МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КРЕМЕНЧУЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ МИХАЙЛА ОСТРОГРАДСЬКОГО

МЕТОДИЧНІ ВКАЗІВКИ
ЩОДО ВИКОНАННЯ РОЗРАХУНКОВО-ГРАФІЧНОЇ РОБОТИ
З НАВЧАЛЬНОЇ ДИСЦИПЛІНИ
«ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ»
ДЛЯ СТУДЕНТІВ ДЕННОЇ ФОРМИ НАВЧАННЯ
ЗІ СПЕЦІАЛЬНОСТІ 123 – «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ»

КРЕМЕНЧУК 2018

Методичні вказівки щодо виконання розрахунково-графічної роботи з навчальної дисципліни «Паралельні та розподілені обчислення» для студентів денної форми навчання зі спеціальності 123 – «Комп'ютерна інженерія»

Укладач к. т. н., доц. О. Г. Славко

Рецензент к. т. н., доц. В. М. Сидоренко

Кафедра комп'ютерних та інформаційних систем

Затверджено методичною радою КрНУ імені Михайла Остроградського
Протокол № 11 від 9 липня 2018 р.

Голова методичної ради _____ проф. В. В. Костін

ЗМІСТ

ВСТУП

Цей навчальний курс є логічним продовженням навчальних курсів «Комп'ютерні системи» і «Програмування». Предметом вивчення навчальної дисципліни є системні рішення, що покладені в основу побудови паралельних алгоритмів, орієнтованих для реалізації на багатопроцесорних обчислювальних системах, а також методи та підходи до організації паралельних обчислень і способи організації обчислювальних кластерів і архітектури багатопроцесорних систем.

У межах навчального курсу розглядаються питання дослідження особливостей функціонування сучасних комп'ютерних систем, орієнтованих на організацію та реалізацію паралельних обчислень. Студенти знайомляться з основами розробки ефективних паралельних алгоритмів, використання паралелізму програм та ін. Також набувають практичних навичок з використання технологій паралельного програмування використовуючи при цьому мову C++ та бібліотеку MPI (Message passing interface), що надає низькорівневий, але в той же час надзвичайно зручний інтерфейс програмування для мережного кластера та базується на ідеології обміну повідомленнями між рівнобіжними процесами, знайомляться з інтегрованим середовищем розробки паралельних програм AdaGIDE.

Паралельні обчислення – це такий спосіб організації комп'ютерних обчислень, за якого програми розробляються як набір взаємодіючих обчислювальних процесів, що працюють паралельно (одночасно). Термін охоплює сукупність питань паралелізму в програмуванні, а також створення ефективних апаратних реалізацій. Теорія паралельних обчислень становить розділ прикладної теорії алгоритмів.

Основна складність у проектуванні паралельних програм – це забезпечення правильної послідовності взаємодій між різними обчислювальними процесами, а також координацію ресурсів, поділюваних між процесами.

Мета та завдання навчальної дисципліни: набуття студентами загальних теоретичних і практичних знань у галузі паралельної обробки інформації, а також з питань, пов'язаних з аналізом методів і алгоритмів паралельної обробки інформації, дослідженням моделей і мов паралельних обчислень, ознайомлення з методами та алгоритмами паралельних обчислень, оволодіння навичками організації паралельної та розподіленої обробки інформації в суперкомп'ютерах та паралельних системах.

Місце навчальної дисципліни у навчальному процесі: навчальний курс «Паралельні та розподілені обчислення» тісно пов'язаний з такими дисциплінами, як «Комп'ютерні системи», «Мережні інформаційні технології», «Дискретна математика», «Програмування» та іншими спеціальними дисциплінами навчального плану.

У результаті вивчення навчальної дисципліни студент повинен:

знати:

– теоретичні та практичні аспекти паралельної обробки інформації;

– особливості організації обчислювальних процесів у паралельних обчислювальних системах;

– структури та принципи побудови програмного забезпечення розподілених систем обробки інформації;

– моделі системи планування обчислювальних процесів;

уміти:

– орієнтуватись у класах сучасних суперкомп'ютерів і моделях паралельної обробки інформації;

– обирати модель, метод та алгоритм, найбільш адекватний для ефективної паралельної обробки для розв'язання того чи іншого класу задач;

– створювати додатки користувача для паралельної обробки інформації із застосуванням спеціалізованих бібліотек і однієї з мов паралельної обробки інформації.

1 ПЕРЕЛІК ЗАВДАНЬ НА РОЗРАХУНКОВО-ГРАФІЧНУ РОБОТУ

Завдання № 1

Тема. Компіляція та запуск програм на обчислювальному кластері

Мета: вивчення та практичне випробовування основних команд компіляції та запуску програм на обчислювальному кластері, дослідження ефективності виконання послідовної програми на однопроцесорному комп'ютері, отримання навичок аналізу інформаційної структури обчислювальних алгоритмів і вибору методів їх розпаралелювання.

Короткі теоретичні відомості

Для того, щоб написати програму, використовуючи елементи паралельного програмування, зокрема, застосовуючи бібліотеку MPI для обміну повідомленнями між процесами проектованого алгоритму, слід мати повну інформацію про середовище, у якому ця програма виконуватиметься.

Основні команди для вивчення оточуючого середовища в ОС Linux:

chown – зміна хазяїна файлу;

batch – виконати команду під час завантаження;

finger/who/users – вивести список працюючих у системі;

kill – зупинити процес;

ls – перегляд каталогу;

chmod – зміна привілеїв файлового доступу;

id – перегляд прав, тобто uid та gid;

logname – отримання імені реєстрації;

uname-a – усе про версії системи;

crontab – задати резерви часу між запусками програм;

ps – ознайомлення зі списком процесів;

sleep – призупинити процес;

passwd – робота з паролем;

uux – виконати команди на віддаленому комп'ютері;

nslookup – мережні настройки;

greg – пошук строки у файлі за заданим ключем;

greg -i – те саме, без урахування реєстру;

chmod – зміна прав доступу до файлу;

cmp – порівнює два файли;

df – показує всі змонтовані драйвери на машині;

diff – показує відмінності між двома файлами;

stty – настройка терміналу;

who – показати, хто в системі;

write – написати повідомлення іншому користувачу;

id – перегляд прав доступу;

uname – дані про систему;

uuname – список хостів.

Компіляція вихідного файла

Компілятор мови С називається gcc. Під час компіляції вихідного файлу потрібно вказувати опцію -c. От як, наприклад, компілюється файл main.c:

```
gcc -c main.c
```

Отриманий об’єктний файл буде називатися main.o.

Компілятор мови C++ називається g++ і працює майже так само, як і gcc:

```
g++  -c  main.cpp
```

Опція -c указує компілятору про необхідність одержати на виході об’єктний файл (він буде називатися reciprocal.o). Без неї компілятор g++ спробує скомпонувати програму і створити бінарний файл.

Компілятори gcc і g++ приймають безліч різних опцій. Одержати їхній повний список можна в інтерактивній документації за допомогою команди:

```
info gcc
```

Компонування об’єктних файлів

Після того як файли main.c скомпільовано, необхідно скомпонувати його. Якщо до проекту входить хоча б один файл C++, компонування завжди здійснюється за допомогою компілятора g++. Якщо ж усі файли написані

мовою C, потрібно використовувати компілятор gcc. У цьому випадку є файли обох типів, тому необхідна команда має такий вигляд:

```
g++ -o main main.o
```

Опція -o задає ім’я файлу, створюваного в процесі компонування. Тепер можна здійснити запуск програми:

```
./main
```

Автоматизація процесу за допомогою GNU-утиліти make

Для утиліти make необхідно вказати цільові модулі, що беруть участь у процесі побудови бінарного файлу, і правила, за якими відбувається цей процес. Також задаються залежності, що визначають, коли конкретний цільовий модуль повинен бути перебудований.

Окрім цільових модулів також повинен існувати модуль clean, що призначений для видалення всіх згенерованих об’єктних файлів і програм. Правило для даного модуля включає команду rm, що видаляє названі файли.

Щоб передати всю цю інформацію утиліті make, необхідно створити файл Makef i 1e.

Хід роботи

1. Переглянути вміст домашнього каталогу і структуру файлової системи.
2. Переглянути наявність інших користувачів у системі.
3. Одержати інформацію про всі процеси, що виконуються, і окремо про процеси даного користувача.
4. Написати, відкомпілювати і настроїти послідовну програму, що здійснює перемножування матриць.
5. Дослідити ефективність виконання програми перемножування матриць для випадків різної розмірності матриць (із зазначенням графічних результатів залежності часу виконання програми від розмірності матриці).
6. Дослідити інформаційну структуру програми перемножування матриць, запропонувати різні методи розподілу операцій між процесорами, вибрати один для реалізації.

Зміст звіту

1. Назва, мета завдання.

2. Результати дослідження середовища оточення користувача.

3. Лістинги та результати роботи програм.

4. Графік залежності часу виконання програми від розмірності матриці.

5. Результати дослідження інформаційної структури програми перемножування матриць і обґрунтування вибору методу розпаралелювання операцій між процесорами.

6. Письмові відповіді на контрольні питання.

Контрольні питання

1. Якими об’єктами маніпулюють операційні системи класу UNIX?

2. Що являє собою оточення процесу?

3. Назвіть основні характеристики користувачів і процесів.

4. Сформулюйте загальні принципи паралелізму.

5. Перелічіть основні команди керування процесами.

Література: [1, 2, 4–10].

Завдання № 2

Тема. Створення паралельних програм з використанням бібліотеки Pthread

Мета: набуття практичних навичок компілювання та налагодження рівнобіжних програм у середовищі ОС Linux, дослідження ефективності виконання рівнобіжної програми на однопроцесорному комп’ютері.

Короткі теоретичні відомості

Поняття процесу. Концептуально потік існує усередині процесу та є меншою одиницею керування програмою. Під час виклику програми Linux

створює для неї новий процес, а в ньому – єдиний потік, що послідовно виконує програмний код. Цей потік може створювати додаткові потоки. Усі вони знаходяться в одному процесі, виконуючи ту ж саму програму, але, можливо, у різних її місцях.

У Linux реалізована бібліотека API-функцій роботи з потоками, що відповідає стандарту POSIX (вона називається Pthreads). Усі функції і типи даних бібліотеки оголошені у файлі <pthread. h>. Ці функції не належать до стандартної бібліотеки мови С, тому для компонування програми потрібно вказувати опцію –lpthread у командному рядку.

Створення потоку

Кожному потоку в процесі призначається власний ідентифікатор. Посилаючись на ідентифікатори потоків, потрібно використовувати тип даних pthread_t.

Функція pthread_create() створює новий потік. Їй передаються наступні параметри:

– вказівник на перемінну типу pthread_t, у якій зберігається ідентифікатор нового потоку;

– вказівник на об’єкт атрибутів потоку. Цей об’єкт визначає взаємодію потоку з іншою частиною програми. Якщо задати його рівним NULL, потік буде створений зі стандартними атрибутами;

– вказівник на потокову функцію;

– значення аргументу потоку (тип void*). Це значення без будь-яких змін передається потоковій функції.

Функція pthread_create () негайно завершується та батьківський потік переходить до виконання інструкції, що виникає після виклику функції. Водночас новий потік починає виконувати потокову функцію (звичайна функція, що містить код потоку). ОС Linux планує роботу обох потоків асинхронно, тому програма не повинна розраховувати на якусь узгодженість між ними.

Завершення потоку

```
void pthread_exit(void *inReturnValue)
```

Виклик цієї функції приводить до завершення потоку. Процес-батько одержує вказівник унаслідок виконання функції.

Звичайне завершення функції і повернення вказівника на void*, еквівалентно виклику функції pthread_exit, що використовується, якщо потрібно завершити потік з функцій, викликаних цією функцією.

Передача даних потоку

Потоковий аргумент – це зручний засіб передачі даних потокам. Але оскільки його тип void*, дані містяться не в самому аргументі. Він лише повинний указувати на структуру чи масив. Найкраще створити для кожної потокової функції власну структуру, у якій визначалися б «параметри», очікувані потоковою функцією.

Обробка завершення потоку

```
int pthread_join( pthread_t inHandle, void **outReturnValue);
```

Виклик цієї функції приводить до блокування батьківського потоку до моменту завершення потоку-нащадка, що відповідає індентифікатору inHandle. В область, зазначену параметром outReturnValue, записується вказівник, повернутий потоком.

`pthread_join` призводить до звільнення ресурсів, відведених на потік. Його також необхідно виконувати також для синхронізації основного потоку і потоків-нащадків.

Семафори

Семафорами називаються загальні змінні цілого типу. Для роботи з семафорами передбачені дві неподільні операції:

– збільшити значення семафора на 1;

– дочекатися, поки значення семафора буде позитивним, і зменшити значення семафора на 1.

Підтримка семафорів у бібліотеці pthreads

```
sem_post(sem_t* semaphor, int flag, int value)
sem_t – тип семафора
```

```
semaphor – семафор,
flag – прапор (0 – усередині процесу, 1 – між процесами)
value – початкове значення
sem_post(sem_t* semaphor) – збільшення семафора
sem_wait(sem_t* semaphor) – зменшення семафора
```

Підтримка критичних секцій у pthreads

Критичною секцією називається фрагмент коду програми, що може одночасно виконуватися тільки одним потоком.

«М'ютекс» – mutex (mutual exclusion, взаємне виключення).

Оголошення та ініціалізація:

```
pthread_mutex_t – тип для взаємного виключення;
pthread_mutex_init(pthread_mutex_t* mutex, void* attribute);
pthread_mutex_destroy(pthread_mutex_t* mutex);
```

Захоплення та звільнення мютекса:

```
pthread_mutex_lock(pthread_mutex_t* mutex);
pthread_mutex_unlock(pthread_mutex_t* lock);
```

Звільнення м'ютекса може бути здійснено тільки потоком, що й захопив його.

Приклад множення матриць

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_mutex_t mut;
static int N, nrow;
static double *A, *B, *C;
void setup_matrices ()
{
  int i, j;
  A = malloc (N * N * sizeof (double));
  B = malloc (N * N * sizeof (double));
  C = malloc (N * N * sizeof (double));
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++){
      A[i * N + j] = 1;
      B[i * N + j] = 2;
    }
}
void print_result ()
{
…
}
void * worker (void *arg)
{
  int i, j;
```

```
  while (nrow < N)
    {
      int oldrow;
      pthread_mutex_lock (&mut);
      oldrow = nrow;
      nrow++;
      pthread_mutex_unlock (&mut);
  for (i = 0; i < N; i++)
      {
        int j;
        double t = 0.0;

        for (j = 0; j < N; j++)
          t += A[oldrow * N + j] * B[j * N + i];
        C[oldrow * N + i] = t;
      }
    }
  return NULL;
}
main (int argc, char *argv[])
{
  int i, nthreads;
   pthread_t *threads;
  pthread_mutex_init(&mut, NULL);
  nthreads = atoi (argv[1]);
  threads = malloc (nthreads * sizeof (pthread_t));
  N = atoi (argv[2]);
  setup_matrices();
  for (i = 0; i < nthreads; i++)
    pthread_create (threads + i, NULL, worker, NULL);
   for (i = 0; i < nthreads; i++)
    pthread_join (threads[i], NULL);
  if (argc > 3)
    print_result ();
   pthread_mutex_destory(&mut); }
```

Хід роботи:

1. Налагодити і відкомпілювати приведений приклад з використанням бібліотеки PTHREADS.

2. Написати і відлагодити рівнобіжну програму, що робить перемножування матриць використовуючи системний виклик FORK.

3. Дослідити ефективність виконання програми перемноження матриць для випадків різної розмірності матриць для обох варіантів програм (із зазначенням графічних результатів залежності часу виконання програми від розмірності матриці).

Зміст звіту

1. Назва, мета завдання.
2. Результати дослідження користувальницького середовища.
3. Лістинги і результати роботи програм.
4. Графік залежності часу виконання програми від розмірності матриці.
5. Порівняння результатів дослідження інформаційної структури програми перемножування матриць для послідовної (з попереднього завдання) та паралельної програм.
6. Письмові відповіді на контрольні питання.

Контрольні питання

1. Яка різниця між потоком та процесом? Які ви знаєте характеристики потоків?
2. Що таке семафори?
3. Коли використовують критичні секції?

Література: [1–3, 7–12].

Завдання № 3

Тема. Дослідження основних функцій MPI

Мета: набуття практичних навичок побудови простих паралельних програм, написаних за допомогою інтерфейсу програмування MPI, дослідження ефективності реалізації різних видів пересилань даних.

Короткі теоретичні відомості

Система програмування MPI відноситься до класу MIMD з індивідуальною пам’яттю, тобто до багатопроцесорних систем з обміном повідомленнями. MPI має такі особливості:

– MPI – бібліотека, а не мова. Вона визначає імена, виклики процедур і

результати їхньої роботи. Програми, що пишуться на FORTRAN, C, і C++ компілюються звичайними компіляторами, але пов’язані з MPI-бібліотекою.

– MPI – опис, а не реалізація. Усі постачальники рівнобіжних комп’ютерних систем пропонують реалізації MPI для своїх машин як безкоштовні, і вони можуть бути отримані з Інтернету. Правильна MPI-програма повинна виконуватися на всіх реалізаціях без зміни.

– MPI відповідає моделі багатопроцесорній ЕОМ з передачею повідомлень.

Початок і завершення.

Існує декілька функцій, що використовуються в будь-якому, навіть найкоротшому додатку MPI. Вони не стільки передають дані, скільки забезпечують передачу:

1. Ініціалізація бібліотеки. Одна з перших інструкцій у функції main (головна функція додатка):

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

Вона одержує адреси аргументів, переданих до функції main від операційної системи і параметри командного рядка. У кінець командного рядка програма mpirun додає ряд інформаційних параметрів, що вимагає MPI_Init.

2. Аварійне закриття бібліотеки. Викликається, якщо користувальницька програма завершується через помилки часу виконання, пов’язаних з MPI:

```
MPI_Abort( параметр області зв’язку, код помилки MPI );
```

Виклик MPI_Abort з будь-якої задачі примусово завершує роботу ВСІХ задач, приєднаних до заданої області зв’язку.

3. Нормальне закриття бібліотеки:

```
MPI_Finalize();
```

Настійно рекомендується не забувати вписувати цю інструкцію перед поверненням із програми, тобто:

– перед викликом стандартної функції C – exit;

– перед кожним MPI_Init-оператором return у функції main;

– якщо функції main призначений тип void, і вона не закінчується

оператором return, те MPI_Finalize() необхідно поставити в кінець main.

4. Дві інформаційні функції: повідомляють розмір групи (тобто загальну кількість задач, приєднаних до її області зв'язку) і порядковий номер задачі:

```
int size, rank;
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

Функції MPI

У MPI базисною операцією передавання даних є операція:

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

де (buff, count, dtype) − кількість (count) об'єктів типу dtype, що починаються з адреси buff у буфері посилання; dest – номер одержувача в групі, обумовленої комунікатором comm; tag − ціле число, використовуване для опису повідомлення; comm – ідентифікатор групи процесів і комунікаційний контекст.

Базисною операцією приймання є операція:

```
int MPI_Recv (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status
```

де (buf, count, dtype) описують буфер приймача, як у випадку MPI_Send; source – номер процесу-відправника повідомлення в групі, обумовленої комунікатором comm; status – містить інформацію щодо фактичного розміру повідомлення, джерела і тега.

Якщо в кластері використовуються SMP-вузли, то для організації обчислень можливі два варіанти:

1. Для кожного процесора в SMP-вузлі утворюється окремий MPI-процес. MPI-процеси усередині цього вузла обмінюються повідомленнями через загальну пам'ять (необхідно настроїти MPICH відповідно).

2. На кожному вузлі запускається тільки один MPI-процес. Усередині кожного MPI-процесу створюється розпаралелювання в моделі «загальної пам'яті», наприклад, за допомогою директив OpenMP.

Чим більше функцій містить бібліотека MPI, тим більше можливостей надається користувачу для написання ефективних програм. Однак для

написання більшої кількості програм принципово досить наступних шести функцій:

`MPI_Init` – ініціалізація MPI;

`MPI_Comm_size` – визначення числа процесів;

`MPI_Comm_rank` – визначення процесом власного номера;

`MPI_Send` – посилання повідомлення;

`MPI_Recv` – одержання повідомлення;

`MPI_Finalize` – завершення програми MPI.

Як приклад рівнобіжної програми, написаної у стандарті MPI для мови C, розглянемо програму обчислення числа π.

```
#include "mpi.h"
#include <math.h>
int main ( int argc, char *argv[ ] )
{
     int n, myid, numprocs, i;
     double mypi, pi, h, sum, x, t1, t2,
     PI25DT = 3.14159265358979323846264 3;
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
     while (1)
     {
       if (myid == 0)
       { printf ("Enter the number of intervals: (0 quits) ");
          scanf ("%d", &n);
          t1 = MPI_Wtime();
       }
       MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
       if (n == 0) break;
       else
       { h = 1.0/ (double) n;
          sum = 0.0;
          for (i = myid +1; i <= n; i+= numprocs)
          { x = h * ( (double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
          }
          mypi = h * sum;
          MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
          MPI_COMM_WORLD);
          if (myid == 0)
          { t2 = MPI_Wtime();
            printf   ("pi   is   approximately   %.16f.   Error   is
            %.16f\n",pi, fabs(pi - PI25DT));
            printf ("`time is %f seconds \n", t2-t1);
         }
```

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

У програмі після декількох рядків визначення перемінних випливають три рядки, що є в кожній MPI-програмі:

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Звертання до MPI_Init має бути першим звертанням у MPI-програмі, воно встановлює «середовище» MPI. У кожному виконанні програми може виконуватися тільки один виклик MPI_Init.

Комунікатор MPI_COMM_WORLD описує склад процесів і зв'язок між ними. Виклик MPI_Comm_size повертає в numprocs число процесів, що користувач запустив у цій програмі. Значення numprocs – розмір групи процесів, пов'язаної з комунікатором MPI_COMM_WORLD. Процеси в будь-якій групі нумеруються послідовними цілими числами, починаючи з 0.

Викликаючи MPI_ Comm_rank, кожен процес з'ясовує свій номер (rank) у групі, що пов'язана з комунікатором. Потім головний процес (який має myid = 0) одержує від користувача значення числа прямокутників n:

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Перші три параметри відповідно позначають адресу, кількість і тип даних. Четвертий параметр указує номер джерела даних (головний процес), п'ятий параметр – назву комунікатора групи. Отже, після звертання до MPI_Bcast усі процеси мають значення n і власні ідентифікатори, що є достатнім для кожного процесу, щоб обчислити mypi − свій внесок в обчислення π. Для цього кожен процес обчислює область кожного прямокутника, що починається з myid + l. Потім усі значення mypi, обчислені індивідуальними процесами, підсумовуються за допомогою виклику Reduce:

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
```

Перші два параметри описують джерело й адресу результату. Третій і четвертий параметр описують число даних (1) і їх тип, п'ятий параметр – тип

арифметико-логічної операції, шостий – номер процесу для розміщення результату. Потім керування передається на початок циклу. Цим користувачу надається можливість задати нове n і підвищити точність обчислень. Коли користувач друкує нуль у відповідь на запит про новий n, цикл завершується, і всі процеси виконують:

```
MPI_Finalize()
```

Після неї будь-які операції MPI не виконуватимуться. Функція MPI_Wtime() використовується для вимірювання часу виконання ділянки програми, розташованого між двома включеннями в програму цієї функції.

Хід роботи

1. Відкомпілювати і перевірити ефективність стандартних прикладів MPI на різній кількості процесорів.

2. Реалізувати програму, у якій кожен процесор друкує кількість процесів у групі та свій номер у ній.

3. Визначити максимально припустиму довжину повідомлення.

4. Реалізувати скалярний добуток розподілених між процесорами векторів.

5. Пінг-понг. Змоделювати послідовний обмін повідомленнями між двома процесами, заміряти час на одну ітерацію обміну, визначити залежність часу від довжини повідомлення. Визначити базові характеристики комунікаційної мережі кластера: латентність (час на передачу повідомлення нульової довжини) і максимально досяжну пропускну здатність (кількість мегабайт за секунду; на повідомленнях якої довжини вона досягається?)

6. Спланувати структуру обміну даними для програми перемноження матриць.

Зміст звіту

1. Назва, мета завдання.

2. Результати дослідження основних функцій MPI.

3. Графік залежності часу затраченого на одну ітерацію обміну даними від довжини повідомлення.

4. Обґрунтування вибору структури обміну даними для програми перемноження матриць.

5. Письмові відповіді на контрольні питання.

Контрольні питання

1. Методи розпаралелювання та моделі програм, підтримувані MPI.

2. Властивості MPI, що забезпечують кросплатформенність рівнобіжних програм і незалежність від обчислювальних систем.

3. Команда компіляції MPI-программ.

4. Команда запуску MPI-программ.

Література: [1–5, 8–12].

Завдання № 4

Тема. Дослідження функцій для забезпечення колективного обміну даними

Мета: дослідити ефективність виконання послідовної програми на однопроцесорному комп’ютері; отримати навички аналізу інформаційної структури обчислювальних алгоритмів і вибору методів розпаралелювання.

Короткі теоретичні відомості

Колективні функції

Термін «колективні» у MPI передбачає три групи функцій:

– функції колективного обміну даними;

– точки синхронізації, чи бар’єри;

– функції підтримки розподілених операцій.

У колективній функції одним з аргументів є комунікатор. Виклик колективної функції є коректним, тільки якщо зроблений з усіх процесів-абонентів відповідної області зв'язку, і саме з цим комунікатором як аргумент. У цьому і полягає колективність: або функція викликається всім колективом процесів, або ніким; третього не існує.

Крапки синхронізації, вони ж бар'єри

Цим займається усього одна функція:

```
int MPI_Barrier( MPI_Comm comm );
```

MPI_Barrier зупиняє виконання задачі, доти, поки не буде викликана з усіх інших задач, приєднаних до вказаного комунікатора. Вона гарантує, що до виконання наступної за MPI_Barrier інструкції кожна задача приступить одночасно з іншими.

Деякі інші колективні функції, залежно від реалізації можуть володіти, а можуть і не мати властивість одночасно повертати керування всім процесам; але для них ця властивість є побічною і необов'язковою – якщо потрібна синхронність, використовуйте тільки MPI_Barrier.

Функції колективного обміну даними

Основні особливості та відмінності від комунікацій типу «точка-точка»:

– на примання і передавання працюють одночасно ВСІ задачі-абоненти, що вказані комунікатором;

– колективна функція виконує одночасно і приймання, і передавання; вона має велику кількість параметрів, частина яких потрібна для приймання, а частина – для передавання; у різних задачах та чи інша частина ігнорується;

– зазвичай значення ВСІХ параметрів (за винятком адрес буферів) повинні бути ідентичними у всіх задачах;

– MPI призначає ідентифікатор для повідомлень автоматично;

– повідомлення передаються не по вказаному комунікатору, а по тимчасовому комунікатору-дублікату;

– потоки даних колективних функцій надійно ізолюються один від одного та від потоків, створених функціями «точка–точка».

MPI_Bcast розсилає вміст буфера з задачі, що має в зазначеній області зв'язку номер root, у всі інші:

```
int MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

- *buff* – адреса початку буфера, що зберігає передане повідомлення;
- *count* – кількість переданих елементів;
- *datatype* – тип переданих елементів;
- *root* – номер кореневого процесу, тобто номер процесу який передаватиме повідомлення всім іншим процесам;
- *comm* – ідентифікатор групи.

Розглянемо приклад 1. У цьому прикладі кожен не root процес у буфері sbuf зберігає рядок «I am not root». Root-процес кладе собі в буфер sbuf рядок «Hello from root» і розсилає її за допомогою MPI_Bcast всім іншим. Як результат, у кожного не root процесу в буфері виявляється саме це повідомлення.

Приклад 1.

```
# include <mpi.h>
# include <stdio.h>
# include <string.h>

int main(int argc, char** argv)
{
  int numtasks, rank, root;
  char sbuf[20];
  MPI_Status s;

  strcpy(sbuf, "I am not root\0");

  root = 1;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(rank == root)   strcpy(sbuf, "Hello from root\0");

  MPI_Bcast(sbuf, 16, MPI_CHAR, root, MPI_COMM_WORLD);

  if(rank == root) strcpy(sbuf, "I am root\0");

  printf("I am %d. Message recived: %s\n", rank, sbuf);
```

```
  MPI_Finalize();

  return 0;
}
```

MPI_Bcast еквівалентна за результатами такому фрагменту:

```
MPI_Comm_size( communicator, &commSize );
MPI_Comm_rank( communicator, &myRank );
if( myRank == rootRank )
   for( i=0; i<commSize; i++ )
       MPI_Send( buf, count, dataType, i,
           tempMsgTag, communicator );
MPI_Recv( buf, count, dataType, rootRank, tempMsgTag,
   communicator, &status );
```

MPI_Gather («совок») збирає в приймальний буфер задачі root передавальні буфери інших задач. Її аналог:

```
MPI_Send( sendBuf, sendCount, sendType, rootRank, ... );
if( myRank == rootRank ) {
   MPI_Type_extent( recvType, &elemSize );
   for( i=0; i<commSize; i++ )
     MPI_Recv(((char*))recvBuf)+(i*recvCount*elemSize),
           recvCount, recvType, i, ... );
}
```

recvType і sendType можуть бути різні і, отже, задаватимуть різну інтерпретацію даних на приймання та передавання. Задача-приймач також відправляє дані у свій приймальний буфер.

Векторний варіант "совка" – MPI_Gatherv – дозволяє задавати РІЗНУ кількість даних, що відправляється, у різних задачах-відправниках. Відповідно, на приймальній стороні задається масив позицій у приймальному буфері, на яких варто розміщати дані, що надходять, і максимальні довжини порцій даних від усіх задач.

Обидва масиви містять позиції/довжини НЕ в байтах, а в кількості структур типу recvCount. Її аналог:

```
MPI_Send( sendBuf, sendCount, sendType, rootRank, ... );
if( myRank == rootRank ) {
 MPI_Type_extent( recvType, &elemSize );
for( i=0; i<commSize; i++ )
       MPI_Recv( ((char*))recvBuf) + displs[i] * recvCounts[i]
           * elemSize, recvCounts[i], recvType, i, ... );
}
```

MPI_Scatter («розпилювач»): виконує зворотну «совку» операцію – частини передавального буфера з задачі root розподіляються по приймальних буферах усіх задач. Її аналог:

```
if( myRank == rootRank ) {
 MPI_Type_extent( recvType, &elemSize );
     for( i=0; i<commSize; i++ )
        MPI_Send( ((char*)sendBuf) + i*sendCount*elemSize,
          sendCount, sendType, i, ... );
}
MPI_Recv( recvBuf, recvCount, recvType, rootRank, ... );
```

І її векторний варіант – MPI_Scatterv, що розсилає частини неоднакової довжини в приймальні буфери неоднакової довжини.

Для ілюстрації розглянемо приклад 2. У цьому прикладі процес *root* має буфер з чотирьох наборів по 4 числа. Після застосування функції *MPI_Scatter* кожний із процесів одержує в буфері *recbuf* порцію з 4-х чисел.

Приклад 2.

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

# define SIZE 4

int main(int argc, char** argv)
{
  int numtasks, rank, sendcount, recvcount, source;
  float sendbuf[SIZE][SIZE] = {
    {1.0, 2.0, 3.0, 4.0},
    {5.0, 6.0, 7.0, 8.0},
    {9.0, 10.0, 11.0, 12.0},
    {13.0, 14.0, 15.0, 16.0}  };
  float recvbuf[SIZE];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  if (numtasks == SIZE) {
    source = 1;
    sendcount = SIZE;
    recvcount = SIZE;
    MPI_Scatter(sendbuf, sendcount, MPI_FLOAT, recvbuf, recvcount,
                MPI_FLOAT, source, MPI_COMM_WORLD);

    printf("rank= %d  Results: %f %f %f %f\n", rank, recvbuf[0],
           recvbuf[1], recvbuf[2], recvbuf[3]);
  }
```

```
  else
    printf("Must specify %d processors. Terminating.\n", SIZE);

  MPI_Finalize();
}
```

MPI_Allgather аналогічна MPI_Gather, але приймання здійснюється не в одній задачі, а в УСІХ: кожна має специфічний вміст у передавальному буфері, і всі одержують однаковий вміст у буфері приймання.

Як і в MPI_Gather, прийомний буфер послідовно заповнюється даними. Варіант з неоднаковою кількістю даних називається MPI_Allgatherv.

MPI_Alltoall: кожен процес розрізає передавальний буфер на шматки і розсилає шматки іншим процесам; кожен процес одержує шматки від всіх інших і почергово розміщює їх у приймальному буфері. Це «совок» і «розпилювач» в одному флаконі. Векторний варіант називається MPI_Alltoallv.

Колективні функції несумісні з «точка–точка» неприпустимі!

Хід роботи

1. Змоделювати бар'єрну синхронізацію за допомогою пересилань типу «точка-точка» і порівняти ефективність різних варіантів реалізації функції MPI_BARRIER.

2. Порівняти ефективність деяких колективних операцій з їхнім моделюванням за допомогою пересилань типу «точка–точка».

3. Визначити корисність використання глобальних операцій при реалізації обраного варіанта рівнобіжного перемножування матриць.

Зміст звіту

1. Назва, мета завдання.

2. Результати дослідження ефективності різних варіантів реалізації функції MPI_BARRIER із зазначенням графічних результатів.

3. Результати порівняння колективних операцій та їх моделювання за допомогою команд типу «точка–точка» із зазначенням графічних результатів.

4. Письмові відповіді на контрольні питання.

Контрольні питання

1. Поясніть принцип колективного обміну інформацією.
2. Які плюси та мінуси використання синхронізації?
3. Пояснити основні принципи виконання нижче зазначених функцій.

```
MPI_Barrier(..);
MPI_Bcast(…);
MPI_Scatter(…),  MPI_Scatterv(…);
MPI_Gather(…),MPI_Gathrv(…),MPI_Allgather(…)
```

Література: [2–6, 10–12].

Завдання № 5

Тема. Розпаралелювання процесу перемноження матриць

Мета: дослідження процесу виконання класичного прикладу паралельного програмування MPI – задача перемноження матриць; закріплення практичного освоєння функцій парних взаємодій між процесами паралельної програми.

Короткі теоретичні відомості

Множення матриці на вектор і матриці на матрицю є базовими макрооперaціями для багатьох задач лінійної алгебри, наприклад ітераційних методів рішення систем лінійних рівнянь. Тому наведені алгоритми можна розглядати як фрагменти в алгоритмах цих методів. Розмаїтість варіантів алгоритмів виникає від розмаїтості обчислювальних систем і розмірів задач. Розглядаються і різні варіанти завантаження даних у систему: завантаження даних через один комп’ютер; і завантаження даних безпосередньо кожним комп’ютером з дискової пам’яті. Якщо завантаження даних здійснюється через один комп’ютер, то дані зчитуються цим комп’ютером з дискової пам’яті, розрізаються і частини розсилаються по інших комп’ютерах. Але дані можуть бути підготовлені і заздалегідь, тобто заздалегідь розрізані і кожна частина

записана на диск у вигляді окремого файлу зі своїм ім’ям; потім кожен комп’ютер безпосередньо зчитує з диска призначений для нього файл.

Розглянемо алгоритм обчислення отримання матриці C = A x B, де A - матриця n1 x n2, і B – матриця n2 x n3. Матриця результатів C має розмір n1 x n3. Вихідні матриці попередньо розрізані на смуги, смуги записані на дискову пам’ять окремими файлами зі своїми іменами і доступні всім комп’ютерам. Матриця результатів повертається в нульовий процес.

Реалізація алгоритму виконується на кільці з p1 комп’ютерів. Матриці розрізані, як показано на рис. 1: матриця A розрізана на p1 горизонтальних смуг, матриця B розрізана на p1 вертикальних смуг, і матриця результату C розрізана на p1 смуги. Тут передбачається, що в пам’ять кожного комп’ютера завантажується і може знаходитися тільки одна смуга матриці A и одна смуга матриці B.

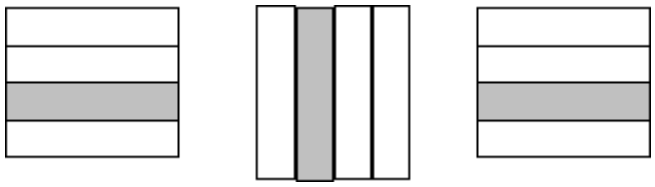

Рисунок 1 – Розрізування даних для рівнобіжного алгоритму добутку двох матриць під час обчислення на кільці комп’ютерів
(виділені смуги розташовані в одному комп’ютері)

Оскільки за умовою в комп’ютерах знаходиться по одній смузі матриць, то смуги матриці B (або смуги матриці A) необхідно «прокрутити» по кільцю комп’ютерів повз смуги матриці A (матриці B).

Кожне зрушення смуг уздовж кільця і відповідне множення зображено на рис. 2 у вигляді окремого кроку.

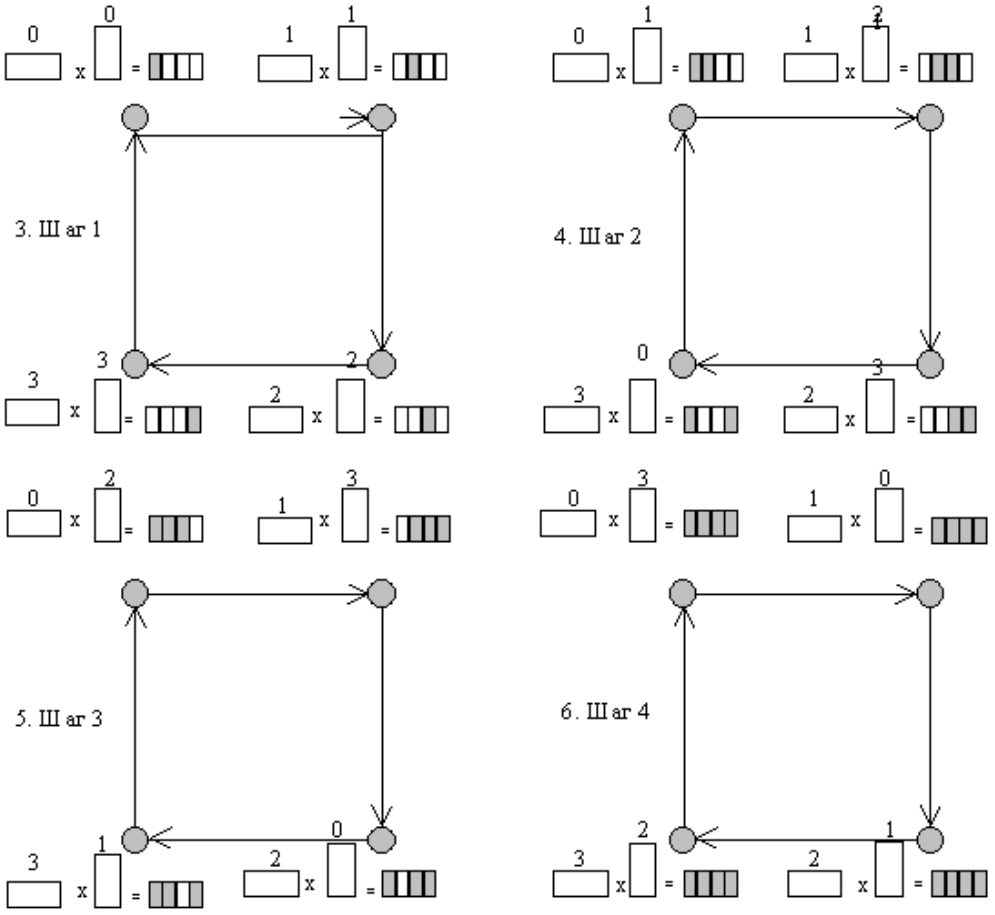

Рисунок 2 – Стадії обчислень добутку матриць у кільці комп’ютерів

На кожному такому кроці обчислюється тільки частина смуги. Процес обчислює на j-му кроці добуток i-ї горизонтальної смуги матриці A і j-ї вертикальної смуги матриці B, добуток отриманий у субматриці (i, j) матриці C.

1. Кожен комп’ютер зчитує з диска відповідну йому смугу матриці A. Нульова смуга повинна зчитуватися нульовим комп’ютером, перша смуга – першим комп’ютером і т. д., остання смуга – зчитується останнім комп’ютером. На рис. 2 смуги матриці A и B пронумеровані.

2. Кожен комп’ютер зчитує з диска відповідну йому смугу матриці B. У цьому випадку нульова смуга повинна зчитуватися нульовим комп’ютером, перша смуга – першим комп’ютером і т. д., остання смуга – зчитується останнім комп’ютером.

3. Обчислювальний крок 1. Кожен процес обчислює одну субматрицю добутку. Вертикальні смуги матриці B зрушуються уздовж кільця комп’ютерів.

4. Обчислювальний крок 2. Кожен процес обчислює одну субматрицю добутку. Вертикальні смуги матриці B зрушуються уздовж кільця комп'ютерів і т. д.

5. Обчислювальний крок p1-1. Кожен процес обчислює одну субматрицю добутку. Вертикальні смуги матриці B зрушуються уздовж кільця комп'ютерів.

6. Обчислювальний крок p1. Кожен процес обчислює одну субматрицю добутку. Вертикальні смуги матриці B зрушуються уздовж кільця комп'ютерів.

7. Матриця C збирається в нульовому комп'ютері.

Якщо «прокручувати» вертикальні смуги матриці B, то матриця C буде розподілена горизонтальними смугами, а якщо «прокручувати» горизонтальні смуги матриці A, те матриця C буде розподілена вертикальними смугами.

Алгоритм характерний тим, що після кожного кроку обчислень здійснюється обмін даними. Нехай tu, tu, і tp час операцій, відповідно, множення, додавання і пересилання одного числа в сусідній комп'ютер. Згідно з наведеними на початку секції позначеннями, сумарний час операцій множень дорівнює:

U = (n1*n2)*(n3*n2)*tu,

сумарний час операцій додавань дорівнює:

S = (n1*n2)*(n3*(n2-1))*ts,

сумарний час операцій пересилань даних по всіх комп'ютерах дорівнює:

P = (n3*n2)*(p1-1)*tp.

Тоді загальний час обчислень дорівнюватиме:

T = (U+S+P)/p1

І відношення часу «обчислень без обмінів» до загального часу обчислень є величина:

K = (U+S)/(U+S+P).

Якщо канали передавання даних повільні, то ефективність алгоритму буде не висока. Тут не враховується час початкового завантаження і вивантаження даних у пам'ять системи. У смугах матриць можуть бути різна

кількість рядків (розходження в один рядок). У великих матрицях цим можна зневажити.

За достатніх ресурсах пам'яті в системі, звичайно ж, краще використовувати алгоритм, у якому мінімізовані обміни між комп'ютерами під час обчислень. Це досягається завдяки дублюванню деяких даних у пам'яті комп'ютерів.

Нижче подана програма множення матриці на вектор на топології «кільце» (приклад 3).

У прикладі 3 передбачається, що кількість рядків матриці A і вектора B поділяються без залишку на кількість комп'ютерів у системі. Вихідна матриця A розміром 20x20 поділяється без залишку на кількість комп'ютерів, у цьому випадку на 4. Вектори B і C теж поділяються на 4 без залишку.

Приклад 3.

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<time.h>
#include <sys/time.h>
/* Задаємо в кожній галузі розміри смуг матриць A і B:  5x20.*/
#define M 20
#define N 5
/* NUM_DIMS-розмір декартової топології. "кільце" - одномірний. */
#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;
    MPI_Comm        ring;
    MPI_Status      st;
    int rt, t1, t2;
/* Ініціалізація бібліотеки MPI*/
    MPI_Init(&argc, &argv);
/* Кожен процес довідується кількість задач, що стартувала, */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* Заповнюємо масив periods для топології  "кільце" */
    for(i=0; i < DIMS; i++) { dims[i] = 0; periods[i] = 1; }
/* Заповнюємо масив dims, де вказуються розміри одномірних решіток
*/
```

```
    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);
 /* Кожен процес визначає свій власний номер: від 0 до (size-1) */
    MPI_Comm_rank(ring, &rank);

 /* Кожен процес знаходить своїх сусідів уздовж кільця, у напрямку
менших значень рангів */
    MPI_Cart_shift(ring, 0, -1, &sour, &dest);
 /* Кожен процес 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();
  /* Кожен процес робить множення своїх смуг матриці і вектора */
  /* Самий зовнішній цикл for(k) - цикл по комп'ютерах */
    for(k = 0; k < size; k++)
     { d = ((rank + k)%size)*N;
 /* Кожен процес робить 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];
         }

 /* Кожен процес передає своїм сусідам з меншим рангом смуги
вектора B. Тобто, смуги вектора B зрушуються уздовж кільця
комп'ютерів */
       MPI_Sendrecv_replace(B, EL(V), MPI_DOUBLE, dest, 12, sour,
                                              12, ring, &st);
     }
/* Множення pfвершене. Кожен процес помножid свою смугу рядків
матриці A на
 * усі 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 ghjwtcb завершують системні процеси, зв'язані з топологією
comm_cart i
  * завершують виконання програми */
    MPI_Comm_free(&ring);
    MPI_Finalize();
    return(0);
  }
```

Хід роботи

1. Уважно вивчити приклад множення матриці на вектор на топології «кільце».

2. Відкомпілювати наведений приклад і запустити на 4 комп’ютерах.

3. Написати і налагодити рівнобіжну програму множення матриці на матрицю в топології «кільце»: A x B = C, за умови, що кількість рядків матриці A і стовпців матриці B повністю ділиться на кількість комп’ютерів.

Зміст звіту

1. Назва, мета завдання.

2. Опис алгоритму роботи програми.

3. Вихідні коди і результати роботи програм.

4. Графік залежності часу виконання програми від розмірності матриці.

5. Письмові відповіді на контрольні питання.

Контрольні питання

1. Який метод розпаралелювання використовується в рівнобіжних алгоритмах множення матриці на вектор і матриці на матрицю з використанням MPI?

2. Як розподіляються елементи матриці і вектора під час множення матриці на вектор на топології «кільце»?

3. Як розподіляються елементи обох матриць під час множення матриці на матрицю на топології «кільце»?

4. Як краще подати в пам’яті комп’ютера другу матрицю, під час множення матриці на матрицю, для прискорення часу розв’язання задачі?

Література: [2–4, 6–10].

2 КРИТЕРІЇ ОЦІНЮВАННЯ ЯКОСТІ ВИКОНАННЯ РОЗРАХУНКОВО-ГРАФІЧНОЇ РОБОТИ СТУДЕНТАМИ

У 8-му семестрі студенти виконують розрахунково-графічну роботу. Загальна кількість балів, яку отримують студенти за виконання розрахунково-графічної роботи, становить 15 балів – сума за захист виконаних завдань в РГР робіт (максимально по 3 бали на кожне завдання РГР).

Шкала оцінювання: національна та ECTS

<table>
<tr><td rowspan="2">Сума балів за всі види навчальної діяльності</td><td rowspan="2">Оцінка ECTS</td><td colspan="2">Оцінка за національною шкалою</td></tr>
<tr><td>Для екзамену, курсового проекту (роботи), практики</td><td>Для заліку</td></tr>
<tr><td>90–100</td><td>A</td><td>Відмінно</td><td rowspan="5">Зараховано</td></tr>
<tr><td>82–89</td><td>B</td><td rowspan="2">Добре</td></tr>
<tr><td>74–81</td><td>C</td></tr>
<tr><td>64–73</td><td>D</td><td rowspan="2">Задовільно</td></tr>
<tr><td>60–63</td><td>E</td></tr>
<tr><td>35–59</td><td>FX</td><td>Незадовільно з можливістю повторного складання</td><td>Не зараховано з можливістю повторного складання</td></tr>
<tr><td>0–34</td><td>F</td><td>Незадовільно з обов’язковим повторним вивченням навчальної дисципліни</td><td>Не зараховано з обов’язковим повторним вивченням навчальної дисципліни</td></tr>
</table>

СПИСОК ЛІТЕРАТУРИ

1. Бекон Д. Операционные системы / Д. Бекон, Е. Харрис. – Киев : Издательская группа BHV, 2004. – 800 с.
2. Вальковский В. А. Распараллеливание алгоритмов и программ. Структурный подход / В.А. Вальковский. – М. : Радио и связь, 1989. – 176 с.
3. Воеводин В. В. Параллельные вычисления / В.В. Воеводин. – СПб. : БХВ-Петербург, 2002. – 608 с.
4. Гергель В. П. Основы параллельных вычислений для многопроцессорных вычислительных систем : учебное пособие / В. П Гергель, Р. Г. Стронгин. – Нижний Новгород : Изд-во ННГУ, 2003. – 184 с.
5. Кейслер С. Проектирование операционных систем для малых ЭВМ. / С. Кейслер. – М. : Мир, 1986. – 680 с.
6. Корнеев В. Д. Параллельное программирование в MPI / В. Д. Корнеев. – Новосибирск : Изд-во СО РАН, 2000. – 213 с.
7. Немнюгин С. А. Параллельное программирование для многопроцессорных вычислительных систем / С. А. Немнюгин, О. Л. Стесик. – СПб. : БХВ-Петербург, 2002. – 400 с.
8. Ортега Дж. Введение в параллельнные и векторные методы решения линейных систем / Дж. Ортега. – М. : Мир, 1991. – 367 с.
9. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. – СПб. : «Русская Редакция», 2000. – 752 с.
10. Стивенс У. Unix: Разработка сетевых приложений / У. Стивенс. – СПб. : Питер, 2004. – 1086 с.
11. Таненбаум Э. Распределенные системы. Принципы и парадигмы. Серия «Классика computer science» / Э. Таненбаум, В. Стеен. – СПб. : Питер, 2003. – 877 с.
12. Таненбаум Э. Современные операционные системы / Э. Таненбаум. –

СПб. : Питер, 2007. – 1040 с.

13. Хьюз К. Параллельное и распределенное программирование на C++ / К. Хьюз, Т. Хьюз. – М. : Издательский дом «Вильямс», 2004. – 672 с.
14. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс. – М. : Издательский дом «Вильямс», 2003. – 512 с.
15. Вахалия Ю. UNIX изнутри / Ю. Вахалия. – СПб. : Питер, 2003. – 844 с.
16. Гольдштейн А. Б. Softswitch. / А. Б. Гольдштейн, Б. С. Гольдштейн. – СПб. : БХВ-Петербург, 2006. – 368 с.
17. Лупин С. А. Технологии паралельного программирования / С. А. Лупин, М. А. Посыпкин. – СПб. : Питер, 2011. – 208 с.
18. Хаггарти Р. Дискретная математика для программистов / Р. Хаггарти. – СПб. : БХВ-Петербург, 2012. – 395 с.
19. Голицына О. А. Программирование на языках высокого уровня / О. А. Голицына, И. И. Попов. – СПб. : Питер, 2010. – 418 с.
20. Вальковский В. А. Синтез паралельних программ и систем на вычислительных моделях / В. А. Вальковский, В. Э. Малышкин. – Новосибирск : Изд-во «Наука», 1988. – 129 с.
21. Малышкин В. Э. Паралельное программирование мультикомпьютеров / В. Э. Малышкин, В. Д. Корнеев. – Новосибирск : Изд-во НТУ, 2006. – 296 с.

Методичні вказівки щодо виконання розрахунково-графічної роботи з навчальної дисципліни «Паралельні та розподілені обчислення» для студентів денної форми навчання зі спеціальності 123 – «Комп'ютерна інженерія»

Укладач к. т. н., доц. О. Г. Славко

Відповідальний за випуск зав. кафедри КІС А. В. Луговой

Підп. до др. _____. Формат 60×84 1/16. Папір тип. Друк ризографія.
Ум. друк. арк. ____. Наклад _____ прим. Зам. №_____. Безкоштовно.

Видавничий відділ
Кременчуцького національного університету
імені Михайла Остроградського
вул. Першотравнева, 20, м. Кременчук, 39600