НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Кафедра обчислювальної техніки

**КУРСОВА РОБОТА**

з дисципліни «Паралельні та розподілені обчислення»

на тему: «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

Студента 3 курсу

групи ІО-01

напряму підготовки 6.050102

«Комп’ютерна інженерія»

Чухно Я. В.

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 2013 рік

Національний технічний університет України

“Київський політехнічний інститут”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки 6.050102 «Комп’ютерна інженерія»

***З А В Д А Н Н Я***

НА КУРСОВУ РОБОТУ СТУДЕНТУ

Чухно Ярославу Вікторовичу

1. Тема роботи «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

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

1. Строк подання студентом роботи 11 травня 2013 р.
2. Вихідні дані до роботи

- процеси в сучасних бібліотеках паралельного програмування

- математична задача МА = МВ\*МС+α\*MO\*(MK\*MU)

- структури ПКС ОП та ПКС ЛП

- мови і бібліотеки програмування: Java, Ада

- засоби організації взаємодії процесів: захищений модуль в Java, механізм

рандеву мови Ада

1. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно

розробити)

- огляд процесів в сучасних бібліотеках паралельного програмування

- розробка і тестування програми ПРГ1 для ПКС ОП

- розробка і тестування програми ПРГ2 для ПКС ЛП

1. 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.

1. Дата видачі завдання \_\_\_\_\_\_\_\_\_\_\_\_\_

*КАЛЕНДАРНИЙ ПЛАН*

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строки виконання етапів КР |
| **1** | Виконяння огляду для розділу 1 | 20.03.2013 |
| **2** | Розробка паралельного алгоритму рішення задачі | 1.04.2013 |
| **3** | Розробка алгоритмів процесів | 6.04.2014 |
| **4** | Розробка схем взаємодії процесів | 13.04.2013 |
| **5** | Розробка програм | 20.04.2013 |
| **6** | Тестування програм | 30.04.2013 |
| **7** | Оформлення КР | 10.05.2013 |
| **8** | Захист КР | 18.05.2013 |

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

( підпис ) (прізвище та ініціали)

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

( підпис ) (прізвище та ініціали)

Зміст

[ВСТУП 5](#_Toc356303581)

[РОЗДІЛ 1. Процеси в сучасних бібліотеках паралельного програмування 6](#_Toc356303582)

[1.1 Поняття процесу 6](#_Toc356303583)

[1.2 Процеси в бібліотеці Win32 8](#_Toc356303584)

[1.3 Процеси в бібліотеці OpenMP 11](#_Toc356303585)

[1.4 Процеси в бібліотеці MPI 17](#_Toc356303586)

[1.5 Процеси в бібліотеці PVM 20](#_Toc356303587)

[1.6 Висновки до розділу 1 24](#_Toc356303588)

[2.1 Розробка паралельного математичного алгоритму. 26](#_Toc356303589)

[2.2 Розробка алгоритмів процесів 27](#_Toc356303590)

[2.3 Розробка схеми взаємодії процесів 28](#_Toc356303591)

[2.4 Розробка ї програми ПРГ1 з використанням заданих засобів синхронізації, налагодження програми та її виконання 28](#_Toc356303592)

[2.5 Проведення досліджень ефективності розробленої програми в реальній 6-ти ядерній системі 29](#_Toc356303593)

[2.6 Висновки до розділу 2: 31](#_Toc356303594)

[РОЗДІЛ 3 Розробка програми ПРГ2 для ПКС із ЛП 32](#_Toc356303595)

[3.1 Побудова паралельного алгоритму. 32](#_Toc356303596)

[3.2 Розробка алгоритмів процесів. 33](#_Toc356303597)

[3.3 Розробка схеми взаємодії процесів 34](#_Toc356303598)

[3.4 Розробка програми ПРГ2 з використанням заданих засобів передачі даних, налагодження програми та її виконання 34](#_Toc356303599)

[3.5 Проведення досліджень ефективності розробленої програми в реальній 6-ти ядерній системі 35](#_Toc356303600)

[3.6 Висновки до розділу 3: 37](#_Toc356303601)

[ОСНОВНІ РЕЗУЛЬТАТИ ТА ВИСНОВКИ 38](#_Toc356303602)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 39](#_Toc356303603)

[ДОДАТКИ 40](#_Toc356303604)

[Додаток А. Блок-схема алгоритму процесу класу ThreadN (ПРГ1) 40](#_Toc356303605)

[Додаток Б. Блок-схема алгоритму процесу типу MainTask (ПРГ2) 41](#_Toc356303606)

[Додаток В. Блок-схема алгоритму процесів типу LeafTask (ПРГ2) 42](#_Toc356303607)

[Додаток Г. Лістинг програми ПРГ1 43](#_Toc356303608)

[Додаток Д. Лістинг програми ПРГ2 49](#_Toc356303609)

ВСТУП

В даній роботі зроблено огляд та порівняння засобів паралельного програмування, для вирішення задач взаємного виключення та синхронізації. Також показано як для паралельних комп’ютерних систем з спільною і локальною пам’яттю та тестувати їх.

В розділі 1 порівнюються засоби для роботи з процесами в таких бібліотеках паралельного програмування, як WIN32, OpenMP, MPI, та PVM. Наведена інформація про всі засоби і методи для роботи з потоками. Розглянуто механізми створення та управління потоками.

В розділі 2 розроблено паралельну програму для обчислення математичної функції в паралельній комп’ютерній системі з загальною пам’яттю. Для програми розроблено алгоритми поведінки потоків, структурні схеми та математичний алгоритм. Проведено тестування програмного продукту та зроблені висновки щодо нього.

В розділі 3 розроблено паралельну програму для обчислення математичної задачі в паралельній комп’ютерній системі з локальною пам’яттю. Показана робота з локальною пам’яттю. Також проведено тестування програмного продукту та зроблені висновки щодо нього.

В додатках присутній серцевий код розроблених алгоритмів та програм. Також наведено список літератури, за допомогою якого була написана ця курсова робота.

# РОЗДІЛ 1. Процеси в сучасних бібліотеках паралельного програмування

* 1. **Поняття процесу**

*Процес* - абстрактне поняття, що включає опис певних дій, пов’язаних з виконанням програми в комп’ютерній системі. Стандарт ISO 9000:2000 Definition визначає процес як сукупність взаємопов’язаних і взаємодіючих дій, які перетворюють вхідні дані у вихідні. Комп’ютерна програма сама по собі є лише пасивною сукупністю інструкцій, в той час як процес – це безпосереднє виконання цих інструкцій.[9] При цьому процес оформлюється так, щоб система керу­вання процесами в ОС могла ефективно перерозподіляти ресурси системи (процесори, пам’ять, прилади введення-виведення, файли та .інше ). Процес характеризується власним набором ресурсів і виділе­ною для нього ділянкою оперативної пам’яті.[1]

Поняття процесу вперше з’явилось в багатозадачних операцій­них системах і сьогодні є фундаментальним для кожної сучасної ОС. В ОС процес був пов’язаний з кожною множиною прикладних або системних програм, які виконуються в комп’ютерній системі. Це дозволяє системі керування процесами ОС, яка розміщується в ядрі ОС, ефективно маніпулювати процесами за допомогою спеціа­льного блоку керування процесом (БКП або РСВ - Process Control Block). Блок керування процесом — динамічна структура даних, яка містить основну інформацію про процес:

* ім’я процесу;
* поточний стан процесу;
* пріоритет процесу;
* місце розміщення процесу в пам’яті;
* ресурси, що пов’язані з процесом та ін.

Стосовно паралельного програмування процеси - це частини однієї програми користувача, які виконуються одночасно. Такі процеси отримали назву *легкі процеси (lightweight processes*). Для позначення легких процесів використовують також терміни *потік* або *задача*. Тому традиційні процеси іноді називають *важкими процесами* (*heavyweight processes).* Реалізація потоків виконання і процесів у різних операційних системах відрізняється одна від однї, але в більшості випадків потік виконання знаходиться всередині процесу. Потік (*thread)* - деяка сутність всередині процесу, яка отримує процесорний час для виконання. У кожному процесі є мінімум один потік. Цей первинний потік створюється системою автоматично при створенні процесу. Далі цей потік може породити інші потоки, а ті в свою чергу нові і т.д. Таким чином, один процес може володіти декількома потоками, і тоді вони одночасно виконують код в адресному просторі одного процесу.[1]

Під час виконання кількох програм у комп’ютерній системі або звичайному комп’ютері відбувається постійне перемикання з одно­го процесу на інший. Те ж саме відбувається і для одиночної пара­лельної програми, якщо вона включає кілька легких процесів (по­токів). Але час, який витрачається на перемикання для легких про­цесів, менший, ніж для важких. Ще одна відмінність важких та легких процесів полягає в тому, що легкі процеси постійно взаємо­діють, оскільки є частинами однієї програми (*тісно зв’язані* проце­си), в той час, як важкі процеси взаємодіють рідко (*слабко зв’язані* процеси).

Засоби роботи з процесами розподіляються на бібліотечні та мовні. Прикладами бібліотек для роботи з процесами є Win32, PVM, МРІ, OpenMP, POSIX, Pthread, які реалізовані у вигляді набору функцій. Вони дозволяють роботу з процесами в будь-яких мовах програмування, зокрема і в тих, які не мають вбудованих засобів роботи з процесами.

Під час свого життєвого циклу процес може проходити визначенні стани. Види станів процесу визначаються кон­кретною ОС і системою керування процесами, але містить здебільшого наступні стани:

* Породження – створення процесу і підготовка до першого виконання в системі;
* Готовності – процес готовий до виконання і чекає звільнення процесора;
* Виконання – процес виконується на процесорі;
* Блокування – процес призупинений через очікування визначеної події;
* Завершення – нормальне або аварійне завершення виконання процесу

Причинами переходів є операції, що виконуються ОС над про­цесом: породження і завершення процесу, блокування і розблокування, завершення кванта часу роботи процесора, операції введення-виведення та ін.

Процес також може мати особливий стан, який отримав назву *тупик* *(deadlock).* Процес у тупиковому стані блокований і очікує на подію, яка ніколи не відбудеться, що зумовлює неможливість його продовження і, як наслідок, - зависання програми в цілому. [1]

Рис 1.1 Діаграма станів процесу

* 1. **Процеси в бібліотеці Win32**

Бібліотека *Win32 (Windows АРІ)* входить до складу ОС Windo­ws і містить набір функцій, які призначені для роботи з процесами.

Для створення потоку в Win32 використовують функції *CreateThread ( )* і *CreateRemoteThread ( ).* Дані функції повертають ідентифікатор процесу, який є унікальним і ідентифікує його в системі. Під час створення потоку визначається початковий адрес коду, з якого має виконуватись потік. Зазвичай, це назва фу­нкції, яка буде виконуватися як процес.

Функція:

*HANDLE їм'я\_Потоку = CreateThread(*

*LPSECURITY\_ATTRIBUTES atr // атрибут безпеки*

*SIZE\_T рс, // розмір стека*

*LPTHREAD\_STERT\_ROUTUNE func , // функція потоку*

*LPVOID args, // аргумент функції потоку*

*DWORD flag, // прапорець*

*LPDWORD id); // ідентифікатор потоку*

створює потік, для якого фактичні параметри визначають ім’я функції та її параметр, атрибути безпеки, початковий розмір стека потоку, прапорець створення.[1]

Приклад створення двох потоків, які будуть виконувати функцію Func :

*// функція, яка виконується в потоці*

*void Func (void)*

*{*

*printf("Потік стартував ");*

*printf("Потік завершився ");*

*}*

*int main(void)*

*{*

*DWORD Tid1, Tid2;*

*HANDLE Thread1, Thread2;*

*// Створення потоків*

*Thread1 = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE) Func,*

*NULL,0, &Tid1);*

*Thread2 = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE) Func,*

*NULL,0, &Tid2);*

*// Закриття потоків*

*CloseHandle(Thread1);*

*CloseHandle(Thread2);*

*}*

Потік виконується доти, доки не відбудеться одна з таких подій:

* функція повертає значення потоку;
* потік викликає функцію *ExitThread ()* ;
* інший потік викликає функцію *ExitProcess ()* ;
* інший потік викликає функцію *TerminâteThread ()* з дескриптором потоку;
* інший потік викликає функцію *TerminateProcess()* з дескриптором процесу.

За допомогою функції *GetExitCodeThread ( )* можна отримати значення стану завершення потоку. Під час виконання потік має стан *STILL\_ACTIVE*.

Пріоритет потоку встановлюється за допомогою функції *SetThreadPriority().* Отримати поточне значення пріоритету процесу можна за допомогою функції *GetThreadPriority ().* Створюючись, потік отримує пріоритет, що дорівнює значенню *TRHREAD\_PRIORITY\_NORMAL.* В таблиці 1.1 описуються рівні пріоритетів потоків.[10]

Таблиця 1.1 Рівні пріоритетів потоків у бібліотеці Win32

|  |  |
| --- | --- |
| Ідентифікатор рівня | Опис |
| THREAD\_PRIORITY\_LOWEST | Пріоритет потоку повинен бути на 2 одиниці менше класу пріоритету процесу |
| THREAD\_PRIORITY\_BELOW\_NORMAL | Пріоритет потоку повинен бути на 1 одиницю менше класу пріоритету процесу |
| THREAD\_PRIORITY\_NORMAL | Пріоритет потоку повинен відповідати класу пріоритету процесу |
| THREAD\_PRIORITY\_ABOVE\_NORMAL | Пріоритет потоку повинен бути на 1 одиницю більше класу пріоритету процесу |
| THREAD\_PRIORITY\_HIGHEST | Пріоритет потоку повинен бути на 2 одиниці більше класу пріоритету процесу |

Потік залишається в системі, поки він не закінчить роботу і всі його дескриптори не будуть закритими за допомогою функції *CloseHandle()*.

Призупинення процесу виконується функціями *Suspend Thread()* і *ResumeThread( ).* Функції *Sleep ()* і *SleepEx()* блокують виконання процесу на заданий інтервал часу. Функція *SwitchToThread ( )* блокує процес і передає керування іншому потоку[1].

* 1. **Процеси в бібліотеці OpenMP**

*OpenMP (Open specifications for Multi-Processing)* – відкритий стандарт для написання паралельних програм для багатопроцесорних обчислювальних систем із загальною оперативною пам'яттю.[1]

Стандарт OpenMP був розроблений в 1997 році, як API, орієнтований на написання переносних багатопотокових програм. Спочатку OpenMP призначався для мови Fortran, але в 1998 році включив у себе і C / C + +.

Розробкою стандарту займається організація OpenMP ARB (ARchitecture Board), до якої увійшли представники найбільших компаній - розробників SMP-архітектур і програмного забезпечення. OpenMP задуманий як стандарт для програмування на масштабованих SMP-системах (SSMP, ccNUMA, etc.) в моделі загальної пам'яті (shared memory model)[6].

OpenMP - це набір спеціальних директив компілятору, бібліотечних функцій і змінних середовища. Компілятор, який підтримує OpenMP, перетворює вихідний код і вставляє відповідні виклики функцій для паралельного виконання цих областей коду.

OpenMP простий у використанні і включає лише два базові типи конструкцій: директиви pragma і функції біліотеки 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.[4]

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

OpenMP використовується модель паралельного виконання "розгалуження-злиття" (fork-join). Програма починається виконанням одного потоку, який називається початковим (initial) потоком. Початковий потік виконується послідовно. Коли потік досягає директиви *parallel* він створює команду потоків, яка складається з початкового потоку і нуля або більше додаткових потоків, і стає госпдарем (master) створеної команди. Всі члени команди виконують код структурної області, пов'язаної з директивою *parallel* (паралельної області). В кінці паралельної області розміщується неявний бар'єр. Тільки потік-господар продовжує виконання після завершення паралельної області.

Число потоків в команді, що виконуються паралельно, можна контролювати декількома способами. Один з них - використання змінної середовища *OMP\_NUM\_THREADS*. Інший спосіб - виклик процедури *omp\_set\_num\_threads ()*. Ще один спосіб - використання виразу *num\_threads* в поєднанні з директивою *parallel*.

У програмі може знаходиться будь-яку кількість директив *parallel*. Паралельні області можуть бути вкладені одна в одну. Якщо вкладений паралелізм заборонений або не підтримується реалізацією, нова команда буде складатися тільки з потоку, який зустрів вкладену директиву *parallel*.

Команда потоків, яка зустріла конструкцію розподілу роботи, розділяє роботу задану цією конструкцією між потоками команди, і ця робота виконується потоками спільно, замість того щоб виконуватися повністю кожної ниткою. Після конструкції розподілу роботи всі нитки продовжують виконання коду паралельної області.[]

Директива *parallel* створює паралельну область для структурованого блоку, який розміщується після цієї директиви. Паралельна область задається за допомогою запису[7]:

*# pragma omp parallel опція [[[,] опція] ...]*

*Структурований блок*

Можливі опції директиви *parallel*:

*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 продовжується обробка програми впопередньому режимі;

*num\_threads* (цілочисельне вираз) - явне задання кількості потоків, які будуть виконуватися в паралельній області; за замовчанням 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 визначено;

*firstprivate* (список) - задає список змінних, для яких породжується локальна копія в кожній нитки; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-господареві;

*shared* (список) - задає список змінних, спільних для всіх потоків;

*copyin* (список) - задає список змінних, оголошених як *threadprivate*, які при вході в паралельну область ініціалізуються значеннями відповідних змінних в потоці-господареві;

*reduction* (оператор: список) - задає оператор і список спільних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно до типу оператора (для адативних 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[7].

Ця директива повідомляє компілятору, що структурований блок коду 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* потоки, кожен потік отримує свій унікальний номер, причому породжувальний потік отримує номер 0 і стає основним поток групи ("майстром"). Решта потоків отримують в якості номера цілі числа від 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 проводиться 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[4].

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.

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

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

Дуже часто паралельна область не містить нічого, крім конструкції розподілу праці між потоками

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. Вираз в опції *num\_threads* повинен бути цілочисловим;
* Викидання виключення виконаний усередині паралельної області повинен викликати обробку виключення в рамках однієї паралельної області, і того ж потоку, який викинув виключення.

Директива *master* виділяє ділянку коду, яка буде виконана тільки потоком-майстром. Решта потоків просто пропускають дану ділянку і продовжують роботу з оператора, розташованого слідом за нею. Неявній синхронізації дана директива не припускає.[]

*# pragma omp master*

*Ділянка коду*

Директива *sections* використовується для завдання кінцевого паралелізму. Директива *sections* містить набір структурованих блоків, які розподіляються по потокам в групі. Кожен структурований блок виконується один раз одним з потоків в групі.[]

*# pragma omp sections опція [[[,] опція] ...]*

*{*

*# pragma omp section*

*структурованний блок*

*# pragma omp section*

*структурованний  блок*

...

}

Директива *section* задає ділянку коду усередині секції sections для виконання однієї ниткою.

*# pragma omp section*

* 1. **Процеси в бібліотеці MPI**

Бібліотека (інтерфейс) *MPI (Message Passing Interface)* містить набір функцій, що дозволять організувати роботу з процесами в мовах, які не мають вбудованих засобів програмування процесів. У разі застосування бібліотеки MPI використовується мова послідовного програмування типу С або Фортран, процеси в якій програмуються за допомогою засобів інтерфейсу MPI.[1]

Формально MPI-підхід заснований на включенні в програмні модулі викликів функцій спеціальної бібліотеки і завантажувача паралельно виконуваного коду в обчислювальні вузли (ОВ).

MPI не забезпечує механізмів задавання початкового розміщення процесів по обчислювальних вузлах (процесорам); це повинен явно задати програміст або скористатися якимсь стороннім механізмом[4].

Розробники MPI піддаються жорсткій критиці за зайву громіздкість і складність для реального (прикладного) програміста. Інтерфейс виявився складним і для реалізації; в підсумку в даний час практично не існує реалізацій MPI, які в повній мірі забезпечують поєднання обмінів з обчисленнями.

MPI не є революційно новим способом програмування для паралельних комп'ютерів. Швидше, це спроба зібрати кращі властивості багатьох існуючих систем передачі повідомлень, поліпшити їх, якщо необхідно, і стандартизувати.

Під паралельної програмою в рамках MPI розуміється множина процесів, які одночасно виконуються. Всі процеси породжуються один раз, утворюючи паралельну частину програми. Кожен процес працює у своєму адресному просторі, ніяких загальних змінних або даних у MPI немає. Процеси можуть виконуватися на різних процесорах, але на одному процесорі можуть розташовуватися і кілька процесів (у цьому випадку їх виконання здійснюється у режимі поділу часу). У кінцевому випадку для виконання паралельної програми може використовуватися один процесор - як правило, такий спосіб застосовується для початкової перевірки правильності паралельної програми.

Кожен процес паралельної програми породжується на основі копії одного і того ж програмного коду (модель SPMP - Single Programm Multiple Processes). Даний програмний код, представлений у вигляді виконуваної програми, повинен бути доступний в момент запуску паралельної програми на всіх використовуваних процесорах. Вихідний програмний код для виконуваної програми розробляється на алгоритмічних мовах C або Fortran з використанням тієї чи іншої реалізації бібліотеки MPI.

Таблиця 1.2 Основні функції MPI

|  |  |
| --- | --- |
| **Функції** | **Дії** |
| **MPI\_Init()** | Підключити до MPI (параметри args і argv визначають аргументи програ­ми). Завжди викликається першою |
| **MPI\_Comm\_size ( )** | Визначити кількість процесів, які потрі­бно запустити |
| **MPI\_Coim\_rank ( )** | Отримати ранг (номер) процесу в групі (в комунікаторі) |
| **MPI\_Finalize()** | Завершити виконання програми |

Кількість процесів і число використовуваних процесорів визначається в момент запуску паралельної програми засобами середовища виконання MPI-програм і в ході обчислень мінятися не може (в стандарті MPI-2 передбачається можливість динамічної зміни кількості процесів). Всі процеси програми послідовно перенумеровані від 0 до p-1, де p є загальна кількість процесів. Номер процесу також називають рангом процесу.[4]

Усі ресурси MPI бібліотеки (функції, типи, константи) розпо­чинаються з префікса *МРІ\_*.[1]

Загальна структура MPI програми має вигляд:

*#include "mpi.h"*

*int main(int args, char\* argv [])*

*{*

*/\* Локальні описання \*/*

*MPI\_lnit(&args, &argv);*

*/\* Тіло програми \*/*

*MPI\_Finalize(); return 0;*

*}*

Основні функції MPI бібліотеки, потрібні для створення про­цесів, наведено в таблиці 1.2.

Процес у MPI має назву *задача* Задачі в MPI можуть бути об’єднані в іменовану групу. Об’єкт - група дозволяє звертатися до групи як єдиного цілого (наприклад, відправляти повідомлення всім задачам групи), а також визначати дії, які виконуються тільки членами групи[1].

Функція *MPI\_Group(MPI\_Comm\_World, &size)* створює групу.

У рамках групи всі задачі мають унікальні ідентифікатори *(ра­нги) -* впорядковані числа, які розпочинаються з нуля. Спочатку всі задачі належать до однієї базової групи, з якої потім формують­ся нові ГРУПИ- MPI надає набір функцій для роботи з групами.

Створюючи в додатку задачі, їх прив’язують до спільної діля­нки зв’язку. Для цього використовують поняття *комунікатор -* опис ділянки зв’язку процесів, які об’єднані в групу, а також у різні групи. Програма може вміщувати кілька ділянок зв’язку. Нуме­рація процесів всередині ділянки зв’язку незалежна. Комунікатор може бути використаний як параметр MPI-функції і для обмеження сфери її дії тільки заданою ділянкою зв’язку. Крім того, комуніка­тор забезпечує вимоги безпеки. MPI автоматично створює комуні­катор *MPI\_Comm\_World*, який є базовим для кожного додатка і створюється автоматично під час виклику функції MPI\_Init ( ) .

Функції:

*int MPI\_Comm\_size (MPI\_Comm\_World, &size) ;*

*int MPI\_Coiran\_rank(MPI\_Comm\_World, &rank) ;*

для комунікатора MPI\_Comm\_World повертають значення: size розміру групи (кількість задач, що приєднані до ділянки зв’язку) і rank - порядковий номер задачі, яка викликає цю функцію.[1]

* 1. **Процеси в бібліотеці PVM**

Бібліотека *PVM (Parallel Virtual Machine)* була розроблена університетом штату Теннессі, Oak Ridge National Laboratory і університетом Еморі. Перша версія була написана в ORNL в 1989 році, і після того, як її удосконалили в університеті Теннессі, версія 2 була випущена в березні 1991 року. Версія 3 була випущена в березні 1993 року, і кращі надійнісь і портативність.[7] Дозволяє розробляти додат­ки для множини гетерогенних комп’ютерів, з’єднаних мережею, розглядаючи їх як одну велику паралельну комп’ютерну систему. Система містить великий набір засобів для керування процесами і ресурсами, пуску і завершення задач, засоби синхронізації задач, конфігурування PVM. Підтримується мовами Фортран, С, C++.[1]

Підтримка різних гетерогенних комп’ютерів - основна особли­вість PVM. Програма, що написана для однієї архітектури, може виконуватися на іншій без модифікації. PVM додаток - це набір взаємодіючих задач (процесів), які забезпечують розв’язання однієї задачі користувача. Модель обчислень у PVM - набір взаємодіючих послідовних задач, кожна задача має особисту нить керування, вза­ємодія задач виконується за допомогою посилання повідомлень.[1]

До складу PVM можна включати досить різнорідні обчислювальні машини, несумісні з архітектури даних. Інакше кажучи, паралельної віртуальною машиною може стати як окремо взятий персональний комп'ютер (ПК), так і локальна мережа, включає в себе суперкомп'ютери з паралельною архітектурою, універсальні ЕОМ, графічні робочі станції і все ті ж малопотужні ПК. Важливо лише, щоб про включаються в PVM обчислювальних засобах була інформація у використовуваному програмному забезпеченні PVM. Завдяки цього програмного забезпечення користувач може вважати, що він спілкується з однією обчислювальною машиною, в якій можливо паралельне виконання безлічі задач.[4]

У PVM вузол системи з комп’ютерам має назву *хост.* PVM мі­стить дві компоненти: PVM бібліотеку і демон, який має розмі щуватися в кожному хості PVM. Для запуску PVM додатка необ­хідно запустити PVM і сконфігурувати віртуальну машину.[1]

Постулати, взяті за основу для PVM, наступні:

1. Конфігурується користувачем пул хостів: обчислювальні задачі програми виконуються із залученням набору машин, які вибираються користувачем для даної програми PVM. Як однопроцесорні машини, так і апаратне забезпечення мультипроцесорів (включаючи комп'ютери з розділеною і розподіленою пам'яттю) можуть бути складовою частиною пулу хостів. Пул хостів може змінюватися додаванням і видаленням машин в процесі роботи (важлива можливість для підтримки мінімального рівня помилок).
2. Прозорість доступу до обладнання: прикладні програми можуть «бачити» апаратне середовище як групу віртуальних обчислювальних елементів без атрибутів або експлуатувати за вибором можливості специфічних машин із пулу хостів шляхом «переміщення» певних розрахункових задач на найбільш підходящі для їх вирішення комп'ютери.
3. Обчислення, здійснені за допомогою процесів: одиницею паралелізму в PVM є задача (часто, але не завжди збігається з процесом в системі UNIX) - незалежний послідовний потік управління, який може бути або комунікаційним, або обчислювальним. PVM не містить і не нав'язує карти зв'язків процесів; характерно, що складові завдання можуть виконуватися на одному процесорі.
4. Модель явного обміну повідомленнями: групи обчислювальних задач, кожна з яких виконує частину «навантаження» програми - використовується декомпозиція за даними, функціях або гібридна, - взаємодіють, явно посилаючи повідомлення один одному і приймаючи їх. Довжина повідомлення обмежена тільки розміром доступної пам'яті.
5. Підтримка гетерогенності: система PVM підтримує гетерогенність системи машин, мереж і програм. У відношенні механізму обміну повідомленнями PVM допускає повідомлення, які містять дані більше одного типу, для обміну між машинами з різними представленням даних.
6. Підтримка мультипроцесорів: PVM використовує оригінальні можливості обміну повідомленнями для мультипроцесорів з метою отримання вигоди від використання базового обладнання. Виробники часто підтримують власні, оптимізовані для своїх систем PVM, які стають комунікаційними в їх загальній версії. [3]

PVM додаток - композиція послідовних програм, кожна з яких пов’язана з одним або кількома процесами в паралельній програмі. Ці програми компілюються індивідуально для кожного хосту PVM. Одна із задач (що ініціалізує) запускається вручну й активізує інші задачі додатка. PVM додаток може мати кілька структур: зірку, майстер-робітника та ін.[1]

Задача в PVM запускається або вручну, або це виконується з іншої задачі. Динамічне створення задачі виконується за допомо­гою PVM функції pvm spawn (). Задача, що викликає функцію pvm spawn (), є батьківською задачею, створювана задача - дочір­ньою задачею. Створюючи дочірню задачу, потрібно вказати:

* машину, на якій буде виконуватися дочірня задача;
* шлях (path) до виконавчого файлу;
* кількість копій дочірньої задачі;
* масив аргументів.

Задачі в PVM мають унікальний ідентифікатор (Tid), який отримують під час створення задачі. Ідентифікатор використову­ють для взаємодії задач.

Формат функції *pvm spawn ()* :

*num = pvm\_spawn(Child, Args, Flag, Where, HowMany, &Tids);*

Параметри:

* *Child* - ім’я exec файлу, котрий визначає виконання створюваної задачі. Має резидентно знаходитись на хості, де буде виконуватися.
* *Args* - покажчик на масив аргументів задачі. Має зна­чення null, якщо аргументів немає.
* *Flag* - застосовують для задання режимів виконання процесів, що запускаються.
* *Where* - ім’я хосту або типу архітектури, де буде виконуватися задача, що створена залежно від значення змінної *Flag*; тобто визначається місце запуску процесу.
* *HowMany* - кількість ідентичних дочірніх задач для запуску.
* *Tids* - ідентифікатор (TID) дочірньої задачі.

Функція *pvm\_spawn ()* запускає копії виконуваного файлу *Child*.

Приклади створення процесів:

*T1 = pvm\_Spawn("user/rew/work", 0, 1, "Dina", 2, &tidl);*

*T2 = pvmjSpawn("user/rew/work", 0, 1, "Zond", 4, &tid2);*

Дві створені задачі відповідають двом і чотирьом копіями програми "work" для двох хостів - *Dina* і *Zond*. Обидві машини мають містити файл *work.e*xe в директоріях "*user/rew/"*. Значення T1 і T2 - номери задач, TID задачі будуть розміщені в tidl і tid2[1].

* 1. **Висновки до розділу 1**

1. Виконаний аналіз засобів представлення процесів в бібліотеках паралельного програмування на прикладі Win32, OpenMP, MPI та PVM, який показує, що всі ці бібліотеки працюють з «легкими» процесами (потоками);
2. На основі виконаного аналізу засобів для створення процесів в сучасних бібліотеках паралельного програмування можна стверджувати , що на сьогодні всі популярні бібліотеки паралельного програмування мають потужну базу розробки паралельних програм;
3. Такі бібліотеки паралельного програмування ,як MPI і PVM, дозволяють розробляти прогамне забезпечення для розподілених комп’ютерних систем, що зробило їх популярними на сьогоднішній час.

**РОЗДІЛ 2 Розробка програми ПРГ1 для ПКС із ЗП**

## 2.1 Розробка паралельного математичного алгоритму.

ЗП

1

MA, MB

P-1

4

2

MC, α

3

MO

P

MK, MU

…….

…….

Рис. 2.1 Структура маштабованої ПКС із ЗП

Вираз для побудови паралельного алгоритму:

MA=MB\*MC + α\*MO\*(MK\*MU )

З якого побудований паралельний алгоритм:

MAH=MB\*MCH+ α\*MO\*(MK\*MUH )

Спільні ресурси: α, MB, МО, MK.

## 2.2 Розробка алгоритмів процесів

Таблиця 2.1 Алгоритм роботи процесів

|  |  |  |
| --- | --- | --- |
| **№** | **Задача Т(i)** | **КД і ТС** |
|  | Якщо i=1: введення даних MB |  |
| Якщо i=2: введення даних α, MC |
| Якщо i=3: введення даних MO |
| Якщо i=P: введення даних MK, MU |
|  | Якщо i=1: сигнал задачам Т(2)-Т(Р) про введення | S2-1, S3-1... SP-1 |
| Якщо i=2: сигнал задачам Т(1),Т(3)-Т(Р) про введення | S1-1, S3-1… SP-1 |
| Якщо i=3: сигнал задачам Т(1),Т(2),Т(4)-Т(Р) про введення | S1-1, S2-1, S4-1... SP-1 |
| Якщо i=P: сигнал задачам Т(1)- Т(Р-1) про введення | S1-1… S(P-1)-1 |
|  | Якщо i=1: чекати на сигнал про введення від задач T(2), T(3) та T(P) | W2-1 ,W3-1,WP-1 |
| Якщо i=2: чекати на сигнал про введення від задач T(1), T(3) та T(P) | W1-1 ,W3-1,WP-1 |
| Якщо i=3: чекати на сигнал про введення від задач T(1), T(2) та T(P) | W1-1 ,W2-1,WP-1 |
| Якщо i=4..P-1: чекати на сигнал про введення від задач T(1), T(2), T(3) та T(P) | W1-1 ,W2-1,W3-1,  WP-1 |
| Якщо i=P: чекати на сигнал про введення від задач T(1), T(2), T(3) | W1-1 ,W2-1,W3-1 |
|  | Копіювання αі=α, MBі =MB, МОі =MO, MKі=MK. | КД |
|  | Обчислення MAH=MB\*MCH+ α\*MO\*(MK\*MUH ) |  |
|  | Якщо i=1..Р-1: сигнал про завершення обчислень задачі T(P) | SP-1 |
| Якщо i=Р: чекати на сигнал про завершення обчислень задачами Т(j), де j=1..P-1 | W1-2 ,W2-2,W3-2,  W4-1… W(P-1)-1 |
|  | Якщо i=Р: вивести MA |  |

## 2.3 Розробка схеми взаємодії процесів

F1=4

getMB

inputEndSignal

CalcEndSignal

setMB

getMO

getAlpha

setMO

setAlpha

 WaitForCalc

 WaitForInput

F2=P-1

T(1)..T(P)

T(P)

T(2)

F1=F1+1

F2=F2+1

**Monitor**

getMK

setMK

T(1).. T(P)

T(1)

T(1).. T(P)

T(3)

T(1).. T(P)

T(P)

T(1).. T(P)

T(1),T(2),

T(3), T(P)

T(1)..T(P-1)

На рисунку 2.2 Структурна схема взаємодії процесів для ПРГ1.

## 2.4 Розробка ї програми ПРГ1 з використанням заданих засобів синхронізації, налагодження програми та її виконання

Для реалізації алгоритму обчислення матричного рівняння на паралельній комп’ютерній системі зі спільною пам’яттю , структура якої зображена на рис. 2.1, була використана мова програмування Java.

Для вирішення проблем синхронізації та доступу до спільних ресурсів була використана реалізація концепції монітора засобами мови Java. Для цього був створений клас *Monitor.java*, структура якого зображена на рис. 2.2 і в якому спільні ресурси(α, MB, МО, MK) описуються за допомогою приватних полів. Доступ до приватних полів здійснюється за допомогою синхронізованих (*synchronized*) методів, що забезпечує вирішення завдання взаємного виключення. Для синхронізації потоків були використані методи *wait()*, *notify(), notifyAll().* Вся програма включає наступні класи : *Test.java, Monitor.java, ThreadN.java.,* де *Test.java –* клас головного потоку*, Monitor.java* – реалізація монітора, а *ThreadN.java –* клас , що описує роботу потоків згідно алгоритму із табл. 2.1.

Лістинг програми ПРГ1 та блок – схеми алгоритмів процесів і програми наведені у Додатках .

## 2.5 Проведення досліджень ефективності розробленої програми в реальній 6-ти ядерній системі

Тестування проводилось на паралельній комп’ютерній системі (ПКС), характеристики якої наведені в Табл. 2.2. Для розрахунку часу виконання на не паралельній комп’ютерній системі було модифіковано програму для ПКС з спільною пам’яттю (зроблено введення/виведення в одному потоці та прибрані всі паралельні засоби для задач синхронізації та взаємного виключення). Під час тестування було заміряно час виконання програми, результати наведені в Табл. 2.3, та розраховано коефіцієнти прискорення та ефективності, Табл. 2.4 та Табл. 2.5 відповідно.

Табл. 2.2 Характеристики стенду

|  |  |
| --- | --- |
| Процесор | AMD® Phenom 2™ X6 1055T |
| Оперативна пам’ять | DDR3 1333 MHz |
| Операційна система | Microsoft Windows 7, 6.1.7601 SP1 |

Результати проведених досліджень ефективності розробленої програми:

Таблиця 2.3. Час виконання обчислень програмою:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 | P=5 | P=6 |
| 600 | 5.951c | 3.076 c | 2.168 c | 1.877 c | 1.775 c | 1.246 c |
| 1600 | 246.4 c | 127.4 c | 91.1 c | 75.2 c | 71.1 c | 50.1 c |
| 2400 | 952.2 c | 492.9 c | 344.65 c | 296.6 c | 267.5 | 176.2 |

Таблиця 2.4. Значення коефіцієнтів прискорення:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 | P=5 | P=6 |
| 600 | 1 | 1.93 | 2.75 | 3.15 | 3.35 | 4.77 |
| 1600 | 1 | 1.93 | 2.7 | 3.27 | 3.46 | 4.91 |
| 2400 | 1 | 1.93 | 2.76 | 3.2 | 3.56 | 5.11 |

Таблиця 2.5. Значення коефіцієнтів ефективності:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 | P=5 | P=6 |
| 600 | 100% | 96.5% | 91.7% | 78.75% | 67% | 79.5% |
| 1600 | 100% | 96.5% | 90% | 81.75 % | 69.2% | 81.8% |
| 2400 | 100%c | 96.5% | 92% | 80 % | 71.2% | 85.2% |

На основі проведених досліджень побудовано наступні графіки залежностей коефіцієнтів прискорення та ефективності від кількості процесорів (рис. 2.3 та рис. 2.4):

Рис. 2.3. Залежність коефіцієнтів прискорення від кількості процесорів.

Рис. 2.4. Залежність коефіцієнтів ефективності від кількості процесорів.

## 2.6 Висновки до розділу 2:

1. В результаті було розроблено паралельну програму для обчислення математичного виразу. Дивлячись на рис.2.3 та 2.4 можна зробити висновок, що паралельна програма розроблена правильно, хоча ефективність .
2. Коефіцієнт прискорення лінійно зростає зі збільшенням кількості ядер. На рис. 2.3 видно, що з кожним наступним додаванням ядер, коефіцієнт ефективності спадає менш інтенсивно. Тобто із збільшенням ядер зменшується інтенсивність спадання ефективності, що означає, що можливо досягнути деякого критичного коефіцієнта ефективності, і нижче нього спадати вже не буде. Це говорить про гарний потенціал масштабованості програмного забезпечення.
3. Коефіцієнт ефективності зменшується зі зростанням кількості ядер. Але при цьому при збільшенні кількості ядер він спадає менш інтенсивно. Це означає, що ОС віддає перевагу більш пріоритетним задачам.

# РОЗДІЛ 3 Розробка програми ПРГ2 для ПКС із ЛП

## 3.1 Побудова паралельного алгоритму.

1

P-2

2

P

P-1

**……**

Рис. 3.1 Структура маштабованої ПКС із ЛП

Вираз для побудови паралельного алгоритму:

MA=MB\*MC + α\*MO\*(MK\*MU )

З якого побудований паралельний алгоритм:

MAH=MB\*MCH+ α\*MO\*(MK\*MUH )

## 3.2 Розробка алгоритмів процесів.

Таблиця 3.1 Алгоритм роботи P процесу

|  |  |  |
| --- | --- | --- |
| **№** | **Задача типу MainTask** | **Прийом/Передача** |
|  | Введення даних α, MB, MC, MO, MK, MU |  |
|  | Передати α, MB, MO, MK, MCH, MUH задачам Т(1)..Т(Р-1)(задачі типу LeafTask) | S1-1…S(P-1)-1 |
|  | Обчислення MAH=MB\*MCH+ α\*MO\*(MK\*MUH) |  |
|  | Прийняти від задач Т(1)..Т(Р-1) (задачі типу LeafTask) результат MAH | W1-1…W(P-1)-1 |
|  | Вивести МА | S2-1 |

Таблиця 3.2 Алгоритм роботи 1..P-1 процесів

|  |  |  |
| --- | --- | --- |
| **№** | **Задачі типу LeafTask** | **Прийом/Передача** |
|  | Прийняти від задачі T(P)(задача типу МainTask) дані α, MB, MO, MK, MCH, MUH | WP-1 |
|  | Обчислення MAH=MB\*MCH+ α\*MO\*(MK\*MUP) |  |
|  | Передати задачі Т(Р) (задачі типу MainTask) результат MAH | SP-1 |

## 3.3 Розробка схеми взаємодії процесів

DataInput

MA,α,MB,MC,MO,MK,MU

Result

**…**

DataInput

T(1)

T(P-1)

T(P)

**…**

α, MB,MO,MK,MCH ,MUH

α, MB,MO,MK,MCH ,MUH

MAH

MAH

На рисунку 3.2 зображено розроблену схему взаємодії процесів.

## 3.4 Розробка програми ПРГ2 з використанням заданих засобів передачі даних, налагодження програми та її виконання

В результаті розробки алгоритмів задач(табл. 3.1 та табл. 3.2) та схеми їх взаємодії(рис. 3.2) розроблено паралельну програму для вирішення заданої математичної задачі на ПКС з ЛП (рис. 3.1), за допомогою засобів механізму рандеву в мові Ада. Для вирішення задачі масштабованості системи було прийняте рішення використовувати задачний тип в Аді, і для цього було розроблено два типа задач: MainTask для P задачі, яка містить пристрій введення-виведення та LeafTask для P-1 p Для пересилки даних між задачами використовуються захищені входи:

* Result – вхід в задачі Р для прийняття результатів MAH  від решит задач;
* DataInput – вхід в задачах 1..Р для прийняття введених даних від задачі Р;

Лістинг програми ПРГ2 та блок – схеми алгоритмів процесів і програми наведені у Додатках .

## 3.5 Проведення досліджень ефективності розробленої програми в реальній 6-ти ядерній системі

Тестування проводилось на ПКС, характеристики якої наведені в Табл. 3.3. Для розрахунку часу виконання на не паралельній комп’ютерній системі було модифіковано програму для ПКС з локальною пам’яттю (зроблено введення/виведення в одному потоці та прибрані всі паралельні засоби для задач синхронізації та взаємного виключення). Під час тестування було заміряно час виконання програми, результати наведені в Табл. 3.4, та розраховано коефіцієнти прискорення та ефективності, Табл. 3.5 та Табл. 3.6.

Табл. 3.3 Характеристики стенду

|  |  |
| --- | --- |
| Процесор | AMD® Phenom 2™ X6 1055T |
| Оперативна пам’ять | DDR3 1333 MHz |
| Операційна система | Microsoft Windows 7, 6.1.7601 SP1 |

Результати проведених досліджень ефективності розробленої програми:

Таблиця 3.4. Час виконання обчислень програмою:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 | P=5 | P=6 |
| 600 | 7.737c | 3.853 c | 2.59 c | 2.247 c | 2.122 c | 1.513 c |
| 1600 | 257.84 c | 130.88 c | 87.41 c | 76.44 c | 71.49 c | 59.73 c |
| 2400 | 874.4 c | 427.19 c | 292.45 c | 257.04 c | 230.11 с | 169.11 |

Таблиця 3.5. Значення коефіцієнтів прискорення:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 | P=5 | P=6 |
| 600 | 1 | 2 | 2.98 | 3.44 | 3.65 | 5.11 |
| 1600 | 1 | 1.97 | 2.94 | 3.37 | 3.61 | 5.18 |
| 2400 | 1 | 2.01 | 2.99 | 3.4 | 3.79 | 5.17 |

Таблиця 3.6. Значення коефіцієнтів ефективності:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 | P=5 | P=6 |
| 600 | 100% | 100% | 99.3% | 86% | 73% | 85.17% |
| 1600 | 100% | 98.5% | 98% | 84.25% | 72.2% | 59.73 c |
| 2400 | 100% | 100.5% | 99.7% | 85% | 75.8% | 169.11 |

На основі проведених досліджень побудовано наступні графіки залежностей коефіцієнтів прискорення та ефективності від кількості процесорів (рис. 3.3 та рис. 3.4):

Рис. 3.3. Залежність коефіцієнтів прискорення від кількості процесорів.

Рис. 3.4. Залежність коефіцієнтів ефективності від кількості процесорів.

## 3.6 Висновки до розділу 3:

1. В результаті було розроблено паралельну програму для обчислення математичного виразу. Дивлячись на рис.3.3 та 3.4 можна зробити висновок, що паралельна програма розроблена правильно.
2. Коефіцієнт прискорення системи з локальною пам’яттю лінійно зростає зі збільшенням кількості процесорів. Це пояснюється тим, що ядра між собою з'єднані попарно, і дані між ними передаються досить швидко.
3. Коефіцієнт ефективності одно пропорційний коефіцієнту прискорення, це видно на рис.3.4. Це пояснюється тим, що ядра неефективно використовують час виконання завдання.

# ОСНОВНІ РЕЗУЛЬТАТИ ТА ВИСНОВКИ

1. В даній роботі було проведено порівняння засобів створення процесів в бібліотеках паралельного програмування Win32, openMP, MPI та PVM. Можна зробити висновок, що зручніше використовувати бібліотеки MPI та PVM у паралельному програмуванні, так як вони мають більше різномані засоби для роботи з розподыленими ситемами. Але більш надійнішим та стабільнішим залишається механізм рандеву.
2. У другому розділі було розроблено паралельну програму яка працює зі спільною пам'яттю за допомогою бзасобів мови Java, на основі цієї програми було проведено тестування.
3. У третьому розділі було розроблено паралельну програму яка працює з локальною пам'яттю за допомогою механізму рандеву мови Ада, на основі цієї програми було проведено тестування.
4. Тестування у розділах 2 та 3 показало, що при збільшені кількості ядер процесора збільшується коефіцієнт прискорення, але спадає коефіцієнт ефективності
5. У порівнянні ефективності обох паралельних програм, можна сказати, що програма спільною пам'яттю трохи швидше обчислює математичний вираз ніж програма зі локальною пам'яттю, хоча при збільшені N перевага переходить до програми з механізмом рандеву, Це можна пояснити тим, що справа може бути не лише у часі на пересилку даних, а також мовним компілятором, крім цього Java використовує JVM , що значно зповільнює роботу Java програм .

.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Жуков І. А., Корочкін О. В. Паралельні та розподілені обчислення. Навч. Посіб. –«Корнійчук», 2005.-226с
2. МЕТОД И ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ ПРИ РЕШЕНИИ ПРИКЛАДНЫХ ЗАДАЧ[Електронне джерело] : archive.nbuv.gov.ua – Режим доступу: <http://archive.nbuv.gov.ua/portal/natural/Ipz/2010_3/Chistyakov.pdf>
3. Система PVM[Електронне джерело]: docstore.mik.ua– Режим доступу: <http://docstore.mik.ua/manuals/ru/linux_parallel/node211.html>
4. Учебное пособие Параллельное программирование на основе библиотек[Електронне джерело]: edu.chpc.ru – Режим доступу: [http://edu.chpc.ru/parallel/main.html#mainch3.html](http://edu.chpc.ru/parallel/main.html%23mainch3.html)
5. Message Passing Interface [Електронне джерело]: wikipedia.org – Режим доступу: <http://ru.wikipedia.org/wiki/Message_Passing_Interface>
6. OpenMP [Електронне джерело]: Wikipedia.org – Режим доступу: <http://ru.wikipedia.org/wiki/OpenMP>
7. OpenMP [Електронне джерело]: computing.llnl.gov – Режим доступу: <https://computing.llnl.gov/tutorials/openMP/>
8. Parallel Virtual Machine [Електронне джерело]: wikipedia.org – Режим доступу:<http://en.wikipedia.org/wiki/Parallel_Virtual_Machine>
9. Process (computing) [Електронне джерело]: wikipedia.org – Режим доступу: <http://en.wikipedia.org/wiki/Process_(computing)>
10. Process and Thread Functions[Електронне джерело]: msdn.microsoft.com– Режим доступу: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms684847(v=vs.85).aspx>
11. Windows API [Електронне джерело]: Wikipedia.org – Режим доступу: <http://ru.wikipedia.org/wiki/Windows_API>

# ДОДАТКИ

## Додаток А. Блок-схема алгоритму процесу класу ThreadN (ПРГ1)



## Додаток Б. Блок-схема алгоритму процесу типу MainTask (ПРГ2)



## Додаток В. Блок-схема алгоритму процесів типу LeafTask (ПРГ2)



## Додаток Г. Лістинг програми ПРГ1

Test.java

1. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2. \*\*\*\*\*\*\*\*\*\*\*\* Test PRG1.PRO2 \*\*\*\*\*\*\*

3. \*\*\*\*\*\*\*\*\* Yaroslav Chukhno \*\*\*\*\*\*\*\*

4. \*\*\*\*\*\*\*\*\* group IO-01 \*\*\*\*\*\*\*\*\*\*\*\*

5. \*\*\*\*\*\*\* MA=MB\*MC+a\*MO\*(MR\*MU) \*\*\*\*\*

6. \*\*\*\*\*\*\*\*\*\*\* 11.05.13 \*\*\*\*\*\*\*\*\*\*\*\*\*\*

7. \*/

8.

9. public class Test {

10.

11. static int N=10;

12. static int P=4;

13. static int H=N/P;

14.

15. static int [][] MA=new int[N][N];

16. static int [][] MC;

17. static int [][] MU;

18. static int [][] MBUF=new int[N][N];;

19. static long startTime;

20. static long finishTime;

21.

22. public static void printVector(int [] V){

23. if(V.length <= 12){

24. System.out.print("[");

25. for(int i=0; i<V.length; ++i){

26. System.out.print(V[i]);

27. if(i != V.length-1){

28. System.out.print(",");

29. }else{

30. System.out.println("]");

31. }

32. }

33. }

34. }

35.

36. public static void printMatrix(int [][] MX){

37. if(MX.length <= 12){

38. for(int i=0; i<MX.length; ++i){

39. System.out.print("[");

40. for(int j=0; j < MX.length; ++j){

41. System.out.print(MX[i][j]);

42. if(j != MX.length-1){

43. System.out.print(",");

44. }else{

45. System.out.println("]");

46. }

47. }

48. }

49. }

50. }

51.

52. public static int[] inputVector(int value){

53. int [] V=new int[N];

54. for (int i=0; i< V.length; ++i)

55. V[i]=value;

56. return V;

57. }

58.

59. public static int[][] inputMatrix(int value){

60. int [][] MX=new int[N][N];

61. for (int i=0; i< MX.length; ++i){

62. for(int j=0; j< MX.length; ++j){

63. MX[i][j] =value;

64. }

65. }

66. return MX;

67. }

68.

69.

70. public static void calculation(int part, int [][] MB, int alpha, int [][] MO,int [][] MR){

71. int last=H\*(part+1);

72. if(part==Test.P-1)

73. last=Test.N;

74.

75. for(int i= H\*part; i< last; ++i){

76. for(int j=0; j< N ;++j)

77. for (int k = 0; k < N; ++k)

78. MBUF[i][j] += MR[k][j] \* MU[i][k];

79. }

80.

81. for(int i= H\*part; i< last; ++i)

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

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

84. MA[i][j] += MO[k][j] \* MBUF[i][k];

85. }

86. MA[i][j] \*= alpha;

87. }

88.

89. for(int i= H\*part; i< last; ++i){

90. for(int j=0; j< N ;++j)

91. for (int k = 0; k < Test.N; ++k)

92. MA[i][j] += MB[k][j] \* MC[i][k];

93. }

94. }

95.

96. public static void main(String [] args) throws InterruptedException {

97. System.out.println("Program started");

98. startTime=System.currentTimeMillis();

99. Monitor monitor=new Monitor(P);

100. ThreadN [] threads=new ThreadN[P];

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

102. threads[i]=new ThreadN(i+1,monitor);

103. threads[i].start();

104. }

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

106. threads[i].join();

107. }

108.

109. finishTime=System.currentTimeMillis();

110. System.out.println("Working time is "+(finishTime-startTime)+"ms");

111. System.out.println("Program finished");

112. }

113. }

Monitor.java

1. /\*\*

2. \* @class monitor for

3. \*/

4. public class Monitor {

5. private int alpha;

6. private int [][] MB,MO,MR;

7. private int F1=0;

8. private int F2=0;

9. private int P;

10.

11. public Monitor(int P){

12. this.P=P;

13. }

14.

15. public synchronized void inputEndSignal() {

16. F1++;

17. if(F1>=4)

18. notifyAll();

19. }

20.

21. public synchronized void waitForInput(){

22. try{

23. if(F1<4)

24. wait();

25. }catch(Exception e){

26. e.printStackTrace();

27. }

28. }

29.

30. public synchronized void calcEndSignal(){

31. F2++;

32. if(F2>=P-1)

33. notifyAll();

34. }

35.

36. public synchronized void waitForCalc(){

37. try{

38. if(F2<P-1)

39. wait();

40. }catch(Exception e){

41. e.printStackTrace();

42. }

43. }

44.

45. public synchronized void setAlpha(int value){

46. alpha=value;

47. }

48.

49. public synchronized void setMB(int value){

50. MB=Test.inputMatrix(value);

51. }

52.

53. public synchronized void setMO(int value) {

54. MO=Test.inputMatrix(value);

55. }

56.

57. public synchronized void setMR(int value) {

58. MR=Test.inputMatrix(value);

59. }

60.

61. public int getAlpha(){

62. return alpha;

63. }

64.

65. public int [][] getMB(){

66. return MB.clone();

67. }

68.

69. public int [][]getMO(){

70. return MO.clone();

71. }

72.

73. public int [][]getMR(){

74. return MR.clone();

75. }

76. }

ThreadN.java

1.

2. public class ThreadN extends Thread {

3. int number;

4. int alpha;

5. int [][] MB;

6. int [][] MO;

7. int [][] MR;

8. Monitor monitor;

9. public ThreadN(int number, Monitor monitor){

10. this.number=number;

11. this.monitor=monitor;

12. }

13.

14. //operations in Thread P

15. public void run(){

16. System.out.println("Thread"+number+" started");

17.

18. //\*\*\*\*\*\*\*\*\*\*data inputting \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

19. switch(number){

20. case 1:

21. monitor.setMB(1);

22. //\*\*\*\*\*\*\*\*signal about end of inputting in Thread 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

23. monitor.inputEndSignal();

24. break;

25.

26. case 2:

27. Test.MC=Test.inputMatrix(1);

28. monitor.setAlpha(1);

29. //\*\*\*\*\*\*\*\*signal about end of inputting Thread 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

30. monitor.inputEndSignal();

31. break;

32.

33. case 3:

34. monitor.setMO(1);

35. //\*\*\*\*\*\*\*\*signal about end of inputting Thread 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

36. monitor.inputEndSignal();

37. break;

38.

39. default:

40. if(number==Test.P){

41. monitor.setMR(1);

42. Test.MU=Test.inputMatrix(1);

43. //\*\*\*\*\*\*\*\*signal about end of inputting Thread P \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

44. monitor.inputEndSignal();

45. }

46. }

47.

48.

49. //\*\*\*\*\*\*\*\*waiting for end of all inputting \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

50. monitor.waitForInput();

51.

52. //\*\*\*\*\*\*\*\*\*copying of data \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

53. alpha=monitor.getAlpha();

54. MB=monitor.getMB();

55. MO=monitor.getMO();

56. MR=monitor.getMR();

57.

58. //\*\*\*\*\*\*\*\*\*calculation\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

59. Test.calculation(number-1,MB,alpha,MO,MR);

60.

61. if(number!=Test.P){

62.

63. //\*\*\*\*\*\*\*\*signal about end of calculation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

64. monitor.calcEndSignal();

65. }else{

66.

67. //\*\*\*\*\*\*\*\*wait for end of calculation in others threads \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

68. monitor.waitForCalc();

69.

70. //\*\*\*\*\*\*\*\*outputting of results \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

71. Test.printMatrix(Test.MA);

72. }

73. System.out.println("Thread"+number+"finished");

74. }

75. }

## Додаток Д. Лістинг програми ПРГ2

Source file: ..\..\..\..\coursetest.ada Tue May 14 13:53:39 2013

1 --------TEST PRG2. Kursova PRO ----

2 ----------- Ada Randevouse ---------

3 ----------- group io-01 ---------

4 ------------ Chukhno Yaroslav -----------

5 -------------01.05.2013 -----------------

6

7 with Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Float\_Text\_IO,

8 Ada.Synchronous\_Task\_Control, Ada.Calendar;

9

10 use Ada.Text\_IO, Ada.Integer\_Text\_IO,Ada.Float\_Text\_IO,

11 Ada.Synchronous\_Task\_Control, Ada.Calendar;

12

13 Procedure Course is

14 N: integer := 10;

15 P: integer := 6;

16 h: Integer := N/p;

17 startTime: Time;

18 FinishTime: Time;

19 WorkTime: Duration;

20 ------------types------------------------------------------

21 type Vector is array(1 .. N) of integer;

22

23 type All\_Matrix is array (positive range<>) of Vector;

24 subtype Matrix is All\_Matrix (1 .. N);

25 subtype Matrix\_H is All\_Matrix (1 .. H);

26 subtype Matrix\_HN is All\_Matrix (1 ..(H+(N mod P)));

27 -----------------------------------------------------------

28

29 procedure Input\_Vector (V: out Vector;

30 numb: in Integer) is

31 begin

32 for i in 1..N loop

33 V(i):=Numb;

34 end loop;

35 end Input\_Vector;

36

37 procedure Input\_Matrix (MY: out Matrix; numb: in Integer) is

38 begin

39 for i in 1..N loop

40 for j in 1..N loop

41 MY(i)(j):=Numb;

42 end loop;

43 end loop;

44 end Input\_Matrix;

45

46 procedure Output\_Vector (V: in Vector) is

47 begin

48 if(N<=12) then

49 for i in 1..N loop

50 put(V(i));

51 New\_Line;

52 end loop;

53 end if;

54 end Output\_Vector;

55

56 procedure Output\_Matrix (MY: in Matrix) is

57 begin

58 if (N<=12) then

59 for i in 1..N loop

60 for j in 1..N loop

61 put(MY(i)(j));

62 put(" ");

63 end loop;

64 new\_Line;

65 end loop;

66 end if;

67 end Output\_Matrix;

68

69 procedure Output\_MatrixH (MY: in Matrix\_H) is

70 begin

71 if (N<=12) then

72 for i in 1..h loop

73 for j in 1..N loop

74 put(MY(i)(j));

75 put(" ");

76 end loop;

77 new\_Line;

78 end loop;

79 end if;

80 end Output\_MatrixH;

81

82 procedure Calculation(Alpha: in integer;

83 MB,MO,MR: in Matrix;

84 MCh,MUh: in matrix\_h;

85 MAh: out Matrix\_H) is

86

87 MRMUh: Matrix\_H;

88 begin

89 for i in 1..H loop

90 for j in 1..N loop

91 MRMUh(i)(j):=0;

92 for k in 1..N loop

93 MRMUh(i)(j):=MRMUh(i)(j)+mR(k)(j)\*MUh(i)(k);

94 end loop;

95 end loop;

96 end loop;

97

98 for i in 1..H loop

99 for j in 1..N loop

100 MAh(i)(j):=0;

101 for k in 1..N loop

102 MAh(i)(j):=MAh(i)(j)+mB(k)(j)\*MCh(i)(k);

103 end loop;

104 end loop;

105 end loop;

106

107 for i in 1..H loop

108 for j in 1..N loop

109 for k in 1..N loop

110 MAh(i)(j):=MAh(i)(j)+alpha\*mO(k)(j)\*MRMUh(i)(k);

111 end loop;

112 end loop;

113 end loop;

114 end Calculation;

115

116 procedure Calculation1(Alpha: in integer;

117 MB,MO,MR: in Matrix;

118 MCh,MUh: in matrix\_hN;

119 MAh: out Matrix\_HN) is

120

121 MRMUh: Matrix\_HN;

122 begin

123 for i in 1..(H+(N mod P)) loop

124 for j in 1..N loop

125 MRMUh(i)(j):=0;

126 for k in 1..N loop

127 MRMUh(i)(j):=MRMUh(i)(j)+mR(k)(j)\*MUh(i)(k);

128 end loop;

129 end loop;

130 end loop;

131

132 for i in 1..(H+(N mod P)) loop

133 for j in 1..N loop

134 MAh(i)(j):=0;

135 for k in 1..N loop

136 MAh(i)(j):=MAh(i)(j)+mB(k)(j)\*MCh(i)(k);

137 end loop;

138 end loop;

139 end loop;

140

141 for i in 1..(H+(N mod P)) loop

142 for j in 1..N loop

143 for k in 1..N loop

144 MAh(i)(j):=MAh(i)(j)+alpha\*mO(k)(j)\*MRMUh(i)(k);

145 end loop;

146 end loop;

147 end loop;

148 end Calculation1;

149

150 PROCEDURE START IS

151 ----------------Tasks specification -----------------------------------

152 ---------Task types----------------------------

153 task type LeafTask(taskNumber: Integer) is

154 entry DataInput(inAlpha: in integer;

155 inMB,inMO,inMR: in Matrix;

156 inMCh,inMUh: in matrix\_h);

157 end LeafTask;

158

159

160 task type MainTask is

161 entry Result(Part: in integer;

162 MAh: in matrix\_h);

163 end MainTask;

164

165 type LeafTask\_ptr is access LeafTask;

166 Tasks: array(1..P-1) of LeafTask\_Ptr;

167 T\_P: MainTask;

168 ------------------------BODY-------------------------------------------

169 ----------------------------Leaf TASK -------------------------------------

170

171 task body LeafTask is

172 Alpha\_X: Integer;

173 MAh,MCh,MUh: Matrix\_H;

174 MB\_X,MO\_X,MR\_X: Matrix;

175 begin

176 put\_Line("T" & Integer'Image(TaskNumber) & " is started");

177

178 accept DataInput(inAlpha: in integer;

179 inMB,inMO,inMR: in Matrix;

180 inMCh,inMUh: in matrix\_h) do

181 Alpha\_X:=inAlpha;

182 MB\_X:=inMB;

183 MO\_X:=inMO;

184 MR\_X:=InMR;

185 MCh:=inMCh;

186 MUh:=inMUh;

187 end DataInput;

188 put\_Line("Accepted1!!!");

189 Calculation(Alpha\_X,MB\_X,MO\_x,MR\_x,MCh,MUh,MAh);

190 T\_P.Result(TaskNumber-1,MAh);

191 put\_Line("T" & Integer'Image(TaskNumber) & " is finished");

192 end LeafTask;

193

194 ----------------------------Main task type-------------------------------------

195

196 task body MainTask is

197 Alpha\_P: Integer;

198 MA,MC,MU,MB\_P,MO\_P,MR\_P: Matrix;

199 begin

200 put\_Line("T" & Integer'Image(P) & " is started");

201 ----------------data inputting -----------------------------------------

202

203 Alpha\_P:=1;

204 Input\_Matrix(MB\_P,1);

205 Input\_Matrix(MC,1);

206 Input\_Matrix(MO\_P,1);

207 Input\_Matrix(MR\_P,1);

208 Input\_Matrix(MU,1);

209 put\_Line("Data inputted");

210 ----------Data sending --------------------------------------------------

211 for i in 0..P-2 loop

212 Tasks(I):= new LeafTask(I+1);

213 Tasks(i).DataInput(Alpha\_P,MB\_P,MO\_P,MR\_P,MC(i\*h+1..(I+1)\*H),MU(i\*h+1..(i+1)\*H));

214 end loop;

215 Put\_Line("Data sent");

216 ---------calculation----------------------------------------------------

217 Calculation1(Alpha\_P,MB\_P,MO\_P,MR\_P,MC((P-1)\*H+1..N),MU((P-1)\*H+1..N),MA((P-1)\*H+1..N));

218

219 for i in 1..P-1 loop

220 accept Result(Part: in integer;

221 MAh: in Matrix\_H) do

222 MA(part\*H+1..(Part+1)\*H):=MAh;

223 end Result;

224 end loop;

225

226 output\_Matrix(MA);

227 put\_Line("T" & Integer'Image(P) & " is finished");

228 FinishTime:=Clock;

229 WorkTime:=FinishTime-startTime;

230 put\_Line("Time is");

231 Put(Float(WorkTime),4,3,0);

232 end MainTask;

233 begin

234 null;

235 end start;

236

237 begin

238 StartTime:=Clock;

239 start;

240 end Course;