НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

КИЇВСЬКИЙ ПОЛІТЕХНИЧНИЙ ІНСТИТУТ

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра обчислювальної техніки

К У Р С О В А Р О Б О Т А

*з дисципліни:* «Паралельні та розподілені обчислення»

*на тему:* «Розробка програмного забезпечення для

паралельних комп’ютерних систем»

Виконав: Земцов А.А.

Група: ІО- 93

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_ Корочкін О. В.

Захищено з оцінкою\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_2012 р.

Київ - 2012 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

КИЇВСЬКИЙ ПОЛІТЕХНИЧНИЙ ІНСТИТУТ

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра обчислювальної техніки

***ЗАВДАННЯ НА КУРСОВУ РОБОТУ***

*з дисципліни:* «Паралельні та розподілені обчислення»

*на тему:* «Розробка програмного забезпечення для

паралельних комп’ютерних систем»

Дата отримання 11.04.2012р.

Термін виконання 11.05.2011р.

Керівник роботи доц.. Корочкін О.В.

ЗАВДАННЯ

1.Порівняння MPI, PVM, Механізм “Рандеву”.

2.Розробка програмного забезпечення для масштабованих паралельних комп’ютерних систем. Мови та бібліотеки паралельного програмування Ада і Open MP. Засоби організації взаємодії процесів: механізм посилки повідомлень – Рандеву на мові Ада; Критична секція, бар’єр – Open MP. Математична задача: MA=max(MZ)\*MC + α\*MO\*MX.

3. Тестування програмного забезпечення в багатоядерної комп’ютерної системі

ПЕРЕЛІК ГРАФИЧНОЇ ДОКУМЕНТАЦІЇ

1. Структурна схема ПКС ОП

2. Структурна схема ПКС ЛП

3. Схема алгоритму основної програми для ПКС ОП

4. Схеми алгоритмів процесів для ПКС с ОП

5. Схема алгоритму основної програми для ПКС ЛП

6. Схеми алгоритмів процесів для ПКС ЛП

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Жуков І., Корочікн О. Паралельні та розподілені обчислення. – Київ,

Корнійчук, 2005. – 226 с.

2. Корочкин А.В. Ада 95: Введение в программирование. - Киев; Свит, .

1998. - 260 с.

ПІДПИС КЕРІВНИКА РОБОТИ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Технічне завдання на курсову роботу**

1. Область застосування

1. Розроблюване програмне забезпечення може бути застосовано в галузі паралельних математичних обчислень для масштабованих комп'ютерних систем з загальної та локальної пам'яттю.

2. Підстави для розробки

Підставою для розробки служить ТЗ на курсову роботу.

3. Мета курсової роботи

Метою даної роботи є закріплення отриманих знань з дисципліни "Паралельні та розподілені обчислення ", а також отримання навичок і досвіду в розробці програмного забезпечення для комп'ютерних систем з різною структурною організацією.

4. Призначення роботи

Розробка програмного забезпечення (ПЗ) для ПКС з спільною та локальною пам'ятью. Проведення досліджень ефективності ПЗ.

5. Вихідні дані та виконання роботи

5.1. Завдання для розділу 1: Порівняння MPI, PVW, Механізм Рандеву.

5.2 Завдання для розділів 2 і 3: Математична задача:

MA= max(MZ)\*MC + α\*MO\*MX. матриці розмірності N.

5.3. Структура паралельної комп'ютерної системи з спільною пам'яттю (рисунок. 1).

5. 4. Структура паралельної комп'ютерної системи з локальною пам'яттю (рисунок. 2).

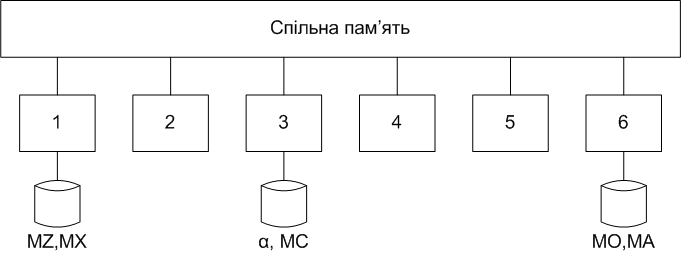


Рисунок.1. Структура ПКС СП

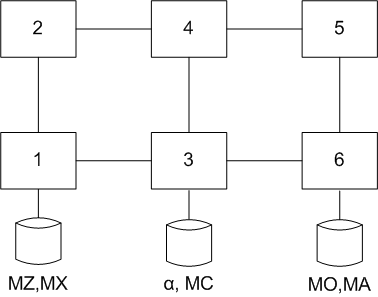


Рисунок.2. Структура ПКС ЛП

5.5. Мови та бібліотеки програмування та засоби організації взаємодії процесів: мова “C++” та бібліотека “Open MP” для розділу 2, мова “Ада” та механізм “Рандеву” - для розділу 3.

У КР необхідно:

Розробити алгоритм розв'язання заданого математичної задачі і досліджувати паралельний властивості завдання в рамках концепції необмеженого паралелізму для N і Р, де N - розмірність матриць, Р - кількість процесорів.

Виконати розробку програми вирішення заданого математичного виразу для заданих структур ПКС на заданих язиках (біблотеках) програмування. Описати структуру взаємодії завдань.

Виконати дослідження ефективності ПЗ завдання в залежності від значень Р і N. При цьому підрахувати часи виконання математичної задачі на різній кількості процессоров.Определіть значення коефіцієнтів прискорення (Кп) та ефективності (Ке).

6. Вимоги до складу і параметрів технічних засобів

Тестування ПЗ виконується з використання багатоядерної ПКС

7. Вимоги до інформаційної та програмної сумісності

Програма повинна працювати під управлінням ОС Windows.

8. Спеціальні вимоги

Тестування виконується на 4-х ядерної системі

9. Етапи і стадії розробки

- Узгодження технічного завдання

- Виконання курсової роботи.

- Захист курсової роботи.

ЗМІСТ

[РОЗДІЛ 1. ПОРІВНЯННЯ MPI, PVM, Рандеву 8](#_Toc324502572)

[1.1 Засоби MPI 8](#_Toc324502573)

[1.2 Засоби PVM 12](#_Toc324502574)

[1.3 Засоби Рандеву](#_Toc324502575) 15

[1.4 Порівняння засобів MPI, PVM, Рандеву 2](#_Toc324502576)8

Висновки до Розділу 1 [2](#_Toc324502576)8

[РОЗДІЛ 2. РОЗРОБКА ПЗ ДЛЯ ПКС З СП](#_Toc324502591) 29

[2.1 Побудова паралельного алгоритму. 29](#_Toc324502579)

[2.2 Розроблення алгоритмів роботи кожного процесу. 29](#_Toc324502580)

[2.3 Розроблення схеми взаєиоії процесів 3](#_Toc324502581)1

[2.4 Розроблення програми 3](#_Toc324502582)2

[2.5 Проведення досліджень ефективності розробленої програми. 3](#_Toc324502583)2

[Висновки до Розділу 2 3](#_Toc324502584)4

[РОЗДІЛ 3. РОЗРОБКА ПЗ ДЛЯ ПКС З ЛП](#_Toc324502591) 35

[3.1 Побудова паралельного алгоритму. 3](#_Toc324502586)5

[3.2 Розроблення алгоритмів роботи кожного процесу. 3](#_Toc324502587)5

[3.3 Розроблення схеми взаємодії процесів 3](#_Toc324502588)7

[3.4 Розроблення програми](#_Toc324502589) 38

[3.5 Проведення досліджень ефективності розробленої програми.](#_Toc324502590) 38

[Висновки до Розділу 3 4](#_Toc324502591)0

[ВИСКНОВКИ 4](#_Toc324502591)1

[СКОРОЧЕННЯ 4](#_Toc324502593)2

[ЛІТЕРАТУРА 4](#_Toc324502593)3

[ДОДАТКИ 44](#_Toc324502595)

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 1порівнюється бібліотеки програмування MPI, PVM, Рандеву. Для порівняння 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 у 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 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 Порівняння MPI, PVM, Рандеву**

**1.1 Засоби MPI**

MPI - бібліотека функцій, призначена для підтримки роботи паралельних процесів.  
    Основним способом взаємодії паралельних процесів в таких системах є передача повідомлень один одному. Це й відображено в назві даної технології - Message Passing Interface (інтерфейс передачі повідомлень). Стандарт MPI фіксує інтерфейс, який повинен дотримуватися як системою програмування на кожній обчислювальної платформі, так і користувачем при створенні своїх програм.  
     Інтерфейс MPI підтримує створення паралельних програм в стилі MIMD (Multiple Instruction Multiple Data), що передбачає об'єднання процесів з різними вихідними текстами.  
      Проте писати і налагоджувати такі програми дуже складно, тому на практиці програмісти набагато частіше використовують SPMD-модель (Single Program Multiple Data) паралельного програмування, в рамках якої для всіх паралельних процесів використовується один і той же код. В даний час все більше і більше реалізацій MPI підтримують роботу з нитками.  
     Оскільки MPI є бібліотекою, то при компіляції програми необхідно прилінкувати відповідні бібліотечні модулі. Це можна зробити в командному рядку або скористатися передбаченими в більшості систем командами або скриптами mpich, mpiCC,, і mpif77/mpif90.  
     Всі процеси породжуються один раз, утворюючи паралельну частину програми. В ході виконання MPI-програми породження додаткових процесів або знищення існуючих не  
допускається (в MPI-2.0 така можливість з'явилася). Кожен процес працює у своєму адресному просторі, ніяких загальних змінних або даних в MPI немає. Основним способом взаємодії між процесами є явна посилка повідомлень.  
 Для локалізації взаємодії паралельних процесів програми можна створювати групи процесів, надаючи їм окреме середовище для спілкування - комунікатор. Склад утворених груп довільний. Групи можуть повністю збігатися, входити одна в іншу, не перетинатися або перетинатися частково.  
     Кожен процес MPI-програми має в кожній групі, в яку він входить, унікальний атрибут номер процесу, який є цілим невід'ємним числом. За допомогою цього атрибута відбувається значна частина взаємодії процесів між собою. Зрозуміло, що в одному і тому ж комунікаторі всі процеси мають різні номери. Але оскільки процес може одночасно входити в різні комунікатори, то його номер в одному комунікаторі може відрізнятися від його номера в іншому. Звідси стають зрозумілими два основних атрибута процесу: комунікатор і номер в комунікаторі. Якщо група містить n прочісування, то номер будь-якого процесу в даній групі лежить в межах від 0 до n-1.

     Основним способом спілкування процесів між собою є явна посилка повідомлень. Повідомлення - це набір даних деякого типу. Кожне повідомлення має кілька атрибутів, зокрема, номер процесу-відправника, номер процесу-одержувача, ідентифікатор повідомлення та інші. Одним з важливих атрибутів повідомлення є його ідентифікатор або тег. За ідентифікатором процес, який приймає повідомлення, наприклад, може відрізнити два повідомлення, яке прийшло до нього від одного і того ж процесу. Сам ідентифікатор повідомлення є цілим невід'ємним числом, який знаходиться в діапазоні від 0 до MPI\_TAG\_UP, при чому гарантується, що MPI\_TAG\_UP не менше 32767. Для роботи з атрибутами повідомлень введений масив (у мові Сі - структура), елементи якого дають доступ до їх значень.  
     В останньому аргументі (в мові Сі - в повернутому значенні функції) більшість процедур MPI повертають інформацію про успішність завершення. В випадку успішного виконання повертається значення MPI\_SUCCESS, інакше - код помилки. Вид помилки, яка сталася при виконанні процедури. Можна буде визначити з її опису. Зумовлені значення, які відповідають різним помилковим ситуацій, перераховані у файлі mpif.h.  
 Атрибути повідомлення - номер процесу-відправника, номер процесу-одержувача і ідентифікатор повідомлення. Для них заведена структура MPI\_Status, що містить три поля: MPI\_Source (номер процесу відправника), MPI\_Tag (ідентифікатор повідомлення), MPI\_Error (код помилки); можуть бути і додаткові поля.  
 Ідентифікатор повідомлення (msgtag) - атрибут повідомлення, що є цілим невід'ємним числом, які лежать в діапазоні від 0 до 32767. Процеси об'єднуються в групи, можуть бути вкладені групи. Усередині групи всі процеси перенумеровані. З кожною групою асоційований свій комунікатор.  
 Тому при здійсненні пересилання необхідно вказати ідентифікатор групи, всередині якої виробляється ця пересилка. Всі процеси містяться в групі з визначеним ідентифікатором MPI\_COMM\_WORLD.  
  
       Прийом / передача повідомлень з блокуванням  
int MPI\_Send (void \* buf, int count, MPI\_Datatype datatype, int dest, int msgtag, MPI\_Comm comm)  
    • buf - адреса початку буфера посилки повідомлення  
    • count - число переданих елементів в повідомленні  
    • datatype - тип переданих елементів  
    • dest - номер процесу-одержувача  
    • msgtag - ідентифікатор повідомлення  
    • comm - ідентифікатор групи  
  
 Блокуюча посилка повідомлення з ідентифікатором msgtag, що складається з count елементів типу datatype, процесу з номером dest. Всі елементи повідомлення розташовані підряд в буфері buf. Значення count може бути нулем. Тип переданих елементів datatype повинен вказуватися за допомогою визначених констант типу. Дозволяється передавати повідомлення самому собі.  
 Блокування гарантує коректність повторного використання всіх параметрів після повернення з підпрограми. Вибір способу здійснення цієї гарантії: копіювання в проміжний буфер або безпосередня передача процесу dest, залишається за MPI. Слід спеціально зазначити, що повернення з підпрограми MPI\_Send не означає ні того, що повідомлення вже передано процесу dest, ні того, що повідомлення покинуло процесорний елемент, на якому виконується процес, який виконав MPI\_Send.  
  
int MPI\_Recv (void \* buf, int count, MPI\_Datatype datatype, int source, int msgtag, MPI\_Comm comm, MPI\_Status \* status)  
    • OUT buf - адреса початку буфера прийому повідомлення  
    • count - максимальне число елементів в прийнятому повідомленні  
    • datatype - тип елементів прийнятого повідомлення  
    • source - номер процесу-відправника  
    • msgtag - ідентифікатор прийнятого повідомлення  
    • comm - ідентифікатор групи  
    • OUT status - параметри прийнятого повідомлення  
  
 Прийом повідомлення з ідентифікатором msgtag від процесу source з блокуванням. Число елементів в прийнятому повідомленні не повинно перевищувати значення count. Якщо число прийнятих елементів менше значення count, то гарантується, що в буфері buf змінюються тільки елементи, що відповідають елементам прийнятого повідомлення. Якщо потрібно дізнатися точне число елементів в повідомленні, то можна скористатися підпрограмою MPI\_Probe.  
 Блокування гарантує, що після повернення з підпрограми всі елементи повідомлення прийняті і розташовані в буфері buf.

Як номера процесу-відправника можна вказати зумовлену константу MPI\_ANY\_SOURCE - ознака того, що підходить повідомлення від будь-якого процесу. В якості ідентифікатора прийнятого повідомлення можна вказати константу MPI\_ANY\_TAG - ознака того, що підходить повідомлення з будь-яким ідентифікатором. Якщо процес посилає два повідомлення іншому процесу і обидва ці повідомлення відповідають одному і тому ж викликом MPI\_Recv, то першим буде прийнято те повідомлення, яке було надіслано раніше.  
  
       Прийом / передача повідомлень без блокування  
int MPI\_Isend (void \* buf, int count, MPI\_Datatype datatype, int dest, int msgtag, MPI\_Comm comm, MPI\_Request \* request)  
    • buf - адреса початку буфера посилки повідомлення  
    • count - число переданих елементів в повідомленні  
    • datatype - тип переданих елементів  
    • dest - номер процесу-одержувача  
    • msgtag - ідентифікатор повідомлення  
    • comm - ідентифікатор групи  
    • OUT request - ідентифікатор асинхронної передачі  
      Передача повідомлення, аналогічна MPI\_Send, проте повернення з підпрограми відбувається відразу після ініціалізації процесу передачі без очікування обробки всього повідомлення, що знаходиться в буфері buf. Це означає, що не можна повторно використовувати даний буфер для інших цілей без отримання додаткової інформації про завершення даної посилки.  
       Закінчення процесу передачі можна визначити за допомогою параметра request і процедур MPI\_Wait і MPI\_Test.  
Повідомлення, відправлене будь-який з процедур MPI\_Send і MPI\_Isend, може бути прийняте будь-який з процедур MPI\_Recv і MPI\_Irecv.  
  
int MPI\_Irecv (void \* buf, int count, MPI\_Datatype datatype, int source, int msgtag, MPI\_Comm comm, MPI\_Request \* request)  
    • OUT buf - адреса початку буфера прийому повідомлення  
    • count - максимальне число елементів в прийнятому повідомленні  
    • datatype - тип елементів прийнятого повідомлення  
    • source - номер процесу-відправника  
    • msgtag - ідентифікатор прийнятого повідомлення  
    • comm - ідентифікатор групи  
    • OUT request - ідентифікатор асинхронного прийому повідомлення  
   Прийом повідомлення, аналогічний MPI\_Recv, проте повернення з підпрограми відбувається відразу після ініціалізації процесу прийому без очікування отримання повідомлення в буфері buf. Закінчення процесу прийому можна визначити за допомогою параметра request і процедур MPI\_Wait і MPI\_Test.

Розробники MPI створили механізм конструювання нових типів навіть більш універсальний, ніж наявний у мові програмування. Дійсно, у всіх мені відомих мовах програмування осередки всередині агрегатного типу (масиву або структури):  
- Не налазять один на одного,  
- Не розташовуються з розривами (вирівнювання полів в структурах не в рахунок).  
У MPI зняті обидві ці обмеження! Це дозволяє досить химерно "вирізати", зокрема, фрагменти матриць для передачі, і розміщувати прийняті дані між власних. У специфікації MPI наведено приклад створення користувальницького описувача типу, передача матриці з використанням якого призводить до її транспонування.  
Виграш від використання механізму конструювання типів очевидний - краще один раз викликати функцію прийому / передачі зі складним шаблоном, ніж двадцять разів - з простими.  
  
      Колективна взаємодія.  
В операціях колективної взаємодії процесів беруть участь всі процеси комунікатора. Відповідна процедура повинна бути викликана кожним процесом, може, зі своїм набором параметрів. Повернення з процедури колективної взаємодії може відбутися в той момент, коли участь процесу в даній операції вже закінчено. Як і для блокуючих процедур, повернення означає те, що дозволений вільний доступ до буфера прийому або посилки, але не означає ні того, що операція завершена іншими процесами, ні навіть того, що вона ними розпочата (якщо це можливо за змістом операції).

Оптимізовані бібліотеки для стандартних мов.

В цьому випадку оптимізація взагалі може бути прихована від проблемного програміста. Чим більший обсяг роботи всередині програми відводиться підпрограмам такій  
бібліотеці, тим більшим буде підсумковий виграш у швидкості її (програми) роботи. Власне ж програма пишеться на звичайній мові програмування без жодних згадок про MPI, і будується стандартним компілятором. Від програміста потрібно лише вказати для компонування ім'я бібліотечного файлу MPI, і запускати через MPI-завантажувач. Популярні бібліотеки обробки матриць, такі як Linpack, Lapack і ScaLapack, вже переписані під MPI.

**1.2 Засоби PVM**

Бібліотека PVM (Parallel Virtual Machine) дозволяє працювати з безліччю гетерогенних комп'ютерів, з'єднаних мережею, розглядаючи їх як велику паралельну комп'ютерну систему. Система має великий набір засобів для керівництва процесами і ресурсами, пуску і завершення задач, засобами синхронізації задач. Підтримується вона мовами Фортран, С, С++.  
   Підтримка різних гетерогенних комп'ютерів - основна особливість цієї бібліотеки.

Взаємодія процесів (задач) в PVM виконується на основі отриманих повідомлень. В рамках однієї PVM задача може послати (прийняти) повідомлення від будь-якої іншої задачі. Крім двоточкової взаємодії PVM забезпечує трансляцію даних від однієї задачі до кількох.  
      В системі PVM\_3 кожна задача (якій відповідає виконуваний файл), запущена на деякому процесорі, ідентифікується цілим числом, яке називається ідентифікатором завдання (далі використовується позначення TID) і за змістом схоже на ідентифікатор процесу в операційній системі Unix. Конкретні значення TID несуттєві, важливо лише, щоб всі задачі, запущені в PVM, мали різні TID. Відзначимо тут, що копії одного виконуваного файлу, запущені паралельно на N процесорах PVM, створюють N завдань з різними TID.  
  
      Організація посилки повідомлення пов'язана з виконанням таких дій:  
  
        1. ініціалізація буфера, що використовується для повідомлення  
  
        2. упаковка переданих даних  
  
        3. пересилання повідомлення в буфер одержувача  
  
 Для прийнятої моделі взаємодії задач в PVM вважається, що в межах однієї PVM будь-яка задача може передавати повідомлення будь-якій іншій задачі, причому, розміри і кількість таких повідомлень в принципі не обмежені. Це припущення істотно спрощує реалізацію PVM на конкретних обчислювальних комплексах, тому що при цьому контроль переповнення буферних пристроїв і масивів залишається тільки у введенні приватних операційних систем.  
       Для підвищення ефективності міжзадачного обміну інформацією передбачено використання декількох алгоритмів. Зокрема, можна використовувати алгоритм блокованої передачі, при якому функція "Надіслати повідомлення" повертає значення (тобто завершує роботу) тільки  
після того отримання позитивної чи негативної квитанції від одержувача повідомлення.  
 Такий алгоритм передачі з очікуванням повідомлення про доставку кращий у тих випадках, коли довге повідомлення передається кількома порціями, а також при обміні командами, послідовність виконання яких у часі повинна бути строго фіксованою.  
      При використанні не блокуючих алгоритмів передачі і прийому повідомлень зменшуються простої процесорів, викликані очікуванням  
реакції "співрозмовника". Особливо великий ефект це дає на приймальній стороні при невідомому часу приходу повідомлення. Можна організувати роботу приймального процесора так, щоб він в очікуванні повідомлення виконував поточну роботу, лише час від часу опитуючи приймальний буфер.  
      Суттєвим є обставина, що при передачі послідовності повідомлення від однієї задачі до іншої порядок прийому повідомлення завжди збігається з порядком їх передачі. Більш того, якщо до звернення до функції "Прийняти повідомлення" в приймальний буфер приймаючої завдання записано кілька повідомлень, то функція "Прийняти повідомлення" поверне посилання на перше прийняте повідомлення.  
      Пам'ять для буферних масивів на передавальній і приймальній стороні виділяється динамічно, отже, максимальний обсяг повідомлень обмежується лише обсягом доступної пам'яті. Якщо одне із задач, запущена в PVM, не може отримати необхідну пам'ять для спілкування з іншими задачами то вона видає користувачеві відповідне повідомлення про помилку ("cannot get memory"), але інші задачі про цю подію не сповіщаються і можуть, наприклад, продовжувати посилати їй повідомлення.  
      Крім обміну повідомленнями між двома задачами в PVM передбачені можливість широкомовної передачі повідомлень від однієї задачі декільком іншим задачам, а також можливості синхронізації дій у групі задач і спільне виконання задачами групи деяких операцій над розподіленими даними (фрагменти оброблюваних даних розподілені між задачами - членами групи).  
      Задача запускається або вручну, або це виконується з використанням іншої задачі. Динамічне створення задачі виконується за допомогою функції pvm\_spawn (). Задача, яке викликає функцію pvm\_spawn (), вважається батьківською задачею, а створювана задача - породженою. Створюючи породжену програму необхідно вказати машину, на якій буде виконуватися ця програма, шлях (path) до виконавчого файлу, кількість копій породжуваної програми, масив аргументів.  
      При посилці повідомлення від однієї задачі до іншої всередині PVM потрібно виконати як мінімум 3 дії: проініціалізувати буфер для переданого повідомлення, упакувати передані дані відповідно до форматів їх подання і переслати повідомлення в буфер одержувача.  
      Здійснення повідомлень  
      При посилці повідомлення йому присвоюється ідентифікатор, який приймає цілі невід'ємні значення і відіграє для повідомлень ту ж роль, що і величина tid для задач в PVM. У той же час, на відміну від задач різні повідомлення можуть мати однакові ідентифікатори, оскільки є гарантована можливість розрізняти їх за іншими ознаками (tid відправника або одержувача, час приходу і т.д.).  
  
      Int info = pvm\_send (int tid, int msgtag);  
  
      Тут надалі параметр info повертає значення info = 0, або код помилки info <0.  
      Прийом повідомлень  
      Якщо Ви отримали повідомлення, по-перше, потрібно визначити алгоритм прийому (блокований або неблокований), потім, прийнявши повідомлення, його потрібно розпакувати, тобто відновити передані фрагменти даних відповідно до їх типів.  
      Для функції прийому повідомлень можливі наступні значення параметрів.  
      Якщо tid = -1, то приймається будь-яке повідомлення з ідентифікатором msgtag, спрямоване до даної задачі. Якщо msgtag = -1, то приймається будь-яке повідомлення, направлене до даної задачі з ідентифікатором tid. Якщо tid = msgtag = -1, то приймається будь-яке повідомлення від будь-якої задачі, спрямоване до даної задачі.

**1.3 Засоби рандеву**

Концепція рандеву  
 Як один з механізмів забезпечення надійного між задачного обміну даними і взаємної синхронізації роботи завдань, Ада надає механізм рандеву. Основна ідея механізму рандеву досить проста. У специфікації завдання публікуються різні входи (entry) у завдання, 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.  
 Необхідно звернути увагу на несиметричність такого механізму взаємодії. Це означає, що в процесі взаємодії одна з задач розглядається як сервер, а друга - як клієнт, причому задача-сервер не може бути ініціатором початку взаємодії.  
 У найпростішому випадку, коли розглядається взаємодія тільки двох задач, задача-клієнт, що бажає звернутися до іншої задачі (задачі-серверу), ініціює звернення до входу задачі-сервера. Після цього задача-сервер відгукується на звернення задачі-клієнта, беручи звернення до цього входу. Таким чином, взаємодія двох задач здійснюється в ситуації, коли задача-клієнт звернулася до входу, а задача-сервер готова прийняти це звернення. Цей спосіб взаємодії двох задач називається рандеву.  
 Оскільки задача-клієнт і задача-сервер виконуються незалежно один від одного, то немає ніякої гарантії, що обидві задачі виявляться в точці здійснення рандеву одночасно. Тому, якщо задача-сервер опинилася в точці рандеву, але при цьому немає жодного звернення до входу (запиту на взаємодію), то вона повинна чекати появи такого звернення. Аналогічно, якщо задача-клієнт звертається до входу, а задача-сервер не готова обслужити таке звернення, то задача-клієнт повинна чекати, поки задача-сервер обслужить це звернення. У процесі очікування як задача-клієнт, так і задача-сервер не займають ресурси процесора, перебуваючи в стані, який називають призупиненим або станом блокування.  
 У випадках, коли виклики до входу задача-сервера здійснюють одразу кілька задач-клієнтів, ці виклики ставляться в чергу. Порядок обслуговування такої черги залежить від відповідності конкретній реалізації Ада-системи вимогам додатка D (Annex D) стандарту Ada95, в якому зазначаються вимоги для систем реального часу. Якщо реалізація Ада-системи не забезпечує відповідності цим вимогам, то черга обслуговується в порядку надходження викликів (FIFO - First-In-First-Out).

Селекція прийняття рандеву  
 Припустимо, що нам необхідна задача-сервер, яка буде обслуговувати безліч задач-клієнтів. Припустимо також, що задача-сервер повинна мати не один, а безліч входів для надання різних сервісів задачам-клієнтам, а один з входів буде вказувати на необхідність завершення роботи задачі-сервера, тобто задача-сервер повинна виконуватися до тих пір, поки не отримає явної команди на завершення своєї роботи.  
Приклад:

|  |
| --- |
| **task** Server\_Task **is**  **entry** Service\_1 *[ параметри для Service\_1 ]* ;  **entry** Service\_2 *[ параметри для Service\_2 ]* ;  . . .  **entry** Service\_N *[ параметри для Service\_N ]* ;  **entry** Stop;  **end task**;  **task body** Server\_Task **is**  . . .  **begin**  **loop**  **accept** Service\_1 *[ параметри для Service\_1 ]* **do**  . . .  **end** Service\_1;  **accept** Service\_2 *[ параметри для Service\_2 ]* **do**  . . .  **end** Service\_2;  . . .  **accept** Service\_N *[ параметри для Service\_N ]* **do**  . . .  **end** Service\_N;  **accept** Stop **do**  **exit** ; -- вихід з циклу та завершення задачі    **end** Stop;  **end loop**;  **end** Server\_Task; |

Однак при уважному розгляді логіки роботи даного прикладу виявляється, що задача-сервер буде блокуватися (припинятися) в кожній інструкції прийняття рандеву, очікуючи надходження виклику на відповідному вході від будь-якої задачі-клієнта. Причому, перебуваючи в стані очікування, задача-сервер ніяк не реагуватиме на наявність і надходження викликів від задач-клієнтів на інших входах. Отже, для ситуацій, які подібні до показаної в цьому прикладі, необхідна можливість селекції (або вибору) прийняття рандеву на входах задачі-сервера.  
 Для забезпечення селекції прийняття рандеву, Ада надає різні варіанти інструкції відбору з очікуванням, яка задається за допомогою зарезервованого слова select. Використання інструкції відбору в тілі задачі-сервера дозволяє:

• одночасно очікувати більше одного рандеву;

• виконувати таймаут, якщо за вказаний період часу не надійшло

жодного виклику рандеву;

• здійснювати рандеву тільки в разі наявності виклику рандеву;

• завершувати виконання задачі за відсутності задач-клієнтів, які

потенційно можуть викликати рандеву;

Приклад використання інструкції відбору в тілі задачі-сервера:

|  |
| --- |
| **task** Server\_Task **is**  **entry** Service\_1 *[ параметри для Service\_1 ]* ;  **entry** Service\_2 *[ параметри для Service\_2 ]* ;  . . .  **entry** Service\_N *[ параметри для Service\_N ]* ;  **entry** Stop;  **end task**;  **task body** Server\_Task **is**  . . .  **begin**  **loop**  **select**  **accept** Service\_1 *[ параметри для Service\_1 ]* **do**  . . .  **end** Service\_1;  . . . -- додаткова послідовність інструкцій, яка виконується  -- після прийняття рандеву на вході Service\_1  **or**  **accept** Service\_2 *[ параметри для Service\_2 ]* **do**  . . .  **end** Service\_2;  **or**  . . .  **or**  **accept** Service\_N *[ параметри для Service\_N ]* **do**  . . .  **end** Service\_N;  **or**  **accept** Stop;  **exit** ; -- вихід з циклу та завершення задачі  **end select**  **end loop**;  **end** Server\_Task; |

Як видно з початкового тексту прикладу, інструкції прийняття рандеву вказуються як альтернативи вибору інструкції відбору (це подібно інструкції case).  
 В даному випадку при виконанні інструкції відбору задача-сервер циклічно "опитує" свої входи на наявність виклику рандеву від задач-клієнтів (без блокування в стані очікування виклику рандеву на якомусь вході). Опитування триває до тих пір, поки не буде виявлений виклик рандеву на якомусь вході, який відповідає одній з перерахованих інструкцій прийняття рандеву. Після виявлення виклику виконується відповідна альтернатива.  
Слід звернути увагу, що якщо в процесі опитування, виконуваного інструкцією відбору, одночасно з'являться два і більше виклику рандеву, то інструкція відбору вибере для обробки тільки один з викликів причому правила вибору альтернативи для такого випадку не визначаються стандартом. Іншими словами, коли інструкція відбору поміщена в тіло циклу, при одночасному появі двох і більше викликів рандеву, інструкція відбору буде здійснювати обробку тільки одного виклику рандеву протягом одного "витка" циклу, а одночасно надішлені виклики рандеву будуть поставлені в чергу порядок якої не визначається стандартом.  
 У даному прикладі цікаву ситуацію представляє альтернатива прийняття рандеву на вході Stop. В цьому випадку відбувається вихід з нескінченного циклу виконання інструкції відбору, що, в свою чергу, призводить до завершення роботи задачі-сервера. Якщо в процесі завершення роботи задачі-сервера надійде виклик рандеву на якійсь із входів Service\_1 - Service\_N, то задача-клієнт, що викликає рандеву, швидше за все отримає виняток Tasking\_Error. Недолік такого підходу полягає в тому, що завершення роботи задачі-сервера вимагає явної вказівки виклику рандеву на вході Stop.  
Щоб позбутися від необхідності явної вказівки виклику рандеву на вході Stop, можна використовувати інструкцію відбору, в якій вказується альтернатива завершення завдання:

|  |
| --- |
| **task** Server\_Task **is**  **entry** Service\_1 *[ параметри для Service\_1 ]* ;  . . .  **entry** Service\_N *[ параметри для Service\_N ]* ;  **end task**;  **task body** Server\_Task **is**  . . .  **begin**  **loop**  **select**  **accept** Service\_1 *[ параметри для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **accept** Service\_N *[ параметри для Service\_N ]* **do**  . . .  **end** Service\_N;  **or**  **terminate**; -- завершення роботи задачі  **end select**  **end loop**;  **end** Server\_Task; |

В такому випадку альтернатива завершення задачі буде завершувати роботу задачі-сервера, коли відсутні виклики рандеву і відсутні задачі-клієнти, які потенційно здатні викликати рандеву з задачею-сервером. Таким чином, для завершення роботи задачі-сервера не потрібно явного виконання ніяких спеціальних дій. Слід також враховувати, що альтернатива завершення роботи задачі повинна вказуватися останньою в списку альтернатив інструкції відбору.  
 Може виникнути ситуація, коли необхідно, щоб в процесі очікування виклику рандеву від задач-клієнтів задача-сервер виконувала будь-які додаткові дії. Для цього можна використовувати інструкцію відбору, в якій замість альтернативи завершення роботи використовується розділ else. Використання такого варіанту інструкції відбору може мати вигляд

|  |
| --- |
| **loop**  **select**  **accept** Service\_1 *[ параметри для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **accept** Service\_N *[ параметри для Service\_N ]* **do**  . . .  **end** Service\_N;  **else**  . . . -- послідовність інструкцій, що виконується якщо  -- немає жодного виклику рандеву  **end select**  **end loop**; |

Подібним чином послідовність інструкцій, зазначена в розділі else, може бути використана для організації "фонової" роботи задачі-сервера.  
Ще одним різновидом інструкції відбору служить інструкція відбору, що використовує альтернативу таймауту (або затримки). Така інструкція відбору має наступний вигляд:

|  |
| --- |
| **loop**  **select**  **accept** Service\_1 *[ параметри для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **accept** Service\_N *[ параметри для Service\_N ]* **do**  . . .  **end** Service\_N;  **or**  **delay** 1.0;  . . . -- послідовність інструкцій таймаута, яка виконується  -- у випадку коли немає жодного виклику рандеву  -- последовательность инструкций таймаута,  -- протягом однієї секунди  **end select**  **end loop**; |

Підхід, демонстрований цим прикладом, зручно використовувати для організації "сторожового таймера", що сигналізує про некоректну роботу програмного забезпечення.  
 Альтернатива таймауту, зазначена в інструкцій відбору, дозволяє задачі-серверу виконувати певну послідовність дій, якщо протягом зазначеного інтервалу часу не надійшло жодного виклику рандеву. Інтервал часу вказується в альтернативах таймауту аналогічно інтервалу часу в інструкціях затримки виконання (необхідно зауважити, що незважаючи на зовнішню подібність, не слід плутати альтернативи таймауту інструкцій відбору до інструкцій затримки виконання). Таким чином, може бути вказаний відносний (як у прикладі вище) або абсолютний інтервал часу. Якщо вираз, що вказує відносний інтервал часу, має від'ємне або нульове значення або значення абсолютного інтервалу часу оцінюється як минуле, то альтернатива таймауту інструкції відбору може бути розцінена як еквівалент розділу else.  
В одній інструкції відбору, допускається наявність більше однієї альтернативи таймауту. При цьому, буде оброблятися та альтернатива таймауту, яка має найменший інтервал часу. Відзначимо також, що в межах однієї інструкції відбору не допускається спільне використання різних альтернатив таймауту для яких одночасно задані відносні і абсолютні інтервали часу.  
 При застосуванні інструкцій відбору слід враховувати, що використання альтернативи завершення роботи завдання (terminate), альтернативи таймауту (delay) і розділу else в межах однієї інструкції відбору є взаємно виключними. Крім того, будь-який варіант інструкції відбору зобов'язаний утримувати хоча б одну альтернативу вибору, в якій зазначена інструкція прийняття рандеву accept.  
 Ще однією особливістю інструкцій відбору є опціональна можливість вказівки додаткової перевірки будь-якої умови для альтернатив прийняття рандеву, завершення роботи задачі та таймауту. Для вказівки такої перевірки використовується конструкція виду: "when умова =>", - де перевіряюча умова описується за допомогою логічного вираження, результат обчислення якого повинен мати значення зумовленого логічного типу Standard.Boolean. Як правило, таку перевірку називають захисною або охоронною (guard), а її використання може мати наступний вигляд:

|  |
| --- |
| **declare**  . . .  Service\_1\_Counter : Integer;  . . .  Service\_N\_Counter : Integer;  . . .  **begin**  . . .  **loop**  . . .  **select**  **when** (Service\_1\_Counter > 0) =>  **accept** Service\_1 *[ параметри для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **when** (Service\_N\_Counter > 100) =>  **accept** Service\_N *[ параметри для Service\_N ]* **do**  . . .  **end** Service\_N;  **end select**  **end loop**;  . . .  **end**; |

Обчислення значення логічного виразу здійснюється на початку виконання інструкції відбору. Якщо результат обчислення виразу True, то відповідна альтернатива має право бути обраною інструкцією відбору, якщо результат False, альтернатива вибрана не буде, причому навіть у тому випадку, коли будь-яка задача-клієнт здійснила виклик відповідного входу і чекає обслуговування. Слід також врахувати, що у випадку, коли перевірки умов використовуються для всіх альтернатив інструкції відбору, і результати перевірок всіх умов мають значення False, це призведе до порушення виключення Program\_Error.  
Альтернативу відбору можна назвати відкритою, коли для неї не вказана додаткова перевірка умови або коли значення результату перевірки умови є True. В іншому випадку альтернативу відбору можна назвати закритою.

Селекція виклику рандеву.  
 Інструкції відбору select можуть використовуватися не тільки для селекції прийняття рандеву в задачі-сервері, а й для селекції виклику рандеву в задачі-клієнті. У подібних випадках різні форми інструкцій відбору дозволяють задачі-клієнту виконувати умовний виклик рандеву, тимчасовий виклик рандеву або асинхронну передачу управління.

Простий приклад інструкції відбору для виконання умовного виклику рандеву на вході Service\_1 задачі-сервера Server\_Task:

|  |
| --- |
| **select**  Server\_Task.Service\_1 *[ параметри для Service\_1 ]* ;  **else**  Put\_Line ("Server\_Task is busy!");  **end select**; |

В цьому випадку, якщо задача-сервер Server\_Task не готова до негайного прийому рандеву на вході Service\_1, то виклик рандеву буде скасований і виконається послідовність інструкцій в розділі else, яка виводить повідомлення "Server\_Task is busy!".  
 Інструкція відбору, призначена для умовного виклику рандеву на вході здачі-серверу, як правило, використовується для багаторазового звернення до задачі-серверу. Таким чином, загальний вид її застосування може бути наступним:

|  |
| --- |
| **loop**  **select**  Server\_Task.Service\_1 *[ параметри для Service\_1 ]* ;  . . . -- опціональна послідовність інструкцій,  -- яка виконується після рандеву  **exit**;  **else**  . . . -- послідовність інструкцій, яка  -- виконується у випадку неможливості  -- здійснення рандеву  **end select**;  **end loop**; |

Інструкція відбору для тимчасового виклику рандеву дозволяє задавати очікування прийняття рандеву на вході задачі-сервера протягом заданого інтервалу часу. Приклад такої інструкції для виконання тимчасового виклику рандеву на вході Service\_1 задачі-сервера Server\_Task може мати вигляд:

|  |
| --- |
| **select**  Server\_Task.Service\_1 *[ параметри для Service\_1 ]* ;  **or**  **delay** 5.0;  Put\_Line ("Server\_Task has been busy for 5 seconds!");  **end select**; |

В даному випадку, якщо протягом зазначеного інтервалу часу (тут заданий відносний інтервал часу тривалістю 5 секунд) задача-сервер Server\_Task не прийняла рандеву на вході Service\_1, то виклик рандеву скасовується і виконується послідовність інструкцій, задана альтернативою затримки, яка в цьому прикладі видасть повідомлення "Server\_Task has been busy for 5 seconds!".

Замість відносного інтервалу може бути зазначено абсолютне значення часу:

|  |
| --- |
| **select**  Server\_Task.Service\_1 *[ параметри для Service\_1 ]* ;  **or**  **delay until** Christmas;  Put\_Line ("Server\_Task has been busy for ages!");  **end select**; |

Слід зауважити, що у разі відносного інтервалу часу, зазначена затримка матиме нульове або негативне значення, чи заданий абсолютне значення часу розцінюється як минуле, виконання інструкції буде здійснюватися подібно виконанню інструкції відбору для умовного виклику рандеву.  
 Інструкція відбору для тимчасового виклику рандеву зручно використовувати як сторожового таймера. Таким чином, її загальний вигляд може бути наступним:

|  |
| --- |
| **loop**  **select**  Server\_Task.Service\_1 *[ параметри для Service\_1 ]* ;  . . . -- опціональна послідовність інструкцій,  -- яка виконується після рандеву  **exit**;  **or**  **delay** *інтервал\_часу*  . . . -- послідовність інструкцій, яка  -- виконується у випадку неможливості  -- здійснення рандеву  **end select**;  **end loop**; |

Зауважимо, що замість конструкції "delay інтервал\_часу", може бути використана конструкція "delay until інтервал\_часу", яка вказує абсолютне значення часу.  
 Інструкція відбору для асинхронної передачі управління (введена стандартом Ada95) дозволяє задачі-клієнту продовжити виконання будь-якого коду в той час, коли виклик рандеву очікує обслуговування.

Приклад використання такої інструкції:

|  |
| --- |
| **select**  **delay** 5.0;  Put\_Line ("Server\_Task is not serviced the Service\_1 yet");  **then abort**  Server\_Task.Service\_1 *[ параметри для Service\_1 ]* ;  **end select**; |

В даному випадку починається виконання інструкцій, розташованих між "then abort" і "end select", тобто виклик рандеву із задачею-сервером Server\_Task на вході Service\_1. При цьому, якщо закінчення інтервалу часу з інструкції delay (або delay until), розташованої після select, відбудеться раніше завершення виконання Server\_Task.Service\_1, то виконання Server\_Task.Service\_1 примусово переривається і виконується виведення повідомлення "Server\_Task is not serviced the Service\_1 yet".  
Використання асинхронної передачі управління не обмежена багатозадачністю, наприклад:

|  |
| --- |
| **select**  **delay** 5.0;  Put\_Line ("So\_Big\_Calculation abandoned!");  **then abort**  So\_Big\_Calculation;  **end select**; |

У цьому випадку використання асинхронної передачі дозволяє перервати невизначено довгу процедуру “So\_Big\_Calculation” що виконується нескінченному циклі.  
 При селекції виклику рандеву за допомогою інструкцій відбору (для здійснення умовного виклику рандеву, тимчасового виклику рандеву або асинхронної передачі управління) слід пам'ятати, що якщо викликана задача вже завершена, то виконання інструкції відбору призведе до порушення виключення Tasking\_Error.

Проблеми механізму рандеву  
 Незважаючи на велику кількість можливостей, що надаються моделлю механізму рандеву Ada83, який передбачає розвинений високорівневий підхід для синхронізації задач, що дозволяє уникнути численні методологічні складності, викликані застосуванням низькорівневих примітивів, подібних семафора і сигналами, цей механізм має деякі недоліки.  
 Уявімо собі класичний приклад, коли необхідно забезпечити взаємодію двох задач, одна з яких виконує читання даних з будь-якого пристрою, а друга - здійснює обробку цих даних. В літературі подібне взаємовідношення завдань, як правило, називають "постачальник-споживач". Механізм рандеву є одночасно як механізмом синхронізації, так і механізмом міжзадачного обміну даними. Необхідність передачі даних між двома задачами, які зазвичай виконуються асинхронно, вимагає призупинення виконання задач для здійснення "переговорів". Отже, для ефективної організації подібної взаємодії необхідна буферизація даних, переданих від однієї задачі до іншої. Звичайним рішенням у цій ситуації є створення проміжної задачі, яка служить буфером і управляє потоком даних, а отже, завжди доступна для рандеву з обома задачами, які потребують обмін даними. При цьому слід врахувати, що рандеву є відносно тривалої операцією, і необхідність наявності додаткової задачі-буфера призводить до невиправданого зниження продуктивності.  
 Виходячи з таких міркувань, набагато краще виглядає можливість, щоб задача-виробник (та задача, яка читає дані з пристрою) могла просто залишати дані в будь-якій змінній для задачі-приймача (та задачі, яка обробляє прийняті дані), після чого задача -приймач могла б так само просто ці дані прочитати. Однак, звичайна змінна для цієї мети не підходить, оскільки необхідно запобігти модифікацію вмісту змінної двома задачами, що виконуються одночасно.  
 Ще одним недоліком рандеву є те, що в деяких ситуаціях може виникнути інверсія абстракції, що приводить до замкнутої взаємної блокування задач. Крім того, з методологічної точки зору, ідеологія механізму рандеву строго орієнтована на управління і знаходиться поза новітніх об'єктно-орієнтованих підходів.  
 Засобом, який дозволяє організувати взаємно виключний доступ до даних з різних задач, які виконуються одночасно, є захищені модулі, які були введені стандартом Ada95. Характерною особливістю захищених модулів є забезпечення ними синхронізованого доступу до приватних даних, однак на противагу задачам, які є активними сутностями, захищені модулі - пасивні.

**1.4 Порівняння засобів MPI, PVM, Рандеву**

Бібліотеки PVM та MPI - найбільш поширені засоби програмування для комп'ютерних систем з розподіленою пам'ять, а також для розподілених комп'ютерних систем.

MPI - вельми розгалужений інструментарій. Те, що в бібліотеці PVM та в механізмі Рандеву реалізовано одним-єдиним способом, в MPI може бути зроблено декількома, про які йдеться: спосіб А простий у використанні, але не дуже ефективний; спосіб Б складніше, але ефективніше; а спосіб В складніше і ефективніше за певних умов ".

Колективна взаємодія краще реалізована в PVM та Рандеву, там можуть взаємодіяти лише кілька процесів, а в MPI взаємодіють всі процеси.

Вибір бібліотеки багато в чому залежить від задачі, яку необхідно вирішити. PVM надійніший оскільки там багато перевірки виконуються автоматично, в MPI їх треба виконувати самому, правда, якщо вони не потрібні заощаджується час, а механізм рандеву заснован на високорівневих концепціях рандеву, він надає засоби блокування і таймаутів, а також засоби для виконання вибіркового перестроювання черг клієнтів та аварійного завершення.

# Висновки до Розділу 1

Розглянуто реалізацію бібліотек MPI, PVM та механізм посилки повідомлень “Рандеву” у мовах програмування. Яка суттєва різниця між ними, чим відрізняються механізми для посилки повідомлень та організацій задач. Переваги та недоліки бібліотек. Та для яких задач краще використовувати ту чи іншу бібліотеку.

Бібліотеки MPI та PVW більш сучасніші для паралельного програмування, але мова Ада була створена ще у 1979-1980 роках, вона вважається більшь надійнішою та стабільнішою, саме тому вона досі використовується, як для звичайного програмування, так і для паралельного.

**РОЗДІЛ 2 Розробка програмного забезпечення для ПКС з ЗП**

**2.1 Побудова паралельного алгоритму.**

Вираз для побудови паралельного алгоритму:

MA=max(MZ)\*MC + α\*MO\*MX

З якого побудований паралельний алгоритм:

1. ai=max(MZH), i=(1..P);
2. a=max(a, ai), i=(1..P);
3. MAH=a\*MCH+ α\*MO\*MXH;

Н=N/P

Спільні ресурси: a, α, МО.

**2.2 Розробка алгоритмів роботи кожного процесу**

**Алгоритм задачі N1**

1. Введення MZ, MX
2. Сигнал про введення MZ, MX в задачі Т1
3. Чекати введення MC, α в задачі T3
4. Чекати введення МО в задачі Т6
5. Обчислення a1=max(MZH)
6. Обчислення a=max(a,a1)
7. Сигнал про завершення обчислення a в задачі T1
8. Чекати завершення обчислень a в задачах T2-T6
9. Копіювання a1=a; α1= α; MO1=MO;
10. Обчислення MAH=a1\*MCH + α1\*MO1\*MXH
11. Сигнал задачі T6 про завершення обчислення MAH

**Алгоритм задачі N3**

1. Введення MC, α
2. Сигнал про введення MC, α в задачі Т3
3. Чекати введення MZ, MX в задачі T1
4. Чекати введення МО в задачі Т6
5. Обчислення a3=max(MZH)
6. Обчислення a=max(a,a3)
7. Сигнал про завершення обчислення a в задачі T3
8. Чекати завершення обчислень a в задачах T1,T2,T4-T6
9. Копіювання a3=a; α3= α; MO3=MO;
10. Обчислення MAH=a3\*MCH + α3\*MO3\*MXH
11. Сигнал задачі T6 про завершення обчислення MAH

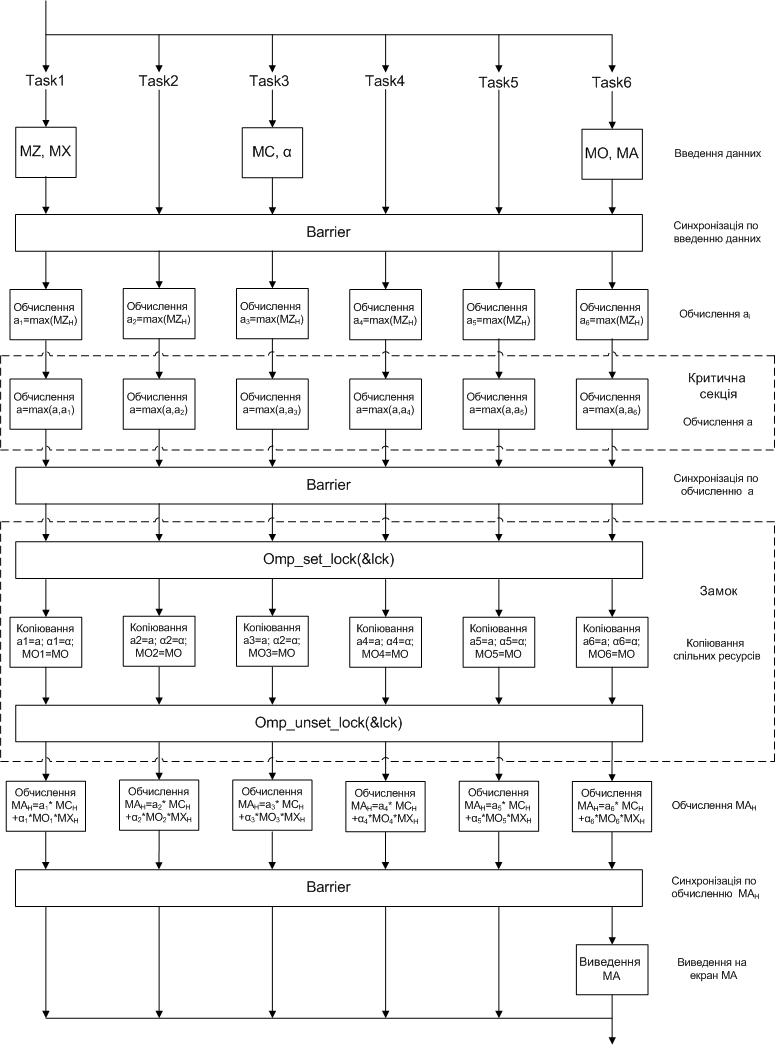
**Алгоритм задачі N6**

1. Введення MO
2. Сигнал про введення MO в задачі Т6
3. Чекати введення MZ, MX в задачі Т1
4. Чекати введення MC, α в задачі T3
5. Обчислення a6=max(MZH)
6. Обчислення a=max(a,a6)
7. Сигнал про завершення обчислення a в задачі T6
8. Чекати завершення обчислень a в задачах T1-T5
9. Копіювання a6=a; α6= α; MO6=MO;
10. Обчислення MAH=a6\*MCH + α6\*MO6\*MXH
11. Чекати завершення обчислень MAH в задачах T1-T5
12. Виведення MA

**Алгоритм задач N2,4,5**

1. Чекати введення MZ, MX в задачі Т1
2. Чекати введення MC, α в задачі T3
3. Чекати введення МО в задачі Т6
4. Обчислення ai=max(MZH)
5. Обчислення a=max(a,ai)
6. Сигнал про завершення обчислення a в задачі T2,T4,T5
7. Чекати завершення обчислень a в задачах T1, T3, T6
8. Копіювання ai=a; αi= α; MOi=MO;
9. Обчислення MAH=ai\*MCH + αi\*MOi\*MXH
10. Сигнал задачі T6 про завершення обчислення MAH

**2.3 Розроблення схеми взаємодії процесів**



На рисунку 2.2 зображено розроблену схему взаємодії процесів.

**2.4 Розробка паралельної програми з використанням заданих засобів синхронізації, налагодження програми та її виконання**

Для реалізації алгоритму обчислення матричного рівняння на обчислювальній системі зі спільною пам’яттю було використано мову C++ до якої було підключено бібліотеку Open MP (Open Multi-Processing).

Для вирішення проблем синхронізації та доступу до спільного ресурсу було використані такі бібліотечні механізми:

“Бар'єрна синхронізація” - досягнувши кінця регіону, всі потоки блокуються до тих пір, поки останній потік не завершить свою роботу;

“Критична секція” - частина програми, в якій є звернення до спільних ресурсів. Два потоки не повинні одночасно перебувати в критичній секції, отже якщо один потік знаходиться в критичній секції, інші блокуються при спробі зайти у критичну секцію;

“Замок” – реалізація критичної секції.

Лістинг програми наведено у Додатку Д.

**2.5 Проведення досліджень ефективності розробленої програми в реальній 4-х ядерній системі**

Тестування проводилось на паралельній комп’ютерній системі (ПКС), характеристики якої наведені в Табл. 2.1. 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. 2.2, та розраховано коефіцієнти прискорення та ефективності, Табл. 2.3 та Табл. 2.4 відповідно.

Табл. 2.1 Характеристики стенду

|  |  |
| --- | --- |
| Процесор | Intel® Core™ i5-2400 Processor[26] |
| Материнська плата | ASUS P8H67-M LX[27] |
| Оперативна пам’ять | DDR3 1333 MHz |
| Операційна система | Microsoft Windows 7 SP1 Ultimate[28] |

Результати проведених досліджень ефективності розробленої програми:

Таблиця 2.1. Час виконання обчислень програмою:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 |
| 1000 | 4.5 c | 2.3 c | 1.6 c | 1.4 c |
| 2000 | 37.3 c | 18.8 c | 12.6 c | 11 c |
| 3000 | 125.1 c | 63.2 c | 42.6 c | 36.9 c |

Таблиця 2.2. Значення коефіцієнтів прискорення:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 |
| 1000 | 1 | 1.95 | 2.8 | 3.21 |
| 2000 | 1 | 1.98 | 2.96 | 3.39 |
| 3000 | 1 | 1.97 | 2.93 | 3.39 |

Таблиця 2.3. Значення коефіцієнтів ефективності:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 |
| 1000 | 100% | 97,5% | 93.3% | 80.25% |
| 2000 | 100% | 99.0% | 98.6% | 94.75% |
| 3000 | 100% | 98.5% | 97.6% | 94.75% |

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.

**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.

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, окрім розмірності N = 1000, через малий час обчислення. Це означає, що ОС віддає перевагу більш пріоритетним задачам.

**РОЗДІЛ 3 Розробка програмного забезпечення для ПКС з ЛП**

**3.1 Побудова паралельного алгоритму.**

Вираз для побудови паралельного алгоритму:

MA=max(MZ)\*MC + α\*MO\*MX

З якого побудований паралельний алгоритм:

1. ai=max(MZH), i=(1..P);
2. a=max(a, ai), i=(1..P);
3. MAH=a\*MCH+ α\*MO\*MXH;

Н=N/P

Спільні ресурси: a, α, МО.

**3.2 Розроблення алгоритмів роботи кожного процесу.**

**Алгоритм задачі N1**

1. Введення MZ, MX
2. Обміняти MZ4H, MX4H на α, MCH, MO з Т3
3. Передати T2 – α, MO, MZH, MXH, MCH
4. Обчислення a=max(MZH)
5. Передати T3 – a1
6. Прийняти від T3 - а
7. Обчислення MAH=a\*MCH + α\*MO\*MXH
8. Передати T3 - MAH

**Алгоритм задачі N2**

1. Прийняти від T1 – α, MO, MZH, MXH, MCH
2. Обчислення a2=max(MZH)
3. Передати T4 – a2
4. Прийняти від T4 – а
5. Обчислення MAH=a\*MCH + α\*MO\*MXH
6. Передати T4 - MAH

**Алгоритм задачі N3**

1. Введення α, MC
2. Обміняти α, MC2H на MO з Т6
3. Передати T6 – MZH, MXH
4. Передати T4 – α, MCH, MO, MXH, MZH
5. Обчислення a3=max(MZH)
6. Прийняти від T1 – a1
7. a3 = max(a3 , a1)
8. Передати T6 – a3
9. Прийняти від T6 – а
10. Передати T1 - a
11. Обчислення MAH=a\*MCH + α\*MO\*MXH
12. Прийняти від T1 - MAH
13. Передати T6 – MA2H

**Алгоритм задачі N4**

1. Прийняти від T3 – α, MCH, MO, MXH, MZH
2. Обчислення a4=max(MZH)
3. Прийняти від T2 – a2
4. a4 = max(a4 , a2)
5. Передати T5 – a4
6. Прийняти від T5 – а
7. Передати T2 - a
8. Обчислення MAH=a\*MCH + α\*MO\*MXH
9. Прийняти від T2 - MAH
10. Передати T5 – MA2H

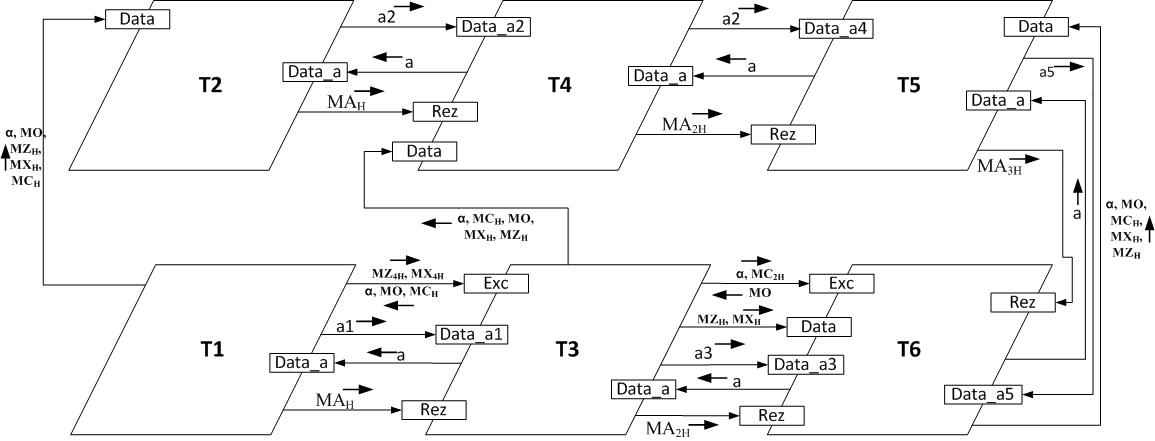
**Алгоритм задачі N5**

1. Прийняти від T6 – α, MO, MCH, MXH, MZH
2. Обчислення a5=max(MZH)
3. Прийняти від T3 – a3
4. a6 = max(a6 , a3)
5. Передати T6 – a5
6. Прийняти від T6 – а
7. Обчислення MAH=a\*MCH + α\*MO\*MXH
8. Прийняти від T4 – MA2H
9. Передати T6 – MA3H

**Алгоритм задачі N6**

1. Введення MO
2. Прийняти від Т3 - MX2H, MZ2H
3. Передати T5 – α, MO, MCH, MXH, MZH
4. Обчислення a5=max(MZH)
5. Прийняти від T5 – a5
6. a6 = max(a5 , a6)
7. Прийняти від T3 – a3
8. a = max(a6 , a3)
9. Передати T3 – a
10. Передати T5 - a
11. Обчислення MAH=a\*MCH + α\*MO\*MXH
12. Прийняти від T3 - MA2H
13. Прийняти від T5 - MA3H
14. Виведення MA

**3.3 Розроблення схеми взаємодії процесів**



На рисунку 3.2 зображено розроблену схему взаємодії процесів.

**3.4 Розробка паралельної програми з використанням заданих засобів передачі даних, налагодження програми та її виконання**

В результаті розробки алгоритмів задач та схеми їх взаємодії розроблено паралельну програму для вирішення заданої математичної задачі за допомогою засобів механізму рандеву в мові Ада. Для пересилки даних використовуються захищені входи. Є шість видів потоків – Т1, Т2, T3, T4, T5, T6. В залежності від номеру потоку для нього визначені входи, які пересилають або отримують дані.

Лістинг наведено в додатку Е.

**3.5 Проведення досліджень ефективності розробленої програми в реальній 4-х ядерній системі**

Тестування проводилось на ПКС, характеристики якої наведені в Табл. 3.1. 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. 3.2, та розраховано коефіцієнти прискорення та ефективності, Табл. 3.3 та Табл. 3.4.

Табл. 3.1 Характеристики стенду

|  |  |
| --- | --- |
| Процесор | Intel® Core™ i5-2400 Processor |
| Материнська плата | ASUS P8H67-M LX[27] |
| Оперативна пам’ять | DDR3 1333 MHz |
| Операційна система | Microsoft Windows 7 SP1 Ultimate[28] |

Результати проведених досліджень ефективності розробленої програми:

Таблиця 3.1. Час виконання обчислень програмою:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 |
| 1000 | 8.35 c | 5.77 c | 3.77 c | 2.99 c |
| 2000 | 62.83 c | 45.81 c | 30.06 c | 23.43 c |
| 3000 | 251.83 c | 154.4 c | 101.29 c | 80.61 c |

Таблиця 3.2. Значення коефіцієнтів прискорення:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 |
| 1000 | 1 | 1.44 | 2.21 | 2.79 |
| 2000 | 1 | 1.36 | 2.09 | 2.68 |
| 3000 | 1 | 1.63 | 2.48 | 3.12 |

Таблиця 3.3. Значення коефіцієнтів ефективності:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P=1 | P=2 | P=3 | P=4 |
| 1000 | 100% | 72% | 73,6% | 69.75% |
| 2000 | 100% | 68% | 69.6% | 67% |
| 3000 | 100% | 81.5% | 82.6% | 78% |

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.

**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.

# 

# ВИСНОВКИ

1. В даній роботі було проведено порівняння бібліотек паралельного програмування MPI, PVM ті механізму Рандеву. Можна зробити висновок, що зручніше використовувати бібліотеки MPI та PVM у паралельному програмуванні, так як вони мають більше різноманітних засобів. Але більш надійнішим та стабільнішим залишається механізм рандеву.
2. У другому розділі було розроблено паралельну програму яка працює зі спільною пам'яттю за допомогою бібліотеки Open MP, 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 набагато швидше обчислює математичний вираз ніж програма зі локальною пам'яттю, це можна пояснити тим, що справа може бути не лише у часу на пересилку даних, а також мовним компілятором. У другому у другому етапі використовується Microsoft Visual Studio який працює швидше за Object Ada який використовується у третьому етапі.

### 

**СКОРОЧЕННЯ**

1. Ке - коефіцієнт ефективності.
2. Кп – коефіцієнт прискорення.
3. КР – курсова робота.
4. ЛП – локальна пам’ять.
5. ОС – операційна система.
6. ПЗ – програмне забезпечення.
7. ПКС – паралельна комп’ютерна система.
8. СП – спільна пам’ять.
9. ТЗ – технічне завдання.

ПЗ – програмне забезпечення.

# 

**ЛІТЕРАТУРА**

1. Жуков І. А., Корочкін О. В. – “Паралельні та розподілені обчислення “–Київ: «Корнійчук» 2005. – 224 с.
2. Корочкин А.В. Ада 95: Введение в программирование. - Киев; Свит, . 1998. - 260 с.
3. [Камерон Хьюз, Трейси Хьюз](http://www.ozon.ru/context/detail/id/1919696/#tab_person) – “Параллельное и распределенное программирование с использованием C++”
4. [Брайан Керниган, Деннис Ритчи](http://www.ozon.ru/context/detail/id/2480925/#tab_person) – “The C Programming language”
5. Самюел П. Харбисон, Гай Л. Стил – “Язык С”
6. Антонов, Шпаковский, Серикова, Корнеев – “Сборник "MPI"”
7. Классификация параллельных вычислительных систем [З мережі] http://ru.wikipedia.org/wiki/Классификация параллельных вычислительных систем
8. Многозадачность [З мережі]

http://www.ada-ru.org

1. Параллельное программирование [З мережі]

http://msdn.microsoft.com/ru-ru/library/ee856607.aspx

1. Справочник Visual Studio [З мережі]
2. http://msdn.microsoft.com/ru-ru/library/scesz732.aspx
3. ReaderWriterLockSlim Class (System.Threading) [З мережі] http://msdn.microsoft.com/ru-ru/library/system.threading.readerwriterlockslim(v=vs.110).aspx
4. LockRecursionException Class (System.Threading) [З мережі] http://msdn.microsoft.com/ru-ru/library/system.threading.lockrecursionexception(v=vs.110).aspx
5. Barrier Class (System.Threading) [З мережі] http://msdn.microsoft.com/ru-ru/library/system.threading.barrier(v=vs.110).aspx
6. BarrierPostPhaseException Class (System.Threading) [З мережі] http://msdn.microsoft.com/ru-ru/library/system.threading.barrierpostphaseexception(v=vs.110).aspx
7. Microsoft Corporation: Software, Smartphones, Online, Games, Cloud Computing, IT Business Technology, Downloads [З мережі] http://www.microsoft.com/en-us/default.aspx
8. Intel Core i5-2400 Processor [З мережі] http://www.intel.com
9. ASUS - Motherboards- ASUS P8H67-M LX [З мережі] http://www.asus.com

**ДОДАТКИ**

Додаток А. Алгоритм основної програми для ПКС з СП

Додаток Б. Алгоритми процесів в програмі для ПКС з СП

Додаток В. Алгоритми процесів в програмі для ПКС з СП

Додаток Г. Алгоритми процесів в програмі для ПКС з СП

Додаток Ґ. Алгоритми процесів в програмі для ПКС з СП

Додаток Д. Алгоритми процесів в програмі для ПКС з СП

Додаток Е. Алгоритми процесів в програмі для ПКС з СП

Додаток Є. Алгоритм основної програми для ПКС з ЛП

Додаток Ж. Алгоритми процесів основної програми для ПКС з ЛП

Додаток З. Алгоритми процесів основної програми для ПКС з ЛП

Додаток И. Алгоритми процесів основної програми для ПКС з ЛП

Додаток І. Алгоритми процесів основної програми для ПКС з ЛП

Додаток Ї. Алгоритми процесів основної програми для ПКС з ЛП

Додаток Й. Алгоритми процесів основної програми для ПКС з ЛП

Додаток К. Лістинг програми з СП

#include "stdafx.h"

#include <iostream>

#include "omp.h"

#include "time.h"

using namespace std;

#pragma comment(linker, "/stack:160000000")

const int N = 6;

const int P = 6;

const int H = N/P;

int a = -999999999;

int\*\* MA = new int\*[N];

int \*\*MZ = new int\*[N];

int \*\*MX = new int\*[N];

int \*\*MC = new int\*[N];

int \*\*MO = new int\*[N];

int alfa;

void inMatrix(int\*\* matr) {

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

matr[i] = new int[N];

}

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

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

matr [i][j] = 1;

}

}

}

void cloneM(int \*\*MT, int \*\*MM) {

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

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

MT [i][j] = MM [i][j];

}

}

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

clock\_t start=clock();

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

MA[i] = new int[N];

}

printf("Main thread: started\n");

omp\_lock\_t lck;

omp\_init\_lock(&lck);

omp\_set\_num\_threads(P);

#pragma omp parallel

{

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

printf("Thread %d: started\n", num);

//ввод данных

switch (num) {

case 0:

inMatrix(MZ);

inMatrix(MX);

break;

case 2:

inMatrix(MC);

alfa = 1;

break;

case 5:

inMatrix(MO);

break;

}

//синхронизация по вводу

#pragma omp barrier

//счет ax=max(MZ)

int ax = -999999999;

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

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

if (MZ[i][j] > ax) {

ax = MZ[i][j];

}

}

}

//критическая секция

#pragma omp critical

{

if (ax > a) {

a = ax;

}

}

//синхронизация по счету a

#pragma omp barrier

int \*\*MOx = new int \*[N];

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

MOx[i] = new int[N];

}

//замок

omp\_set\_lock(&lck);

cloneM(MOx, MO);

omp\_unset\_lock(&lck);

//счет MA

int sum;

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

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

sum = 0;

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

sum += alfa \* MX[i][j] \* MOx[j][z];

}

sum += a\*MC[j][i];

MA[i][j] = sum;

}

}

//синхронизация по выводу

#pragma omp barrier

//вывод результата

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

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

cout << MA[i][j] << "\t";

printf("Thread %d: finished\n", num);

}

printf("Main thread: finished\n");

cout<< "time = " <<(long double) (clock() - start) / CLOCKS\_PER\_SEC << " sec " << endl;

return 0;

}

Додаток Л. Лістинг програми з ЛП.

--Земцов А.А гр.IO-93

with Ada.TEXT\_IO; use Ada.TEXT\_IO;

with Ada.INTEGER\_TEXT\_IO; use Ada.INTEGER\_TEXT\_IO;

PROCEDURE LAB5 IS

N: INTEGER := 3600;

P: INTEGER := 6;

H: INTEGER := N/P;

TYPE VECTOR IS ARRAY (INTEGER RANGE <>) OF INTEGER;

TYPE MATRIX IS ARRAY (INTEGER RANGE <>) OF VECTOR(1..N);

TASK T1 IS

ENTRY exc(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: out matrix; mz1: out matrix);

ENTRY dataA(a1: in integer);

END T1;

TASK T2 IS

ENTRY data(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: in matrix; mz1: in matrix);

ENTRY dataA(a1: in integer);

END T2;

TASK T3 IS

ENTRY exc(alfa1: out integer; mc1: out matrix; mo1: in matrix);

ENTRY res(MA1: in matrix);

ENTRY dataA(a1: in integer);

ENTRY max(a1: in integer);

END T3;

TASK T4 IS

ENTRY data(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: in matrix; mz1: in matrix);

ENTRY res(MA1: in matrix);

ENTRY dataA(a1: in integer);

ENTRY max(a1: in integer);

END T4;

TASK T5 IS

ENTRY data(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: in matrix; mz1: in matrix);

ENTRY res(MA1: in matrix);

ENTRY dataA(a1: in integer);

ENTRY max(a1: in integer);

END T5;

TASK T6 IS

ENTRY data(MX1: in matrix; MZ1: in matrix);

ENTRY res5(MA1: in matrix);

ENTRY res3(MA1: in matrix);

ENTRY max(a1: in integer);

END T6;

---------------------------------------------------------------------------------------------------

TASK BODY T1 IS

alfa, a, z1: INTEGER;

MA: matrix(1..H);

MC: MATRIX(1..H\*2);

MO: MATRIX(1..N);

MX: MATRIX(1..N);

MZ: MATRIX(1..N);

BEGIN

PUT\_LINE(" T1 STARTED ");

a:= -999999;

FOR I IN 1..N LOOP

FOR J IN 1..N LOOP

MX(I)(J) := 1;

MZ(I)(J) := 1;

END LOOP;

end loop;

--2

ACCEPT exc(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: out matrix; mz1: out matrix) do

ALFA:= ALFA1;

MC:= MC1;

MO:= MO1;

MX1:= MX(H\*2+1..H\*6);

MZ1:= MZ(H\*2+1..H\*6);

END EXC;

--3

T2.data(alfa, mc(H+1..H\*2), mo, mx(H+1..H\*2), mz(H+1..H\*2));

FOR I IN 1..H LOOP

FOR J IN 1..N LOOP

IF MZ(I)(J) > a THEN

a:= MZ(I)(J);

END IF;

END LOOP;

END LOOP;

T3.MAX(a);

ACCEPT DATAA(A1: in integer) do

A:= A1;

END DATAA;

for i in 1..H loop

for j in 1..N loop

Z1:= 0;

for k in 1..N loop

z1:= z1+ mX(i)(j)\*mO(j)(k);

end loop;

MA(I)(J):= a\*MC(I)(J)+ALFA\*Z1;

end loop;

end loop;

T3.RES(MA);

PUT\_LINE(" T1 FINISHED ");

END T1;

---------------------------------------------------------------------------------------------------

TASK BODY T2 IS

alfa, a, z1: INTEGER;

MA: matrix(1..H);

MC: MATRIX(1..H);

MO: MATRIX(1..N);

MX: MATRIX(1..H);

MZ: MATRIX(1..H);

BEGIN

PUT\_LINE(" T2 STARTED ");

a:= -999999;

--2

ACCEPT DATA(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: IN matrix; mz1: IN matrix) do

ALFA:= ALFA1;

MC:= MC1;

MO:= MO1;

MX:= MX1;

MZ:= MZ1;

END DATA;

FOR I IN 1..H LOOP

FOR J IN 1..N LOOP

IF MZ(I)(J) > a THEN

a:= MZ(I)(J);

END IF;

END LOOP;

END LOOP;

T4.MAX(a);

ACCEPT DATAA(A1: in integer) do

A:= A1;

END DATAA;

for i in 1..H loop

for j in 1..N loop

Z1:= 0;

for k in 1..N loop

z1:= z1+ mX(i)(j)\*mO(j)(k);

end loop;

MA(I)(J):= a\*MC(I)(J)+ALFA\*Z1;

end loop;

end loop;

T4.RES(MA);

PUT\_LINE(" T2 FINISHED ");

END T2;

---------------------------------------------------------------------------------------------------

TASK BODY T3 IS

alfa, a, z1: INTEGER;

MA: matrix(1..H\*2);

MC: MATRIX(1..N);

MO: MATRIX(1..N);

MX: MATRIX(1..H\*4);

MZ: MATRIX(1..H\*4);

BEGIN

PUT\_LINE(" T3 STARTED ");

a:= -999999;

FOR I IN 1..N LOOP

FOR J IN 1..N LOOP

MC(I)(J) := 1;

END LOOP;

end loop;

ALFA:= 1;

--2

ACCEPT exc(alfa1: out integer; mc1: out matrix; mo1: in matrix) do

ALFA1:= ALFA;

MC1:= MC(H\*4+1..H\*6);

MO:= MO1;

END EXC;

--3

T1.exc(alfa, mc(1..H\*2), mo, mx, mz);

T6.data(mx(H\*2+1..H\*4), mz(H\*2+1..H\*4));

T4.data(alfa, mc(H\*3+1..H\*4), mo, mx(H+1..H\*2), mz(H+1..H\*2));

FOR I IN 1..H LOOP

FOR J IN 1..N LOOP

IF MZ(I)(J) > a THEN

a:= MZ(I)(J);

END IF;

END LOOP;

END LOOP;

ACCEPT MAX(A1: in integer) do

IF A1 > A THEN

A:= A1;

END IF;

END MAX;

T6.MAX(a);

ACCEPT DATAA(A1: in integer) do

A:= A1;

END DATAA;

T1.dataA(a);

for i in 1..H loop

for j in 1..N loop

Z1:= 0;

for k in 1..N loop

z1:= z1+ mX(i)(j)\*mO(j)(k);

end loop;

MA(I)(J):= a\*MC(I)(J)+ALFA\*Z1;

end loop;

end loop;

ACCEPT RES(MA1: in MATRIX) do

MA(H+1..H\*2):= MA1;

END RES;

T6.RES3(MA);

PUT\_LINE(" T3 FINISHED ");

END T3;

---------------------------------------------------------------------------------------------------

TASK BODY T4 IS

alfa, a, z1: INTEGER;

MA: matrix(1..H\*2);

MC: MATRIX(1..H);

MO: MATRIX(1..N);

MX: MATRIX(1..H);

MZ: MATRIX(1..H);

BEGIN

PUT\_LINE(" T4 STARTED ");

a:= -999999;

--2

ACCEPT DATA(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: IN matrix; mz1: IN matrix) do

ALFA:= ALFA1;

MC:= MC1;

MO:= MO1;

MX:= MX1;

MZ:= MZ1;

END DATA;

--3

FOR I IN 1..H LOOP

FOR J IN 1..N LOOP

IF MZ(I)(J) > a THEN

a:= MZ(I)(J);

END IF;

END LOOP;

END LOOP;

ACCEPT MAX(A1: in integer) do

IF A1 > A THEN

A:= A1;

END IF;

END MAX;

T5.MAX(a);

ACCEPT DATAA(A1: in integer) do

A:= A1;

END DATAA;

T2.dataA(a);

for i in 1..H loop

for j in 1..N loop

Z1:= 0;

for k in 1..N loop

z1:= z1+ mX(i)(j)\*mO(j)(k);

end loop;

MA(I)(J):= a\*MC(I)(J)+ALFA\*Z1;

end loop;

end loop;

ACCEPT RES(MA1: in MATRIX) do

MA(H+1..H\*2):= MA1;

END RES;

T5.RES(MA);

PUT\_LINE(" T4 FINISHED ");

END T4;

---------------------------------------------------------------------------------------------------

TASK BODY T5 IS

alfa, a, z1: INTEGER;

MA: matrix(1..H\*3);

MC: MATRIX(1..H);

MO: MATRIX(1..N);

MX: MATRIX(1..H);

MZ: MATRIX(1..H);

BEGIN

PUT\_LINE(" T5 STARTED ");

a:= -999999;

--2

ACCEPT DATA(alfa1: in integer; mc1: in matrix; mo1: in matrix; mx1: IN matrix; mz1: IN matrix) do

ALFA:= ALFA1;

MC:= MC1;

MO:= MO1;

MX:= MX1;

MZ:= MZ1;

END DATA;

--3

FOR I IN 1..H LOOP

FOR J IN 1..N LOOP

IF MZ(I)(J) > a THEN

a:= MZ(I)(J);

END IF;

END LOOP;

END LOOP;

ACCEPT MAX(A1: in integer) do

IF A1 > A THEN

A:= A1;

END IF;

END MAX;

T6.MAX(a);

ACCEPT DATAA(A1: in integer) do

A:= A1;

END DATAA;

T4.dataA(a);

for i in 1..H loop

for j in 1..N loop

Z1:= 0;

for k in 1..N loop

z1:= z1+ mX(i)(j)\*mO(j)(k);

end loop;

MA(I)(J):= a\*MC(I)(J)+ALFA\*Z1;

end loop;

end loop;

ACCEPT RES(MA1: in MATRIX) do

MA(H+1..H\*3):= MA1;

END RES;

T6.RES5(MA);

PUT\_LINE(" T5 FINISHED ");

END T5;

---------------------------------------------------------------------------------------------------

TASK BODY T6 IS

alfa, a, z1: INTEGER;

MA: matrix(1..N);

MC: MATRIX(1..H\*2);

MO: MATRIX(1..N);

MX: MATRIX(1..H\*2);

MZ: MATRIX(1..H\*2);

BEGIN

PUT\_LINE(" T6 STARTED ");

a:= -999999;

FOR I IN 1..N LOOP

FOR J IN 1..N LOOP

MO(I)(J) := 1;

END LOOP;

end loop;

T3.exc(alfa, mc, mo);

--2

ACCEPT DATA(mx1: IN matrix; mz1: IN matrix) do

MX:= MX1;

MZ:= MZ1;

END DATA;

T5.data(alfa, mc(1..H), mo, mx(1..H), mz(1..H));

--3

FOR I IN H+1..H\*2 LOOP

FOR J IN 1..N LOOP

IF MZ(I)(J) > a THEN

a:= MZ(I)(J);

END IF;

END LOOP;

END LOOP;

ACCEPT MAX(A1: in integer) do

IF A1 > A THEN

A:= A1;

END IF;

END MAX;

ACCEPT MAX(A1: in integer) do

IF A1 > A THEN

A:= A1;

END IF;

END MAX;

T5.dataA(a);

T3.dataA(a);

for i in H..H\*2 loop

for j in 1..N loop

Z1:= 0;

for k in 1..N loop

z1:= z1+ mX(i)(j)\*mO(j)(k);

end loop;

MA(I+H\*4)(J):= a\*MC(I)(J)+ALFA\*Z1;

end loop;

end loop;

ACCEPT RES3(MA1: in MATRIX) do

MA(1..H):= MA1(1..H);

MA(H\*2+1..H\*3):= MA1(H+1..H\*2);

END RES3;

ACCEPT RES5(MA1: in MATRIX) do

MA(H+1..H\*2):= MA1(1..H);

MA(H\*3+1..H\*4):= MA1(H+1..H\*2);

MA(H\*4+1..H\*5):= MA1(1..H);

END RES5;

IF N < 9 THEN

FOR i IN 1..N LOOP

FOR J IN 1..N LOOP

PUT(MA(i)(J));

PUT(" ");

END LOOP;

NEW\_LINE;

END LOOP;

ELSE

PUT(MA(1)(1));

NEW\_LINE;

END IF;

PUT\_LINE(" T6 FINISHED ");

END T6;

BEGIN

NULL;

END LAB5;