НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

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

(повна назва кафедри, циклової комісії)

**КУРСОВА РОБОТА**

з дисципліни «Паралельні та розподілені обчислення»

(назва дисципліни)

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

Студента (ки) 3 курсу \_\_\_\_\_\_ групи

напряму підготовки 050102 «Комп’ютерна інженерія»

Івченко Є.Г.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(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- 2015 рік

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 6.050102 «Комп’ютерна інженерія»

# (шифр і назва)

## З А В Д А Н Н Я

### НА КУРСОВУ РОБОТУ СТУДЕНТУ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Івченко Євгеній Германович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(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 18 травня 2015 р.

3. Вхідні дані до роботи

- засоби роботи з процесами в бібліотеці OpenMP

- математична задача МА = МВ\*МС

- структури ПКС ОП та ПКС ЛП

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

- огляд засобів роботи з процесами в бібліотеці OpenMP

- 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 \_\_\_\_\_\_\_XXX\_\_\_\_!!!!\_\_

#### КАЛЕНДАРНИЙ ПЛАН

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання Розділу 1 | 23.02.2015 |
| 2 | Виконання Розділу 2 | 23.03.2015 |
| 3 | Виконання Розділу 3 | 23.04.2015 |
| 4 | Тестування програм ПРГ1 та ПРГ2 | 10.05.2015 |
| 7 | Оформлення КР | 17.05.2015 |
| 8 | Захист КР | 18.05.2015 |

**Студент \_\_\_\_\_\_\_\_\_ \_\_\_**Івченко Є.Г.\_\_\_\_\_\_\_\_\_\_\_

( підпис ) (прізвище та ініціали)

**Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_** \_\_\_\_\_\_\_\_\_\_\_\_Корочкін О.В.\_\_\_\_\_\_\_\_\_\_

( підпис ) (прізвище та ініціали)

**ЗМІСТ**

РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ З ПРОЦЕСАМИ В БІБЛІОТЕЦІ OPEN MP……………………………………………………………………..……5

* 1. Історія створення та розробки бібліотеки OpenMP…………………5
  2. Основні принципи роботи бібліотеки OpenMP……………………...6
  3. Засоби програмування процесів в бібліотеці OpenMP……………..7
  4. Синхронізація процесів в бібліотеці OpenMP……………………...16
  5. Висновки до розділу 1………………………………………………..20

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС ОП…………………20

* 1. Розробка паралельного математичного алгоритму……………… 20
  2. Розробка алгоритмів процесів……………………………………… 22
  3. Розробка схеми взаємодії процесів…………………………………24
  4. Розробка програми ПРГ1…………………………………………… 26
  5. Тестування програми ПРГ1 …………………………………………27
  6. Висновки до розділу 2 …………………………………………… 29

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП ……..………… 30

* 1. Розробка паралельного математичного алгоритму……………… 30
  2. Розробка алгоритмів процесів…………………………………… 32
  3. Розробка схеми взаємодії процесів………………………… … 34
  4. Розробка програми ПРГ2……………………………………… 36
  5. Тестування програми ПРГ2……………………………………… 37
  6. Висновки до розділу 3………………………………………………39

ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ДО РОБОТИ………………………40

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ……………………………………. 43

ДОДАТКИ……………………………………………………............................. 45

**ВСТУП**

Курсова робота по дисципліні «Паралельні і розподілені обчислення» складається з трьох розділів.

В першому розділі «Порівняння реалізації механізму семафорів в мовах і бібліотеках паралельного програмування» описаний механізм семафорів в паралельних програмах, а також його застосування в різних мовах та бібліотеках програмування.

Другий та третій розділи присвячені розробці програми для обчислення математичної задачі в паралельній комп’ютерній системі зі спільною та локальною пам’яттю відповідно. Програмне забезпечення для комп’ютерної системи зі спільною та локальною пам’яттю розроблено на мові Аdа із використанням механізму захощенного модуля та Рандеву. Проведено тестування отриманих програмних продуктів і зроблено висновки по їх ефективності.

Лістинги та алгоритми розроблених програм наведено у додатках.

**РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ С ПРОЦЕСАМИ В БІБЛІОТЕЦІ OPENMP.**

OpenMP - це інтерфейс прикладного програмування для створення багатопоточних додатків, призначених в основному для паралельних обчислювальних систем зі спільною пам'яттю. OpenMP складається з набору директив для компіляторів і бібліотек спеціальних функцій.

* + 1. **Історія створення та розробки бібліотеки Open MP**

Стандарти OpenMP розроблялися протягом останніх 15 років для архітектури із загальною пам'яттю. Опис стандартів OpenMP та їх реалізації при програмуванні на алгоритмічних мовах Fortran і C / C ++. Найбільш повно питання програмування на OpenMP розглянуті в монографіях. В останні роки дуже активно розробляється розширення стандартів OpenMP для паралельних обчислювальних систем з розподіленою пам'яттю. В кінці 2005 року компанія Intel анонсувала продукт Cluster OpenMP, який реалізує розширення OpenMP для обчислювальних систем з розподіленою пам'яттю. Цей продукт дозволяє оголошувати області даних, доступні всім вузлам кластера, і здійснювати передачу даних між вузлами кластера неявно за допомогою протоколу Lazy Release Consistency. OpenMP дозволяє легко і швидко створювати багатопотокові програми на алгоритмічних мовах Fortran і C / C ++. При цьому директиви OpenMP аналогічні директивам препроцесора для мови C / C ++ і є аналогом коментарів у алгоритмічній мові Fortran. Це дозволяє в будь-який момент розробки паралельної реалізації програмного продукту при необхідності повернутися до послідовного варіанту програми. Відзначимо, що компанія Intel не є винахідником цього підходу - раніше такі спроби робилися в Японії.

АБЗАЦЫ!! ОТСТУПЫ!!! ПЕРЕНОСЫ!!

* 1. **Основні принципи роботи бібліотеки OpenMP.**

Будь-яка програма, послідовна або паралельна, складається з набору областей двох типів: послідовних областей і областей розпаралелювання. При виконанні послідовних областей породжується тільки один головний потік (процес). У цьому ж потоці ініціюється виконання програми, а також відбувається її завершення. У послідовній програмі в областях розпаралелювання породжується також тільки один, головний потік, і цей потік є єдиним протягом виконання всієї програми. У паралельній програмі в областях розпаралелювання породжується ціла низка паралельних потоків. Породжені паралельні потоки можуть виконуватися як на різних процесорах, так і на одному процесорі обчислювальної системи. В останньому випадку паралельні процеси (потоки) конкурують між собою за доступ до процесора. Управління конкуренцією здійснюється планувальником операційної системи за допомогою спеціальних алгоритмів. В операційній системі Linux планувальник завдань здійснює обробку процесів за допомогою стандартного карусельного (round-robin) алгоритму. При цьому тільки адміністратори системи мають можливість змінити або замінити цей алгоритм системними засобами. Таким чином, в паралельних програмах в областях розпаралелювання виконується ряд паралельних потоків. Принципова схема паралельної програми зображена на малюнку.

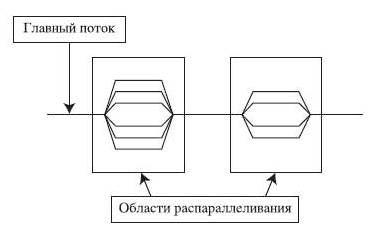


Рис. 1.1 Принципова схема паралельної програми.

При виконанні паралельної програми робота починається з ініціалізації та виконання головного потоку (процесу), який у міру необхідності створює і виконує паралельні потоки, передаючи їм необхідні дані. Паралельні потоки з однієї паралельної області програми можуть виконуватися як незалежно один від одного, так і з пересилкою та отриманням повідомлень від інших паралельних потоків. Остання обставина ускладнює розробку програми, оскільки в цьому випадку програмісту доводиться займатися плануванням, організацією і синхронізацією посилки повідомлень між паралельними потоками. Таким чином, при розробці паралельної програми бажано виділяти такі області розпаралелювання, в яких можна організувати виконання незалежних паралельних потоків. Для обміну даними між паралельними процесами (потоками) в OpenMP використовуються загальні змінні. При зверненні до загальних змінним в різних паралельних потоках можливе виникнення конфліктних ситуацій при доступі до даних. Для запобігання конфліктів можна скористатися процедурою синхронізації (synchronization). При цьому треба мати на увазі, що процедура синхронізації - дуже дорога операція по тимчасових витратах і бажано по можливості уникати її або застосовувати якомога рідше. Для цього необхідно дуже ретельно продумувати структуру даних програми.

Виконання паралельних потоків в паралельній області програми починається з їх ініціалізації. Воно полягає у створенні дескрипторів породжуваних потоків і копіювання всіх даних з області даних головного потоку в області даних створюваних паралельних потоків. Ця операція надзвичайно трудомістка - вона еквівалентна приблизно трудомісткості 1000 операцій. Ця оцінка надзвичайно важлива при розробці паралельних програм c допомогою OpenMP, оскільки її ігнорування веде до створення неефективних паралельних програм, які виявляються найчастіше повільніше їх послідовних аналогів. Справді: для того щоб отримати виграш у швидкодії паралельної програми, необхідно, щоб трудомісткість паралельних процесів в областях розпаралелювання програми істотно перевершувала б трудомісткість породження паралельних потоків. В іншому випадку ніякого виграшу за швидкодією отримати не вдасться, а часто можна опинитися навіть і в програші.

Завершення виконання паралельних потоків управління програмою знову передає виконання головному потоку. При цьому виникає проблема коректної передачі даних від паралельних потоків головного. Тут важливу роль відіграє синхронізація завершення роботи паралельних потоків, оскільки в силу цілого ряду обставин час виконання навіть однакових по трудомісткості паралельних потоків непередбачувано (воно визначається як історією конкуренції паралельних процесів, так і поточним станом обчислювальної системи). При виконанні операції синхронізації паралельні потоки, вже завершили своє виконання, простоюють і чекають завершення роботи самого останнього потоку. Природно, при цьому неминуча втрата ефективності роботи паралельної програми. Крім того, операція синхронізації має трудомісткість, порівнянну з трудомісткістю ініціалізації паралельних потоків, тобто еквівалентна приблизно трудомісткості виконання 1000 операцій.

На підставі викладеного вище можна зробити наступний важливий висновок: при виділенні паралельних областей програми та розробці паралельних процесів необхідно, щоб трудомісткість паралельних процесів була не менш 2000 операцій ділення. В іншому випадку паралельний варіант програми буде програвати у швидкодії послідовній програмі. Для ефективної працюючої паралельної програми ця межа має бути істотно перевищена.

* 1. **Засоби програмування процесів в бібліотеці OpenMP.**

За рахунок ідеї "часткового розпаралелювання" OpenMP ідеально підходить для розробників, бажаючих швидко розпаралелити свої обчислювальні програми з великими паралельними циклами. Розробник не створює нову паралельну програму, а просто додає в текст послідовної програми OpenMP директиви. Передбачається, що OpenMP-програма на однопроцесорній платформі може бути використана в якості послідовної програми, тобто немає необхідності одночасно підтримувати послідовну і паралельну версії. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs), текст яких наведено в специфікаціях.

OpenMP простий у використанні і включає лише два базових типи конструкцій: директиви pragma і функції виконуючого середовища OpenMP. Директиви pragma, як правило, вказують компілятору, як реалізувати паралельне виконання блоків коду. Всі ці директиви починаються з фрази pragma omp. Як і будь-які інші директиви pragma, вони ігноруються компілятором, що не підтримують конкретну технологію - в даному випадку OpenMP. Кожна директива може мати кілька додаткових атрибутів. Окремо специфікуються атрибути для призначення класів змінних, які можуть бути атрибутами різних директив.

Функції OpenMP служать в основному для зміни і отримання параметрів оточення. Крім того, OpenMP включає API-функції для підтримки деяких типів синхронізації. Щоб задіяти ці функції OpenMP бібліотеки періоду виконання (виконуючого середовища), в програму потрібно включити заголовний файл omp.h. Якщо ж використовується в додатку тільки OpenMP-директиви pragma, включати цей файл не потрібно.

Директива parallel створює паралельну область для наступного за нею структурованого блоку, паралельна область задається за допомогою запису:

#pragma omp\_parallel [опції] {тіло}

Можливі опції:

* if (умова) - виконання паралельної області за умовою. Входження в паралельну область здійснюється тільки при виконанні деякої умови. Якщо умова не виконана, то директива не спрацьовує і триває обробка програми в колишньому режимі;
* num\_threads (цілочисленний вираз) - явне завдання кількості потоків, які будуть виконувати паралельну область; за замовчуванням вибирається останнє значення, встановлене за допомогою функції omp\_set\_num\_threads (), або значення змінної OMP\_NUM\_THREADS;
* default (shared | none) - всім змінним у паралельній області, яким явно не призначений клас, буде призначений клас shared; none означає, що всім змінним у паралельній області клас повинен бути призначений явно;
* private (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* firstprivate (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* shared (список) - задає список змінних, загальних для всіх потоків;
* copyin (список) - задає список змінних, оголошених як threadprivate, які при вході в паралельну область ініціалізуються значеннями відповідних змінних в потоці-майстра;
* reduction (оператор: список) - задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно типу оператора (для адитивних операцій - 0 або його аналоги, для мультиплікативних операцій - 1 або її аналоги); над локальними копіями змінних після виконання всіх операторів паралельної області виконується заданий оператор; оператор це: +, \*, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску.

Ця директива повідомляє компілятор, що структурований блок коду(тіло) повинен бути виконаний паралельно, в декількох потоках. Кожен потік буде виконувати один і той же потік команд, але не один і той же набір команд - все залежить від операторів, керуючих логікою програми, таких як if-else.

При вході в паралельну область породжуються нові OMP\_NUM\_THREADS-1 потоків, кожний потік отримує свій унікальний номер, причому породжуючий потік отримує номер 0 і стає основним потоком групи ("майстром"). Решта потоків отримують в якості номера цілі числа з 1 до OMP\_NUM\_THREADS-1. Кількість потоків, що виконують дану паралельну область, залишається незмінним до моменту виходу з області. При виході з паралельної області виробляється неявна синхронізація і знищуються всі потоки, крім майстра.

Якщо один з потоків паралельної області зустрічає іншу директиву parallel, то він створює нову групу потоків, згідно з правилами, і стає основним потоком нової групи.

Якщо виконання потоку аварійно переривається всередині паралельної області, то також переривається виконання всіх потоків у всіх групах. Порядок переривання роботи потоків не визначений. Вся робота, виконана групою до останньої бар'єрної синхронізації, гарантовано буде виконана. Обсяг виконаної роботи, виконаної кожним потоком після останньої бар'єрної синхронізації, до аварійного завершення роботи потоків не визначений.

Всі породжені потоки виконують один і той же код, відповідний до паралельної області. Передбачається, що в SMP-системі потоки будуть розподілені по різним процесорам (однак це, як правило, перебуває у віданні операційної системи).

Під час виконання будь потік може призупинити виконання своє неявне завдання в точці планування завдань (task scheduling point) і переключитися на виконання будь-якої явно-згенерованої задачі перш ніж відновити виконання неявної задачі.

Потік може дізнатися свій номер за допомогою виклику бібліотечної функції omp\_get\_thread\_num.

Дуже часто паралельна область не містить нічого, крім конструкції поділу роботи (тобто конструкція поділу роботи тісно вкладена в паралельну область). У цьому випадку можна вказувати не дві директиви, а вказати одну комбіновану.

Наприклад, якщо всередині паралельної області міститься тільки один паралельний цикл або одна конструкція sections, то можна використовувати укорочену запис: parallel for або parallel sections. При цьому допустима вказівка ​​всіх опцій цих директив, за винятком опції nowait.

Обмеження для директиви parallel наступні:

* Програма не повинна залежати від будь-якого порядку визначення опцій паралельної директиви, або від будь-яких побічних ефектів визначення опцій;
* Тільки одна опція if може бути присутня в директиві;
* Тільки одна опція num\_threads може бути присутня в директиві. Вираження в опції num\_threads має бути цілочисловим;
* Виклик винятку виконаний всередині паралельної області повинен викликати обробку виключення в рамках однієї паралельної області, і того ж потоку, який викликав виняток.

Якщо в паралельній області зустрівся оператор циклу, то, згідно загальним правилам, він буде виконаний всіма потоками поточної групи, тобто кожний потік виконає всі ітерації даного циклу. Для розподілу ітерацій циклу між різними потоками можна використовувати директиву for.

#pragma omp for [опція[,опція] ... ]

--цикл for

Ця директива відноситься до блоку що йде слідом за даною директивою.

Можливі опції:

* private (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* firstprivate (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* lastprivate (список) - змінним, перерахованим у списку, присвоюється результат з останнього витка циклу;
* reduction (оператор: список) - задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно типу оператора (для адитивних операцій - 0 або його аналоги, для мультиплікативних операцій - 1 або її аналоги); над локальними копіями змінних після завершення всіх ітерацій циклу виконується заданий оператор; оператор це: +, \*, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску;
* schedule (type [, chunk]) - опція задає, яким чином ітерації циклу розподіляються між потоками;
* collapse (n) - опція вказує, що n послідовних вкладених циклів асоціюються з даною директивою; для циклів утворюється загальний простір ітерацій, яке ділиться між потоками; якщо опція collapse не задана, то директива відноситься тільки до одного безпосередньо наступному за нею циклу;
* ordered - опція, що говорить про те, що в циклі можуть зустрічатися директиви ordered; в цьому випадку визначається блок всередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть в послідовному циклі;
* nowait - наприкінці паралельного циклу відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається тільки тоді, коли всі вони досягнуть даної точки; якщо в подібній затримці немає необхідності, опція nowait дозволяє потокам, що вже дійшли до кінця циклу, продовжити виконання без синхронізації з іншими.

На вигляд паралельних циклів накладаються досить жорсткі обмеження. Зокрема, передбачається, що коректна програма не повинна залежати від того, який саме потік яку ітерацію паралельного циклу виконає. Не можна використовувати побічний вихід з паралельного циклу. Розмір блоку ітерацій, зазначений в опції schedule, не повинен змінюватися в рамках циклу.

Ці вимоги введені для того, щоб OpenMP міг при вході в цикл точно визначити число ітерацій. Якщо директива паралельного виконання стоїть перед гніздом циклів, що завершуються одним оператором, то директива діє тільки на самий зовнішній цикл. Ітеративна змінна розподіленого циклу за змістом повинна бути локальною, тому в разі, якщо вона специфікована загальною, то вона неявно робиться локальною при вході в цикл. Після завершення циклу значення ітеративної змінної циклу не визначено, якщо не вказано в опції lastprivate.

Наступний приклад демонструє використання директиви for. У послідовній області ініціалізуються три вихідних масивів A, B, C. В паралельній області дані масиви оголошені загальними. Допоміжні змінні i і n оголошені локальними. Кожний потік присвоїть змінній n свій порядковий номер. Далі за допомогою директиви for визначається цикл, ітерації якого будуть розподілені між існуючими потоками. На кожній i-ій ітерації даний цикл складе i-ті елементи масивів A і B і результат запише в i-ий елемент масиву C. Також на кожній ітерації буде надрукований номер потоку, що виконав дану ітерацію.

Приклад 1.3.1

#include <stdio.h>   
#include <omp.h>    
int main(int argc, char \*argv[])   
{   
    int A[10], B[10], C[10], i, n;   
// заповнимо початковий масив   
    for (i = 0; i < 10; i++)   
    {   
        A[i] = i;   
        B[i] = 2 \* i;   
        C[i] = 0;   
    }   
    #pragma omp parallel shared(A, B, C) private(i, n)   
    {   
// отримаємо номер потоку   
        n = omp\_get\_thread\_num();   
        #pragma omp for   
        for (i = 0; i < 10; i++)   
        {   
            C[i] = A[i] + B[i];   
            printf("Потік \%d додав елементи з номером %d\n", n, i);   
        }   
    }   
}

Якщо в паралельній області який-небудь ділянку коду повинен бути виконаний лише один раз, то його потрібно виділити директивою single.

#pragma omp single [опція[, опція]...]

--тіло

Можливі опції:

* private (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* firstprivate (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* copyprivate (список) - після виконання нитки, що містить конструкцію одно-, нові значення змінних списку будуть доступні всім однойменною приватним змінним (private та firstprivate), описаним на початку паралельної області і використовуваним усіма її потоками; опція не може використовуватися спільно з опцією nowait; змінні списку не повинні бути перераховані в опціях private і firstprivate даної директиви single;
* nowait - після виконання виділеної ділянки відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається тільки тоді, коли всі вони досягнуть даної точки; якщо в подібній затримці немає необхідності, опція nowait дозволяє потокам, що вже дійшли до кінця ділянки, продовжити виконання без синхронізації з іншими.

Доволі часто зустрічаються блоки коду, доступ до яких бажано надавати тільки одному потоку, - наприклад, блоки коду, що відповідають за запис даних у файл. У багатьох таких ситуаціях не має значення, який потік виконає код, важливо лише, щоб цей потік був єдиним. Для цього в OpenMP служить директива single.

Обмеження для директиви single наступні:

* Один потік буде виконувати фрагмент, а всі інші потоки будуть очікувати завершення його роботи, якщо тільки не вказана опція nowait.
* Опція copyprivate не повинна використовуватися разом з опцією nowait;
* Тільки одна опція nowait може бути використана в директиві single;
* Виняток викликаний в області single має бути оброблений в рамках однієї області single, одним і тим же ж потоком.

Директива sections використовується для задання кінцевого (неітеративного) паралелізму. Директива sections містить набір структурованих блоків, які розподіляються по потокам в групі. Кожен структурований блок виповнюється один раз, одним з потоків в групі.

#pragma omp sections [опція[, опція] ...]   
{   
#pragma omp section

--тіло   
#pragma omp

--тіло

...   
}

При виконанні цього коду OpenMP спочатку створює групу потоків, а потім розподіляє між ними обробку ітерацій циклу, після виконання якого потоки починають паралельну обробку розділів коду, що залишилися. Якщо кількість розділів програмного коду буде більше числа потоків, обробка декількох розділів буде відкладена до тих пір, поки не з'являться вільні потоки. На відміну від планування циклів, розподіл навантаження між потоками при обробці паралельних розділів коду здійснюється і контролюється OpenMP. Програмісту залишається тільки вибрати, які змінні будуть загальними, а які - індивідуальними, і передбачити вираження зменшення аналогічно сегменту з організацією циклів.

Можливі опції:

* private (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* firstprivate (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* lastprivate (список) - змінним, перерахованим у списку, присвоюється результат, отриманий в останній секції;
* reduction (оператор: список) - задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно типу оператора (для адитивних операцій - 0 або його аналоги, для мультиплікативних операцій - 1 або її аналоги); над локальними копіями змінних після завершення всіх секцій виконується заданий оператор; оператор це: +, \*, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску;
* nowait - наприкінці блоку секцій відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається тільки тоді, коли всі вони досягнуть даної точки; якщо в подібній затримці немає необхідності, опція nowait дозволяє потокам, що вже дійшли до кінця своїх секцій, продовжити виконання без синхронізації з іншими.

Директива section задає ділянку коду, усередині секції sections, для виконання одним потоком.

#pragma omp section

Перед першою ділянкою коду в блоці sections директива section не обов'язкова. Які саме потоки будуть задіяні для виконання якої секції не визначається. Якщо кількість потоків більше кількості секцій, то частина потоків, для виконання секцій, не буде задіяна. Якщо кількість потоків менше кількості секцій, то деяким (або усім) ниткам дістанеться більше однієї секції.

Директива task застосовується для виділення окремої незалежної завдання.

#pragma omp task опція[[[,] опція] ...]

--тіло

Поточний потік виділяє як завдання асоційований з директивою блок операторів. Завдання може виконуватися негайно після створення або бути відкладено на невизначений час і виконуватися по частинах. Розмір таких частин, а також порядок виконання частин різних відкладених завдань визначається реалізацією.

Можливі опції:

* if (умова) - породження нового завдання тільки при виконанні деякої умови; якщо умова не виконується, то завдання буде виконано поточним потоком і негайно;
* untied - опція означає, що у разі відкладання завдання може бути продовжена будь-яким потоком з числа виконуючих дану паралельну область; якщо дана опція не вказана, то завдання може бути продовжено тільки потоком-майстром;
* default (shared | none) - всім змінним в задачі, яким явно не призначений клас, буде призначений клас shared; none означає, що всім змінним в задачі клас повинен бути призначений явно;
* private (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* firstprivate (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* shared (список) - задає список змінних, загальних для всіх потоків.

Наступний приклад показує як пройти деревоподібну структуру використовуючи директиву task. Відзначимо, що функція траверс (traverse) повинна бути викликана з паралельної області для різних зазначених завдань, які будуть виконуватися паралельно. Також відзначимо, що завдання виконуються не в зазначеному порядку, оскільки тут не використовуються директиви синхронізації. Таким чином, припущення, що обхід буде зроблений в тому ж порядку, що і в послідовному коді, є невірним.

Приклад 1.3.2

struct node   
{   
    struct node \*left;   
    struct node \*right;   
};    
extern void process(struct node \*);   
void traverse( struct node \*p )   
{   
    if (p->left)   
        #pragma omp task // p є firstprivate-змінною за замовченням   
            traverse(p->left);   
    if (p->right)   
        #pragma omp task // p є firstprivate-змінною за замовченням  
            traverse(p->right);   
    process(p);   
}

Для гарантованого завершення в точці виклику всіх запущених завдань використовується директива taskwait.

#pragma omp taskwait

Потік, що виконав дану директиву, призупиняється до тих пір, поки не будуть завершені всі раніше запущені даним потоком незалежні завдання.

Директива taskyield вказує, що поточна задача може бути припинена на користь виконання інших завдань.

#pragma omp taskyield

* 1. **Синхронізація процесів в бібліотеці OpenMP**

У попередніх розділах при згадуванні директив роботи з циклами проблема синхронізації вже зачіпалася. По-перше, було зазначено, що ця процедура є дуже трудомісткою і порівнянна з трудомісткістю ініціалізації паралельних потоків. Тому бажано користуватися синхронізацією якомога рідше.

По-друге, було відзначено, що неявно (за замовчуванням) синхронізація паралельних процесів забезпечується при виконанні циклів в паралельному режимі. Була згадана директива nowait для усунення неявній синхронізації при завершенні циклів. Проте користуватися цією директивою слід вельми і вельми акуратно, попередньо проаналізувавши порядок роботи програми і переконавшись, що скасування синхронізації не приведе до псування даних і непередбачуваних результатів.

Механізм роботи синхронізації можна описати таким чином. При ініціалізації набору паралельних процесів в програмі встановлюється контрольна точка (аналогічна контрольній точці у відладчику), в якій програма очікує завершення всіх породжених паралельних процесів. Відзначимо, що поки всі паралельні процеси свою роботу не завершили, програма не може продовжити роботу за точкою синхронізації. А оскільки всі сучасні високопродуктивні процесори є процесорами конвеєрного типу, стає зрозумілою і висока трудомісткість процедури синхронізації. Справді, поки не завершені всі паралельні процеси, програма не може розпочати підготовку завантаження конвеєрів процесорів. Ось це-то і веде до великих втрат при синхронізації процесів, аналогічних втрат при роботі умовних операторів у звичайній послідовній програмі.

Директиви master виділяють ділянку коду, яку буде виконано тільки потоком-майстром. Решта потоків просто пропускають дану ділянку і продовжують роботу з оператора, розташованого слідом за ним. Неявну синхронізацію дана директива не передбачає.

#pragma omp master

За допомогою директив critical оформляється критична секція програми. Критична секція забороняє одночасне виконання структурованого блоку більш ніж одним потоком.

#pragma omp critical ім’я[()]

--тіло

В один момент часу в критичній секції може перебувати не більше одного потоку. Якщо критична секція вже виконується будь-яким потоком, то всі інші потоки, які виконали директиву для секції з даним ім'ям, будуть заблоковані, поки потік не закінчить виконання даної критичної секції. Як тільки потік що працював вийде з критичної секції, один із заблокованих на вході потоків увійде в неї. Якщо на вході в критичну секцію стояло кілька потоків, то випадковим чином вибирається один з них, а решта заблокованих потоків продовжують очікування.

Всі неіменовані критичні секції умовно асоціюються з одним і тим же ім'ям. Всі критичні секції, що мають одне і теж ім'я, розглядаються єдиної секцією, навіть якщо знаходяться в різних паралельних областях. Побічні входи і виходи з критичної секції заборонені.

Наступний приклад ілюструє застосування директиви critical. Змінна n оголошена поза паралельної області, тому за замовчуванням є спільною. Критична секція дозволяє розмежувати доступ до змінної n. Кожна нитка по черзі присвоїть n свій номер і потім надрукує отримане значення.

Приклад 1.4.1

#include <stdio.h>   
#include <omp.h>   
int main(int argc, char \*argv[])   
{   
    int n;   
    #pragma omp parallel   
    {   
        #pragma omp critical   
        {   
            n = omp\_get\_thread\_num();   
            printf("Потік %d\n", n);   
        }   
    }   
}

Якби в прикладі не була вказана директива critical, результат виконання програми був би непередбачуваний. З директивою critical порядок виведення результатів може бути довільним, але це завжди буде набір одних і тих же чисел від 0 до OMP\_NUM\_THREADS-1. Звичайно, подібного ж результату можна було б добитися іншими способами, наприклад, оголосивши змінну n локальною, тоді кожний потік працював би зі своєю копією цієї змінної. Однак у виконанні цих фрагментів суттєва різниця.

Якщо є критична секція, то в кожен момент часу фрагмент буде оброблятися лише якимось одним потоком. Решта потоків, навіть якщо вони вже підійшли до даної точки програми і готові до роботи, чекатимуть своєї черги. Якщо критичної секції немає, то всі потоки можуть одночасно виконати дану ділянку коду. З одного боку, критичні секції надають зручний механізм для роботи із загальними змінними. Але з іншого боку, користуватися ним потрібно обачно, оскільки критичні секції додають послідовні ділянки коду в паралельну програму, що може знизити її ефективність.

Найпоширеніший спосіб синхронізації в OpenMP - бар'єр. Він оформляється за допомогою директиви barrier. Директива barrier дає всім потокам вказівку очікувати один одного перед тим, як вони продовжать виконання за бар'єром.

#pragma omp barrier

Потоки, які виконують поточну паралельну область, дійшовши до цієї директиви, зупиняються і чекають, поки всі потоки не дійдуть до цієї точки програми, після чого розблокуються і продовжують працювати далі. Крім того, для розблокування необхідно, щоб всі потоки, які синхронізуються завершили всі породжені ними завдання (task).

Частим випадком використання критичних секцій на практиці є оновлення загальних змінних. Наприклад, якщо змінна sum є загальною і оператор виду sum = sum + expr знаходиться в паралельній області програми, то при одночасному виконанні даного оператора декількома потоками можна отримати некоректний результат. Щоб уникнути такої ситуації можна скористатися механізмом критичних секцій або спеціально передбаченою для таких випадків директивою atomic.

#pragma omp atomic capture

--тіло

Дана директива відноситься до оператора присвоювання, що йде безпосередньо за нею (на використовувані в якому конструкції накладаються досить зрозумілі обмеження), гарантуючи коректну роботу із загальною змінної, що стоїть в його лівій частині. На час виконання оператора блокується доступ до даної змінної, всім запущеним в даний момент потокам, крім потоку, що виконує операцію. Атомарної є тільки робота зі змінною в лівій частині оператора присвоєння, при цьому обчислення в правій частині не зобов'язані бути атомарними.

Наступний приклад ілюструє застосування директиви atomic. У даному прикладі проводиться підрахунок загальної кількості породжених потоків. Для цього кожний потік збільшує на одиницю значення змінної count. Для того, щоб запобігти одночасній зміні декількома потоками значення змінної, що стоїть в лівій частині оператора присвоювання, використовується директива atomic.

Приклад 1.4.2

#include <stdio.h>   
#include <omp.h>   
int main(int argc, char \*argv[])   
{   
    int count = 0;   
    #pragma omp parallel   
    {   
        #pragma omp atomic   
        count++;   
    }   
    printf("Кількість потоків: %d\n", count);   
}

Директиви ordered визначають блок всередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть в послідовному циклі.

#pragma omp ordered

--тіло

Блок операторів відноситься до самого внутрішнього з осяжниг циклів, а в паралельному циклі повинна бути задана опція ordered. Потік, що виконує перший ітерацію циклу, виконує операції даного блоку. Потік, що виконує будь-яку наступну ітерацію, повинен спочатку дочекатися виконання всіх операцій блоку усіма потоками, які виконують попередні ітерації. Може використовуватися, наприклад, для впорядкування висновку від паралельних потоків.

* 1. **Висновки до розділу 1**

1. Виконано аналіз принципів роботи паралельних програм та бібліотека OpenMP. Досліджено, що паралелізм є доволі важливим в умовах вирішення, великих за обсягом, задач. Проте доволі часто використання паралелізму не є виправданим. Оскільки, при виділенні паралельних ділянок програми і розробці паралельних процесів необхідно, щоб трудомісткість паралельних процесів була не менше 2000 операцій поділу. В іншому випадку паралельний варіант програми буде програвати у швидкодії послідовної програмі. Ресурсо-затратність не завжди позитивно впливає на роботу програми, яка використовує паралелізм.
2. Виявлено, що проблема синхронізації паралельних потоків важлива не тільки для паралельного програмування з використанням OpenMP, а й для всього паралельного програмування в цілому. Проблема полягає в тому, що структурний паралельний блок за визначенням має одну точку виходу, за якою зазвичай знаходиться послідовний структурний блок. Обчислення в послідовному блоці, як правило, можуть бути продовжені, якщо завершені всі процеси в паралельному структурному блоці і їх результати коректно передані в послідовний блок. Саме для забезпечення такої коректної передачі даних і необхідна процедура синхронізації паралельних потоків.

**РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС ОП**

У цьому розділі буде розглянуто та проаналізовано поставлену математичну задачу, розроблено алгоритм рішення та схему взаємодії потоків для системі з спільною пам’яттю, а на основі вищезазначених досліджень буде створена паралельна програма.

ЗАВДАННЯ???

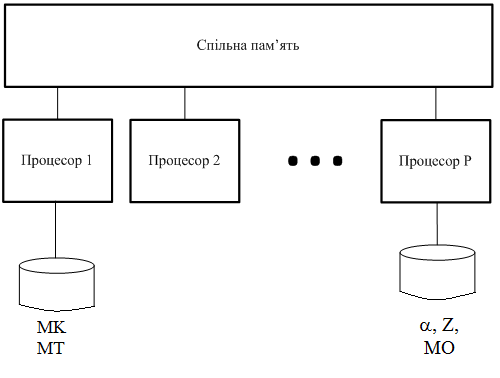


Рис. 2.1 – Структура ПКС з СП

2.**1 Розробка паралельного математичного алгоритму**

Математична задача, для якої необхідно створити паралельну програму:

MA=max(Z)(MO\*MK) + \*MT.

Перш за все розроблюється математичний алгоритм, аналізується наявність спільних ресурсів.

1. ei=max(ZH), i=(1,p);

e=max(e,ei);

Спільний ресурс: e;

1. MAH=e\*(MO\*MKH)+аlpha\*MTH;

Спільний ресурс: аlpha, e, MO;

**2.3 Розробка алгоритмів процесів**

Наступним етапом є створення алгоритму для кожного паралельного процесу із зазначенням точок синхронізації та критичних ділянок.

Задача Т1:

1. Введення MK, MT.
2. Сигнал всім Т про завершення вводу. S2-p,1
3. Очікування введення даних в ТP. Wp,1
4. Розрахунок.

* e1=max(ZH);
* e=max(e,e1); КД

1. Сигнал всім Т про завершення обчислень в Т1. S2-p,2
2. Очікування завершення розрахунку всіх T. W2-p,2
3. Копії: КД

* e;
* alpha;
* MO;

1. Розрахунок.

MAH=e\*(MO\*MKH)+аlpha\*MTH;

1. Сигнал про завершення розрахунку для ТP. Sp,3

Задача Тi(1<i<p):

1. Очікування завершення вводу T1 і TP. Wp1,1
2. Розрахунок.

* e1=max(ZH);
* e=max(e,e1); КД

1. Сигнал про завершення розрахунку в Тi. S
2. Очікування завершення розрахунку всіх Т. S
3. Копії:

* e;
* alpha;
* MO;

1. Розрахунок.

MAH=e\*(MO\*MKH)+аlpha\*MTH;

1. Сигнал про завершення розрахунку для ТP. S

Задача ТP:

1. Введення Z, аlpha, MO.
2. Сигнал всім Т про завершення вводу. S1-p,1
3. Очікування завершення вводу в Т1. W1,1
4. Розрахунок.

* eр=max(ZH);
* e=max(e,ep); КД

1. Сигнал про завершення розрахунку в ТP. S1-p-1,2
2. Очікування завершення розрахунку всіх Т. W1-p-1,2
3. Копії: КД

* e;
* alpha;
* MO;

1. Розрахунок.

MAH=e\*(MO\*MKH)+аlpha\*MTH;

1. Очікування завершення розрахунку всіх Т. W1-p-1,3
2. Виведення MA.

**2.4 Розробка схеми взаємодії процесів**

На підставі даного алгоритму будується схема взаємодії процесів. Як засіб синхронізації обраний механізм монітора, зображеного на рисунку 2.1, в програмі створений клас з даною назвою, який реалізує виконання задач взаємного виключення і синхронізації, представлених вище в алгоритмі.

ДЕ ОПИС ВЗАЄМОДІЇ????

РОЗРОБКА ПРГ1? ДЕ?

e

MO

WriteData

CopyMO

CopyAlpha

WriteE

Alpha

CopyE

WaitIn

WaitE

WaitCalc

SignalIn

SignalCalc

SignalE

Tp

T1..Tp

T1..Tp

T1..Tp

T1..Tp

T1..Tp

T1..Tp

Tp

T1,Tp

T1..Tp

T1..Tp

F1

F2

F3

Рис. 2.2 – Структурна схема взаємодії процесів для ПРГ1.

* 1. **Тестування ПРГ1**

Для дослідження необхідно визначити час виконання розробленої програми в реальній ПКС. Тому при тестуванні послідовно використовуються 1, 2, 4, 6 процесорів, для яких визначаються час виконання програми зі спільною пам’яттю . При цьому встановлюється декілька значень розмірності векторів (матриць) N = 1000, 2000, 3000. В таблиці 2.1 відображаються значення часу для різних N та Р.

Таблиця 2.1 Час виконання програми в ПКС з СП (в мс)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N//P | Т1 | Т2 | Т4 | Т6 |
| 900 | 28797 | 15491 | 8283 | 6214 |
| 1800 | 382763 | 202642 | 103574 | 78124 |
| 2400 | 1234432 | 761296 | 380569 | 292537 |

Базуючись на отриманих результатах, вираховуємо значення коефіцієнтів прискорення . У таблиці 2.2 приведено результати цих розрахунків.

Таблиця 2.2. Значення Кп для ПКС з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N//P | Кількість процесорів (P) | | | |
| 1 | 2 | 4 | 6 |
| 900 | 1 | 1,8589 | 3,4766 | 4,6342 |
| 1800 | 1 | 1,8888 | 3,6955 | 4,8994 |
| 2400 | 1 | 1,6214 | 3,2436 | 4,2197 |

На основі отриманих коефіцієнтів прискорення будується графік змінення КП в залежності від Р та N. Графік наведено на рис. 2.2. На осі абсцис розташована розмірність векторів та матриць, ординатами зазначено коефіцієнт прискорення. Для позначення результатів різних багатопроцесорних систем використовуються різні типи ліній та маркеров.

Таблиця 2.3 Значення Ке для програми ПКС з СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N//P | Кількість процесорів (P) | | | |
| 1 | 2 | 4 | 6 |
| 900 | 1 | 92,94 | 86,91 | 77,23 |
| 1800 | 1 | 94,44 | 92,38 | 81,65 |
| 2400 | 1 | 81,07 | 81,09 | 70,32 |

На основі отриманих коефіцієнтів прискорення будується графік змінення КП в залежності від Р та N. Графік наведено на рис. 2.2. На осі абсцис розташована розмірність векторів та матриць, ординатами зазначено коефіцієнт прискорення. Для позначення результатів різних багатопроцесорних систем використовуються різні типи ліній та маркерів.

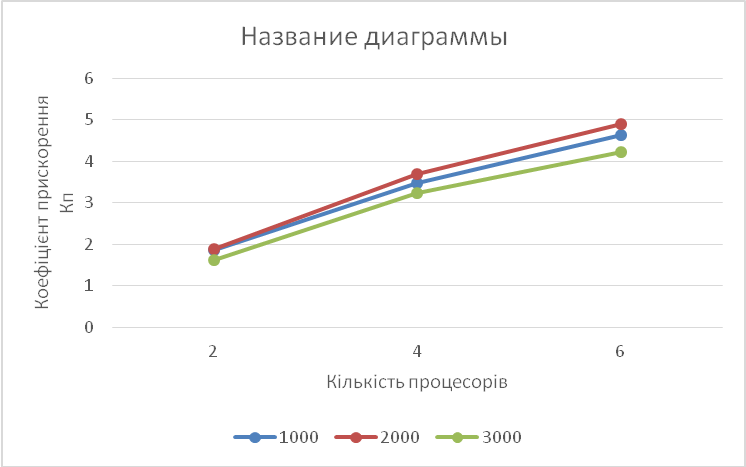


Рисунок. 2.2 - Графік залежності коефіцієнта прискорення для ПКС з СП.

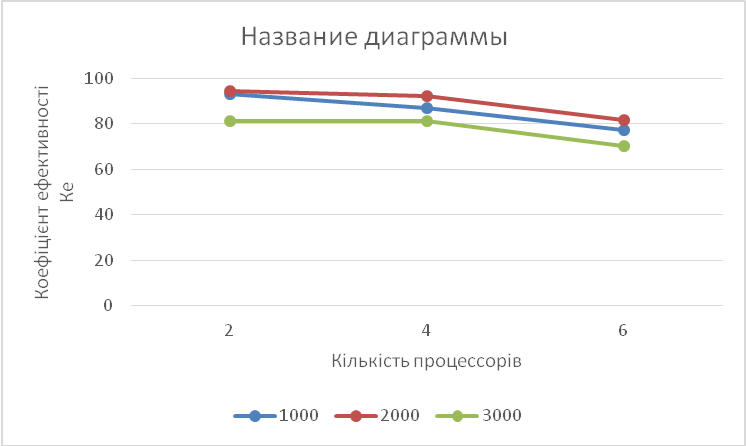


Рисунок 2.3 - Графік залежності коефіцієнта прискорення від розмірності матриць і векторів для ПКС з СП.

**Висновки до розділу 2**

**?**

**РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП**

У цьому розділі буде розглянуто та проаналізовано поставлену математичну задачу, розроблено алгоритм рішення та схему взаємодії задач, а на основі вищезазначених досліджень буде створена паралельна програма для системи з локальною пам’яттю.

…

ЗАВДАННЯ ?

P/3

MA, MO,

Z, a

MK, MT

Рисунок 3.1 – Структура ПКС з ЛП

**3.1 Розробка паралельного математичного алгоритму**

Математична задача, для якої необхідно створити паралельну програму:

MA=max(Z)(MO\*MK) + \*MT.

Перш за все розроблюється математичний алгоритм, аналізується наявність спільних ресурсів.

1. ei=max(ZH), i=(1,p);

e=max(e,ei);

Спільний ресурс: e;

1. MAH=e\*(MOH\*MK)+аlpha\*MTH;

Спільний ресурс: аlpha, e, MO;

**3.2 Розробка алгоритмів процесів**

Наступним етапом є створення алгоритму для кожного паралельного процесу із зазначенням точок синхронізації та критичних ділянок.

**Задача Т1:**

1. Введення *MК, MТ.*
2. Передати MK 4H, MT4H в Т2.
3. Прийняти MO, Z4H, a від T2.
4. Передати a, Z2H, MK2H, MT2H, MO в Т1.1.
5. Обчислення1



1. Прийняти e1.1 від Т1.1.
2. Обчислення2



1. Передати e1 в Т2.
2. Прийняти e1 від Т2.
3. Передати e1 в Т1. 1.
4. Обчислення3



1. Прийняти МА2H від Т1.1.
2. Передати МА4H в ТР.

**Задача Тi:**

1. Прийняти a, Z4H, MK4H, MT4H, MO від Ti.
2. Передати a, Z2H, MK2H, MT2H, MO в Тi.1.
3. Обчислення1



1. Прийняти ei.1 від Тi.1.
2. Обчислення2



1. Передати ei в Тi+1.
2. Прийняти ei від Тi+1.
3. Передати ei в Тi. 1.
4. Обчислення3



1. Прийняти МА2H від Тi.1.
2. Передати МА4H в ТР.

**Задача Тi.1:**

1. Прийняти a, Z2H, MK2H, MT2H, MO від Ti.
2. Передати a, ZH, MKH, MTH, MO в Тi.2.
3. Обчислення1



1. Прийняти ei.1 від Тi.1.
2. Обчислення2



1. Передати ei в Тi.
2. Прийняти ei від Тi.
3. Прийняти ei від Тi.2.
4. Обчислення3



1. Передати МА2H в Тi.

**Задача Тi.2:**

1. Прийняти a, ZH, MKH, MTH, MO від Ti.
2. Обчислення1



1. Обчислення2



1. Передати ei в Тi.1.
2. Прийняти ei від Тi.1.
3. Обчислення3



1. Передати МАH в Тi.1.

**Задача ТP:**

1. Введення *MO, Z, a..*
2. Передати Z 4H, MO, a в Тp-1.
3. Прийняти MK4H, MT4H від Tp-1.
4. Передати a, Z2H, MK2H, MT2H, MO в Тp.1.
5. Обчислення1



1. Прийняти ep.1 від Тp.1.
2. Обчислення2



1. Передати ep в Тp-1.
2. Прийняти ep від Тp-1.
3. Передати ep в Тp. 1.
4. Обчислення3



1. Прийняти МА2H від Тp.1.
2. Очікування всіх МА.
3. Вивід МА.

Для реалізації поставленого завдання використовується механізм Ada рандеву. Основна ідея механізму рандеву: у специфікації задачі публікуються різні входи (entry) у задачу, в яких вона готова чекати звернення до неї від інших задач. Далі, в тілі задачі вказуються інструкції прийняття звернень до відповідних входів, зазначених у специфікації цієї задачі. Механізм рандеву є одночасно як механізмом синхронізації, так і механізмом міжзадачного обміну даними.

**3.3 Розробка схеми взаємодії процесів**

Базуючись на алгоритмі задач та концепції рандеву, створюється схема взаємодії задач, зображена на рис.3.1.

**ОПИСАТЬ ВЗАИМОДЕЙСТВИЕ!!!!!**

MO

MAH

MK, MT

GetMO

Result

T(1)

Data

Result

sTask2

Data

Result

sTask3

**.**

**.**

**.**

MT

MA

GetMK

Result

T(P)

MT2h, MO2h, MK,a

Data

Result

sTask2

MOh, MK, a, MTh

MAh

Data

Result

sTask3

MA2h

e,MA2h

e,MAh

MA, MO, a, Z

GetMC

MC

Z, MT2h, MO2h, MK,a

MOh, MK, Z, a, MTh

GetZ

GetMT

MK

Geta

Z

Z

a

Рис. 3.1. Схема взаємодії задач

Лістинг програми розміщується в Додатку Б «Лістинг програми ПКС ЛП».

**3.4 Розробка програми**

**………………………………………………..**

**3.5 Тестування ПРГ2**

Для дослідження необхідно визначити час виконання розробленої програми в реальній ПКС. Тому при тестуванні послідовно використовуються 1, 2, 4, 6 процесорів, для яких визначаються час виконання Програми 1. При цьому встановлюється декілька значень розмірності векторів (матриць) N = 900, 1800, 2400. В таблиці 3.1 відображаються значення часу для різних N та Р.

Таблиця 3.1 - Час виконання програми в ПКС з ЛП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N//P | Т1 | Т2 | Т4 | Т6 |
| 900 |  |  |  |  |
| 1800 |  |  |  |  |
| 2400 |  |  |  |  |

Базуючись на отриманих результатах, вираховуємо значення коефіцієнтів прискорення . У таблиці 3.2 приведено результати цих розрахунків.

Таблиця 3.2. Значення Кп для ПКС з ЛП

|  |  |  |  |
| --- | --- | --- | --- |
| N//P | Т2 | Т4 | Т6 |
| 900 |  |  |  |
| 1800 |  |  |  |
| 2400 |  |  |  |

На основі отриманих коефіцієнтів прискорення будується графік змінення КП в залежності від Р та N. Графік наведено на рис. 3.2. На осі абсцис розташована розмірність векторів та матриць, ординатами зазначено коефіцієнт прискорення. Для позначення результатів різних багатопроцесорних систем використовуються різні типи ліній та маркерів.

Розрахуємо коефіцієнти ефективності та заповнимо їми таблицю 3.3. Для розрахунку використовуються формули  .

Таблиця 3.3 - Значення Ке для програми ПКС з ЛП

|  |  |  |  |
| --- | --- | --- | --- |
| N//P | Т2 | Т4 | Т6 |
| 900 |  |  |  |
| 1800 |  |  |  |
| 2400 |  |  |  |

На основі отриманих коефіцієнтів ефективності будується графік змінення Ке в залежності від Р та N. Графік наведено на рис. 3.3. На осі абсцис розташована розмірність векторів та матриць, ординатами зазначено коефіцієнт ефективності. Для позначення результатів різних багатопроцесорних систем використовуються різні типи ліній та маркерів.

**3.6 Висновки до розділу 3.**

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1.Історія створення OpenMP [Електронний ресурс] .— Режим доступу:

2.Жуков І.А., Корочкін О.В. Паралельні та розподілені обчислення: Навч.посібник. – К.: Корнійчук, 2014. – 50-53 с.

3.Розробка та використання бібліотеки OpenMP [Електронний ресурс] .— Режим доступу:

4.Використання бібліотеки OpenMP [Електронний ресурс] .— Режим доступу: https://software.intel.com/ru-ru/articles/more-work-sharing-with-openmp/

5. Засоби роботи з процесами в бібліотеці OpenMP [Електронний ресурс]. — Режим доступу: http://edu.chpc.ru/parallel/mainse5.html

6. Синхронізація процесів [Електронний ресурс]. — Режим доступу: http://www.intuit.ru/studies/courses/1112/232/lecture/6025

**МАЛО!!!!!! 10-15**