); }

int\* matcpy(int \*src) { return arrcpy(src, N \* N); }