Г. О. Козуб, В. Ю. Козуб

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ
ДЕРЖАВНИЙ ЗАКЛАД
„ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА”

Г. О. Козуб, В. Ю. Козуб

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Методичні рекомендації до виконання лабораторних робіт для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 122 „ Комп’ютерні науки ”

Старобільськ
ДЗ „ЛНУ імені Тараса Шевченка”
2022

УДК 004.42(072))

Р е ц е н з е н т и :

Гоменюк С.І. **–** доктор технічних наук, професор, декан математичного факультету Запорізького національного університету.

Могильний Г. А. **–** кандидат технічних наук, доцент, директор Інституту фізики, математики та інформаційних технологій ДЗ „ Луганський національний університет імені Тараса Шевченка

Козуб Г. О. Паралельні та розподілені обчислення**:** Методичні рекомендації до виконання лабораторних робіт. для здоб. першого рівня вищої освіти спец. 122 “Комп’ютерні науки ”/ Козуб Г.О., В. Ю. Козуб, Держ. закл. „Луган. нац. ун-т імені Тараса Шевченка”. – Луганськ : ДЗ „ЛНУ імені Тараса Шевченка”, 2022. – 125 с.

Методичні рекомендації структуровано відповідно до розділів освітньої програми курсу „Паралельні та розподілені обчислення” для спеціальності 122 „Комп’ютерні науки” кафедри фізико-технічних технологій та інформатики ДЗ ЛНУ імені Тараса Шевченка. Посібник охоплює основні принципи побудови паралельних програм, розпаралелювання алгоритмів розв’язання СЛАР ітераційними та прямими методами у MPI та рішення практичних завдань з застосуванням технології OpenMP.

Навчальний посібник призначений для студентів фізико-математичного та технічного профілю, учителів ліцеїв, коледжів, гімназій, слухачів курсів підвищення кваліфікації, а також для самоосвіти.

УДК 004.42(072)

Рекомендовано до друку Вченою радою
Луганського національного університету імені Тараса Шевченка
(протокол № 6 від 28 січня 2022 р.)

ЗМІСТ

ВСТУП

Суперкомп'ютерні технології та високопродуктивні обчислення з використанням паралельних обчислювальних систем стають важливим фактором науково-технічного прогресу; їх застосування набуває загального характеру.

Знання сучасних тенденцій розвитку комп'ютерних та апаратних засобів для досягнення паралелізму, вміння розробляти моделі, методи та програми паралельного вирішення завдань обробки даних слід віднести до важливих кваліфікаційних характеристик сучасного фахівця з комп'ютерних наук, комп'ютерної інженерії, прикладної математики, інформатики та інженерії програмного забезпечення.

Вивчення освітнього компоненту (ОК) „Паралельні та розподілені обчислення" є невід'ємною частиною у загальному процесі навчання здобувачів освіти спеціальностей 122 – „Комп'ютерні науки". Головна мета полягає у формуванні базових знань з технології програмування, вивчення математичних моделей, методів і технологій паралельного програмування та розподіленого обчислення для багатопроцесорних обчислювальних систем в обсязі, достатньому для успішного початку робіт в області паралельного програмування. Знання, отримані при вивченні цього ОК, забезпечать професійну підготовку фахівців в галузі комп'ютерних наук.

Завдання ОК „Паралельні та розподілені обчислення"– сформувати у студентів теоретичні знання про методи програмування для паралельних комп'ютерів, організацію обчислень у багатопроцесорних системах, практичні уміння розробляти математичні моделі паралельних алгоритмів, аналізувати, оптимізувати, настроювати програми, використовуючи при цьому мову С, С++ , бібліотеку MPI, технології Open MP.

За результатами вивчення ОК здобувачі освіти повинні **знати:**

− теоретичні основи організації паралельних і розподілених обчислювальних процесів, розпаралелювання алгоритмів, перетворення послідовних програм в паралельні;

− основні поняття та принципи створення багатопотокового додатку, основні керівництва для визначення потоків та типи паралельних архітектур;

− фундаментальні концепції, парадигми і основні принципи функціонування мовних, інструментальних і обчислювальних засобів інженерії програмного забезпечення;

− проблематику організації паралельних і розподілених обчислень;

− знання методів розробки в якості освоєння засобів паралельного програмування у мовах Java, C++.

вміти:

– використовувати концепції паралельної обробки інформації;

– оцінювати складові ефективності алгоритмів функціонування комп'ютеризованих систем;

– використовувати розподілену парадигму проектування програмного забезпечення;

– знаходити паралелізм і розподіляти операції та дані алгоритму між процесорами;

– установлювати порядок виконання операцій та обміну даними;

– використовувати інструментальні засоби для організації паралельних і розподілених обчислювальних процесів;

– реалізовувати синхронні або асинхронні паралельні процеси з використанням бібліотек MPI / OpenMP, стандартними засобами мов програмування C++, Java для розподілених обчислень;

– створювати ефективний паралельний код.

– виконувати паралельні та розподілені обчислення, застосовувати чисельні методи та алгоритми для паралельних структур, мови паралельного програмування при розробці та експлуатації паралельного та розподіленого програмного забезпечення.

Однією із основних навчальних форм є лабораторні роботи, які відіграють провідну роль у формуванні навичок та застосуванні набутих знань з паралельних та розподілених обчислень. Лабораторні заняття логічно продовжують вивчення

тематик, розпочатих на лекціях. Усі форми лабораторних занять призначені для відпрацювання практичних дій.

Методичні рекомендації з лабораторного практикуму структуровані відповідно до навчальної програми освітнього компоненту „Паралельні та розподілені обчислення” і складаються з шестьох лабораторних робіт, що охоплюють питання першого модуля „Основи паралельного програмування”.

Зміст освітнього компонента.

№	Змістовні модулі та їх структура	денна				заочна			
		загальна кількість	лекції	лабораторні	самостійна	загальна кількість	лекції	лабораторні	самостійна
Модуль 1. Основи паралельного програмування									
1.1.	Вступ. Огляд архітектури багатопроцесорних обчислювальних систем.	10	2		8		10		10
1.2.	Засоби програмування багатопроцесорних систем. Паралельні обчислювальні методи	12	2	2	8	12	1	2	9
1.3.	Базові алгоритми паралельних обчислень. Середовище паралельного програмування MPI. Колективні операції точності виконання арифметичних операцій.	18	4	4	10	18	1	1	16
1.4.	Похідні типи даних і передача упакованих даних. Робота з групами і комунікаторами. Топологія процесів.	14	4	4	6	14	1	1	12
1.5.	Основи OpenMP. Директиви синхронізації. Процедури і змінні середовища OpenMP. Технологія Fork-Join	16	4	2	10	16	1		15
	Разом 1 модуль	**70**	**16**	**12**	**40**	**70**	**4**	**4**	**62**

Методичні рекомендації до лабораторної роботи, містять назву, мету, основні теоретичні відомості та завдання для індивідуального виконання. Для кращого розуміння теми, яка розглядається, кожна лабораторна робота містить приклади програм, результати їх виконання. Наприкінці кожна робота має питання для самоперевірки отриманих знань. Зміст лабораторних занять передбачає роботу в комп'ютерних класах та самостійну роботу здобувачів освіти. Для самостійної роботи здобувачам освіти запропоновано теми, відповідна література, питання для самоперевірки, а також опис функцій бібліотеки MPI для виконання завдань. Додатки містять зразок оформлення звіту та матеріали для виконання лабораторних робіт. Здобувачі самостійно опановують теми у вільний від навчання час, у разі виникнення питань він повинен звертатися за допомогою до викладачів під час консультації, яка поводиться он-лайн згідно за графіком.

Методичні рекомендації можуть бути корисними для студентів інших спеціальностей у якості посібника для самостійного опанування технологій паралельного програмування.

ОСНОВИ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

Лабораторна робота № 1
Побудова простих паралельних програм

Мета - дати уявлення про побудову простих паралельних програм на мові паралельного програмування MPI; уявлення про паралельні програми, що налаштовуються на розмір обчислювальної системи, як на параметр; практичне освоєння функцій парних і колективних взаємодій між гілками паралельної програми.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Методи розпаралелювання і моделі програм, які підтримуються MPI

Найважливішою особливістю MPI є те, що користувач при написанні своїх паралельних програм не повинен враховувати архітектурні особливості конкретних мультікомп'ютерів, оскільки MPI надає користувачеві віртуальний мультікомп'ютер з розподіленою пам'яттю і з віртуальної мережею зв'язку між віртуальними комп'ютерами. Користувач замовляє кількість комп'ютерів, необхідних для вирішення його завдання, і визначає топологію зв'язків між цими комп'ютерами. MPI реалізує це замовлення на конкретній фізичній системі. Обмеженням є обсяг оперативної пам'яті фізичного мультікомп'ютера. Таким чином користувач працює у віртуальному середовищі, що забезпечує переносимість його паралельних програм. Система MPI є бібліотекою засобів паралельного програмування для мов C, C++ і Fortran 77.

Однією з цілей, переслідуваних при вирішенні задач на обчислювальних системах, в тому числі і на паралельних, - є ефективність. Ефективність паралельної програми істотно залежить від співвідношення часу обчислень до часу комунікацій між комп'ютерами (при обміні даними). І чим менше в процентному відношенні частка часу, витраченого на

комунікації, в загальному часу обчислень, тим більше ефективність. Для паралельних систем з передачею повідомлень оптимальне співвідношення між обчисленнями і комунікаціями забезпечують методи крупнозернистого розпаралелювання, коли паралельні алгоритми будуються з великих і рідко взаємодіючих блоків [2-8]. Завдання лінійної алгебри, завдання, які вирішуються сітковими методами і багато інших, досить ефективно розпаралелювати грубозернистими методами.

MPMD - модель обчислень. MPI - програма являє собою сукупність автономних процесів, що функціонують під управлінням своїх власних програм і взаємодіють за допомогою стандартного набору бібліотечних процедур для передачі і прийому повідомлень. Таким чином, в найзагальнішому випадку MPI - програма реалізує MPMD - модель програмування (Multiple program - Multiple Data).

SPMD - модель обчислень. Всі процеси виконують в загальному випадку різні гілки однієї і тієї ж програми. Такий підхід обумовлений тією обставиною, що задача може бути досить природним чином розбита на підзадачі, які вирішуються одним алгоритмом. На практиці найчастіше зустрічається саме ця модель програмування (Single program - Multiple Data) [1,12].

Останню модель інакше можна назвати моделлю розпаралелювання за даними. Коротко, суть цього способу полягає в наступному. Вихідні дані завдання розподіляються по процесах (гілкам паралельного алгоритму), а алгоритм є одним і тим же у всіх процесах, але дії цього алгоритму розподіляються відповідно до наявних в цих процесах даних. Розподіл дій алгоритму полягає, наприклад, в привласненні різних значень змінним одних і тих же циклів в різних гілках, або у виконанні в різних гілках різної кількості витків однихі тих же циклів і т.п. Іншими словами, процес в кожній гілці прямує різними шляхами виконання на тій же самій програмі.

Надалі замість "паралельна програма" писатимемо скорочено: *n*-програма.

ПРИКЛАД 1.1

Кожна гілка *n*-програми виводить на екран свій ідентифікаційний номер і розмір замовленої паралельної системи, тобто кількість віртуальних комп'ютерів, в кожен з яких завантажується гілка *n*-програми.

```
#include<stdio.h>
#include<mpi.h>
 int main(int argc, char **argv)
  { int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("SIZE = %d RANK = %d\n",size,rank);
    MPI_Finalize();
    return(0);
}
```

Зверніть увагу на:

1.`#include<mpi.h>` - для приєднання MPI-бібліотеки.

2.`main(int argc, char **argv)` – заголовок головної функції (запис зазначених параметрів обов'язкове).

3.`MPI_Init(&argc, &argv)` – MPI-функція для ініціалізації MPI-бібліотеки; записується в програмі один раз спочатку програми.

4.`MPI_Finalize()` – MPI-функція (без параметрів) для завершення MPI- процесів; записується в кінці програми.

Рядки цих чотирьох пунктів в MPI-програмі обов'язкові.

`MPI_Comm_size(MPI_COMM_WORLD, &size)`- MPI-функція, що записує в змінну `size` кількість запущених паралельних гілок.

`MPI_Comm_rank(MPI_COMM_WORLD, &rank)`- MPI-функція, що записує в змінну `rank` номер паралельної гілки.

Завдання 1.1

Скомпілювати і запустити програму прикладу 1 на 4-х і на 8-й комп'ютерах (умовно).

ПРИКЛАД 1.2

Гілка з номером 0 пересилає дані (в даному випадку число 10) гілки з номером 3. Гілка 3 друкує на екран свій номер і прийняте число від нульової гілки.

```
#include<stdio.h>
#include<mpi.h>
 int main(int argc, char **argv)
  { int size, rank, a, b;
    MPI_Status st;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0)
    { a = 10;
      MPI_Send(&a,1,MPI_INT,3,15,MPI_COMM_WORLD);
    }
   else
    { if(rank == 3)
       { MPI_Recv(&b,1,MPI_INT,0,15,MPI_COMM_WORLD,&st);
         printf("Vetv= %d b= %d\n",rank,b);
       }
    }
   MPI_Finalize();
   return(0);
 }
```

Завдання 1.2

Скомпілювати і запустити програму прикладу 2 на 4-х комп'ютерах.

ПРИКЛАД 1.3

Гілка з номером 0 пересилає дані (в даному випадку число 10) останньої гілки в безлічі запущених гілок. Остання гілка друкує на екран свій номер і прийняте число від нульової гілки.

```
#include<stdio.h> #include<mpi.h>
int main(int argc, char **argv)
{ int size, rank, a, b; MPI_Status st;
MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank); if(rank
     == 0)
```

```
     { a = 10;
     MPI_Send(&a,1,MPI_INT,size-
     1,15,MPI_COMM_WORLD);
     }
else
{ if(rank == size-1)
{ MPI_Recv(&b,1,MPI_INT,0,15,MPI_COMM_WORLD,&st);
     printf("Vetv= %d b= %d\n",rank,b);
     }
}
MPI_Finalize(); return(0);
}
```

Зверніть увагу на умову задачі і на те, що останній номер гілки в безлічі запущених гілок дорівнює `size-1`.

Це простий приклад паралельної програми, яка налаштовується на розмір обчислювальної системи, як на параметр. Тобто програму без переробок і без перекомпіляції можна запускати на будь-якій кількості комп'ютерів і вона буде видавати правильний результат. Основними параметрами в паралельній програмі є: 1) розмір обчислювальної системи – `size;` та 2) номер гілки – `rank.`

В даному випадку цих параметрів досить, що б програма була настроюється автоматично налагоджувальною на розмір системи. Бажано прагнути писати самоналагоджувальна програми.

Завдання 1.3

Скомпілювати і запустити програму прикладу 3 на 4-х і на 8-й комп'ютерах.

ПРИКЛАД 1.4

Гілка з номером 0 пересилає дані (в даному випадку число 10) всім гілкам в безлічі запущених гілок. Усі гілки друкують на екран свій номер і прийняте число від нульової гілки.

Перший варіант програми.

```
#include<stdio.h>
#include<mpi.h>
 int main(int argc, char **argv)
  { int size, rank, a;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0)
     {            a            =            10;
       MPI_Bcast(&a,1,MPI_INT,0,MPI_COMM_WORLD);
     }
     else
     { MPI_Bcast(&a,1,MPI_INT,0,MPI_COMM_WORLD);
         printf("Vetv= %d a= %d\n",rank,a);
     }
    MPI_Finalize();
    return(0);
  }
```

Другий варіант програми.

```
#include<stdio.h>
#include<mpi.h>
 int main(int argc, char **argv)
  { int size, rank, a;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0)
      a                =                10;
    MPI_Bcast(&a,1,MPI_INT,0,MPI_COMM_WORLD);

    printf("Vetv= %d a= %d\n",rank,a);

    MPI_Finalize();
    return(0);
  }
```

Завдання 1.4

Скомпілювати і запустити обидві програми прикладу 4 на 4-х і на 8-й комп'ютерах.

Завдання до лабораторної роботи № 1

Завдання 1.5. Програмування конвеєра.

Гілка 0 пересилає деякі дані до гілки 1, гілка 1 прийняті дані передає гілці 2, гілка 2 прийняті дані передає гілці 3 і т.д. по ланцюжку збільшення номерів. І, нарешті, гілка 0 приймає дані, що пересилаються, від гілки size-1 і виводить на екран свій номер і прийняті дані. Тобто пересилання інформації по кільцю комп'ютерів. У цьому прикладі потрібно написати алгоритм програму, що автоматично налаштовується на розмір обчислювальної системи, тобто алгоритм повинен бути незалежним від числа комп'ютерів і програма повинна запускатися на будь-якому допустимій кількості комп'ютерів.

Завдання 1.6. Програмування кільцевих зрушень даних

Усі гілки одночасно пересилають деякі свої дані гілкам з номерами на одиницю більшими, ніж гілки що передавали дані. Тобто пересилання інформації уздовж кільця комп'ютерів, див. рис.1.1. У цьому прикладі потрібно написати алгоритм програму, що автоматично налаштовується на розмір обчислювальної системи, тобто алгоритм повинен бути незалежним від числа комп'ютерів і програма повинна запускатися на будь-якому допустимій кількості комп'ютерів.

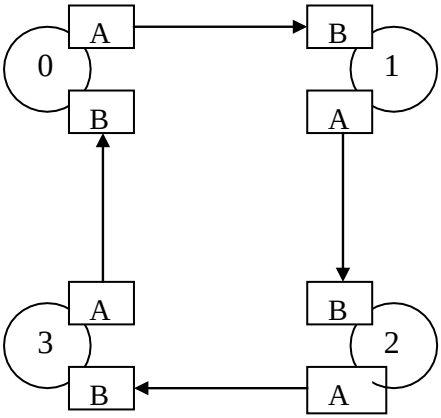


Рис. 1.1. Схема пересилання інформації уздовж кільця комп'ютерів

Контрольні питання до лабораторної роботи №1

1. Методи розпаралелювання і моделі програм, які підтримуються MPI.
2. Властивості MPI, що забезпечують переносимість паралельних програм і незалежність від обчислювальних систем.
3. Пояснити основні принципи виконання нижче зазначених функцій.

Парні взаємодії:

3.5 `MPI_Isend (...), MPI_Irecv (...), MPI_Wait (...), MPI_Test (...);`

3.6 `MPI_Send (...), MPI_Recv (...);`

3.7 `MPI_Ssend (...);`

3.8 `MPI_Sendrecv (...);`

Колективні взаємодії:

3.9 `MPI_Barrier (..);`

3.10 `MPI_Bcast (...);`

3.11 `MPI_Scatter (...), MPI_Scatterv (...);`

3.12 `MPI_Gather (...), MPI_Gathrv (...), MPI_Allgather (...);`

4. Команда компіляції MPI-програм.
5. Команда запуску MPI-програм.
6. Дії системи по команді mpirun.

Зміст звіту (Зразок оформлення звіту (ДодатокА))

1. Мета роботи.
2. Завдання до роботи.
3. Відповіді на контрольні питання.
4. Текст розробленого програмного забезпечення.
5. Результати тестування: вхідні дані та результати роботи програми.
6. Висновки, що відображають особисто отримані результати виконання роботи, їх критичний аналіз.

До звіту додаються файли проекту.

Лабораторна робота № 2

Завдання топологій. Приклади паралельних програм множення матриці на вектор на різних топологіях

Мета - практичне освоєння програмування паралельних процесів, що виконуються на декартовій топології зв'язків і топології "граф". Освоєння функцій завдання декартової топології і топології "граф". Освоєння методів розпаралелювання алгоритмів розв'язання задач, таких як множення матриці на вектор. Цей алгоритм є, у свою чергу, макрооперацією в ітераційних задачах. У лабораторній роботі наведено три приклади. Один простий приклад пов'язаний із взаємодіями паралельних процесів на декартових структурах, два інших приклади пов'язані з різними способами паралельного множення матриці на вектор на топологіях "кільце" і "повний граф".

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Під віртуальною топологією розуміється програмно реалізована топологія у вигляді конкретного графа, наприклад: кільце, решітка, тор, зірка, дерево і взагалі довільний граф на існуючій фізичній топології. Віртуальна топологія забезпечує дуже зручний механізм найменування процесів, пов'язаних комунікатором, і є потужним засобом відображення процесів обладнання системи. Віртуальна топологія в MPI може задаватися лише групі процесів, об'єднаних комунікатором.

Потрібно розрізняти віртуальну топологію процесів та топологію основного, фізичного обладнання. Механізм віртуальних топологій значно спрощує та полегшує написання паралельних програм, робить програми легшими для читання та зрозумілими. Користувачеві при цьому не потрібно програмувати схему фізичних зв'язків процесорів, а лише програмувати схему віртуальних зв'язків між процесами. Відображення віртуальних зв'язків на фізичні зв'язки здійснює система, що робить паралельні програми машинно-незалежними та легко переносимими. Функції у цьому розділі здійснюють

лише машинно-незалежне відображення. (Функції створення декартових топологій дивись Додаток В).

ПРИКЛАД 2.1. Для освоєння програмування Декартових топологій і освоєння суміщених функцій приймання-передавання даних.

У прикладі виконується зсув даних сусіднім гілкам уздовж обох координат комп'ютерів на топології двовимірний тор на один крок, тобто всі гілки паралельної програми одночасно передають дані сусіднім гілкам, наприклад, у бік збільшення значень уздовж однієї та уздовж іншої координати комп'ютерів.

```
#include <mpi.h>
#include <stdio.h>
#define DIMS 2
int main(int argc, char** argv)
  { int rank, size, i, A, B, dims[DIMS];
    int periods[DIMS], sourc1, sourc2, dest1, dest2;
    int reorder = 0;
    MPI_Comm tor_2D;
    Статус MPI_Status;
    MPI_Init(&argc, &argv);
```

/* Кожна гілка дізнається про загальну кількість гілок */

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

/* і свій номер: від 0 до `(size-1)` */

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    A = rank;
    B = -1;
```

/* Обнуляємо масив dims і заповнюємо масив `periods` для топології "двовимірний тор" */

```
  for(i = 0; i < DIMS; i++) { dims[i] = 0; periods[i]
= 1; }
```

/* Заповнюємо масив dims, у якому вказуються розміри решітки */

```
    MPI_Dims_create(size, DIMS, dims);
```

/* Створюємо топологію "двовимірний тор" з комунікатором top2D */

```
    MPI_Cart_create(MPI_COMM_WORLD,   DIMS,   dims,
periods, reorder, & tor_2D);
```

/* Кожна гілка знаходить своїх сусідів уздовж координат, у напрямку більших значень * координат */

```
    MPI_Cart_shift(tor_2D, 0, 1, &sourc1,
&dest1);
    MPI_Cart_shift(tor_2D, 1, 1, &sourc2, &dest2);
```

/* Кожна гілка одночасно передає свої дані (значення змінної A) сусідній гілці
*** з більшим значенням координати і приймає дані в B від сусідньої гілки з меншим значенням координати вздовж "кільця". */**

```
     MPI_Sendrecv(&A, 1, MPI_INT, dest1, 2, &B, 1,
MPI_INT, sourc1, 2, tor_2D, &status);
    printf("rank = %d B=%d\n", rank, B);
    MPI_Sendrecv(&A, 1, MPI_INT, dest2, 2, &B, 1,
MPI_INT, sourc2, 2,  tor_2D, &status);
```

/* Кожна гілка друкує свій номер (він же і був надісланий сусідній гілці) і значення змінної B (ранг сусідньої гілки) */

```
    printf("rank = %d B=%d\n", rank, B);
```

/* Усі гілки завершують системні процеси, пов'язані з топологією top_2D і завершують виконання програми */

```
    MPI_Comm_free(&tor_2D);
    MPI_Finalize();
```

```
return 0;}
```

У наведеному прикладі функцій завдання топології, знаходження сусідів і взаємодій див. у п. "Основні функції MPI" у Лаб.1. та Додаток В.

ПРИКЛАД 2.2. Множення матриці на вектор на топології "кільце".

Це завдання вирішується методом розпаралелювання за даними (або декомпозиції даних). Тобто вихідні дані розрізаються на частини за кількістю комп'ютерів і ці частини розподіляються по комп'ютерах. Тут у прикладі головним є сам алгоритм множення, а розподіл даних і генерація даних задаються безпосередньо в програмі.

Задано вихідну матрицю A і вектор B. Обчислюється добуток `C = A x B`, де A - матриця `[N x M]`, і B - вектор [M] за формулою:

$$c_i = \sum_{j=0}^{M} a_{ij} b_j \quad (i = 0,...,N)$$

Початкові матриця і вектор попередньо розрізані на смуги і кожна гілка генерує свої частини. Схема розподілу даних по комп'ютерах наведена нижче на рис. 2.1.

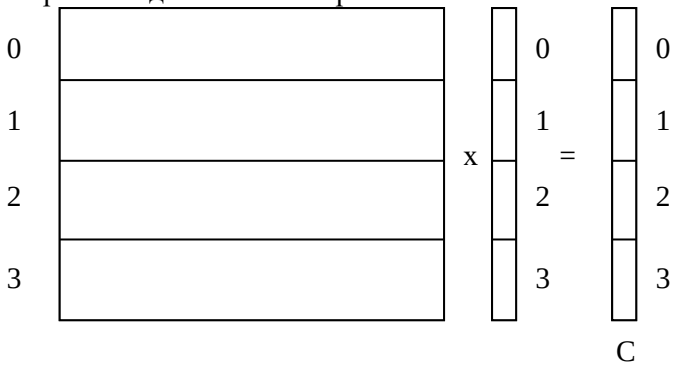


Рис. 2.1. Розрізання даних для паралельного алгоритму.

Реалізація алгоритму виконується на системі з P (на

малюнку 4-х) комп'ютерів. Матриця A, вектор B і вектор C розрізані на P (на малюнку 4-е) горизонтальних смуг. Тут припускається, що в пам'ять кожного комп'ютера завантажується і може перебувати тільки одна смуга матриці A і одна смуга вектора B. На малюнку 2.1 цифрами позначено номери комп'ютерів, у пам'ять яких завантажуються відповідні смуги матриці та шматки векторів.

Схема множення матриці на вектор

Оскільки кожен комп'ютер має тільки частини вектора, то кожен комп'ютер може множити тільки частини своїх рядків на ці шматочки векторів. Наприклад, A = [20 x 20], B = [20], і кількість комп'ютерів P = 4, то в 0-му комп'ютері в початковий момент перебувають 5 перших рядків матриці та частина вектора: 0÷4, у 1-му □ 5 наступних рядків матриці та частина вектора: 5÷9, у 2-му 5 наступних рядків матриці та частина вектора: 10÷14, і в 3-му 5 наступних рядків матриці та частина вектора: 15÷19. Щоб кожен рядок у кожному комп'ютері був помножений на весь вектор, необхідно "прокрутити" повз кожен комп'ютер усі шматочки вектора. Це можна зробити за 4 кроки (за кількістю комп'ютерів), послідовно після кожного множення передаючи частини векторів по кільцю.

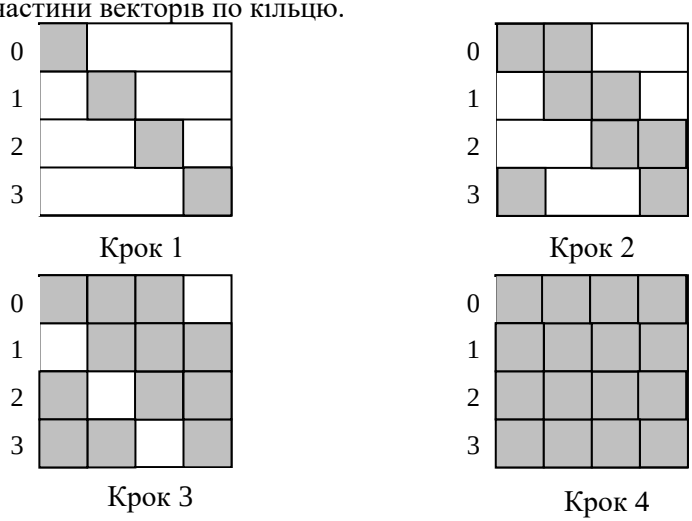


Рис 2.2. Схема множення матриці на вектор

На рис.2.2 заштрихована область позначає числа, що множаться на кожному кроці в кожному комп'ютері.

Нижче подано паралельну програму множення матриці на вектор на кільці.

```
/* У прикладі передбачається, що кількість рядків матриці A
і вектора B діляться без залишку
 * на кількість комп'ютерів у системі. Вихідна матриця A
розміром 20x20 ділиться без залишку на
 * кількість комп'ютерів, у цьому випадку на 4-е. Вектори B і
C теж діляться на 4-е без залишку.
 * У цьому разі завдання запускаємо на 4 комп'ютерах.
 */
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<time.h>
#include <sys/time.h>

/* Задаємо в кожній гілці розміри смуг матриць A і B:  5x20.
(Тут передбачається, що розміри
 * смуг однакові у всіх гілках. */

#define M 20
#define N 5

/* NUM_DIMS - розмір декартової топології.  "кільце" -
одновимірний тор. */

#define DIMS 1
#define EL(x) (sizeof(x) / sizeof(x[0][0]))

/* Задаємо смуги вихідних матриць.  */

static double A[N][M], B[N], C[N];
int main(int argc, char **argv)
  { int rank,size,i,j,k,i1,d,sour,dest;
    int dims[DIMS];
```

```
    int periods[DIMS];
    int new_coords[DIMS];
    int reorder = 0;
    Кільце     MPI_Comm;
    MPI_Status st;
    int rt, t1, t2;
```

/* Ініціалізація бібліотеки MPI*/

```
    MPI_Init(&argc, &argv);
```

/* Кожна гілка дізнається кількість завдань у додатку, що стартував */

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

/* Обнуляємо масив dims і заповнюємо масив periods для топології "кільце" */

```
    for(i=0; i < DIMS; i++) { dims[i] = 0; periods[i]
= 1; }
```

/* Заповнюємо масив dims, де вказуються розміри одновимірної решітки */

```
    MPI_Dims_create(size, DIMS, dims);
```

/* Створюємо топологію "кільце" з communicator(ом) comm_cart */

```
    MPI_Cart_create(MPI_COMM_WORLD,   DIMS,   dims,
periods, reorder,  &ring);
```

/* Кожна гілка визначає свій власний номер: від 0 до (size-1) */

```
    MPI_Comm_rank(ring, &rank);
```

/* Кожна гілка знаходить своїх сусідів уздовж кільця, у

напрямку менших значень рангів */

```
    MPI_Cart_shift(ring, 0, -1, &sour, &dest);
```

/* Кожна гілка генерує смуги вихідних матриць A і B, смуги C обнуляє */

```
    for(j = 0; j < N; j++)
     { for(i = 0; i < M; i++)
         A[j][i] = 3.0; B[j] = 2.0;
       C[j] = 0,0;
     }
```

/* Засікаємо початок множення матриць */

```
     t1 = MPI_Wtime();
```

/* Кожна гілка здійснює множення своїх смуг матриці та вектора */

/* Самий зовнішній цикл for(k) - цикл по комп'ютерах */

```
    for(k = 0; k < size; k++)
     { d = ((rank + k)%size)*N;
```

/* Кожна гілка здійснює множення своєї смуги матриці A на поточну смугу матриці B */

```
       for(j = 0; j < N; j++)
         { for(i1=0, i = d; i < d+N; i++, i1++)
            C[j] += A[j][i] * B[i1];
         }
```

/* Кожна гілка передає своїм сусіднім гілкам з меншим рангом смуги вектора B.

/* Тобто смуги вектора B зсуваються вздовж кільця комп'ютерів */

```
       MPI_Sendrecv_replace(B,  EL(V),  MPI_DOUBLE,
dest, 12, sour,
                                        12, ring, &st);
     }
```

/* Множення завершено. Кожна гілка помножила свою смугу рядків матриці A на
*** всі смуги вектора B. Засікаємо час і результат друкуємо */.**

```
    t2 = MPI_Wtime();
    rt = t2 - t1;
     printf("rank = %d Time = %d\n", rank, rt);
```

/* Для контролю друкуємо перші N елементів результату */

```
    for(i = 0; i < N; i++)
      printf("rank = %d RM = %6.2f\n "rank,C[i]);
```

/* Усі гілки завершують системні процеси, пов'язані з топологією comm_cart і
*** завершують виконання програми */**

```
    MPI_Comm_free(&ring);
    MPI_Finalize();
    return(0);
  }
```

Звернути увагу на спосіб "прокручування" даних по процесорах і на формулу початкового і кінцевого значення змінної циклу і (у самому внутрішньому циклі). А так само звернути увагу на функцію виміру часу рахунку: `MPI_Wtime()`.

ПРИКЛАД 2.3. Множення матриці на вектор на топології "повний граф".

Множення матриці на вектор у топології "повний граф". Задано вихідну матрицю A і вектор B. Обчислюється добуток `C = A x B`, де A - матриця `[NxM]`, і B - вектор [M]. Початкову матрицю попередньо розрізано на смуги, як на рис. 2.1, а вектор B дубльовано в кожній гілці. Кожна гілка генерує свої частини. Програма робить таке: множить матрицю на вектор і отримує розподілений по комп'ютерах результат - матрицю C; потім розрізані частини матриці C з'єднує в єдиний вектор, який записується у вектор B у всіх комп'ютерах. (Це модель однієї ітерації в ітераційних алгоритмах).

У програмі наведено приклад завдання топології "повний граф".

/* У прикладі передбачається, що кількість рядків матриці A і B діляться без залишку
*** на кількість комп'ютерів у системі.**
*** В даному випадку завдання запускаємо на 4 комп'ютерах. */**

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<time.h>
#include <sys/time.h>
```

/* Задаємо в кожній гілці розміри смуг матриць A і B. */

```
   #define M 20
   #define N 5
```

/* NUM_DIMS - розмір декартової топології. "кільце" - одновимірний тор. */

```
   #define DIMS 1
   #define EL(x) (sizeof(x) / sizeof(x[0][0]))
```

/* Задаємо смуги вихідних матриць. У кожній гілці, в даному випадку, вони однакові */

```
static double A[N][M], B[M], C[N];

int main(int argc, char** argv)
  {         int         rt,         t1,         t2,
rank,size,*index,*edges,i,j,reord=0;
    MPI_Comm comm_gr;
 /* Ініціалізація бібліотеки MPI*/
    MPI_Init(&argc, &argv);
```

/* Кожна гілка дізнається кількість завдань у додатку, що стартував */

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

/* Виділяємо пам'ять під масиви для опису вершин (index) і

```
    ребер (edges) у топології граф */.
    index = (int *)malloc(size * sizeof(int));
    edges = (int *)malloc(((size-1)+(size-1) * 3) *
sizeof(int));
```

/* Заповнюємо масиви для опису вершин і ребер для топології граф і задаємо топологію "граф". */

```
    index[0] = size - 1;
    for(i = 0; i < size-1; i++)
      edges[i] = i+1;
      v = 0;
    for(i = 1; i < size; i++)
      { index[i] = (size - 1) + i * 3;
        edges[(size - 1) + v++] = 0;
        edges[(size - 1) + v++] = ((i-2)+(size-
1))%(size-1)+1;
        edges[(size - 1) + v++] = i % (size-1) + 1;
       }

    MPI_Graph_create(MPI_COMM_WORLD, size, index,
edges, reord,

&comm_gr);
```

/* Кожна гілка визначає свій власний номер: від 0 до (size-1) */

```
    MPI_Comm_rank(comm_gr, &rank);
```

/* Кожна гілка генерує смуги вихідних матриць A і B, смуги C обнуляє */

```
    for(i = 0; i < M; j++)
      { for(j = 0; j < N; i++)
         { A[j][i] = 3.0;
           C[j] = 0,0;
```

```
          }
        B[i] = 2.0;
      }
```

/* Засікаємо початок множення матриць */

```
    t1 = MPI_Wtime();
```

/* Кожна гілка здійснює множення своїх смуг матриці та вектора */

```
    for(j = 0; j < N; j++)
     { for(i = 0; i < M; i++)
         C[j] += A[j][i] * B[i];
     }
```

/* З'єднуємо частини вектора С і записуємо їх у вектор В в усі комп'ютери */

```
    MPI_Allgather(C,    N,    MPI_DOUBLE,    B,    N,
MPI_DOUBLE, comm_gr);
```

/* Кожна гілка друкує час розв'язання і нульова гілка друкує вектор В */

```
    t2 = MPI_Wtime();
    rt = t2 - t1;
    printf("rank = %d Time = %d\n",rank,rt);
    if(rank == 0)
     { for(i = 0; i < M; i++)
         printf("B = %6.2f\n",B[i]);
     }
    MPI_Finalize();
    return(0);
 }
```

Звернути увагу на функцію завдання топології графів: `MPI_Graph_create`, і функцію виміру часу рахунку: `MPI_Wtime()`.

Завдання до лабораторної роботи № 2

Мета - дати уявлення про побудову простих паралельних програм на мові паралельного програмування MPI; уявлення про паралельні програми, що налаштовуються на розмір обчислювальної системи як на параметр; закріпити практичне освоєння функцій парних взаємодій між гілками паралельної програми.

Завдання 1. Опрацювання прикладів із пункту "Приклади паралельних програм" цієї ж лабораторної.

Уважно опрацювати приклади 2.1, 2.2, 2.3. Приклади 2.2 і 2.3 (Множення матриці на вектор на топологіях "кільце" і "повний граф") відкомпілювати і запустити на 4-х комп'ютерах.

Завдання 2. Паралельне множення матриці на вектор на топології "кільце".

Розробити алгоритм, написати й налагодити паралельну програму множення матриці на вектор у топології "кільце": A x B = C, за умови, що кількість рядків матриці та елементів вектора націло не ділиться на кількість комп'ютерів. Наприклад, матриця A розміром [22x22] і вектор B розміром [22] на чотирьох комп'ютерах.

Завдання 3. Паралельне множення матриці на матрицю на топології "кільце".

Варіант 1.

Розробити алгоритм, написати й налагодити паралельну програму множення матриці на матрицю в топології "кільце": A x B = C, за умови, що кількість рядків матриці A та стовпців матриці B націло ділиться на кількість комп'ютерів. Наприклад, матриця A розміром [20x20] і матриця B розміром [20x20] на чотирьох комп'ютерах.

Варіант 2.

Розробити алгоритм, написати й налагодити паралельну програму множення матриці на матрицю в топології "кільце": A x B = C, за умови, що кількість рядків матриці A та стовпців матриці B націло не ділиться на кількість комп'ютерів. Наприклад, матриця A розміром [22x20] і матриця B розміром [20x22] на чотирьох комп'ютерах.

Завдання 4. Порівняльні часові характеристики двох алгоритмів множення матриці на вектор на топологіях "кільце" і "повний граф".

Як паралельні програми взяти приклади 2.2 і 2.3 (Множення матриці на вектор на топологіях "кільце" і "повний граф"). Для обох програм побудувати невеликі графіки залежності часу розв'язання задачі від розмірів матриці та вектора. Обидві програми запустити послідовно для матриць: `A = [300 x 300], [500 x 500], [700 x 700], [1000 x 1000]` і відповідних цим матрицям векторів. Відкомпілювати і запустити на 4-х комп'ютерах, засікти час і побудувати зазначені графіки.

Контрольні запитання до лабораторної роботи №2

1. Для чого потрібні віртуальні топології?
2. Як задаються "декартові" топології в MPI?
3. Як задаються топології "граф" у MPI?
4. Який метод розпаралелювання використовується в паралельних алгоритмах множення матриці на вектор і матриці на матрицю з використанням MPI?
5. Як розподіляються елементи матриці та вектора при множенні матриці на вектор на топології "кільце"?
6. Як розподіляються елементи обох матриць при множенні матриці на матрицю на топології "кільце"?
7. Як краще уявити в пам'яті комп'ютера другу матрицю, при множенні матриці на матрицю, для прискорення часу розв'язання задачі?

Зміст звіту (Зразок оформлення звіту (ДодатокА))

1. Мета роботи.
2. Завдання до роботи.
3. Відповіді на контрольні питання.
4. Текст розробленого програмного забезпечення.
5. Результати тестування: вхідні дані та результати роботи програми.
6. Висновки, що відображають особисто отримані результати виконання роботи, їх критичний аналіз.

До звіту додаються файли проекту.

Лабораторна робота № 3
Прості приклади паралельних програм в OpenMP

Мета - дати уявлення про побудову простих паралельних програм на мові паралельного програмування OpenMP; практичне освоєння основних директив мови.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Інтерфейс OpenMP задуманий як стандарт для програмування на масштабованих SMP-системах (SSMP,ccNUMA, etc.) у моделі спільної пам'яті (shared memory model). В стандарт OpenMP входять специфікації набору директив компілятора, процедур і змінних середовища. Прикладами систем із загальною пам'яттю, масштабованих до великої кількості процесорів, можуть слугувати суперкомп'ютери Cray Origin2000 (до 128 процесорів), HP 9000 V-class (до 32 процесорів в одному вузлі, а в конфігурації з 4 вузлів - до 128 процесорів), Sun Starfire (до 64 процесорів).

Основне джерело інформації - сервер http://www.openmp.org/. На сервері доступні специфікації, статті, навчальні матеріали, посилання.

Інформацію про методи розпаралелювання і моделі програм, підтримувані OpenMP, а також базовий потоковий паралелізм, явний паралелізм та моделі вітвлення та об’єднання з прикладами представлено у Додатку Г(перед виконанням Лаб.р.3 опрацювати матеріали).

Базовий потоковий паралелізм *(більше див. Додаток Г)*

Загальнодоступний процес пам'яті може складатися з множинних потоків, кілька ниток управління, які мають загальний адресний простір, але різні потоки команд і роздільні стеки. У найпростішому випадку, процес складається з однієї нитки. Нитки іноді називають також потоками, легковагими процесами, `LWP (light-weight processes)`. OpenMP заснований на існуванні множинних потоків у загальнодоступній пам'яті, що програмує парадигму.

Явний паралелізм *(більше див. Додаток Г)*

OpenMP має явну (не автоматичну) модель програмування, пропонуючи програмістові повне управління щодо розпаралелювання.

Модель Fork-Join (Розгалуження - Об'єднання) *(більше див. Додаток Г)*

OpenMP використовує Fork-Join модель паралельного виконання (рис.3.1): Усі програми OpenMP починаються як єдиний процес: головний потік. Головний потік виконується послідовно, поки не стикаються з першою областю паралельної конструкції.

***Fork (ВІТВЛЕННЯ*)**: головний потік створює групу паралельних потоків.

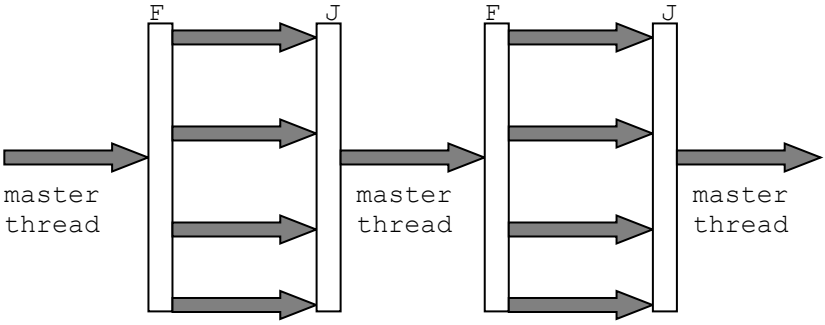


Рис. 3.1. модель Fork-Join

Інструкції в програмі, які включені паралельною конструкцією області{*регіону*}, тоді виконані паралельно серед різних потоків групи

***Join (ОБ'ЄДНАННЯ*):** Коли потоки групи завершують інструкції в області паралельної конструкції, вони синхронізуються і закриваються, залишаючи тільки головний потік.

ПРИКЛАД 3.1

Кожен процес (гілка п-програми) виводить на екран свій ідентифікаційний номер і кількість замовлених паралельних процесів.

```
#include<omp.h>
#include<stdio.h>

main ()
{
int size, rank;
```

/* Створення безлічі паралельних процесів і в кожному з них задаються
*** свої приватні змінні size і rank */.**

```
#pragma omp parallel private(size, rank)
  {
```

/* Кожен процес знаходить свій порядковий номер і виводить його на екран */

```
  rank = omp_get_thread_num();
  printf("Hello World from thread = %d\n", rank);
```

/* Головний процес - master виводить на екран кількість процесів */

```
  if (rank == 0)
    {
    size = omp_get_num_threads();
    printf("Number of threads = %d\n", size);
    }
```

```
 } /* Завершення паралельної частини */
}
```

ПРИКЛАД 3.2

Паралельне підсумовування елементів двох векторів. Підсумовування здійснюється в циклі. У програмі застосовується комбінація паралельного циклу та редукованої операції за всіма процесами.

```
#include <omp.h>
#include<stdio.h>

main ()
{
int i, n;
float a[100], b[100], sum;
```

/* Ініціалізація елементів векторів */

```
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;
```

/* Створення безлічі паралельних процесів і розпаралелювання
*** циклу по витках. При виході з циклу всі значення змінної sum**
*** підсумовуються за всіма процесами. */**

```
 #pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
```

/* Головний процес виводить на екран значення sum */

```
printf("Sum = %f\n",sum);
}
```

ПРИКЛАД 3.3

Приклад аналогічний попередньому: Паралельне підсумовування елементів двох векторів. Але підсумовування здійснюється в циклі в окремій підпрограмі. У програмі застосовується комбінація паралельного циклу і редукованої операції за всіма процесами.

```
#include <omp.h>
#include<stdio.h>

#define VECLEN 100
float a[VECLEN], b[VECLEN], sum;
```

/* Підпрограма, в якій підсумовуються елементи векторів */

```
float dotprod ()
{
int i,rank;

rank = omp_get_thread_num();
#pragma omp for reduction(+:sum)
  for (i=0; i < VECLEN; i++)
    {
    sum = sum + (a[i]*b[i]);
    printf("rank = %d i=%d\n",rank,i);
    }
return(sum);
}

main ()
{
int i;
```

/* Ініціалізація елементів векторів */

```
for (i=0; i < VECLEN; i++)
  a[i] = b[i] = 1.0 * i;
sum = 0.0;
```

/* Створення безлічі паралельних процесів */

```
#pragma omp parallel
  sum = dotprod();

printf("Sum = %f\n",sum);

}
```

ПРИКЛАД 3.4

Приклад аналогічний прикладу 3.2: Паралельне підсумовування елементів двох векторів. Але підсумовування здійснюється в окремих секціях.

```
#include <omp.h>
#include<stdio.h>

#define N 50

main ()
{
int i, size, rank;
float a[N], b[N], c[N];
```

/* Ініціалізація елементів векторів */

```
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
```

/* Створення безлічі паралельних процесів */

```
#pragma omp parallel shared(a,b,c)
private(i,rank,size)
  {
```

/* Кожен процес знаходить свій порядковий номер і виводить його на екран */

```
 rank = omp_get_thread_num();
  printf("Thread %d starting...\n",rank);
```

/* Директива завдання секцій */

```
  #pragma omp sections nowait
    {
```

```
/* Секція 0*/
    #pragma omp section
      for (i=0; i < N/2; i++)
        {
        c[i] = a[i] + b[i];
        printf("rank  =  %d  i=  %d  c[i]=  %f\n",
rank,i,c[i]);
        }
/* Секція 1*/
    #pragma omp section
      for (i=N/2; i < N; i++)
        {
        c[i] = a[i] + b[i];
        printf("rank  =  %d  i=  %d  c[i]=  %f\n",
rank,i,c[i]);
        }

    } /* Завершення блоку секцій */

  if (rank == 0)
    {
    size = omp_get_num_threads();
    printf("Number of threads = %d\n", size);
    }
  } /* Завершення паралельної частини */
}
```

ПРИКЛАД 3.5

Приклад паралельного множення матриці на вектор.

```
#include <omp.h>
#include<stdio.h>

#define M 10

main ()
{
float A[M][M], b[M], c[M];
int i, j, rank;
/* Ініціалізація даних */
for (i=0; i < M; i++)
```

```
  {
  for (j=0; j < M; j++)
    A[i][j] = (j+1) * 1.0;
  b[i] = 1.0 * (i+1);
  c[i] = 0.0;
  }
printf("\Виведення значень матриці A і вектора b на
екран:\n");
for (i=0; i < M; i++)
  {
  printf(" A[%d]= ",i);
  for (j=0; j < M; j++)
    printf("%.1f ",A[i][j]);
  printf(" b[%d]= %.1f\n",i,b[i]);
  }
```

/* Створення безлічі паралельних процесів і в кожному з них задаються
*** свої приватні змінні rank та i*/.**

```
#pragma omp parallel shared(A,b,c,total)
private(rank,i)
  {
  rank = omp_get_thread_num();
```

/* Директива розпаралелювання циклу за витками */

```
#pragma omp for private(j)
  for (i=0; i < M; i++)
    {
    for (j=0; j < M; j++)
      c[i] += (A[i][j] * b[j]);
```

/* Кожен процес виводить свій порядковий номер, значення витка циклу і
*** значення результуючого вектора на кожному витку циклу і всередині**
*** критичної секції */**

```
  #pragma omp critical
      {
      printf("rank=   %d   i=   %d   c[%d]=%.2f\n",
rank,i,c[i]);
      }
    } /* Кінець паралельного циклу */
  } /* Завершення паралельної конструкції */
}
```

ЗАВДАННЯ

Ретельно вивчити програми прикладів. Скомпілювати та запустити всі програми прикладів.

Завдання до лабораторної роботи № 3

Мета - дати уявлення про побудову простих паралельних програм мовою паралельного програмування OpenMP; закріпити практичне засвоєння директив мови.

Завдання 1. Опрацювання прикладів із пункту "Приклади паралельних програм" цієї ж лабораторної.

Уважно опрацювати приклади 3.1-3.5. Відкомпілювати і запустити на 2-х процесах.

Завдання 2. Паралельне множення матриці на вектор

Варіант 1.
Розробити алгоритм, написати й налагодити паралельну програму множення матриці на вектор з використанням розподілу робіт для паралельних процесів директивою sections. Використовувати алгоритм прикладу 3.5.

Варіант 2.
Розробити алгоритм, написати й налагодити паралельну програму множення матриці на вектор з використанням розподілу робіт для паралельних процесів із безпосереднім завданням роботи ("вручну"). Використовувати алгоритм прикладу 3.5.

Завдання 3. Паралельне множення матриці на матрицю.

Варіант 1.
Розробити алгоритм, написати й налагодити паралельну програму множення матриці на матрицю з використанням

директиви розпаралелювання циклу по витках.

Варіант 2.
Розробити алгоритм, написати та налагодити паралельну програму множення матриці на матрицю з використанням розподілу робіт для паралельних процесів із безпосереднім завданням роботи ("вручну").

Завдання 4. Порівняльні часові характеристики двох алгоритмів множення матриці на вектор мовою MPI і мовою OpenMP.

Як паралельні програми взяти паралельні програми: "множення матриці на вектор в MPI на топології "кільце"" і "множення матриці на вектор в OpenMP (один із варіантів)". Для обох програм побудувати невеликі графіки залежності часу розв'язання задачі від розмірів матриці та вектора. Обидві програми запустити послідовно для матриць: `A = [300 x 300], [500 x 500], [700 x 700], [1000 x 1000]` і відповідних цим матрицям векторів. Відкомпілювати і запустити на 2-х комп'ютерах, засікти час і побудувати зазначені графіки.

Контрольні запитання до лабораторної роботи №3

1. Методи розпаралелювання і моделі програм, підтримувані OpenMP.
2. Пояснити основні принципи виконання нижче зазначених директив.

Головна директива завдання ниток:

```
#pragma omp parallel [clause clause ...]
{
... ...
   }
clause: if(умова)
private(list)
shared(list)
firstprivate(list)
copyin(list)
```

```
reduction(operator:list)
```

Директиви поділу робіт за нитками

```
#pragma omp for [clause clause ...]
{
... . .
   }
clause: schedule(type[,chink])
ordered
private(list)
shared(list)
firstprivate(list)
lastprivate(list)
reduction(operator:list)
nowait
```

Паралельні секції

```
#pragma omp sections [clause clause ...]
{ . . . . . .
.
#pragma omp section
{ . . . . . .
.
}
#pragma omp section
{ . . . . . .
.
} } } // - кінцеві секції
clause: private(list)
firstprivate(list)
lastprivate(list)
reduction(operator:list)
nowait
```

Виконання однією ниткою

```
#pragma omp single [clause clause ...]
{
... ...
   }
clause: private(list)
firstprivate(list)
nowait
```

Явне управління розподілом роботи
Директиви синхронізації

```
#pragma omp master
{
. . . .
   }
#pragma omp critical[name]
{
. . . .
   }
#pragma omp barrier
#pragma omp atomic
```

3. Як задаються паралельні процеси під час множення матриці на вектор і матриці на матрицю в OpenMP?

Зміст звіту (Зразок оформлення звіту (ДодатокА))

1. Мета роботи.
2. Завдання до роботи.
3. Відповіді на контрольні питання.
4. Текст розробленого програмного забезпечення.
5. Результати тестування: вхідні дані та результати роботи програми.
6. Висновки, що відображають особисто отримані результати виконання роботи, їх критичний аналіз.

До звіту додаються файли проекту.

Лабораторна робота № 4
Розв'язання систем лінійних алгебраїчних рівнянь ітераційними методами

Мета - практичне освоєння методів розв'язування систем лінійних алгебраїчних рівнянь ітераційними методами. У лабораторній роботі наведено два приклади. Один - розв'язування СЛАР методом простої ітерації; другий - розв'язування СЛАР методом спряжених градієнтів. У цьому розділі наведено формули, за якими здійснюються обчислення в обох методах і

послідовні програми розв'язування СЛАР цими методами.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Розв'язання СЛАР методом простої ітерації. Наведено формули і послідовні програми цього методу.

Дано систему лінійних алгебраїчних рівнянь:

$$\begin{aligned} &a_{00}x_0 + a_{01}x_1 + \ldots + a_{0n}x_n = f_0 \\ &a_{10}x_0 + a_{11}x_1 + \ldots + a_{1n}x_n = f_1 \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &a_{n0}x_0 + a_{n1}x_1 + \ldots + a_{nn}x_n = f_n \end{aligned} \qquad (1)$$

Короткий запис:

$$Ax = f \qquad (2)$$

Коріння цієї системи методом простої ітерації обчислюються за такою формулою:

$$x_i^{k+1} = x_i^k - \tau * (\sum_{j=0}^{N} a_{ij}x_j^k - f_i) \ldots . i = 0,\ldots,N \qquad (3)$$

Тут верхній індекс позначає номер ітераційного кроку, і - номер кореня або правої частини. τ - ітераційний крок.
Завершення обчислень за умовою:

$$\frac{\left\| \sum_{j=0}^{N} a_{ij}x_{ij}^k - f_i \right\|}{\|f\|} < \varepsilon \qquad (4)$$

Зауваження.

$$\|f\| = \sqrt{\sum_{i=0}^{N} (f_i * f_i)} \qquad (5)$$

Чисельник виразу (4) обчислюється за аналогічною формулою.

ПРИКЛАД 4.1.

Нижче наведено програму для послідовного алгоритму:

```
#include<stdio.h>
#include<sys/time.h>

#define N 100

double A[N][N], F[N], mf, X[N], X1[N], S[N], msf;

main()
 { int i, j;
   long int dt;
   double t, e;
   struct timeval tv1, tv2;
```

/* Генерація даних. Тут задається матриця з елементами, що дорівнюють 1,
*** по діагоналі рівними 2. Матриця береться добре обумовленою.**
*** За правої частини, що дорівнює N+1, усі корені дорівнюватимуть 1. */**

```
   mf = 0;
   for(i = 0; i < N; i++)
    { for(j = 0; j < N; j++)
        (i==j)?(A[i][j]=2):(A[i][j]=1);
      F[i] = N+1;
```

/* Відразу обчислюємо суму квадратів елементів вектора F,
*** тобто підкореневий вираз формули (4). */**

```
      mf += F[i]*F[i];
    }

/* Задаємо крок t і e. */
   t = 0.01;
   e = 0.00001;
```

/* Задаємо початкове наближення коренів. У X1 зберігаються значення коренів

```
 * к+1-й ітерації. */
   for(i = 0; i < N; i++)
    X1[i] = 0,6;
```

/* Засікаємо час початку обчислень. */

```
    gettimeofday(&tv1,NULL);

    робити
      { for(i = 0; i < N; i++)
```

/* У X зберігаються значення коренів к-й ітерації. */

```
         X[i] = X1[i];

        for(msf=0,i = 0; i < N; i++)
         { for(S[i] = 0,j = 0; j < N; j++)
             S[i] += A[i][j] * X[j];
     X1[i] = X[i] - t*(S[i] - F[i]);
```

/* Обчислюємо суму квадратів елементів нев'язки, тобто підкореневе

*** вираз чисельника формули (4). */**

```
           msf += (S[i] - F[i])*(S[i] - F[i]);
         }
      }
    while(msf/mf  >  e*e);  /*  Перевірка  умови  за
формулою (3). */
```

/* Засікаємо час кінця обчислень. */

```
    gettimeofday(&tv2,NULL);
    dt  =  (tv2.tv_sec  -  tv1.tv_sec)  *  1000000  +
tv2.tv_usec - tv1.tv_usec;
```

/* Виводимо на екран час обчислень у секундах */

```
    printf("Time= %d\n",dt);
```

/* Для контролю виводимо 4-е перших кореня */

```
    for(i = 0; i < 4; i++)
     printf(" %f\n",X[i]);
 return(0);
  }
```

Зверніть увагу на функцію виміру часу `gettimeofday(&tv,NULL)`. Цю функцію можна використовувати і для паралельних програм.

ПРИКЛАД 2.2. Розв'язання СЛАР методом спряжених градієнтів. Наведено формули і послідовні програми цього методу.
Дано систему лінійних алгебраїчних рівнянь:

$$Ax = f \tag{6}$$

Цей метод призначений для симетричних матриць: $a_{ij} = a_{ji}$
Обирається початкове наближення x_0 . $r_0 = f - Ax_0$; $z_0 = r0$;
$\alpha_k = (r_{k-1}, r_{k-1}) / (Az_{k-1}, z_{k-1})$ - коефіцієнт (тут к - це номер ітераційного кроку);
$x_k = x_{k-1} + \alpha_k z_{k-1}$ - вектор рішень на к-й поточній ітерації;
$r_k = r_{k-1} - \alpha_k Az_{k-1}$ – вектор нев'язки на к-й поточній ітерації;
$\beta_k = (r_k, r_k) / (r_{k-1}, r_{k-1})$ - коефіцієнт;
$z_k = r_k + \beta_k z_{k-1}$ – вектор спуску на к-й поточній ітерації (спряжний напрямок).

Вихід з ітераційного процесу:

$$\frac{\|r_k\|}{\|f\|} < \varepsilon \tag{7}$$

Нижче наведено програму для послідовного алгоритму.
Зверніть увагу на те як обчислюються скалярні добутки векторів.

```
#include<stdio.h>
#include<time.h>
#include<sys/time.h>

#define M 100

#define E 0.00001
#define T 0.01

double A[M][M], F[M], Xk[M], Zk[M];
double Rk[M], Sz[M], alf, bet, mf;
double Spr, Spr1, Spz;
```

```
int main()
  { int i, j, v;
    struct timeval tv1, tv2;
    long int dt1;
```

/* Генерація даних. Тут задається матриця з елементами, що дорівнюють 1,
*** по діагоналі рівними 2. Матриця береться добре обумовленою і**
*** симетричною. При правій частині, що дорівнює M+1, всі корені будуть рівними 1 */.**

```
    for(mf=0,i = 0; i < M; i++)
      { for(j = 0; j < M; j++)
          { if(i == j)
              A[i][j] = 2.0;
            іще
              A[i][j] = 1.0;
          }
        F[i] = M + 1;
```

/* Відразу обчислюємо суму квадратів елементів вектора F,
*** тобто підкореневий вираз формули (7). */**

```
      mf += F[i] * F[i];
      }
```

/* Задаємо початкове наближення коренів. У Xk зберігаються значення коренів

```
 * к-й ітерації. */
    for(i = 0; i < M; i++)
      Xk[i] = 0,2;
```

/* Задаємо початкове значення r0 і z0 . */

```
    for(i = 0; i < M; i++)
      { for(Sz[i]=0,j = 0; j < M; j++)
          Sz[i] += A[i][j] * Xk[j];
      Rk[i] = F[i] - Sz[i];
      Zk[i] = Rk[i];
      }
```

/* Засікаємо час початку обчислень. */

```
    gettimeofday(&tv1, NULL);

    do
      {
```

/* Обчислюємо чисельник і знаменник для коефіцієнта

```
   * αk = (rk-1 ,rk-1 )/(Azk-1 ,zk-1 ) */.

        Spz = 0;
        Spr = 0;
        for(i = 0; i < M; i++)
          { for(Sz[i]=0, j = 0; j < M; j++)
             Sz[i] += A[i][j] * Zk[j];
          Spz += Sz[i] * Zk[i];
          Spr += Rk[i] * Rk[i];
          }
        alf = Spr/Spz; /* αk */
```

/* Обчислюємо вектор рішення: xk = xk-1 + α zkk-1,
вектор нев'язки: rk = rk-1 - αk Azk-1 і чисельник для βk , що дорівнює (rk ,rk) */

```
        Spr1 = 0;
        for(i = 0; i < M; i++)
          { Xk[i] += alf*Zk[i];
            Rk[i] -= alf*Sz[i];
            Spr1 += Rk[i]*Rk[i];
          }
```

/* Обчислюємо βk */

```
        bet = Spr1/Spr;
```

/* Обчислюємо вектор спуску: zk = rk + β zkk-1 */

```
        for(i = 0; i < M; i++)
          Zk[i] = Rk[i] + bet*Zk[i];
      }
```

/* Перевіряємо умову виходу з ітераційного циклу */

```
    while(Spr1/mf > E*E);
```

/* Засікаємо час кінця обчислень. Час розв'язання задачі виводимо на екран */

```
  gettimeofday(&tv2, (struct timezone*)0);
```

```
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
tv2.tv_usec - tv1.tv_usec;
    printf("Time = %d\n",dt1);

/* Для контролю виводимо 8 перших коренів */
printf("   %f   %f   %f   %f   %f   %f   %f   %f
%f\n",Xk[0],Xk[1],Xk[2],Xk[3],Xk[4],Xk[5],
 Xk[6], Xk[7]);

    return(0);
  }
```

ЗАВДАННЯ

Ретельно вивчити програми прикладів. Скомпілювати та запустити всі програми прикладів на одному комп'ютері.

Завдання до лабораторної роботи № 4

Мета - дати уявлення про розпаралелювання алгоритмів розв'язування систем лінійних алгебраїчних рівнянь ітераційними методами на мовах MPI та OpenMP;

Завдання 1. Опрацювання прикладів із пункту "Приклади паралельних програм" цієї ж лабораторної.

Уважно вивчити приклади 4.1-4.2. Відкомпілювати і запустити на 1-му процесорі.

Завдання 2. Паралельний алгоритм розв'язування СЛАР методом простої ітерації в MPI

Розробити паралельний алгоритм, написати та налагодити паралельну програму розв'язання СЛАР методом простої ітерації в MPI. Використовувати програму для послідовного алгоритму в прикладі 4.1.

Завдання 3. Паралельний алгоритм розв'язування СЛАР методом спряжених градієнтів у MPI

Розробити паралельний алгоритм, написати й налагодити паралельну програму розв'язування СЛАР методом сполучених

градієнтів у MPI. Використовувати програму для послідовного алгоритму в прикладі 4.2.

Завдання 4. Паралельний алгоритм розв'язання СЛАР методом простої ітерації в OpenMP

Розробити паралельний алгоритм, написати та налагодити паралельну програму розв'язання СЛАР методом простої ітерації в OpenMP. Використовувати програму для послідовного алгоритму в прикладі 4.1.

Завдання 5. Паралельний алгоритм розв'язування СЛАР методом спряжених градієнтів у OpenMP.

Розробити паралельний алгоритм, написати й налагодити паралельну програму розв'язування СЛАР методом спряжених градієнтів в OpenMP. Використовувати програму для послідовного алгоритму в прикладі 4.2.

Завдання 6. Порівняльні часові характеристики двох алгоритмів розв'язання СЛАР методом простої ітерації в мові MPI і мові OpenMP.

Як паралельні програми взяти паралельні програми завдань 2 і 4. Для обох програм побудувати невеликі графіки залежності часу розв'язання задачі від розмірів матриці коефіцієнтів. Обидві програми запустити послідовно для матриць: `A = [300 x 300], [500 x 500], [700 x 700], [1000 x 1000]`. Відкомпілювати і запустити на 2-х комп'ютерах, засікти час і побудувати зазначені графіки.

Контрольні запитання до лабораторної роботи № 4

1. Який метод розпаралелювання використовується в паралельних алгоритмах розв'язування СЛАР методами простої ітерації та спряжених градієнтів у MPI?
2. Як розподіляються елементи матриці коефіцієнтів і вектора розв'язків під час розв'язання СЛАР методом простої ітерації в MPI?

3. Як розподіляються елементи матриці коефіцієнтів і вектора розв'язків під час розв'язування СЛАР методом спряжених градієнтів у MPI?
4. Як розпаралелюються алгоритми розв'язування СЛАР методами простої ітерації та спряжених градієнтів в OpenMP?

Лабораторна робота № 5
Розв'язання систем лінійних алгебраїчних рівнянь методом Гауса

Мета - практичне освоєння методів розв'язання систем лінійних алгебраїчних рівнянь (СЛАР) прямими методами. У лабораторній роботі дано пояснення двох способів паралельного рішення СЛАР методом Гаусса, пов'язаних із різними способами розподілу даних по комп'ютерах. У цьому розділі наведено послідовну програму рішення СЛАР методом Гауса.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Потрібно знайти рішення системи лінійних рівнянь алгебри:

$$
\begin{array}{l}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = f_1 \\
a_{21}x_2 + a_{22}x_2 + \ldots + a_{2n}x_n = f_2 \\
\ldots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = f_n
\end{array}
$$

Метод Гауса заснований на послідовному виключенні невідомих. Тут розглядаються два алгоритми розв'язання СЛАР методом Гаусса. Вони пов'язані з різними способами представлення *даних* (матриць коефіцієнтів та правих частин) у розподіленій пам'яті мультикомп'ютера. Схеми розподілу даних по комп'ютерах для обох прикладів наведено нижче. Хоча дані розподілені у пам'яті мультикомп'ютера у кожному алгоритмі по-різному, але обидва вони реалізуються однією й тієї ж топології зв'язку комп'ютерів - " повний граф ".

ПРИКЛАД 5.1. **Рішення СЛАР методом Гауса. Алгоритм перший**

В алгоритмі, представленому в цьому пункті, вихідна матриця коефіцієнтів `A` та вектор правих частин `F` розрізані горизонтальними смугами, як показано на рис. 5.1. Кожна смуга завантажується у відповідний комп'ютер: нульова смуга – у нульовий комп'ютер, перша смуга – у перший комп'ютер, тощо, остання смуга – в `p1-1` комп'ютер. Тут, наприклад, кожна гілка генерує свої частини матриці коефіцієнтів `A` і вектор правих частин `F`.

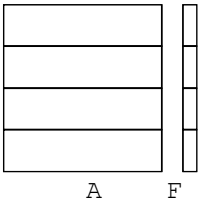


Рис. 5.1. Розрізання *даних* для паралельного алгоритму 1 рішення СЛАР методом Гауса

При прямому ході матриця наводиться до трикутного вигляду послідовно комп'ютерами. Спочатку до трикутного вигляду наводяться рядки в нульовому комп'ютері, причому нульовий комп'ютер послідовно, рядок за рядком, передає свої терміни іншим комп'ютерам, починаючи з першого. Потім до трикутного вигляду наводяться рядки у першому комп'ютері, передаючи рядки іншим комп'ютерам, починаючи з другого, тобто. комп'ютерам з великими номерами, і т. д. Процес поділу рядків на коефіцієнти при x_i не вимагає інформації з інших комп'ютерів.

Після прямого ходу смуги матриці `A` у кожному вузлі матимуть вигляд (рис. 5.2)

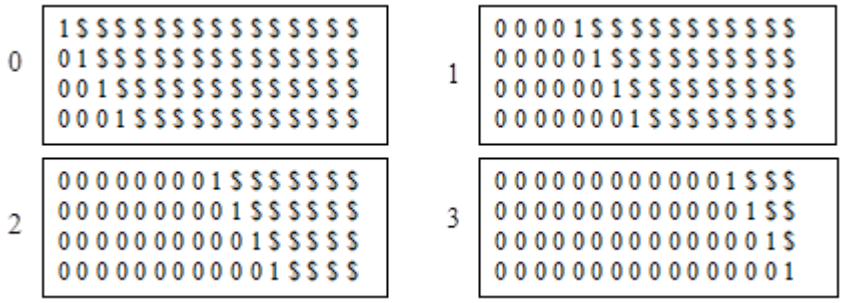


Рис. 5.2. Вид смуг після прямого ходу в алгоритмі 1 рішення СЛАР методом Гауса. Приклад наведено для чотирьох вузлів; $ - речові числа

Аналогічно, послідовно по комп'ютерах, починаючи з останнього за номером комп'ютера здійснюється зворотний хід.

Особливістю цього алгоритму є те, що як при прямому, так і при зворотному ході комп'ютери, які завершили свою частину роботи, переходять у стан очікування, поки не завершать цю роботу інші комп'ютери. Таким чином, обчислювальне навантаження розподіляється по комп'ютерах нерівномірно, незважаючи на те, що *дані* спочатку розподіляються по комп'ютерах приблизно однаково. Простої комп'ютерів значно зменшуються при розподілі матриці циклічними горизонтальними смугами. Цей метод подано у п. 3.4.2.

ПРИКЛАД 5.2. **Рішення СЛАР методом Гауса. Алгоритм другий**

В алгоритмі, представленому тут, вихідна матриця коефіцієнтів розподіляється по комп'ютерах циклічними горизонтальними смугами з шириною смуги в один рядок, як показано на рис. 5.3.

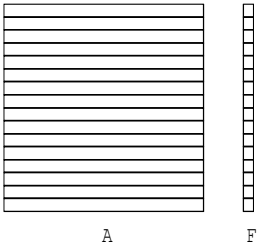


Рис. 5.3. Розрізання *даних* для паралельного алгоритму 2 рішення СЛАР методом Гауса

Нульовий рядок матриці поміщається в комп'ютер 0, перший рядок - в комп'ютер 1, і т. д., (p_1-1) - рядок в комп'ютер p_1-1 (де p_1 кількість комп'ютерів в системі). Потім, $p1-й$ рядок, знову поміщається в комп'ютер 0, $p1+1-й$ рядок - в комп'ютер 1, і т.д.

При такому розподілі даних, що відповідає цьому розподілу, повинен бути і алгоритм. Рядок, який віднімається від решти рядків (після попереднього поділу на потрібні коефіцієнти), назвемо поточним рядком. Алгоритм прямого ходу ось у чому. Спочатку поточним рядком є рядок з індексом 0 у комп'ютері 0, потім рядок з індексом 0 у комп'ютері 1 (тут не потрібно плутати загальну нумерацію рядків у всій матриці та індексацію рядків у кожному комп'ютері; у кожному комп'ютері індексація рядків у масиві починається з нуля) та і т.д., і нарешті, рядок з індексом 0 в останньому за номером комп'ютера. Після чого цикл по комп'ютерах повторюється і поточним рядком стає рядок з індексом 1 в комп'ютері 0, потім рядок з індексом 1 в комп'ютері 1 і т. д. Після прямого ходу смуги матриці в кожному комп'ютері будуть виглядати на рис. 5.4. Приклад наведено для чотирьох вузлів; $ - Речові числа.

Аналогічно, послідовно по вузлах, починаючи з останнього за номером комп'ютера здійснюється зворотний хід.

Особливістю цього алгоритму є те, що як при прямому, так і при зворотному ході комп'ютери є рівномірно завантаженими, ніж у першому методі. Отже, і обчислювальна навантаження розподіляється по комп'ютерам рівномірніше, ніж у першому методі. Наприклад, нульовий комп'ютер, завершивши обробку своїх рядків при прямому ході, чекає, поки інші комп'ютери оброблять тільки по одному, що залишився у них не обробленому рядку, а не повністю оброблять смуги, як у першому алгоритмі.

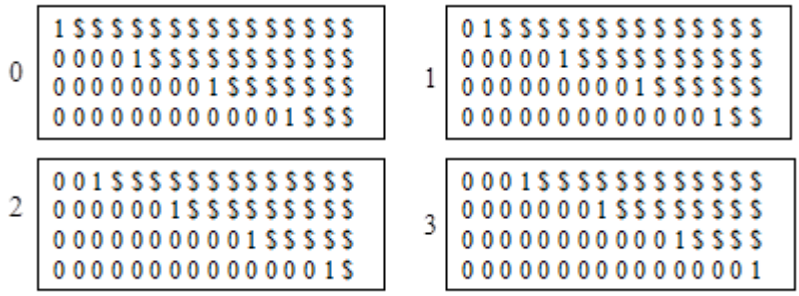


Рис. 5.4. Вид смуг після прямого ходу в алгоритмі 2 рішення СЛАР методом Гауса

Нижче наведено програму рішення СЛАР методом Гауса для послідовного алгоритму:

```
#include<stdio.h>
#include<sys/time.h>

#define M 100

double MA[M][M+1], MAD;

int main()
  { int i, j, v, k;
```

```
/* Змінні для заміру часу рішення */
    struct timeval tv1, tv2;
    long int dt1;

/* Генерація даних */
    for(i = 0; i < M; i++)
      { for(j = 0; j < M; j++)
          {  if(i == j)
              MA[i][j] = 2.0;
            else
              MA[i][j] = 1.0;
          }
        MA[i][M] = 1.0*(M)+1.0;
      }

    gettimeofday(&tv1,NULL);

/* Прямий хід */
        for(k = 0; k < M; k++)
          { MAD = 1.0/MA[k][k];
            for(j = M; j >= k; j--)
              MA[k][j] *= MAD;
            for(i = k+1; i < M; i++)
              for(j = M; j >= k; j--)
               MA[i][j] -= MA[i][k]*MA[k][j];

          }

/* Зворотний хід */
        for(k = M-1; k >= 0; k--)
          { for(i = k-1; i >= 0; i--)
              MA[i][M] -= MA[k][M]*MA[i][k];
          }

/* Засікання часу та друк */
    gettimeofday(&tv2,NULL);
    dt1 = (tv2.tv_sec - tv1.tv_sec)*1000000 + tv2.tv_usec - tv1.tv_use
    printf("Time = %ld\n",dt1);

/* Друк перших восьми коренів */
    printf(" %f %f %f %f\n",MA[0][M],MA[1][M],MA[2][M],MA[3][M]);
    printf(" %f %f %f %f\n",MA[4][M],MA[5][M],MA[6][M],MA[7][M]);

    return(0);
  }
```

Зверніть увагу на функцію вимірювання часу `gettimeofday(&tv,NULL)`. Цю функцію можна використовувати і для паралельних програм.

ЗАВДАННЯ

Ретельно вивчити програми прикладів. Скомпілювати та запустити всі програми прикладів на одному комп'ютері.

Завдання до лабораторної роботи №5

Мета – дати уявлення про розпаралелювання алгоритмів розв'язання систем лінійних рівнянь алгебри прямими методами в MPI.

Завдання 1. Опрацювання прикладів з пункту "Приклади паралельних програм" цієї лабораторної.

Уважно вивчити приклади 5.1-5.2. Відкомпілювати та запустити на 1-му процесорі.

Завдання 2. Паралельний алгоритм № 1 рішення СЛАР методом Гауса в MPI

Розробити паралельний алгоритм, написати та налагодити паралельну програму рішення СЛАР методом Гауса в MPI. У цьому алгоритмі передбачається, що матриця коефіцієнтів розподілено по комп'ютерах горизонтальними смугами. Використовувати програму для послідовного алгоритму на прикладі 5.1.

Завдання 3. Паралельний алгоритм № 2 рішення СЛАР методом Гауса в MPI

Розробити паралельний алгоритм, написати та налагодити паралельну програму рішення СЛАР методом Гауса в MPI. У цьому алгоритмі передбачається, що матриця коефіцієнтів розподілена по комп'ютерах циклічно горизонтальними смугами. Використовувати програму для послідовного алгоритму на прикладі 5.2.

Завдання 4. Порівняльні часові характеристики двох алгоритмів розв'язання СЛАР методом Гауса в MPI.

Як паралельні програми взяти паралельні програми завдань 2 і 3. Для обох програм побудувати невеликі графіки залежності часу вирішення задачі від розмірів матриці коефіцієнтів. Обидві програми запустити послідовно для матриць: `A = [300 x 300], [500 x 500], [700 x 700], [1000 x 1000].` Відкомпілювати та запустити на 4-х комп'ютерах, засікти час та побудувати вказані графіки.

Контрольні запитання до лабораторної роботи № 5

1. Який метод розпаралелювання використовується в паралельних алгоритмах розв'язання СЛАР методом Гауса в MPI?
2. Як розподіляються елементи матриці коефіцієнтів та вектора правих частин під час вирішення СЛАР методом Гауса алгоритмом № 1?
3. Як розподіляються елементи матриці коефіцієнтів та вектора правих частин під час вирішення СЛАР методом Гауса алгоритмом № 2?
4. Який із методів Гауса є більш швидкодіючим?

Зміст звіту (Зразок оформлення звіту (ДодатокА))

1. Мета роботи.
2. Завдання до роботи.
3. Відповіді на контрольні питання.
4. Текст розробленого програмного забезпечення.
5. Результати тестування: вхідні дані та результати роботи програми.
6. Висновки, що відображають особисто отримані результати виконання роботи, їх критичний аналіз.

До звіту додаються файли проекту.

Лабораторна робота № 6
Розв'язання задачі Пуассона методом Зейделя у тривимірній області

Мета - практичне освоєння методів розпаралелювання алгоритмів задач, які вирішуються сіточними методами на прикладі розв'язання задачі Пуасона в тривимірній області.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

У лабораторній роботі дано пояснення способів декомпозиції даних, пов'язаних із різними способами розподілу даних комп'ютерами. У цьому розділі наведено послідовну програму вирішення завдання Пуассона методом Зейделя. Потрібно особливо наголосити, що схема Зейделя є неявною

Постановка задачі

Вирішити рівняння Пуассона

$$\frac{\partial^2\varphi}{\partial x}+\frac{\partial^2\varphi}{\partial y}+\frac{\partial^2\varphi}{\partial z}-a\varphi=\rho(x,y,z) \qquad (1)$$

$$\varphi=\varphi(x,y,z), a>0$$

в області Ω із крайовими умовами 1-го роду

$$\varphi|_G=F(x,y,z)$$

Область рішення Ω має вигляд прямокутного паралелепіпеда з розмірами X_m x Y_m x Z_m (рис. 6.1).

В області введено рівномірну прямокутну сітку з кроками:

$$h_x = X_m/I_m, \quad h_y = Y_m/J_m, \quad h_z = Z_m/K_m,$$

де I_m, J_m, K_m – кількість осередків у напрямках x, y, z.

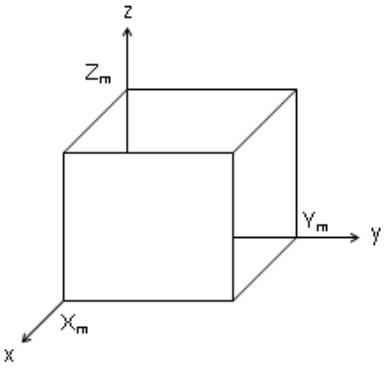


Рис. 6.1. Область рішення Ω

Значення сіткової функції задаються у вузлах сітки

$$\varphi_{i,j,k} = \varphi(x_i, y_j, z_k)$$

де i = 0,…,I_m , j = 0,…,J_m , k = 0,…,K_m.

У вибраній сітці апроксимується рівняння (1):

$$\frac{\varphi_{i+1} - 2\varphi_{i,j,k} + \varphi_{i-1}}{h_x^2} + \frac{\varphi_{j+1} - 2\varphi_{i,j,k} + \varphi_{j-1}}{h_y^2} + \frac{\varphi_{k+1} - 2\varphi_{i,j,k} + \varphi_{k-1}}{h_z^2} - \alpha\varphi = \rho_{i,j,k}$$

i = 1,…,I_m-1 , j = 1,…,I_m-1 , k = 1,…,I_m-1.

Отримана система лінійних рівнянь алгебри вирішується ітераційним методом Зейделя:

$$(\frac{2}{h_x^2} + \frac{2}{h_y^2} + \frac{2}{h_z^2} + a)\varphi_{i,j,k}^{m+1} = \frac{\varphi_{i+1}^{m} + \varphi_{i-1}^{m+1}}{h_x^2} + \frac{\varphi_{j+1}^{m} + \varphi_{j-1}^{m+1}}{h_y^2} + \frac{\varphi_{k+1}^{m} + \varphi_{k-1}^{m+1}}{h_z^2} - \rho_{i,j,k}$$

або

$$\varphi_{ijk}^{m+1} = \frac{\frac{\varphi_{i+1jk}^{m} + \varphi_{i-1jk}^{m+1}}{h_x^2} + \frac{\varphi_{ij+1k}^{m} + \varphi_{ij-1k}^{m+1}}{h_y^2} + \frac{\varphi_{ijk+1}^{m} + \varphi_{ijk-1}^{m+1}}{h_z^2} - \rho_{ijk}}{\frac{2}{h_x^2} + \frac{2}{h_y^2} + \frac{2}{h_z^2} + a}.$$

Тут m номер ітерації. Нагадаємо, що схема Зейделя неявна.

Вхідні параметри: Xm, Ym, Zm, Im, Jm, Km , έ, α функція $\rho(x,y,z)$, φ|ġ.

Ітерації вести до виконання умови:

$$\max |\varphi^{m+1} - \varphi^{m}| < \varepsilon$$

Вихідні дані:

1) $\max |\varphi^{m+1} - \varphi_{tochnoe}|$
2) функція $\varphi^{m+1} - \varphi_{tochnoe}$

Загальна схема розпаралелювання алгоритму задачі Пуассона

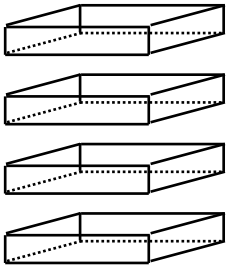
Тут наведено загальну схему вирішення задачі Пуассона в тривимірній області методом Зейделя на паралельній обчислювальній системі. Схема рішення пояснюється на малюнках та наведеному фрагменті основного ітераційного циклу паралельної програми для вирішення задачі Пуассона методом Зейделя у тривимірній області із застосуванням комунікаційних функцій високого рівня.

Завдання вирішується шляхом декомпозиції обчислювального простору на підобласті. Кожен комп'ютер містить відповідну підобласть обчислювального простору. Оскільки обчислення здійснюються за схемою «хрест», зазначені підобласті мають додаткові площини для дублювання граничних площин сусідніх підобластей. Тобто. здійснюється декомпозиція обчислювального простору на підобласті з перекриттям граничних площин. Використовується модель обчислень - SPMD (Single program - Multiple Data. Усі гілки виконуються за однією і тією ж програмою).

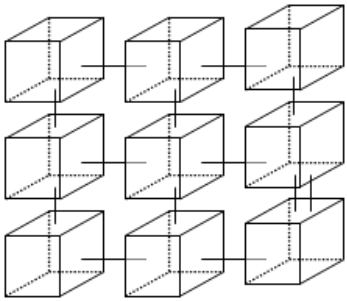
Тут передбачається, що область обчислень задачі прямокутна та задані її граничні значення. Топологія обчислювальної системи, що вирішує це завдання, може бути: "лінійка", або "двовимірна решітка", або "тривимірна решітка" (див. рис. 6.2)

Коротко схема обчислень здійснюється наступним чином. Оскільки метод Зейделя це неявний метод, то загальна схема обчислень представляється у вигляді конвеєра, що починається в

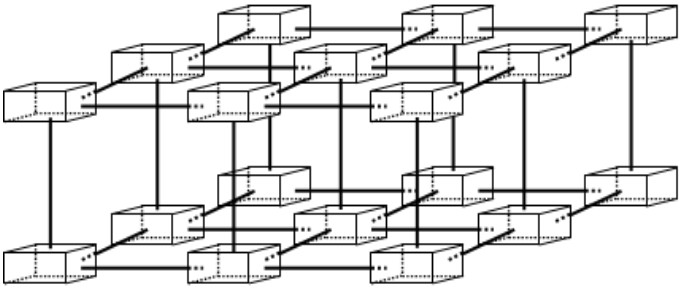
комп'ютері з нульовими координатами і після обміну граничними площинами, що триває в сусідніх комп'ютерах з великими значеннями координат.



а) декомпозиція на на "лінійці"



б) декомпозиція на "двовимірній решітці"



в) декомпозиція на "тривимірній решітці"

Рис. 6.2. Декомпозиція тривимірної області обчислень на ґратах комп'ютерів різної мірності

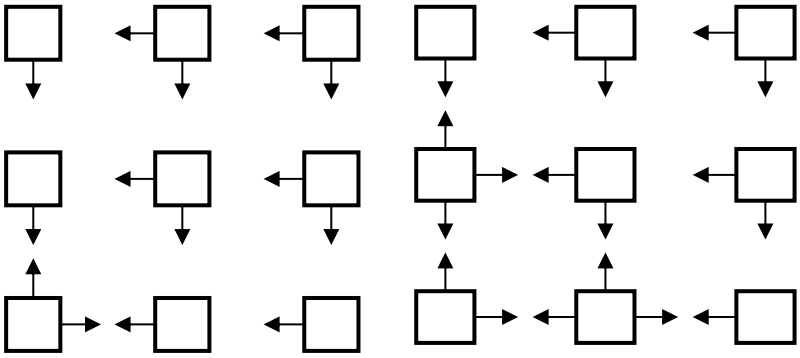
Тобто. обчислення кожної ітерації в точках простору виглядає як хвиля, що починається біля комп'ютерів з меншими координатами і триває до комп'ютерів з великими номерами.

Докладніше опис схеми обчислень.

1. Обчислення кожної і-ї ітерації в точках простору починає комп'ютер із нульовими координатами. (Цей комп'ютер знаходиться підобласть з найменшими координатами всієї області обчислень). Прорахувавши і-ю ітерацію до кордонів із сусідніми підобластями, що знаходяться у сусідніх комп'ютерах, він запускає процес асинхронного пересилання граничних площин (зі значеннями цієї `i-ї` ітерації) своєї підобласті своїм

сусіднім комп'ютерам. Потім запускає процес асинхронного прийому граничних площин від сусідніх комп'ютерів (які запишуться в площині перекриттів). Після чого починаються обчислення `i+1` ітерації і т.д.

2. Решта комп'ютерів процес обчислення кожної `i-ї` ітерації в точках простору починають з прийому граничних площин від сусідніх комп'ютерів з меншими значеннями координат (вздовж кожної координати). Або з очікування прийому, наприклад, у початковий момент (див. рис. 6.3). Після прийому кордонів від сусідів з меншими координатами, спочатку обчислюються граничні площини, сусідні з комп'ютерами з меншими координатами, тобто. з комп'ютерами від яких щойно було отримано кордону. Потім ці обчислені граничні площини передаються асинхронно комп'ютерам з меншими координатами. Прорахувавши `i-ю` ітерацію до кордонів із сусідніми підобластями, які знаходяться у сусідніх комп'ютерах з великими координатами, він запускає (крім останнього комп'ютера вздовж координати) процес асинхронного пересилання граничних площин (зі значеннями цієї `i-ї` ітерації) своєї підобласті цим сусіднім комп'ютерам. Потім він запускає процес асинхронного прийому граничних площин від сусідніх комп'ютерів із великими координатами (які запишуться у площині перекриттів).



а) початок обчислень кожної ітерації; б) продовження обчислень ітерації

Рис. 6.3. Ітераційні обчислення на двовимірній решітці комп'ютерів

Схема обмінів межами показана нижче на рис. 6.4.

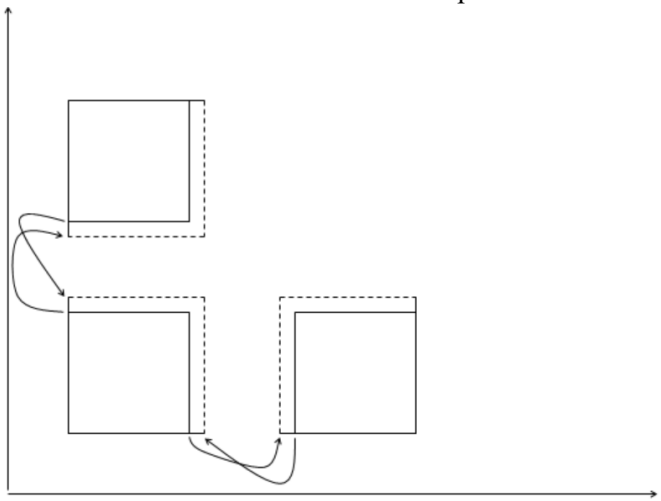


Рис. 6.4. Схема обміну межами між комп'ютерами

Нижче наведена програма розв'язання задачі Пуассона методом Зейделя для послідовного алгоритму.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#include<sys/time.h>
/* Кількість осередків вздовж координат x, y, z */
#define in 20
#define jn 20
#define kn 20

#define a 1

double Fresh(double,double,double);
double Ro(double,double,double);
void Inic();
```

```
/* Виділення пам'яті для 3D простору */
double F[in+1][jn+1][kn+1];

double hx, hy, hz;


/* Функція визначення точного рішення */
 double Fresh(double x,double y,double z)
          { double res;
          res = x + y + z;
          return res;
        }

/* Функція завдання правої частини рівняння */
 double Ro(double x, double y, double z)
        { double d;
          d = -a*(x+y+z);
          return d;
        }

/* Підпрограма ініціалізації меж 3D простору */
 void Inic()
      { int i, j, k;
        for(i = 0; i <= in; i++)
          { for(j = 0; j <= jn; j++)
             { for(k = 0; k <= kn; k++)
                { if((i!=0)&&(j!=0)&&(k!=0)&&(i!=in)&&(j!=jn)&&(k!=kn))
                 { F[i][j][k] = 0;
                 }
                else
                 { F[i][j][k] = Fresh(i*hx,j*hy,k*hz);
                   }
               }
            }
           }
       }
int main()
  {
    double X, Y, Z;
    double max, N, t1, t2;
    double owx, owy, owz, c, e;
    double Fi, Fj, Fk, F1;
```

```
int i, j, k, mi, mj, mk;
    int R, fl, fl1, fl2;
    int it,f;
    long int osdt;

    struct timeval tv1,tv2;

    it = 0;
    X = 2.0;
    Y = 2.0;
    Z = 2.0;
    e = 0.00001;

/* Размеры шагов */
    hx = X/in;
    hy = Y/jn;
    hz = Z/kn;

    owx = pow(hx,2);
    owy = pow(hy,2);
    owz = pow(hz,2);

    c = 2/owx + 2/owy + 2/owz + a;

    gettimeofday(&tv1,(struct timezone*)0);
/* Ініціалізація меж простору */
Inic();

/* Основний ітераційний цикл */
    do
     { f = 1;
       for(i = 1; i < in; i++)
         for(j = 1; j < jn; j++)
        { for(k = 1; k < kn; k++)
            { F1 = F[i][j][k];
              Fi = (F[i+1][j][k] + F[i-1][j][k]) / owx;
              Fj = (F[i][j+1][k] + F[i][j-1][k]) / owy;
              Fk = (F[i][j][k+1] + F[i][j][k-1]) / owz;
              F[i][j][k] = (Fi + Fj + Fk - Ro(i*hx,j*hy,k*hz)) / c;
              if(fabs(F[i][j][k] - F1) > e)
              f = 0;
            }
         }
      it++;
      }
     while(f == 0);
```

```
gettimeofday(&tv2,(struct timezone*)0);
   osdt = (tv2.tv_sec - tv1.tv_sec)*1000000 + tv2.tv_usec-tv1.tv_usec;

    printf("\n in = %d  iter = %d E = %f  T = %ld\n",in,it,e,osdt);

/* Знаходження максимальної розбіжності отриманого наближеного рішення
 * та точного рішення */
    max = 0.0;
      { for(i = 1; i <= in; i++)
         { for(j = 1; j < jn; j++)
           { for(k = 1; k < kn; k++)
               { if((F1 = fabs(F[i][j][k] - Fresh(i*hx,j*hy,k*hz))) > max)
                        { max = F1;
                     mi = i; mj = j; mk = k;
                   }
              }
              }
        }

  printf("Max differ = %f\n in point(%d,%d,%d)\n",max,mi,mj,mk);
      }
      return(0);

  }
```

ЗАВДАННЯ

Ретельно вивчити програму прикладу. Скомпілювати та запустити її на одному комп'ютері

Завдання до лабораторної роботи № 6

Мета – практичне освоєння методів розпаралелювання алгоритмів задач, які вирішуються сіточними методами в MPI.

Завдання 1. Опрацювання прикладу з пункту "Приклади паралельних програм" цієї лабораторної.

Уважно вивчити приклад 6.1. Відкомпілювати та запустити на 1-му процесорі.

Завдання 2. Побудова паралельного алгоритму розв'язання задачі Пуассона методом Зейделя в MPI

Розробити паралельний алгоритм, написати та налагодити паралельну програму розв'язання задачі Пуассона методом Зейделя в MPI.

Рекомендація: паралельний алгоритм реалізувати на лінійці комп'ютерів. Використовувати програму для послідовного алгоритму на прикладі 6.1.

Завдання 3. Запуск паралельного алгоритму розв'язання задачі Пуассона для різних функцій

Розроблений та налагоджений паралельний алгоритм перевірити, наприклад, для функції $f = X^2*Y^2*Z^2$.

Завдання 4. Порівняльні часові характеристики алгоритму розв'язання задачі Пуассона на одному та кількох комп'ютерах.

За паралельну програму взяти вами створену паралельну програму. За паралельну програму, взяти послідовну програму з прикладу попереднього пункту. Обидві програми запустити і засікти час рахунку для одного простору розміром 20*20*20. Засікти час вирішення задачі на 1-му, 2-х та 4-х комп'ютерах.

Контрольні запитання до лабораторної роботи № 6

1. Який метод розпаралелювання використовується для розв'язання задач, які використовують у своїй реалізації сіткові методи?
2. Як можна розподілити простір 3D обчислень на розподіленій обчислювальній системі?
3. Який із паралельних алгоритмів розв'язання задачі Пуассона в 3D на ваш погляд буде більш швидкодіючим: 1) на лінійці комп'ютерів; 2) на двовимірній решітці комп'ютерів; чи 3) на тривимірній решітці комп'ютерів?

Зміст звіту (Зразок оформлення звіту (ДодатокА))

1. Мета роботи.
2. Завдання до роботи.
3. Відповіді на контрольні питання.
4. Текст розробленого програмного забезпечення.
5. Результати тестування: вхідні дані та результати роботи програми.
6. Висновки, що відображають особисто отримані результати виконання роботи, їх критичний аналіз.

До звіту додаються файли проекту.

Теми для підготовки до модульної контрольної роботи №1

1. Класифікація архітектур обчислювальних систем. (Класифікація Флінна)
2. Основні архітектури паралельних комп'ютерів.
3. Моделі паралельного програмування. Основні властивості паралельних алгоритмів.
4. Модель завдання-канал. Основні властивості моделі "завдання / канал".
5. Модель передачі повідомлень.
6. Модель паралелізму даних.
7. Модель розділяється пам'яті.
8. Поняття про детермінізм програми.
9. Модульність. Види композиції модулів і / або програм.
10. Приклад паралельного алгоритму: скінченні різниці.
11. Основні етапи розробки паралельної програми.
12. Декомпозиція. Мети. Види декомпозиції. Підсумки етапу декомпозиції.
13. Комунікація. Види комунікацій: локальні, глобальні, динамічні, асинхронні. Розподілені комунікації і обчислення. Підсумки.
14. Інтеграція. Цілі інтеграції. Укрупнення деталізації. Реплікація обчислень. Підсумки.
15. Відображення. Цілі відображення. Основні стратегії. Підсумки.
16. Програмне забезпечення паралельного програмування. Основні види.
17. Коротка характеристика бібліотеки MPI.
18. Базові функції MPI (мінімальний набір).
19. Бібліотека MPI. Організація прийому / передачі даних між окремими процесами.
20. Бібліотека MPI. Колективні функції.
21. Бібліотека MPI. Глобальні обчислювальні операції над розподіленими даними.

22. Бібліотека MPI. Забезпечення модульності. Комунікатори, групи і галузі зв'язку.
23. Бібліотека MPI. Проблеми використання різних типів даних. Призначені для користувача типи даних.
24. Бібліотека MPI. Похідні типи даних і передача упакованих даних.

ПЕРЕЛІК ТЕМ ДЛЯ САМОСТІЙНОГО ОПРАЦЮВАННЯ

Основні поняття про паралельні та розподілені обчислення

Тема 1 Паралельні та розподілені обчислення, основні поняття

План

1. Сфери застосування і задачі розподіленої та паралельної обробки.
2. Конвеєризація та паралелізм.
3. Засоби для проведення паралельних обчислень.
4. Рівні розпаралелення. Паралельні операції. Основні принципи паралелізму (розпаралелення).
5. Класифікація структур паралельної обробки.

Питання для самоперевірки

1. Історія появи паралелізму.
2. Класифікації паралельних обчислювальних систем.
3. Сутність методів конвеєризації та паралелізму.
4. Основні принципи паралелізму (розпаралелення).
5. Рівні розпаралелювання.
6. Класифікації Хендлера і Флінна. Як співвідносяться класифікації між собою?

Література: [1–6; 18–20]

Тема 2 Архітектура паралельних обчислювальних систем

План

1. Класифікація комп'ютерних систем.
2. Векторно-конвеєрні комп'ютери.
3. Обчислювальні системи з розподіленою пам'яттю.
4. Паралельні комп'ютери зі спільною пам'яттю.
5. Кластери.

Питання для самоперевірки

1. Які існують класифікації комп'ютерних систем?
2. Дати характеристику обчислювальним системам з розподіленою пам'яттю.
3. Охарактеризуйте обчислювальні системи зі спільною пам'яттю.
4. Сутність кластерів та їх призначення.

Література: [7; 19– 23].

Тема 3 Методи оцінювання продуктивності паралельних алгоритмів і систем

План

1. Критерії оцінювання продуктивності паралельних систем.
2. Чинники, які необхідно враховувати під час оцінювання продуктивності паралельних систем.
3. Методи оцінювання продуктивності паралельних систем.
4. Характеристики продуктивності паралельних алгоритмів.
5. Порівняння MIMD і SIMD структур за продуктивністю.

Питання для самоперевірки

1. Які критерії використовують для оцінювання продуктивності паралельних алгоритмів і систем?
2. Які чинники впливають на оцінювання продуктивності паралельних систем?
3. Охарактеризуйте методи оцінювання продуктивності паралельних систем.
4. Сутність закону Амдала.

Література: [1–5; 7; 18–20].

Тема 4 Організація мереж Петрі

План

1. Поняття про мережі Петрі.
2. Прості мережі Петрі
3. Розширені мережі Петрі.
4. Приклади реалізації мереж Петрі.

Питання для самоперевірки

1. Призначення мереж Петрі.
2. Перехід, ребро, вузол, маркування. Стани маркувань.
3. Синхронізація процесів у мережі Петрі.
4. Дуги. Дуга-заперечення. Вага дуг.
5. Яка різниця між простими та розширеними мережами Петрі?

Література: [5; 15; 18; 19; 23].

Тема 5 Концепція паралельних обчислювань. Технології паралельного програмування

План

1. Паралелізм даних.
2. Паралелізм задач.
3. Етапи розробки паралельного алгоритму.
4. Стратегії розміщення задач на процесорах.

Питання для самоперевірки

1. На чому грунтується паралелізм даних? Основні ознаки підходу.
2. Сутність методу паралелізму задач.
3. Охарактеризуйте етапи розробки паралельного алгоритму.
4. Охарактеризуйте стратегію господар/робітник.
5. Охарактеризуйте ієрархічні та децентралізовані стратегії.

Література: [3–4; 8–10; 14; 21].

Тема 6 Структури зв'язку між процесорами

План

1. Основні положення.
2. Шинні мережі.
3. Сітки з комутаторами
4. Структури, що забезпечують зв'язок типу «пункт-пункт».
5. Методи комутацій.

Питання для самоперевірки

1. Яким критеріям має відповідати структура зв'язку між процесорами?
2. Класи структур зв'язку.
3. Охарактеризуйте шинні мережі.
4. Охарактеризуйте мережі з комутаторами.
5. Охарактеризуйте структури, що забезпечують зв'язок типу «пункт- пункт».
6. Методи комутації в комутаційних мережах паралельних систем.

Література: [10; 17–18].

Тема 7 Паралельне програмування з використанням технології MPI

План

1. Призначення MPI. Модель MPI-додатка.
2. Загальна організація MPI. Базові функції MPI. Комунікатори.
3. Функції ініціалізації та завершення роботи.
4. Етапи передачіповідомлень між паралельними процесами MPI. Функції передачі повідомлень між процесами типу «один-одному».
5. Типи даних MPI.
6. Колективні комунікації.
7. Розподілені операції в MPI.
8. Створення нових типів даних MPI.
9. Створення розподілених операцій.
10. Топології процесів. Створення декартової топології процесів у MPI- додатках. Приклад використання декартової топології процесів.

Питання для самоперевірки

1. Загальна будова MPI-програм.
2. Яку структуру називають комунікатором? Стандартні комунікатори MPI?

3. Які є різновиди двухточкового обміну?
4. Процедура для визначення рангу процесу.
5. Як працює буферезований режим передачі повідомлення? У яких випадках рекомендується використовувати буферезований обмін?
6. Як виконується неблокувальний обмін?
7. Яку функцію використовують для створення буфера? Які параметри приймає функція відключення буфера?
8. Колективні обміни, їх характеристика. Як пересилається повідомлення у разі колективного обміну? Як називають процес, який виконує трансляцію розсилки?
9. Віртуальність топологій обміну MPI. Типи топологій в MPI. Як задаються «декартові» топології та топології «граф» у MPI?
10. Операція зрушенням. Типи зрушень даних.
11. Характеристика колективних обмінів.

Література: [7; 8; 10–13; 20].

Паралельне програмування з використанням технологій MPI і OpenMP

Тема 8 Паралельне програмування з використанням технології OpenMP

План

1. Призначення OpenMP. Модель OpenMP-додатка.
2. Директива паралельної обробки parallel.
3. Директива розподілення роботи for. Директиви розподілення роботи sections і section. Директиви single та master. Директиви tasks і taskwait.
4. Директиви синхронізації barrier, ordered, critical, atomic.
5. Спільні та приватні змінні.
6. Функції середовища виконання. Функції блокування і синхронізації. Змінні оточення.
7. Алгоритми планування паралельного виконання циклів (static, dynamic, guided, runtime scheduling).

Питання для самоперевірки

1. Характеристика та основні переваги технології OpenMP.
2. Які компілятори забезпечують підтримку технології OpenMP?
3. Мінімальний набір директив OpenMP, який дозволяє почати розробку паралельних програм.
4. Розпаралелювання циклів у OpenMP. Умови розпаралелювання циклів.
5. Порядок виконання ітерацій в розпаралелювальних циклах OpenMP. Які правила синхронізації обчислень у розпаралелювальних циклах OpenMP?
6. Загальні та локальні змінні потоків.
7. Операція редукції.
8. Способи організації взаємовиключення в OpenMP.
9. Визначення критичної секції. Операції OpenMP для змінних семафорного типу (замків).
10. Як здійснюється в OpenMP розпаралелювання за задачами?
11. Засоби OpenMP для створення та управління створеними потоками.
12. Які переваги надає OpenMP програмісту?
13. Які основні класи змінних використовуються в OpenMP?
14. Як виконується управління спільними та індивідуальними даними в OpenMP?
15. Як планується виконання циклу в OpenMP?

Література: [7; 8; 10–13; 20].

Тема 9 Паралельні методи множення матриці на вектор

План

1. Способи розділення матриць на смуги, прямокутні фрагменти
2. Визначення підзадач, їх інформаційна взаємодія під час виконання обчислень.
3. Масштабування і розподіл підзадач по процесорах.
4. Аналіз ефективності паралельного алгоритму.

5. Реалізація паралельних програм засобами технологій OpenMP та MPI.

Питання для самоперевірки

1. Які існують способи розділення матриць на смуги, прямокутні фрагменти?
2. Як відбувається інформаційна взаємодія під час виконання обчислень?
3. Що таке масштабування підзадач, на якому етапі розпаралелювання алгоритму воно відбувається?
4. Як відбувається розподіл підзадач по процесорах?

Література: [7; 8; 10–13; 19–20].

Тема 10 Паралельні методи матричного множення

План

1. Множення матриць зі стрічковою схемою розділення даних.
2. Алгоритм Фокса множення матриць з блочним розділом даних.
3. Алгоритм Кеннона множення матриць з блочним розділом даних.
4. Реалізація паралельних програм засобами технологій OpenMP та MPI.

Питання для самоперевірки

1. Як відбувається множення матриць зі стрічковимою схемою розділення даних?
2. Поясніть сутність алгоритму Фокса множення матриць з блочним розділом даних.
3. Поясніть сутність алгоритму Кеннона множення матриць з блочним розділом даних

Література: [4–7; 8; 10–13; 24]

Тема 11 Рішення систем лінійних рівнянь

План

1. Метод Гауса розв’язання системи лінійних рівнянь.
2. Ітераційні методи розв’язання систем лінійних рівнянь.
3. Реалізація паралельних програм засобами технологій OpenMP та MPI.

Питання для самоперевірки

4. Стисло охарактеризуйте методи розв’язання системи лінійних рівнянь засобами паралельного програмування.
5. Які функції стандартів OpenMP та MPI використовують для реалізації паралельних програм розв’язання системи лінійних рівнянь різними методами?

Література: [4–7; 8; 10–13; 20].

Тема 12 Паралельні методи сортування

План

1. «Бульбашкове» сортування елементів матриці.
2. Паралельні методи сортування елементів матриці.
3. Реалізація паралельних програм засобами технологій OpenMP та MPI.

Питання для самоперевірки

1. Стисло охарактеризуйте методи сортування елементів матриці засобами паралельного програмування.
2. Які функції стандартів OpenMP та MPI використовують для реалізації паралельних програм сортування елементів матриці різними методами?

Література: [4–7; 8; 10–13; 20–24].

ПИТАННЯ ДЛЯ ПІДСУМКОВОГО КОНТРОЛЮ ЗНАНЬ

1. Сутність, основні цілі паралельної обробки інформації. Різновиди паралельної обробки даних.
2. Основні способи досягнення паралелізму. Дві парадигми та основні моделі паралельного програмування. Рівні розпаралелювання.
3. Методи оцінювання продуктивності паралельних алгоритмів і систем. Ефективність паралельних програм. Закон Амдала.
4. Характеристика етапів розробки паралельного алгоритму.
5. Потік, потік команд, потік даних. Пояснити сутність.
6. Прості та розширені мережі Петрі. Синхронізація процесів у мережі Петрі. Приклади реалізації мереж Петрі.
7. Особливості організації паралельної роботи з використанням технології програмування OpenMP.
8. Характеристика буферезованого режиму передачі повідомлення.
9. Сутність колективного обміну та процесу широкомовної розсилки.
10. Функції блокування і синхронізації технології OpenMP.
11. Директива. Мінімальний набір директив OpenMP для початку розробки паралельних програм.
12. Розпаралелювання циклів у OpenMP, умови розпаралелювання циклів. Можливості OpenMP для управління розподілом ітерацій циклів між потоками.
13. Особливості організації паралельної роботи з використанням технології програмування MPI.
14. Сутність стандарту механізму передачі повідомлень MPI. Модель паралельної програми в MPI.

15. Функції, що забезпечують взаємодію паралельних процесів за допомогою механізму передачі повідомлень у технології MPI. Охарактеризуйте їх за способом виконання.
16. Стандартні комунікатори MPI. Комунікаційні операції типу точка- точка. Комунікаційні функції типу точка-точка.
17. Особливості буферезованого режиму передачі повідомлень.
18. Неблокувальні комунікаційні операції.
19. Колективні обміни. Колективні операції.
20. Функції збирання блоків даних від усіх процесів групи.
21. Функція розподілу блоків даних за всіма процесами групи.
22. Глобальні розрахункові операції над розподіленими даними.
23. Широкомовна розсилка повідомлень. Механізми MPI пересилання даних.
24. Похідні типи даних. Стандартний сценарій визначення і використання похідних типів. Функції для роботи похідними типами.
25. Упаковані дані та їх передача. Поняття «група». Функції роботи з групами.
26. Функції MPI роботи з комунікаторами.
27. Топологія процесів. Типи топологій в MPI.
28. Декартова топологія. Функція для створення комунікатора з декартовою топологією.
29. «Зміщення», типи зміщення даних.
30. Топологія типу «граф» у MPI.

ЛІТЕРАТУРА

1. Минайленко Р.М. Паралельні та розподілені обчислення: навч. посіб. Кропивницький: Видавець Лисенко В. Ф., 2021. 153с.
2. Рольщиков В.Б. Технології розподілених систем та паралельних обчислень. Конспект лекцій. Одеса: ОДЕКУ 2016.155с.
3. Смірнова Н.В., Смірнов В.В. Паралельні та розподілені обчислення: Метод.вказ. до вик. лаб. роб. Кіровоград:КНТУ. 2015. 52с.
4. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. М.: Изд-во МГУ, 2009. 77 с.
5. Ясько, М.М. Навчальний посібник до вивчення курсів “Паралельна обробка даних” та “Мови обчислень та кластерні системи”. Д.: РВВ ДНУ, 2010. 76с.
6. Гергель В.П. Теория и практика параллельных вычислений, 2-е изд. М.: Интуит, 2016. 500 с.
7. Хорстманн, Кей С. Java. Библиотека профессионала. Основы. Санкт-Петербург: ООО "Диалектика", 2019. 864 с.
8. Goetz B. Java Concurrency in Practice. 2006. 424 с.
9. Grama, A., Gupta, A., Kumar V. Introduction to Parallel Computing. Harlow, England: Addison-Wesley.
10. Pacheco, P. Parallel Programming with MPI. - Morgan Kaufmann.
11. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Melon, R. Parallel Programming in OpenMP. Morgan Kaufmann Publishers.
12. MPI: The Complete Reference URL: http://www.netlib.org/ papers/mpi-book/mpi-book.html

13. Foster, Ian. Designing and Building Parallel Programs [Text] / Ian Foster - Addison- Wesley, Inc. Boston, MA, 1995 - 381pp.
14. ScaLAPACK Users' Guide. URL: http://www.netlib.org/scalapack/slug/
15. Quinn M.J. Parallel Programming in C with MPI and OpenMP. McGrawHill, 2004, 544 p.
16. OpenMP Application Program Interface. Version 3.0 May 2008
17. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП«АУТДОР-Шарк», 2021. 188 с.
18. Аксак Н.Г., Руденко О.Г., Гуржій А.М. Паралельні та розподілені обчислення: підруч. Х. : Компанія СМІТ, 2009р.480с.
19. Жуков І., Корочкін О.. Паралельні та розподілені обчислення. К. : Корнійчук, 2005. 226 с.
20. Кузьменко Б. В., Чайковська О. А. Технологія розподілених систем та паралельних обчислень. Навчальний посібник. Частина 1. К. : Видавничий центр КНУКІМ, 2011. 161 с.
21. Демчина М. М. Паралельне програмування : конспект лекцій. Івано-Франківськ : ІФНТУНГ, 2015. 176 с.
22. Дорошенко А. Ю., Фінін Г. С., Цейтлін Г. О. Алгеброалгоритмічні основи програмування. К. : Наук. Думка, 2004. 457 с.
23. Кавун С. В. , Сорбат І. В. Архітектура комп’ютерів. Особливості використання комп’ютерів в ІС : навч. посіб. Харків : Вид. ХНЕУ, 2010. 256 с.
24. Мельник А. О. Архітектура комп’ютера. Луцьк, 2008. 470 с.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ТА ІНТЕРНЕТ-РЕСУРСИ

1. Apache NetBeans. URL: https://netbeans.apache.org/ download/index.html.
2. Java 8 URL: https://java.com/.
3. MPI. URL: https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi.
4. OpenMP Architecture Review Board. URL: http://www.openmp.org.
5. C++ documentation. URL: http://docs.microsoft.com/en-us/cpp/.
6. C++ documentation. URL: http://www.cplusplus.com.
7. JDK 17 Documentation. URL: https://docs.oracle.com/en/ java/ javase/17/.
8. Message passing interface. URL: http://www.mpiforum.org.
9. Параллельное программирование с использованием технологии OpenMP. URL: http://parallel.ru/tech/tech_dev/ OpenMP/examples/.
10. Development of distributed computing. URL: http://www. gridforum.org.
11. CUDA Toolkit Documentation v11.5.1. URL: https://docs. nvidia.com.
12. Сайт: Освітній портал ЛНУ імені Тараса Шевченка – Digital Office". Паралельні та розподілені обчислення. URL: http://do. luguniv.edu.ua/course/view.php?id=32837.
13. Паралельне програмування та його призначення. URL: https://foxminded.ua/paralelne-prohramuvannia/.

НАБІР ФУНКЦІЙ БІБЛІОТЕКИ MPI

MPI - це бібліотека функцій, що забезпечує взаємодію паралельних процесів за допомогою механізму передачі повідомлень. Це достатньо об'ємна і складна бібліотека, що складається приблизно з 130 функцій, до числа яких входять:

- функції ініціалізації і закриття MPI-процесів;
- функції реалізації комунікаційних операцій типу точка-точка;
- функції реалізації колективних операцій;
- функції для роботи з групами процесів і комунікаторами;
- функції для роботи зі структурами даних;
- функції формування топології процесів.

Набір функцій бібліотеки MPI виходить далеко за межі набору функцій, мінімально необхідного для підтримки механізму передачі повідомлень. будь-яка паралельна програма може бути написана з використанням всього 6-MPI функцій, а достатньо повне і зручне середовище програмування створює набір з 24 функцій.

Кожна з MPI-функцій характеризується способом виконання:

1. Локальна функція – виконується усередині конкретного процесу. Її завершення не вимагає комунікацій.

2. Нелокальна функція –для її завершення потрібне виконання MPI- процедури іншим процесом.

3. Глобальна функція - процедуру повинні виконувати всі процеси групи. Недотримання цієї умови може призводити до зависання завдання.

4. Блокуюча функція - повернення керування з процедури гарантує можливість повторного використання параметрів, що беруть участь у виклику. Ніяких змін у стані процесу, що викликав блокуючий запит, до виходу з процедури не буде.

5. Неблокувальна функція - повернення з процедури відбувається негайно, без очікування закінчення операції і до того, як буде дозволене повторне використання параметрів, що

беруть участь у запиті. Завершення неблокувальних операцій здійснюється спеціальними функціями.

Використання бібліотеки MPI має деякі відмінності в мовах C і FORTRAN. У мові C всі процедури є функціями, і більшість з них повертає код помилки. Використовуючи імена підпрограм і іменованих констант, необхідно строго дотримуватися регістра символів. Масиви індексуються з 0. Логічні змінні подаються типом int (`true` відповідає 1, а `false - 0`). Визначення всіх іменованих констант, прототипів функцій і визначення типів виконується підключенням файлу `mpi.h`.

У табл. 1 наведена відповідність зумовлених у MPI типів стандартним типам мови C та Фортран.

Таблиця 1. Відповідність між MPI-типами і типами мови C та Фортран

Тип **MPI**	Тип мови **C**	Тип мови Фортран
MPI_CHAR	**signed char**	**integer(1)**
MPI_SHORT	**signed short int**	**integer(2)**
MPI_INT	**signed int**	**integer(4)**
MPI_LONG	**signed long int**	**integer(8)**
MPI_UNSIGNED_CHAR	**unsigned char**	
MPI_UNSIGNED_SHORT	**unsigned short int**	
MPI_UNSIGNED	**unsigned int**	
MPI_UNSIGNED_LONG	**unsigned long int**	
MPI_FLOAT	**float**	**real(4)**
MPI_DOUBLE	**double**	**real(8)**
MPI_LONG_DOUBLE	**long double**	**real(12)**
MPI_BYTE		**integer(1)**
MPI_PACKED		

Декілька процедур, оформлених у вигляді функцій, код помилки не повертають. Не вимагається строгого дотримання регістра символів в іменах підпрограм і іменованих констант. Масиви за замовчуванням індексуються з 1. Об'єкти MPI, які в мові C є структурами, в мові FORTRAN подаються масивами цілого типу. Визначення всіх іменованих констант і визначення типів виконується підключенням файлу **`mpif.h`**.

В даній реалізації всі інтерфейси процедур MPI

знаходяться в модулі MPI (файл **`MPI.f03`**). Всі процедури MPI реалізовані у вигляді функцій і мають точно такий же вигляд, як і в мові С. Далі всі прототипи функцій будуть описуватись на мові С, а приклади програм будуть наведені на мові Фортран.

У табл. 1 наведена відповідність зумовлених у MPI типів стандартним типам мови FORTRAN. Слід відзначити, що в табл. 2 перерахований обов'язковий мінімум підтримуваних стандартних типів, проте, якщо в базовій системі представлені й інші типи, то їх підтримку здійснюватиме і MPI. Типи MPI_BYTE і MPI_PACKED застосовуються для передачі двійкової інформації без якого- небудь перетворення.

Базові функції MPI

Будь-яка прикладна MPI-програма (застосування) повинна починатися з виклику функції ініціалізації MPI: `MPI_Init`. У результаті виконання цієї функції створюється група процесів, в яку поміщаються всі процеси, і створюється область зв'язку, що описується комунікатором `MPI_COMM_WORLD`. Ця область зв'язку об'єднує всі процеси-застосування. Процеси в групі впорядковані і пронумеровані від 0 до groupsize-1, де groupsize дорівнює числу процесів у групі. Крім того, створюється зумовлений комунікатор MPI_COMM_SELF, що описує свою область зв'язку для кожного окремого процесу.

Синтаксис функції ініціалізації `MPI_Init` значно відрізняється в мовах С і FORTRAN:

Прототип функції MPI_Init на мові С:

```
int MPI_Init(int *argc, char ***argv)
```

На мові FORTRAN це можна використати таким чином:

```
if( MPI_Init(0,0) /= MPI_SUCCESS) stop 'MPI error'
```

Функція завершення MPI-програм MPI_Finalize С:

```
int MPI_Finalize(void)
```

FORTRAN:

```
Ierr = MPI_FINALIZE()
```

Функція закриває всі MPI-процеси і ліквідує всі області зв'язку. Далі всі прототипи функцій MPI будуть наводитись тільки мовою С.

Функція визначення числа процесів у області зв'язку `MPI_Comm_size int MPI_Comm_size(MPI_Comm comm, int *size)`

IN comm - комунікатор;

OUT size - число процесів у області зв'язку комунікатора comm.

Функція повертає кількість процесів у області зв'язку комунікатора comm.

До створення явним чином груп і пов'язаних з ними комунікаторів (розділ

6) єдино можливими значеннями параметра COMM є MPI_COMM_WORLD і MPI_COMM_SELF, які створюються автоматично у процесі ініціалізації MPI. Підпрограма є локальною.

Функція визначення номера процесу **MPI_Comm_rank**

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

IN comm - комунікатор;

OUT Rank - номер процесу, який викликав функцію.

Функція повертає номер процесу, що викликав цю функцію. Номери процесів лежать у діапазоні **`0..size-1`** (значення **`size`** може бути визначене за допомогою попередньої функції). Підпрограма є локальною.

У мінімальний набір слід включити також дві функції передачі й прийому повідомлень.

Функція передачі повідомлення **MPI_Send**

```
int MPI_Send(void* buf, int count MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

IN buf - адреса почала розташування даних, що пересилаються;

IN count - число елементів, що пересилаються;

Ndatatype - тип надісланих елементів;

IN dest - номер процесу-одержувача в групі, пов'язаній з комунікатором comm;

IN tag – ідентифікатор повідомлення;

IN comm - Комунікатор області зв'язку.

Функція виконує посилку count елементів типу datatype повідомлення з ідентифікатором tag процесу dest у області зв'язку комунікатора comm. Змінна buf це, як правило, масив або скаляр. В останньому випадку значення count = 1.

Функція прийому повідомлення **MPI_Recv**

```
int MPI_Recv(void* buf, int count,
MPI_Datatype datatype, int source, int
```

```
 tag, MPI_Comm comm, MPI_Status *status)
```

OUT buf - адреса почала розташування повідомлення, що приймається;

IN datatype - тип елементів повідомлення, що приймається;
IN source - номер процесу-відправника;
IN tag - ідентифікатор повідомлення;
IN comm - комунікатор області зв'язку;
OUT status - атрибути прийнятого повідомлення.

Функція виконує прийом count елементів типу datatype повідомлення з ідентифікатором tag від процесу source у області зв'язку комунікатора comm.

Детальніше операції обміну повідомленнями будуть описані в наступному розділі, а на закінчення цього розділу розглянемо функцію, яка не входить в окреслений нами мінімум, але важлива для розробки ефективних програм. Йдеться про функцію отримання відліку часу - таймер. З одного боку, такі функції є у складі всіх операційних систем, а з іншого – існує цілковите свавілля в їх реалізації. Досвід роботи з різними операційними системами показує, що в разі перенесення програм з однієї платформи на іншу перше (а іноді і єдине), що доводиться переробляти, - це звернення до функцій обліку часу. Тому розробники MPI, добиваючись повної незалежності застосувань від операційного середовища, визначили і свої функції відліку часу.

Функція відліку часу (таймер) **MPI_Wtime**

```
double MPI_Wtime(void)
```

Функція повертає астрономічний час (у секундах), що пройшов з деякого моменту у минулому (точки відліку). Гарантується, що ця точка відліку не буде змінена протягом життя процесу. Для хронометражу ділянки програми виклик функції робиться на початку й кінці ділянки й визначається різниця між показаннями таймера. Це проілюстровано в наступному фрагменті програми на мові С:

```
{
double start, end; start = MPI_Wtime();
... ділянка ..., що хронометрується
end = MPI_Wtime();
printf("Виконання зайняло %f секунд\n", end-
```

```
start);
  }
```

Функція **MPI_Wtick**, що має такий самий синтаксис, повертає розрізнення таймера (мінімальне значення кванта часу).

Комунікаційні операції типу точка-точка

До операцій цього типу належать дві представлені в попередньому розділі комунікаційні процедури. У комунікаційних операціях типу точка-точка завжди беруть участь не більше двох процесів: той, що передає, і той, що приймає. У MPI є безліч функцій, що реалізовують такий тип обмінів. Різноманіття пояснюється можливістю організації таких обмінів безліччю способів. Описані в попередньому розділі функції реалізують стандартний режим з блокуванням.

Блокувальні функції дають можливість виходу з них тільки після повного закінчення операції, тобто процес блокується, поки операція не буде завершена. Для функції посилки повідомлення це означає, що всі дані, що пересилаються, поміщені в буфер (для різних реалізацій MPI це може бути або якийсь проміжний системний буфер, або безпосередньо буфер одержувача). Для функції прийому повідомлення блокується виконання інших операцій, поки всі дані з буфера не будуть поміщені в адресний простір приймального процесу.

Неблокувальні функції передбачають поєднання операцій обміну з іншими операціями, тому неблокувальні функції передачі і прийому по суті є функціями ініціалізації відповідних операцій. Для опитування завершеності операції (і завершення) вводяться додаткові функції.

Як для блокувальних, так і неблокувальних операцій MPI підтримує чотири режими виконання. Ці режими стосуються тільки функцій передачі даних, тому для блокувальних і неблокувальних операцій є по чотири функції посилки повідомлення. У табл. 2 перераховані імена базових комунікаційних функцій типу точка-точка, наявних у бібліотеці MPI.

Таблиця 2. Список комунікаційних функцій типу точка-точка

Режими виконання	З блокуванням	Без блокування
Стандартна посилка	MPI_Send	MPI_Isend
Синхронна посилка	MPI_Ssend	MPI_Issend
Посилка, що буферизує	MPI_Bsend	MPI_Ibsend
Узгоджена посилка	MPI_Rsend	MPI_Irsend
Прийом інформації	MPI_Recv	MPI_Irecv

Табл. 2 демонструє принцип формування імен функцій. До імен базових функцій Send/Recv додаються різні префікси.

Префікс S (synchronous) - означає синхронний режим передачі даних. Операція передачі даних закінчується тільки тоді, коли закінчується прийом даних. Функція нелокальна.

Префікс B (buffered) - означає режим передачі даних, що буферизує. В адресному просторі процесу, що передає дані, за допомогою спеціальної функції створюється буфер обміну, який використовується в операціях обміну. Операція посилки закінчується, коли дані поміщені в цей буфер. Функція має локальний характер.

Префікс R (ready) - узгоджений або підготовлений режим передачі даних. Операція передачі даних починається тільки тоді, коли приймальний процесор виставив ознаку готовності до прийому даних, тобто ініціював операцію прийому. Функція нелокальна.

Префікс I (immediate) - означає неблокувальні операції.

Всі функції передачі і прийому повідомлень можуть використовуватися в будь-якій комбінації один з одним. Функції передачі, що знаходяться в одному стовпці, мають абсолютно однаковий синтаксис і відрізняються тільки внутрішньою реалізацією. Тому надалі розглядатимемо тільки стандартний режим, який в обов'язковому порядку підтримують всі реалізації MPI.

Блокувальні комунікаційні операції

Синтаксис базових комунікаційних функцій MPI_Send і MPI_Recv був наведений у розділі 3.2, тому тут ми розглянемо тільки семантику цих операцій.

У стандартному режимі виконання операція обміну

включає три етапи:

1. Сторона, що передає, формує пакет повідомлення, в який крім інформації, що передається, упаковуються адреса відправника (source), адреса одержувача (dest), ідентифікатор повідомлення (tag) і комунікатор (comm). Цей пакет передається відправником у системний буфер, і на цьому функція посилки повідомлення закінчується.

2. Повідомлення системними засобами передається адресатові.

3. Приймальний процесор витягує повідомлення з системного буфера, коли в нього з'явиться потреба в цих даних. Змістова частина повідомлення поміщається в адресний простір приймального процесу (параметр buf), а службова - в параметр status.

Оскільки операція виконується в асинхронному режимі, адресна частина прийнятого повідомлення складається з трьох полів:

– комунікатора (comm), оскільки кожен процес може одночасно входити в декілька областей зв'язку;

– номера відправника в цій області зв'язку (source);

– ідентифікатора повідомлення (tag), який застосовується для взаємної прив'язки конкретної пари операцій посилки і прийому повідомлень.

Параметр count (кількість елементів повідомлення, що приймаються) в процедурі прийому повідомлення повинен бути не менший, ніж довжина повідомлення, що приймається. При цьому реально прийматиметься стільки елементів, скільки знаходиться в буфері. Така реалізація операції читання пов'язана з тим, що MPI допускає використання розширених запитів:

– для ідентифікаторів повідомлень (MPI_ANY_TAG - читати повідомлення з будь-яким ідентифікатором);

– для адрес відправника (MPI_ANY_SOURCE - читати повідомлення від будь-якого відправника).

Не допускаються розширені запити для комунікаторів. Розширені запити можливі тільки в операціях читання. У цьому відбивається фундаментальна властивість механізму передачі

повідомлень: асиметрія операцій передачі і прийому повідомлень, пов'язана з тим, що ініціатива в організації обміну належить стороні, яка передає.

Таким чином, після читання повідомлення деякі параметри можуть виявитися невідомими, а саме: число надісланих елементів, ідентифікатор повідомлення і адреса відправника. Цю інформацію можна отримати за допомогою параметра status. Змінні status повинні бути явно оголошені в MPI- програмі. У мові C status - це структура типу MPI_Status з трьома полями MPI_SOURCE, MPI_TAG, MPI_ERROR. У мові FORTRAN status - масив типу INTEGER розміру MPI_STATUS_SIZE. Константи MPI_SOURCE, MPI_TAG і MPI_ERROR визначають індекси елементів. Призначення полів змінною status подане в табл. 3.

Таблиця 3. Призначення полів змінною status

Поля **status**	**C**	**FORTRAN**
Процес-відправник	status.MPI_SOURCE	status(MPI_SOURCE)
Ідентифікатор повідомлення	status.MPI_TAG	status(MPI_TAG)
Код помилки	status.MPI_ERROR	status(MPI_ERROR)

Як видно із табл. 3, кількість елементів у змінну status не заноситься. Для визначення числа фактично отриманих елементів повідомлення необхідно використовувати спеціальну функцію MPI_Get_count:

```
int MPI_Get_count (MPI_Status *status, MPI_Datatype
datatype, int *count)
```

IN status - атрибути прийнятого повідомлення;
IN datatype – т ип елементів прийнятого повідомлення;
OUT count - число отриманих елементів.

Підпрограма MPI_Get_count може бути викликана або після читання повідомлення (функціями MPI_Recv, MPI_Irecv), або після опитування факту надходження повідомлення (функціями MPI_Probe, MPI_Iprobe). Операція читання безповоротно знищує інформацію в буфері прийому. При цьому спроба прочитати повідомлення з параметром count меншим,

ніж число елементів в буфері, призводить до втрати повідомлення.

Визначити параметри отриманого повідомлення без його читання можна за допомогою функції MPI_Probe.

```
int MPI_Probe (int source, int tag, MPI_Comm
comm, MPI_Status *status)
```

IN source - номер процесу-відправника;
IN tag - ідентифікатор повідомлення;
IN comm - комунікатор;
OUT status - атрибути опитаного повідомлення.

Підпрограма MPI_Probe виконується з блокуванням, тому завершиться вона лише тоді, коли повідомлення з відповідним ідентифікатором і номером процесу- відправника буде доступне для отримання. Атрибути цього повідомлення повертаються в змінній status. Наступний за MPI_Probe виклик MPI_Recv з тими ж атрибутами повідомлення (номером процесу-відправника, ідентифікатором повідомлення і комунікатором) помістить у буфер прийому саме те повідомлення, наявність якого була опитана підпрограмою MPI_Probe.

У разі використання блокувального режиму передачі повідомлень існує потенційна небезпека виникнення тупикових ситуацій, в яких операції обміну даними блокують один одного. Наведемо приклад некоректної програми на мові Фортран, яка зависатиме за будь-яких умов.

```
Ierr = MPI_COMM_RANK(comm, rank)
IF (rank.EQ.0) THEN
     Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,1,tag
     ,comm, status) Ierr=MPI_SEND(loc(sendbuf),
     count, MPI_REAL, 1, tag, comm)
ELSE IF (rank.EQ.1) THEN
     Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,0,tag
     ,comm, status) Ierr=MPI_SEND(loc(sendbuf),
     count, MPI_REAL, 0, tag, comm)
END IF
```

У даному прикладі обидва процеси (0-й і 1-й) входять у

режим взаємного очікування повідомлення один від одного. Такі тупикові ситуації виникатимуть завжди під час утворення циклічних ланцюжків блокувальних операцій читання.

Приведемо варіант правильної програми

```
Ierr = MPI_COMM_RANK(comm, rank)
IF (rank.EQ.0) THEN
      Ierr=MPI_SEND(loc(sendbuf),count, MPI_REAL, 1,
      tag, comm)
      Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,1,tag,
      comm, status)
ELSE IF (rank.EQ.1) THEN
      Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,0,tag,
      comm,status)
      Ierr=MPI_SEND(loc(sendbuf),count,MPI_REAL,0,tag,
      comm)
END IF
```

Інші комбінації операцій SEND/RECV можуть працювати або не працювати залежно від реалізації MPI (обмін з буферизацією, чи ні).

У ситуаціях, коли потрібно виконати взаємний обмін даними між процесами, безпечніше використовувати суміщену операцію **MPI_Sendrecv.**

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int
source, MPI_Datatypea recvtag, MPI_Comm comm,
MPI_Status *status)
```

IN sendbuf - адреса початку надісланого повідомлення;
IN sendcount - число надісланих елементів;
IN sendtype - тип надісланих елементів;
IN dest - номер процесу, який отримує повідомлення;
IN sendtag - ідентифікатор надісланого повідомлення;
OUT recvbuf - адреса початку повідомлення, що приймається;
IN recvcount - максимальне число елементів, що приймаються;
IN recvtype - тип елементів повідомлення, що приймаються;
IN source - номер процесу-відправника;
IN recvtag - ідентифікатор повідомлення, що приймається;

IN comm - комунікатор області зв'язку;
OUT status - атрибути прийнятого повідомлення.

Функція MPI_Sendrecv суміщає виконання операцій передачі і прийому. Обидві операції використовують один і той же комунікатор, але ідентифікатори повідомлень можуть відрізнятися. Розміщення в адресному просторі процесу даних, що приймаються і передаються, не повинно перетинатися. Дані, що пересилаються, можуть бути різного типу й мати різну довжину.

У тих випадках, коли необхідний обмін даними одного типу із заміщенням надісланих даних тими, що приймаються, зручніше користуватися функцією **MPI_Sendrecv_replace.**

```
MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

INOUT buf - адреса початку повідомлення, що приймається;
IN count - число елементів, що передаються;
IN datatype - тип елементів, що передаються;
IN dest - номер процесу-одержувача;
IN sendtag - ідентифікатор надісланого повідомлення;
IN source - номер процесу-відправника;
IN recvtag - ідентифікатор повідомлення, що приймається;
IN comm - комунікатор області зв'язку;
OUT status - атрибути прийнятого повідомлення.

У даній операції надіслані дані з масиву buf заміщаються даними, що приймаються. Як адресати source і dest в операціях пересилання даних можна використовувати спеціальну адресу MPI_PROC_NULL. Комунікаційні операції з такою адресою нічого не роблять. Застосування цієї адреси буває зручним замість використання логічних конструкцій для аналізу умов посилати/читати повідомлення чи ні. Цей прийом буде використаний нами далі в одному з прикладів, а саме в програмі розв'язування рівняння Лапласа методом Якобі.

Неблокувальні комунікаційні операції

Використання неблокувальних комунікаційних операцій підвищує безпеку з огляду на виникнення тупикових

ситуацій, а також може збільшити швидкість роботи програми за рахунок поєднання виконання обчислювальних і комунікаційних операцій. Для цього комунікаційні операції розділяються на дві стадії: ініціацію операції і перевірку завершення операції.

Неблокувальні операції використовують спеціальний прихований (opaque) об'єкт "запит обміну" (request) для зв'язку між функціями обміну і функціями опитування їх завершення. Для прикладних програм доступ до цього об'єкта можливий тільки через виклики MPI-функцій. Якщо операція обміну завершена, підпрограма перевірки знімає "запит обміну", встановлюючи його в значення MPI_REQUEST_NULL. Зняти запит без очікування завершення операції можна підпрограмою MPI_Request_free.

MPI має набір підпрограм для одночасної перевірки на завершення декількох операцій. Без докладного обговорення приведемо їх перелік (табл. 4).

Таблиця 4. Функції колективного завершення неблокувальних операцій

Виконуванаперевірка	Функції очікування (блокувальні)	Функції перевірки(неблокувальні)
Завершилися операції	MPI_Waitall	MPI_Testall
Завершилася одна операція	MPI_Waitany	MPI_Testany
Завершилася одна із списку тих, що перевіряються	MPI_Waitsome	MPI_Testsome

Крім того, MPI дозволяє для неблокувальних операцій формувати цілі пакети запитів на комунікаційні операції MPI_Send_init і MPI_Recv_init, які запускаються функціями MPI_Start або MPI_Startall. Перевірка завершення виконання проводиться звичайними засобами за допомогою функцій сім'ї WAIT і TEST.

ДОДАТКИ

Додаток А. Зразок оформлення звіту

(Титульна сторінка)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЗ „ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА”

З В І Т

З лабораторної роботи № _

з дисципліни

«Паралельні та розподілені обчислення»

на тему: «_____»

Виконав(ла): здобувач(ка) освіти (Прізвище, ініціали)

групи___курсу, спеціальності

Перевірила: к.т.н., доцент Козуб Г.О.

Старобільськ -Поточний рік

(Продовження додатка А)

Структура оформлення звіту

Мета роботи:

Теоретична частина:

Відповіді на контрольні питання:

1.

Хід виконання роботи:

1.

Результати роботи

Результати роботи додаються до звіту в архівному файлі *ПРО_З_)__Призвіще_Лр№*

Висновки: На цій лабораторній роботі отримав(ла) основні навики _____. Навчився(лась) основних прийомів з _____. Виконав(ла) _____.

Додаток Б. Інструкція підключення MPI до Visual Studio

Для підключення MPI до проекту Visual Studio необхідно зробити наступні дії (шляхи можуть відрізнятися якщо буде змінено шлях за умовчанням під час встановлення):

Крок 1:

Необхідно налаштувати шляхи, для цього переходимо у вкладку Debug - Properties (Налагодження - %Ім'я_проекта% Властивості: (рис.Б1)

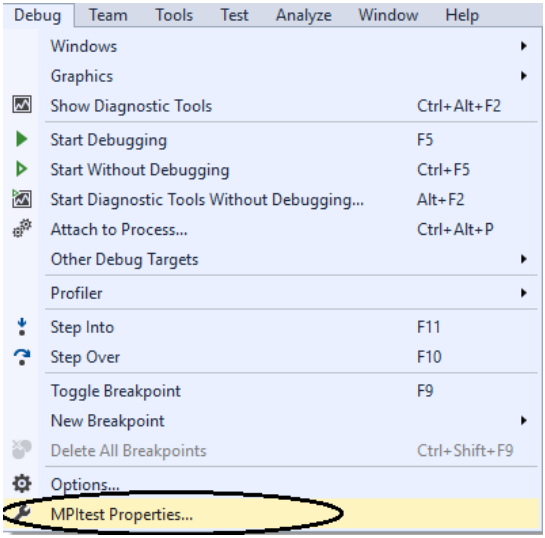


Рис. Б1.

Крок 2:Далі у вкладці **VC++ Directories (Каталоги VC++)** необхідно прописати у полі **Include Directories** (Включені каталоги):

```
“C:\Program Files (x86)\Microsoft SDKs\MPI\Include”
```

В полі Library Directories(Каталоги библиотек):

“C:\Program Files (x86)\Microsoft SDKs\MPIx64”

(Продовження додатка Б)

У полі з бібліотеками, якщо стоїть 32-розрядна версія, замість x64 потрібно прописати x86 (рис. Б2-Б3).

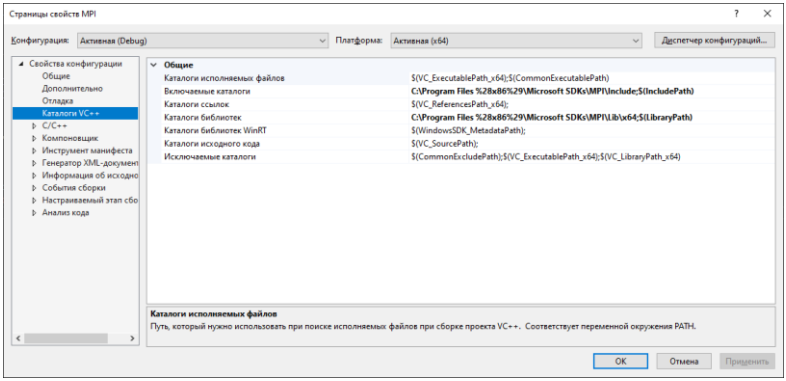


Рис. Б2.

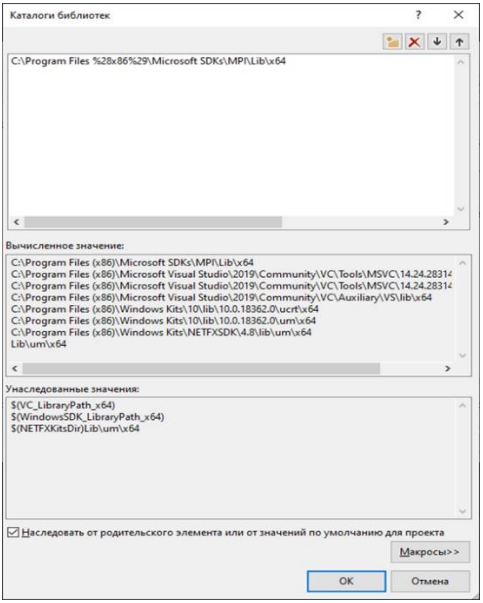


Рис. Б3

(Продовження додатка Б)

Крок 3: у вкладці **Linker – Input(Компоновщик – Ввод)** у полі **Additional Dependencies (Дополнительные зависимости)** необхідно вказати бібліотеку: **msmpi.lib** (рис.Б4)

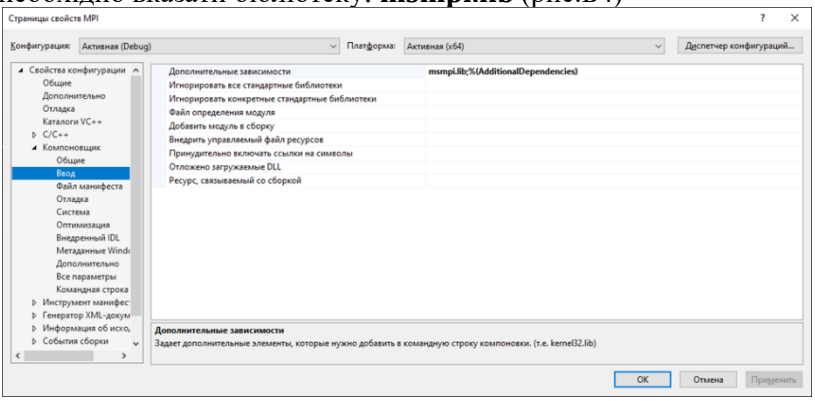


Рис. Б4

Крок 4: Для налаштування запуску необхідно перейти у вкладку **Debugging (Налагодження) та в** полі **Command (Команда)** вказати: "C:\Program Files\Microsoft MPI\Bin\mpiexec.exe"

В полі **Command Arguments (Аргументы команды)** вказати, наприклад, -n 4 $(TargetPath)

Число 4 вказує на кількість процесів (рис.Б5)

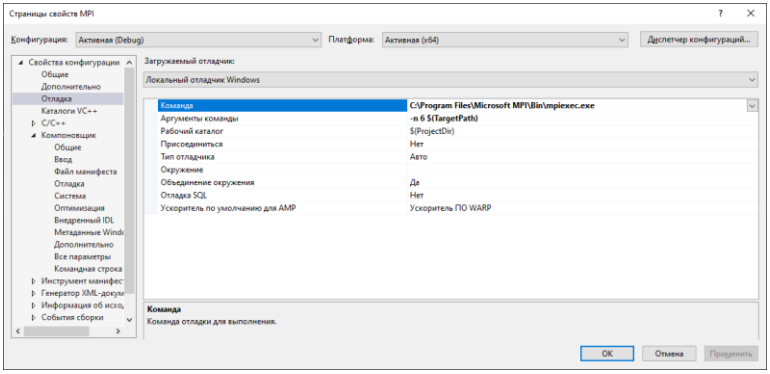


Рис.Б5

Додаток В. Віртуальні топології

У цьому пункті описується механізм віртуальних топологій MPI.

Функції створення декартових топологій

Функція, що конструює декартову топологію

MPI_CART_CREATE використовується для опису декартової структури довільного виміру. Для кожного напряму координати визначається, чи є структура процесу періодичною чи ні.

```
MPI_CART_CREATE(comm_old, ndims, dims, periods,
                                  reorder, comm_cart)
IN  comm_old     вхідний (старий) комунікатор
IN  ndims        кількість вимірювань у декартовій
                 топології
IN  dims         цілий масив розміром ndims,
                 визначальний кількість процесів у
                 кожному вимірі
IN  periods      масив розміром ndims логічних
                 значень, що визначають періодичність
                (true) чи ні (false) у кожному вимірі
IN  reorder      ранги можуть бути перенумеровані
                 (true) чи ні (falsee)
OUT comm_cart     комунікатор нової (створеної)
                  декартової  топології

int MPI_Cart_create(MPI_Comm comm_old, int ndims,
          int *dims, int *periods, int reorder, MPI
                Comm *comm_cart)
```

MPI_CART_CREATE повертає керування новому комунікатору, до якого приєднана інформація декартової топології. Ця функція колективна. Як у випадку з іншими колективними функціями, виклик цієї функції повинен бути синхронізований у всіх процесах. Якщо reorder = false, то номер кожного процесу в новій групі ідентичний його номеру в старій групі. Інакше, функція може пере впорядковувати процеси (можливо щоб вибрати гарне відображення віртуальної топології на фізичну топологію). Якщо повний розмір декартової сітки менший ніж розмір групи comm_old, то деякі процеси повертають MPI_COMM_NULL, за аналогією з

(Продовження додатка В)

MPI_COMM_SPLIT. Запит помилковий, якщо він визначає сітку, яка є більшою, ніж розмір групи comm_old.

0 (0, 0)	1 (0, 1)	2 (0, 2)	3 (0, 3)
4 (1, 0)	5 (1, 1)	6 (1, 2)	7 (1, 3)
8 (2, 0)	9 (2, 1)	10 (2, 2)	11 (2, 3)

Рис. В.1. Зв'язок між рангами та декартовими координатами для 3x4 2D топології. Верхній номер в кожній клітинці - номер процесу, а нижнє значення (рядок, стовпець) – координати

Декартова функція задання сітки

Для декартової топології, функція MPI_DIMS_CREATE допомагає користувачеві вибрати збалансовано розподіл процесів за напрямком координат, залежно від числа процесів в групі, і необов'язкових обмежень, які можуть бути визначені користувачем. Одне можливе використання цієї функції це розбиття усіх процесів (розмір групи MPI_COMM_WORLD) в N-мірну топологію.

```
MPI_DIMS_CREATE(nnodes, ndims, dims)
IN    nnodes       кількість вузлів в сітці
IN    ndims        розмірність декартової топології
INOUT dims         цілочисельний масив, що визначає
                   кількість вузлів в кожній
                   розмірності

int MPI_Dims_create(int nnodes, int ndims, int
    *dims)
```

(Продовження додатка В)

Елементи в масиві dims представляють опис декартової сітки з розмірностями ndims і загальною кількістю вузлів nnodes. Розмірності встановлюються так, щоб бути близько один до одного наскільки можливо, використовуючи відповідний алгоритм дільності. Користувач може обмежувати дію цієї функції, визначаючи елементи масиву dims. Якщо в dims[i] вже записано число, то функція не буде змінювати кількість вузлів в вимірі i; функція модифікує тільки нульові елементи в масиві, тобто де dims[i] = 0. Від'ємні значення елементів dims(i) помилкові. Помилка буде видалятися, якщо nnodes не кратне П dims[i].

(i,dims[i] 0)

Для dims[i] встановлених функцією, dims[i] будуть впорядковані в монотонно зменшуючому порядку. Масив dims підходить для використання, як вхід до функції MPI_CART_CREATE. Функція MPI_DIMS_CREATE локальна. Окремі типові запити показуються в прикладі 1.2

ПРИКЛАД 1.2

dims перед викликом	Функції	dims після повернення
(0,0)	MPI_DIMS_CREATE(6,2,dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7,2,dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6,3,dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(6,2,dims)	помилка

Декартові інформаційні функції

Якщо декартова топологія створена, може виникнути необхідність запитати інформацію стосовно цієї топології. Ці функції наведені нижче і всі вони локальні.

```
MPI_CARTDIM_GET(comm, ndims)
IN  comm        комунікатор з декартовою топологією
OUT ndims       розмірність декартової топології

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

(Продовження додатка В)

MPI_CARTDIM_GET повертає число вимірів декартової топології, пов'язаної з комунікатором comm. Комунікатор з топологією на рис. 1.2 повернув би ndims = 2.

```
MPI_CART_GET(comm, maxdims, dims, periods, coords)
IN  comm          комунікатор з декартовою топологією IN
maxdims       максимальний розмір масивів dims,
                  periods, і coords в викликаній
                  програмі
OUT dims          масив значень, що вказують
                  кількість процесів в кожному вимірі
OUT periods       масив значень, що вказують
                  періодичність (true/false) в
                  кожному вимірі
OUT coords        координати викликаного процесу в
                  декартовій топології

int MPI_Cart_get(MPI_Comm comm, int maxdims, int
                 *dims, int *periods, int *coords)
```

MPI_CARTDIM_GET повертає інформацію щодо декартової топології, пов'язаної з комунікатором comm. maxdims повинен бути принаймні дорівнює ndims, як повертає MPI_CARTDIM_GET. Наприклад на рис. 1.2, dims = (3,4), а в процесі 6 функція поверне coords = (1,2).

Декартові функції транслювання

Функції, наведені у цьому пункті переводять на з рангу в декартові координати топології. Ці запити локальні

```
MPI_CART_RANK(comm, coords, rank)
IN  comm        комунікатор з декартовою топологією
IN  coords      цілісний масив, що визначає
                координати необхідного процесу в
                декартовій топології
OUT rank        ранг потрібного процесу

int MPI_Cart_rank(MPI_Comm comm, int *coords, int
       *rank)
```

(Продовження додатка В)

Для групи процесів з декартовою структурою функція MPI_CART_RANK переводить логічні координати процесів у номери. Ці номери процесів використовують у парних взаємодіях між процесами. coords – масив розміром ndims як повертає MPI_CARTDIM_GET. Наприклад на рис. 1.2 процес з coords = (1,2) повернув би rank = 6. Для вимірювання і з periods(i) = true, якщо координата coords(i) знаходиться поза діапазоном, тобто coords(i) < 0 або coords(i) >= dims(i), вона переміщається назад до інтервалу 0 <= coords(i) < dims(i) автоматично. Якщо топологія на рис. 1.2 періодична в обох розмірностях, процес з coords = (4,6) також повернув би rank = 6. Для неперіодичних розмірностей діапазон поза координат помилковий.

```
MPI_CART_COORDS(comm, rank, maxdims, coords)
IN  comm          комунікатор з декартовою топологією
IN  rank          ранг процесу у торології comm
IN  maxdims       максимальний розмір масивів dims,
                  periods, та coords у програмі, що
                  викликає
OUT coords        цілісний масив, що визначає координати
необхідного процесу в декартовій топології

int MPI_Cart_coords(MPI_Comm comm, int rank, int
         maxdims, int *coords)
```

MPI_CART_COORDS - перетворює номер процесу в координати процесу в топології. Це зворотне відображення MPI_CART_RANK. maxdims можна взяти, наприклад, рівним ndims, що повертається MPI_CARTDIM_GET. Для прикладу на рис. 1.2, процес з rank = 6 поверне coords = (1,2).

Декартова функція зміщення

Якщо в декартовій топології, використовується функція MPI_SENDRECV (див. далі п. 3.2) для зміщення даних уздовж напрямку якої небуть координати, то вхідним аргументом MPI_SENDRECV береться номер процесу source (процесу

(Продовження додатка В)

джерела) для прийому даних, і номер процесу `dest` (процесу призначення) для передачі даних. Операція зміщення в декартовій топології визначається координатою зміщення та розміром кроку зміщення (позитивним або негативним). Функція `MPI_CART_SHIFT` повертає інформацію для вхідних специфікацій, потрібних для виклику `MPI_SENDRECV`. Функція `MPI_CART_SHIFT` локальна.

```
MPI_CART_SHIFT(comm, direction, disp, rank_source,
                  rank_dest)
IN  comm          комунікатор з декартовою топологією
IN  direction     напрямок номер виміру (в топології),
                  де робиться зміщення
IN  disp          напрямок зміщення (> 0: зміщення в
                  бік збільшення номерів кординати
                  direction, < 0: зміщення в бік
                  зменшення номерів кординати
                  direction)
OUT rank_source   ранг процесу джерела
OUT rank_dest     ранг процесу призначення

int MPI_Cart_shift(MPI_Comm comm, int direction, int
               disp, int *rank_source, int *rank_dest
```

Аргумент `direction` вказує вимір, у якому здійснюється зміщення даних. Вимірювання маркуються від 0 до `ndims`-1, де `ndims` - число розмірностей. `disp` вказує напрямок та величину зміщення. Наприклад, у топології "лінійка" або "кільце" з $N \geq 4$ процесами для процесу з номером 1 при `disp = 1` `rank_source = 0`, а `rank_dest = 2`; при `disp = -1` `rank_source = 2`, а `rank_dest = 0`. Для цього процесу 1 при `disp = 2` `rank_source = N-1` для "кільця" і `MPI_PROC_NULL` для "лінійки", а `rank_dest` = 3 для обох структур; при `disp = -2` `rank_source = 3` для обох структур, а `rank_dest = N-1` для "кільця" та MPI_PROC_NULL для "лінійки".

Залежно від періодичності декартової топології у вказаному напрямку координат, `MPI_CART_SHIFT` забезпечує

ідентифікатори `rank_source` і `rank_dest` для кільцевого або не кільцевого зміщення даних. Це вхідні аргументи до функції `MPI_SENDRECV`. Ні `MPI_CART_SHIFT`, а ні `MPI_SENDRECV` не колективні функції. Не потрібно, щоб усі процеси в декартових сітках одночасно викликали `MPI_CART_SHIFT` з тими самими `direction` і `disp` аргументами, але тільки той процес, який посилає відповідно, отримує в наступних запитах до `MPI_SENDRECV`.

Декартова функція розбиття

```
MPI_CART_SUB(comm, remain_dims, newcomm)
IN  comm         communicator декартової топології
IN  remain_dims  i-й елемент remain_dims визначає
                 відповідну i-ю розмірність що
                 включається в підрешітку (true) або
                 не включену (false)
OUT newcomm       communicator созданных подрешеток

int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
MPI_Comm *newcomm)
```

Якщо декартова топологія була створена функцією `MPI_CART_CREATE`, то може використовуватися функція `MPI_CART_SUB` для розбиття групи, пов'язаної комунікатором, на підгрупи, які формують декартові підрешітки меншої розмірності, і будувати для кожної такої підгрупи комунікатор, пов'язаний з підрешітками декартової топології. Цей запит колективний.

ПРИКЛАД 1.4

Припустимо, що `MPI_CART_CREATE(...,comm)` визначає (2 x 3 x 4) решітку. Допустимо remain_dims = (true, false, true). Тоді запит до `MPI_CART_SUB(comm, remain_dims, comm_new)` створить три комунікатори кожен із вісьмома процесами 2 x 4 в декартовій топології. Якщо `remain_dims =(false, false, true)`, то запит до `MPI_CART_SUB(comm, remain_dims, comm_new)` створить шість комунікаторів, що не перетинаються, кожен з чотирма процесами, в одновимірній декартовій топології.

Додаток Г. Основні директиви OpenMP

Основне джерело інформації - сервер http://www.openmp.org/. На сервері доступні специфікації, статті, навчальні матеріали, посилання.

Базовий потоковий паралелізм

Загальнодоступний процес пам'яті може складатися з множинних потоків, кілька ниток управління, які мають загальний адресний простір, але різні потоки команд і роздільні стеки. У найпростішому випадку, процес складається з однієї нитки. Нитки іноді називають також потоками, легковагими процесами, `LWP (light-weight processes)`. OpenMP заснований на існуванні множинних потоків у загальнодоступній пам'яті, що програмує парадигму.

Явний паралелізм

OpenMP має явну (не автоматичну) модель програмування, пропонуючи програмістові повне управління щодо розпаралелювання.

Модель Fork-Join (Розгалуження - Об'єднання)

OpenMP використовує Fork-Join модель паралельного виконання: Усі програми OpenMP починаються як єдиний процес: головний потік. Головний потік виконується послідовно, поки не стикаються з першою областю паралельної конструкції.

Fork (ВІТВЛЕННЯ): головний потік створює групу паралельних потоків. Інструкції в програмі, які включені паралельною конструкцією області{*регіону*}, тоді виконані паралельно серед різних потоків групи

Join (ОБ'ЄДНАННЯ): Коли потоки групи завершують інструкції в області паралельної конструкції, вони синхронізуються і закриваються, залишаючи тільки головний потік.

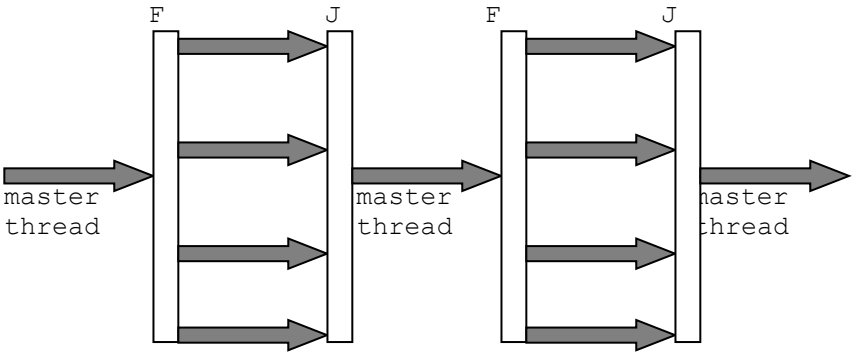


Рис. Г1. Модель Fork-Join

Директиви OpenMP

Директиви OpenMP з точки зору мови С є командами препроцесора:

```
#pragma omp
```

Фортрани є коментарями і починаються з комбінації символів:

```
!$OMP.
```

Формат директиви на C/C++:

```
<команда для препроцесора> <ім'я директиви>
<речення(klausa)>

#pragma omp parallel [clause clause ...]
   {
    .
    .
    .
   }
```

(Продовження додатка Г)

Директиви можна розділити на 3 категорії:

- визначення паралельної секції,
- поділ роботи,
- синхронізація.

Кожна директива може мати кілька додаткових атрибутів - клауз. Окремо специфікуються клаузи для призначення класів змінних, які можуть бути атрибутами різних директив.

Породження ниток

```
#pragma omp parallel [clause clause ...]
  {
   .
   .
   .
  }

!$OMP PARALLEL ... END PARALLEL
```

`clause: if(умова)` – виконання паралельної секції за умовою;

`private(list)` - список змінних локальних у кожній нитці;

`shared(list)` - список змінних загальних для кожної нитки;

`firstprivate(list)` - список змінних, які стають локальними у кожній нитці зі значеннями, раніше присвоєними цим змінним;

`copyin(list)` - список змінних (масивів), які визначені `#pragma omp threadprivate(list),` і які створюються у кожній нитці;

`reduction(operator:list)` - список змінних, з якими виконуються операції узагальнено по всіх нитках.

`list:` - список змінних;

`оператор: – +,-,:,*, . . . ..`

Визначає паралельну область програми. При вході в цю область породжуються нові (N-1), утворюється "команда" з N ниток, а нитка, що породжує, отримує номер 0 і стає основною

ниткою команди (так звана "master thread"). При виході з паралельної області основна нитка чекає завершення інших ниток, і продовжує виконання в одному екземплярі. Передбачається, що в SMP-системі нитки будуть розподілені по різних процесорах (однак це, як правило, перебуває у віданні операційної системи).

Яким чином між породженими нитками розподіляється робота - визначається директивами DO,SECTIONS і SINGLE. Можливе також явне керування розподілом роботи (як і MPI) за допомогою функцій, що повертають номер поточної нитки і загальне число ниток. За замовчуванням (поза цими директивами), код всередині PARALLEL виконується всіма нитками однаково.

Паралельні області можуть бути динамічно вкладеними. За замовчуванням (якщо вкладений паралелізм не дозволено явно), внутрішня паралельна область виконується однією ниткою.

Поділ роботи (work-sharing constructs)

Паралельні цикли

```
#pragma omp parallel [clause clause ...]
  { . . .
    . . .
#pragma omp for [clause clause ...]
  { . . .
    . . .
  }
#pragma omp for [clause clause ...]
  { . . .
  }
 . . .
 . . .
}
```

(Продовження додатка Г)

```
!$OMP DO ... [ENDDO]

clause: schedule(type[,chink])
          замовлено
          private(list)
          shared(list)
          firstprivate(list)
          lastprivate(list)
          reduction(operator:list)
          nowait
```

Визначає паралельний цикл.

Клауза `schedule` визначає спосіб розподілу ітерацій по нитках:

`static,m` - статично, блоками по m ітерацій;

`dynamic,m` - динамічно, блоками по m (кожна нитка бере на виконання перший ще невзятий блок ітерацій)

`guided,m` - розмір блоку ітерацій зменшується експоненціально до величини m

`runtime` - вибирається під час виконання.

`ordered` - (для циклів) реалізується послідовне виконання витків циклу, як у послідовному алгоритмі.

`lastprivate(list)` - змінним присвоюється результат останнього витка циклу.

За замовчуванням, наприкінці циклу відбувається неявна синхронізація; цю синхронізацію можна заборонити за допомогою `nowait (ENDDO NOWAIT)`.

Паралельні секції

```
#pragma omp parallel [clause clause ...]
  { . . .
    . . .
#pragma omp sections [clause clause ...]
  { . . .
    . . .
#pragma omp section
```

(Продовження додатка Г)

```
  { . . .
    . . .
  }
#pragma omp section
   { . . .
    . . .
   }
  } } // - кінцеві секції
 . . .
 . . .
 }

!$OMP СЕКЦІЇ ... END SECTIONS

clause: private(list)
          firstprivate(list)
          lastprivate(list)
          reduction(operator:list)
          nowait
```

Не-ітеративна паралельна конструкція. Визначає набір незалежних секцій коду (т.зв., "кінцевий" паралелізм). Секції відокремлюються одна від одної директивою **`SECTION`**.

Примітка. Якщо всередині parallel міститься тільки одна конструкція `for` або тільки одна конструкція `section`, то можна використовувати скорочений запис: `parallel for` або `parallel section`.

Виконання однією ниткою

```
#pragma omp parallel [clause clause ...]
  { . . .
    . . .
#pragma omp single [clause clause ...]
  { . . .
    . . .
```

(Продовження додатка Г)

```
#pragma omp single [clause clause ...]
  { . . .
    . . .
  }
 }

!$OMP SECTIONS... END SECTIONS

clause: private(list)
           firstprivate(list)
           nowait

SINGLE ... END SINGLE
```

Визначає блок коду, який буде виконаний тільки однією ниткою (першою, яка дійде до цього блоку).

Явне управління розподілом роботи

За допомогою функцій `omp_get_thread_num()` і `omp_get_num_threads()` нитка може дізнатися свій номер і загальну кількість ниток, а потім виконувати свою частину роботи залежно від свого номера (цей підхід широко використовується в програмах на базі інтерфейсу MPI).

Приклад **.** Простий приклад: обчислення числа "Пі". У послідовну програму вставлено два рядки, і вона розпаралелена!

C:

```
#include<omp.h>
#include<stdio.h>

double f(double y)
  { return(4.0/(1.0+y*y));
  }

 main()
  { double w,x,sum,pi;
    int i;
```

(Продовження додатка Г)

```
    int n = 1000;
    w = 1,0/n;
    sum = 0.0;
#pragma  omp  parallel  for  schedule(static,n/2)
private(i,x) \
shared(w) reduction(+:sum)
   for(i=0; i < n; i++)
      { x = w*(i-0.5);
       sum = sum + f(x);
      }
   pi = w*sum;
   printf("pi = %f\n",pi);

 return(0);
 }
```

Тут змінна sum не повинна описуватися в `private(i,x)`, але може бути описана в `shared(w)`.

Фортран:

```
program compute_pi
   parameter (n = 1000)
   integer i
   double precision w,x,sum,pi,f,a
   f(a) = 4.d0/(1.d0+a*a)
   w = 1.0d0/n
   sum = 0.0d0;
!$OMP PARALLEL DO PRIVATE(x) SHARED(w)
REDUCTION(+:sum)
   do i=1,n
       x = w*(i-0.5d0)
       sum = sum + f(x)
   enddo

   pi = w*sum
   print *,'pi = ',pi
   stop
   end
```

(Продовження додатка Г)

Директиви синхронізації

```
#pragma omp master
  {
   .
  }
MASTER ... END MASTER
```

Визначає блок коду, який буде виконано тільки `master`-ом (нульовою ниткою). Вхід/вихід із критичного інтервалу заборонені.

```
#pragma omp critical[name]
  {
   .
  }

CRITICAL ... END CRITICAL
```

Визначає критичну секцію, тобто блок коду, який не повинен виконуватися одночасно двома або більше нитками.
`name` - ім'я критичного інтервалу. Ім'я - глобальний ідентифікатор. Різні критичні інтервали з одним ім'ям обробляються як одна й та сама область (критичний інтервал). Вхід/вихід із критичного інтервалу заборонені.

```
#pragma omp barrier

BARRIER
```

Визначає точку бар'єрної синхронізації, в якій кожна нитка чекає на всіх решти. Найменша інструкція, яка містить `barrier`, має бути структурним блоком. `barrier` не може бути в `for` і `sections`.

Неправильно:

```
for(x ==0)
   #pragma omp barrier
```

(Продовження додатка Г)

Правильно:

```
for(x ==0)
  {   #pragma omp barrier
  }
#pragma omp atomic
  x binop = exper;

 x++
++x
x--
--x
binop - +,*,-,/,&,^,\,>>,<<,or;
exper – скалярное выражение.
```

ATOMIC

Визначає змінну в лівій частині оператора "атомарного" присвоювання, яка має коректно оновлюватися кількома нитками.

```
#pragma omp ordered
```

ORDERED ... END ORDERED

Визначає блок усередині тіла циклу, який має виконуватися в тому порядку, в якому ітерації йдуть у послідовному циклі. Може використовуватися для впорядкування виведення від паралельних ниток.

```
#pragma omp flush(list)
```

FLUSH

Явно визначає точку, в якій реалізація повинна забезпечити однаковий вид пам'яті для всіх ниток. Неявно FLUSH присутній у таких директивах: BARRIER, CRITICAL, END CRITICAL, DO, END DO, PARALLEL, END PARALLEL, SECTIONS, END SECTIONS, SINGLE, END SINGLE, ORDERED, END ORDERED. З метою синхронізації можна також користуватися механізмом замків (locks).

Класи змінних

В OpenMP змінні в паралельних областях програми поділяються на два основні класи:

SHARED (спільні; під ім'ям А всі нитки бачать одну змінну) і

PRIVATE (приватні; під ім'ям А кожна нитка бачить свою змінну).

Окремі правила визначають поведінку змінних під час входу і виходу з паралельної області або паралельного циклу: REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN.

За замовчуванням, усі COMMON-блоки, а також змінні, породжені поза паралельною областю, при вході в цю область залишаються загальними (SHARED). Виняток становлять змінні - лічильники ітерацій у циклі, з очевидних причин. Змінні, породжені всередині паралельної області, є приватними (PRIVATE). Явно призначити клас змінних за замовчуванням можна за допомогою клаузи DEFAULT.

```
shared(list)
```

SHARED – Застосовується до змінних, які необхідно зробити загальними.

```
private(list)
```

PRIVATE – Застосовується до змінних, які необхідно зробити приватними. При вході в паралельну область для кожної нитки створюється окремий екземпляр змінної, який не має жодного зв'язку з оригінальною змінною поза паралельною областю.

```
#pragma omp thredprivate(list)
```

THREADPRIVATE – Застосовується до COMMON-блоків, які необхідно зробити приватними. Директива повинна застосовуватися після кожної декларації COMMON-блока.

(Продовження додатка Г)

```
double A[][],B[];
#pragma omp thredprivate(A,B)
```

`firstprivate`

`FIRSTPRIVATE` – Приватні копії змінної при вході в паралельну область ініціалізуються значенням оригінальної змінної.

`lastprivate`

`LASTPRIVATE` – Після закінчення паралельного циклу або блоку паралельних секцій, нитка, яка виконала останню ітерацію циклу або останню секцію блоку, оновлює значення оригінальної змінної.

`reduction(+:A)`

`REDUCTION(+:A)` – Позначає змінну, з якою в циклі проводиться `reduction`-операція (наприклад, підсумовування). При виході з циклу, ця операція проводиться над копіями змінної у всіх нитках, і результат присвоюється оригінальній змінній. Змінні в цьому списку не повинні описуватися в `private(list)`, але можуть бути описані в `shared(list)`.

`copyin(list)`

`COPYIN` – Застосовується до `COMMON`-блоків, які позначені як threadprivate. При вході в паралельну область приватні копії цих даних ініціалізуються оригінальними значеннями.

Runtime-процедури та змінні середовища

З метою створення переносимого середовища запуску паралельних програм, в OpenMP визначено низку змінних середовища, які контролюють поведінку програми.

(Продовження додатка Г)

В OpenMP передбачено також набір бібліотечних процедур, які дають змогу:

– під час виконання контролювати і запитувати різні параметри,

– що визначають поведінку застосунку (такі як кількість ниток і процесорів, можливість вкладеного паралелізму);

– процедури призначення параметрів мають пріоритет над відповідними змінними середовища.

– використовувати синхронізацію на базі замків `(locks)`.

Змінні середовища

```
export OMP_SCHEDULE "guided,4"
export OMP_SCHEDULE "dynamic"
```

Визначає спосіб розподілу ітерацій у циклі, якщо в директиві DO використано клаузу `SCHEDULE(RUNTIME)`.

```
export OMP_NUM_THREADS=n ( = 4, setenv )
```
Визначає число ниток для виконання паралельних областей програми.

```
export OMP_DYNAMIC TRUE
```
Дозволяє або забороняє динамічну зміну числа ниток.

```
export OMP_NESTED TRUE
```
Дозволяє або забороняє вкладений паралелізм.

Процедури для контролю/запиту параметрів середовища виконання

```
omp_set_num_threads(int num)
```

`OMP_SET_NUM_THREADS` – Дозволяє призначити максимальну кількість ниток для використання в наступній паралельній області (якщо це число дозволено змінювати динамічно).

(Продовження додатка Г)

Викликається з послідовної області програми.

`omp_get_max_threads()`

`OMP_GET_MAX_THREADS` – Повертає максимальне число ниток.

`omp_get_num_threads()`

`OMP_GET_NUM_THREADS` – Повертає фактичне число ниток у паралельній області програми.

`omp_get_num_procs()`

`OMP_GET_NUM_PROCS` – Повертає число процесорів, доступних додатку.

`omp_in_parallel()`

`OMP_IN_PARALLEL` – Повертає `.TRUE`, якщо викликана з паралельної області програми.

`omp_set_dunamic(int dynamic_threads)/omp_get_dinamic()`

`OMP_SET_DYNAMIC` / `OMP_GET_DYNAMIC` – Встановлює/запитує стан прапора, що дає змогу динамічно змінювати число ниток.

`omp_set_nested(int nested)/omp_get_nested()`

`OMP_GET_NESTED` / `OMP_SET_NESTED` – Встановлює/запитує стан прапора, що дозволяє вкладений паралелізм.

(Продовження додатка Г)

Процедури для синхронізації на базі замків

Як замки використовуються загальні змінні типу `INTEGER` (розмір повинен бути достатнім для зберігання адреси). Дані змінні повинні використовуватися тільки як параметри примітивів синхронізації.

```
omp_init_lock(omp_lock_t                    *lock)/
omp_destroy_lock(omp_lock_t *lock)
```

`OMP_INIT_LOCK(var)` / `OMP_DESTROY_LOCK(var)` - Ініціалізує замок, пов'язаний зі змінною var.

```
omp_set_lock(omp_lock_t *lock)
```

`OMP_SET_LOCK(VAR)` – Змушує нитку, що викликала, дочекатися звільнення замка, а потім захоплює його.

```
omp_unset_lock(omp_lock_t *lock)
```

`OMP_UNSET_LOCK(VAR)` – Звільняє замок, якщо він був захоплений ниткою, що його викликала.

```
omp_test_lock(omp_lock_t *lock)
```

`OMP_TEST_LOCK(VAR)` – Пробує захопити вказаний замок. Якщо це неможливо, повертає `.FALSE.`

Специфікація OpenMP для мов C/C++

Специфікація OpenMP для C/C++, випущена на рік пізніше фортранної, містить взагалі аналогічну функціональність.

Необхідно лише зазначити такі моменти:

1) Замість спецкоментарів використовуються директиви компілятора `"#pragma omp"`.

2) Компілятор з підтримкою OpenMP визначає макрос `"_OPENMP"`, який може використовуватися для умовної

компіляції окремих блоків, характерних для паралельної версії програми.
3) Розпаралелювання застосовується до `for`-циклів, для цього використовується директива `"#pragma omp for"`. У паралельних циклах забороняється використовувати оператор `break`.
4) Статичні `(static)` змінні, визначені в паралельній області програми, є спільними `(shared)`.
5) Пам'ять, виділена за допомогою `malloc()`, є спільною (однак покажчик на неї може бути як загальним, так і приватним).
6) Типи та функції OpenMP визначено у включаємому файлі `<omp.h>`.
7) Крім звичайних, можливі також "вкладені" `(nested)` замки - замість логічних змінних використовуються цілі числа, і нитка, що вже захопила замок, при повторному захопленні може збільшити це число.

Приклад розпаралелювання `for`-циклу в C

```
#pragma omp parallel for private(i)
#pragma omp shared(x, y, n) reduction(+: a, b)
for (i=0; i<n; i++)
{
     a = a + x[i];
     b = b + y[i];
}
```

Навчальний посібник

Укладачі
КОЗУБ Галина Олександрівна
КОЗУБ Владислав Юрійович

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Методичні рекомендації до виконання лабораторних робіт для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 122 „ Комп'ютерні науки ”

Редактор – КОЗУБ Г. О.
Комп'ютерний макет – КОЗУБ В.Ю.

Здано до склад. 01.12.2021 р. Підп. до друку 22.12.2021 р.
Формат 60х84 1/16. Папір офсет. Гарнітура Times New Roman.
Друк ризографічний. Ум. друк. арк. 6,8. Наклад 50 прим. Зам. № 00.

Видавець і виготовлювач
Видавництво Державного закладу
„Луганський національний університет імені Тараса Шевченка”
пл. Гоголя, 1, м. Старобільськ, 92703. Тел./факс: (06461) 2-26-70.
e-mail: mail@luguniv.edu.ua
Свідоцтво суб'єкта видавничої справи ДК № 3459 від 09.04.2009 р.