МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ОПТИМАЛЬНОГО КЕРУВАННЯ І ЕКОНОМІЧНОЇ КІБЕРНЕТИКИ

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ MPI

ЕЛЕКТРОННІ МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
до лабораторних робіт студентам
факультету математики, фізики та інформаційних технологій

ОДЕСА
ОНУ
2023

УДК 004.42:004.43(072)
П18

Укладачі:
В. В. Вербіцький, кандидат фізико-математичних наук, доцент, доцент кафедри оптимального керування і економічної кібернетики;
А. Л. Максимов, старший викладач кафедри оптимального керування і економічної кібернетики.

Рецензенти:
В. В. Мороз, кандидат технічних наук, професор кафедри оптимального керування і економічної кібернетики Одеського національного університету імені І. І. Мечникова;
І. М. Шпінарева, кандидат фізико-математичних наук, доцент, доцент кафедри математичного забезпечення комп'ютерних систем імені І. І. Мечникова.

Рекомендовано вченою радою факультету математики, фізики та інформаційних технологій ОНУ імені І. І. Мечникова. Протокол № 4 від 27 березня 2023 р.

П18 **Паралельне** програмування з використанням технології MPI [Електронний ресурс] : електрон. метод. вказівки до лаб. робіт студентів ф-ту математики, фізики та інформаційних технологій / уклад.: В. В. Вербіцький, А. Л. Максимов. – Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2023. – 37 с. – 0,7 МБ.

У методичних рекомендаціях приведено основні відомості щодо створення паралельних додатків для комп'ютерних систем з розподіленою пам'яттю за технологією написання паралельних програм MPI. Методичні вказівки містять завдання та варіанти для лабораторних робіт.

Розраховано для студентів факультету математики, фізики та інформаційних технологій Одеського національного університету імені І. І. Мечникова, що вивчають дисципліни «Паралельні та розподілені обчислення», «Технології розподілених систем та паралельних обчислень», «Паралельні алгоритми обчислювальної математики».

УДК 004.42:004.43(072)

ЗМІСТ

Вступ

Найбільш поширеною технологією програмування для паралельних комп'ютерів з розподіленою пам'яттю на даний час є MPI. Основним способом взаємодії паралельних процесів в таких системах є передача повідомлень один одному. Це й відображено в назві даної технології – Message Passing Interface (інтерфейс передачі повідомлень). Стандарт MPI фіксує інтерфейс, якого повинні дотримуватися як система програмування обчислювальної платформи, так і користувач під час створення паралельних програм.

У MPI 1.1 (опублікований 12 червня 1995 року, перша реалізація з'явилася 2002 року) підтримуються такі функції:

- передача та отримання повідомлень між окремими процесами;
- колективні взаємодії процесів;
- взаємодії у групах процесів;
- реалізація топологій;

У MPI 2.0 (опублікований 18 липня 1997) додатково підтримуються такі функції:

- динамічне породження процесів та управління процесами;
- односторонні комунікації (Get/Put);
- паралельне введення та виведення;
- розширені колективні операції (процеси можуть виконувати колективні операції як усередині одного комунікатора, а й у межах кількох комунікаторів).

Версія MPI 2.1 вийшла на початку вересня 2008 року.

Версія MPI 2.2 вийшла 4 вересня 2009 року.

Версія MPI 3.0 вийшла 21 вересня 2012 року.

Нова версія MPI 4.0 вийшла 9 червня 2021 року.

MPI підтримує роботу з мовами Фортран і Сі. В даному посібнику приклади і описи всіх процедур будуть дані з використанням мови Сі. Повна версія інтерфейсу містить опис більш як 125 процедур і функцій.

Інтерфейс MPI підтримує створення паралельних програм в стилі MIMD (Multiple Instruction Multiple Data), що має на увазі об'єднання

процесів з різними вихідними текстами. Однак писати і налагоджувати такі програми дуже складно, тому на практиці програмісти набагато частіше використовують SPMD-модель (Single Program Multiple Data) паралельного програмування, в межах якої для всіх паралельних процесів використовується один і той самий код. В даний час всі реалізації MPI підтримують роботу з потоками (threads – нитки).

Оскільки MPI є бібліотекою, то при компіляції програми необхідно підключити відповідні бібліотечні модулі. Це можна зробити в командному рядку або скористатися передбаченими в більшості систем командами або скриптами mpicc (для програм на мові C), mpiCC (для програм на мові C++). Опція компілятора "-o name" дозволяє задати ім'я “name” для одержуваного результуючого файлу, наприклад:

```
$ mpicc -o program program.c
```

Приклад MPI-програми:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
   int myrank, size;
   MPI_Init(&argc, &argv); // Ініціалізація MPI
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Отримання розміру комунікатора
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    // Одержуємо номер процесу
     printf("Proc %d of %d\n", myrank, size);
      MPI_Finalize(); // Фіналізація MPI
      puts("Done.");
   return 0;
}
```

Після отримання результуючого файлу необхідно запустити його на необхідній кількості процесорів. Для цього зазвичай надається команда запуску MPI-додатків `mpirun`, наприклад:

```
# mpirun -np N <програма з аргументами>
```

де N – число процесів, яке має бути не більше дозволеного в даній системі числа процесів для одного завдання. Після запуску одна й та сама програма буде виконуватися всіма процесами, що будуть запущені, і результат виконання в залежності від системи буде видаватися на термінал або записуватися в файл з визначеним ім'ям.

Всі додаткові об'єкти: імена процедур, константи, зумовлені типи даних і т. п., що використовуються в MPI, мають префікс MPI_. Зазвичай в назві функцій MPI перша буква після префікса MPI_ пишеться у верхньому регістрі, наступні літери – у нижньому регістрі, а назви констант MPI записуються цілком у верхньому регістрі. Всі описи інтерфейсу MPI зібрані в файлі mpi.h.

MPI-програма – це множина паралельних взаємодіючих процесів. Всі процеси породжуються один раз, утворюючи паралельну частину програми. В ході виконання MPI-програми стандарту MPI-1 породження додаткових процесів або знищення існуючих не допускається (в MPI-2.0 і далі така можливість з'явилася). Кожен процес працює в своєму адресному просторі, ніяких загальних змінних або даних в MPI немає. Основним способом взаємодії між процесами є явне розсилання повідомлень.

1. Комунікатори

Для локалізації взаємодії паралельних процесів програми можна створювати групи процесів, надаючи їм окреме середовище для спілкування – комунікатор. Склад утворених груп довільний. Групи можуть повністю збігатися, входити одна в іншу, не перетинатися або перетинатися частково. Процеси взаємодіють тільки всередині будь-якого комунікатора; повідомлення, відправлені в різних комунікаторах, не перетинаються і не заважають один одному. Комунікатори мають в мові С зумовлений тип MPI_COMM.

При старті програми завжди вважається, що всі породжені процеси працюють в рамках всеосяжного комунікатора, що має визначене ім'я `MPI_COMM_WORLD`. Цей комунікатор існує завжди і служить для взаємодії всіх запущених процесів MPI-програми. Крім

нього, при старті програми є комунікатор MPI_COMM_SELF, що містить тільки один поточний процес, а також комунікатор MPI_COMM_NULL, який не містить жодного процесу. Всі взаємодії процесів протікають в рамках певного комунікатора; повідомлення, передані в різних комунікаторах, ніяк не заважають один одному.

Кожен процес MPI-програми має в кожній групі, в яку він входить, унікальний номер (ранг) процесу, який є цілим невід'ємним числом. За допомогою цього номера відбувається значна частина взаємодій процесів між собою. Ясно, що в одному й тому самому комунікаторі всі процеси мають різні номери. Але, оскільки процес може одночасно входити в різні комунікатори, то його номер в одному комунікаторі може відрізнятися від його номера в іншому. Якщо група містить n процесів, то номер будь-якого процесу в даній групі лежить в межах від 0 до n-1.

Основним способом спілкування процесів між собою є явна посилка повідомлень. Повідомлення – це набір даних деякого типу. Кожне повідомлення має кілька атрибутів, зокрема, номер процесу-відправника, номер процесу-одержувача, ідентифікатор повідомлення та інші. Одним з важливих атрибутів повідомлення є його ідентифікатор або тег. За ідентифікатором процес, який отримує повідомлення, може розрізнити два повідомлення, що прийшли до нього від одного й того самого процесу. Ідентифікатор повідомлення є цілим невід'ємним числом, яке належить діапазону від 0 до MPI_TAG_UP, причому гарантується, що MPI_TAG_UP не менш 32767. В останньому атрибуті повідомлення, що повертається. Більшість MPI-функцій повертають інформацію про успішність завершення роботи. У разі успішного виконання повертається значення MPI_SUCCESS, інакше – код помилки. Вид помилки, яка сталася при виконанні процедури, можна буде визначити з її опису. Коди, які відповідають різним помилковим ситуаціям, перераховані у файлі mpi.h.

2. Функції ініціалізації і завершення роботи

Будь-яка прикладна MPI-програма (додаток) повинна починатися

з виклику функції ініціалізації `MPI_Init()`. В результаті виконання цієї функції створюється група процесів, в яку поміщаються всі процеси, і створюється область зв'язку, що описується визначеним комунікатором `MPI_COMM_WORLD`. Ця область зв'язку об'єднує всі процеси. Процеси в групі впорядковані і пронумеровані від 0 до groupsize - 1, де groupsize дорівнює числу процесів в групі. Крім цього, створюється комунікатор `MPI_COMM_SELF`, що описує свою область зв'язку для кожного окремого процесу.

Синтаксис функції ініціалізації С:

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

У програмах на С кожному процесу при ініціалізації передаються аргументи функції `main()`, отримані з консолі.

Функція завершення MPI програм `int MPI_Finalize(void)` закриває всі MPI-процеси і ліквідує всі області зв'язку.

Функція `int MPI_Comm_size(MPI_Comm comm, int *size)` визначає число `size` процесів в комунікаторі `comm`.

До створення явним чином груп та пов'язаних з ними комунікаторів єдино можливими значеннями параметра `comm` є `MPI_COMM_WORLD`, або `MPI_COMM_SELF`, які створюються автоматично при ініціалізації MPI.

Функція `int MPI_Comm_rank (MPI_Comm comm, int * rank)` визначає номер `rank` процесу в комунікаторі `comm`.
Номери процесів лежать в діапазоні `0 … size-1`. Значення `size` може бути визначено за допомогою функції `MPI_Comm_size()`.

3. Типи даних MPI

MPI допускає можливість запуску процесів паралельної програми на комп'ютерах різних платформ, забезпечуючи при цьому автоматичне перетворення даних при пересиланнях. У Таблиці 1 наведено відповідність визначених в MPI типів стандартних типів мови С.

Таблиця 1

Відповідність між типами MPI і типами мови С

Тип MPI	Тип мови С
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

4. Передача/прийом повідомлень між окремими процесами

Практично всі програми, написані з використанням комунікаційної технології MPI, повинні містити засоби для взаємодії запущених процесів між собою. Така взаємодія здійснюється в MPI за допомогою явного розсилання повідомлень.

Всі функції передачі повідомлень в MPI діляться на дві групи. До однієї з груп входять функції, які призначені для взаємодії тільки двох процесів програми. Такі операції називаються індивідуальними, або операціями типу точка-точка. Функції іншої групи припускають, що в операцію повинні бути залучені всі процеси деякого комунікатора. Такі операції називаються колективними.

Почнемо опис функцій обміну повідомленнями з обговорення операцій типу точка-точка. У таких взаємодіях беруть участь два процеси. Один процес є відправником повідомлення, а інший –

одержувачем. Процес-відправник повинен викликати одну з функцій передачі даних і явно вказати номер в деякому комунікаторі процесу-одержувача, а процес-одержувач повинен викликати одну з функцій прийому із зазначенням того ж комунікатора, причому в деяких випадках він може не знати точний номер процесу-відправника в даному комунікаторі.

Всі функції даної групи, у свою чергу, так само поділяються на два класи: функції з блокуванням (з синхронізацією) і функції без блокування (асинхронні). Функції обміну з блокуванням припиняють роботу процесу до виконання деякої умови, а повернення з асинхронних функцій відбувається негайно після ініціалізації відповідної комунікаційної операції. Неакуратне використання функцій з блокуванням може привести до виникнення тупикової ситуації, тому при цьому потрібна додаткова обережність. Використання асинхронних операцій до тупикових ситуацій не приводить, однак вимагає більш обережного використання масивів даних.

4.1. Передача/прийом повідомлень з блокуванням

Щоб передати повідомлення з блокуванням, процес-відправник викликає функцію

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

Масив `buf`, що складається з count елементів типу datatype, передається процесу з номером `dest` у комунікаторі COMM. Всі елементи повідомлення, що надсилається, повинні бути розташовані послідовно у буфері `buf`. `tag` – ідентифікатор повідомлення. Операція починається незалежно від того, чи була ініціалізована відповідна функція прийому. При цьому повідомлення може бути скопійовано як безпосередньо в буфер прийому, так і поміщено в певний системний буфер (якщо це передбачено в MPI). Значення `count` може бути нулем. Процесу дозволяється передавати повідомлення самому собі, однак це небезпечно і може привести до

виникнення тупикової ситуації.

При пересиланні повідомлень можна використовувати спеціальне значення `MPI_PROC_NULL` для неіснуючого процесу. Операції з таким процесом завершуються негайно з кодом завершення `MPI_SUCCESS`.

Блокування гарантує коректність повторного використання всіх параметрів після повернення з процедури. Це означає, що після повернення з `MPI_Send()` можна використовувати будь-які присутні у виклику даної функції змінні без побоювання зіпсувати передане повідомлення. Вибір способу здійснення цієї гарантії: копіювання в проміжний буфер або безпосередня передача процесу `dest`, залишається за розробником конкретної реалізації MPI.

Слід спеціально зазначити, що повернення з функції `MPI_Send()` не означає ні того, що повідомлення отримано процесом `dest`, ні того, що повідомлення покинуло процесорний елемент, на якому виконується процес, який виконав даний виклик. Надається тільки гарантія безпечної зміни змінних, використаних у виклику даної функції. Подібна невизначеність далеко не завжди влаштовує користувача. Щоб розширити можливості передачі повідомлень, в MPI введені додаткові три функції. Всі параметри у цих функцій такі ж, як і у `MPI_Send()`, однак у кожної з них є своя особливість.

MPI надає наступні модифікації процедури передачі даних з блокуванням `MPI_Send()`:

– `MPI_Bsend()` – передача повідомлення з буферизацією. Якщо прийом повідомлення, що надсилається, ще не був ініціалізований процесом-одержувачем, то повідомлення буде записано в спеціальний буфер, і станеться негайне повернення з функції. Виконання даної функції ніяк не залежить від відповідного виклику функції прийому повідомлення. Проте, функція може повернути код помилки, якщо місця під буфер недостатньо. Про виділення масиву для буферизації повинен піклуватися користувач.

– `MPI_Ssend()` – передача повідомлення з синхронізацією. Вихід з даної процедури станеться тільки тоді, коли прийом повідомлення, що надсилається, буде ініціалізований процесом-одержувачем. Таким чином, завершення передачі з синхронізацією говорить не тільки про можливості повторного використання буфера посилки, але і про гарантоване досягнення процесом-одержувачем точки прийому повідомлення в програмі. Використання передачі повідомлень із синхронізацією може уповільнити виконання програми, але дозволяє уникнути наявності в системі великої кількості великої кількості не прийнятих буферизованих повідомлень.
– `MPI_Rsend()` – передача повідомлення по готовності. Цією функцією можна користуватися тільки в тому випадку, якщо процес-одержувач вже ініціював прийом повідомлення. В іншому випадку виклик функції, взагалі кажучи, є помилковим, і результат її виконання не визначений. Гарантувати ініціалізацію прийому повідомлення перед викликом функції `MPI_Rsend()` можна за допомогою операцій, які здійснюють явну або неявну синхронізацію процесів (наприклад, `MPI_Barrier()`). У багатьох реалізаціях функція `MPI_Rsend()` скорочує протокол взаємодії між відправником і отримувачем, зменшуючи накладні витрати на організацію передачі даних.

Перед використанням функції `MPI_Bsend()` для передачі повідомлення з буферизацією користувач спочатку повинен виділити буфер, наприклад, функцією `malloc()`. Потім треба зареєструвати цей буфер в середовищі MPI функцією

```
int MPI_Buffer_attach (void *buf, int size).
```

Після використання треба від'єднати буфер від середовища MPI функцією

```
int MPI_Buffer_detach (void *buf, int size),
```

а потім звільнити пам'ять, займану буфером, наприклад, функцією `free()`.

Процес-одержувач може прийняти повідомлення за допомогою

функції
int MPI_Recv (void *buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm, MPI_Status *status).
Функція виконує блокуючий прийом в буфер buf не більше count елементів типу datatype від процесу з номером source з ідентифікатором повідомлення tag в комунікаторі comm, із заповненням змінної status атрибутами повідомлення. Якщо число реально прийнятих елементів менше значення count, то гарантується, що в буфері buf зміняться тільки елементи, відповідні елементам прийнятого повідомлення. Якщо кількість елементів в прийнятому повідомленні більше значення count, то виникає помилка переповнення. Якщо потрібно дізнатися точну кількість елементів в прийнятому повідомленні, то можна скористатися функцією MPI_Get_count(). Блокування гарантує, що після завершення процедури MPI_Recv() всі елементи повідомлення вже будуть прийняті в буфер buf. Параметр source може приймати значення MPI_ANY_SOURCE (зумовлена константа MPI). В цьому випадку процес готовий отримати повідомлення від будь-якого процесу. Якщо параметр tag дорівнює зумовленої константі MPI_ANY_TAG, процес приймає повідомлення з будь-яким ідентифікатором.

У мові С status – це структура типу MPI_Status з трьома полями:

- MPI_SOURCE – номер процесу, що послав повідомлення;
- MPI_TAG – ідентифікатор прийнятого повідомлення;
- MPI_ERROR – код помилки прийняття повідомлення або нуль.

У наступному прикладі n паралельних процесів знаходять найбільший елемент в цілочисельному масиві:

```
#include "mpi.h"
#include <stdio.h>
#define NN 20
int main(int argc, char **argv) {
```

```
  int a[NN] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2,
3, 4, 5, 6, 7, 8, 9, 99};
  int i, size, ns, nf;
  int myrank, ranksize;
  int lmax, max;
  MPI_Status status;
   MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &ranksize);
     size = NN/ranksize;
      ns = size*myrank;
       nf = ns + size;
        if (myrank == (size - 1))
            nf = NN;
       lmax = a[ns];
        for (i = ns; i < nf; i++)
             if (lmax < a[i])
                 lmax = a[i];
         printf("My  rank=%d  Local  max=%d  \n",
myrank, lmax);
         if (myrank == 0) {
             max = lmax;
              for (i = 1; i < ranksize; i++) {
                   MPI_Recv(&lmax, 1, MPI_INT, i,
99, MPI_COMM_WORLD, &status);
                    if (max < lmax)
                        max = lmax;
              }
               printf("Master: max=%d \n", max);
         } else
               MPI_Send(&lmax, 1, MPI_INT, 0, 99,
MPI_COMM_WORLD);
          MPI_Finalize();
}
```

4.2. Передача/прийом повідомлень без блокування

У MPI передбачено набір функцій для здійснення *асинхронної передачі даних.* На відміну від блокуючих функцій, повернення з функцій даної групи відбувається відразу після виклику без будь-якої зупинки роботи процесів. На тлі подальшого виконання програми одночасно відбувається і обробка асинхронно запущеної операції. Дана можливість виключно корисна для створення ефективних програм. Справді, якщо програміст знає, що в певний момент в даному процесі йому потрібен масив, що обчислюється в іншому процесі, він заздалегідь виставляє в програмі асинхронний запит на отримання масиву, а до того моменту, коли масив реально потрібний, даний процес може виконувати будь-яку іншу корисну роботу. Знову ж, у багатьох випадках взагалі не обов'язково чекати закінчення посилки повідомлення для виконання наступних обчислень. Для завершення асинхронного обміну потрібен виклик додаткової функції, яка перевіряє, чи завершилася операція, або чекає її завершення. Тільки після цього можна використовувати буфер посилки для інших цілей без побоювання зіпсувати повідомлення, що відправляється. Якщо є можливість операції прийому/передачі повідомлень виконувати на тлі обчислень, то цим, начебто, треба беззастережно користуватися. Однак на практиці не все завжди узгоджується з теорією. Багато що залежить від конкретної реалізації. На жаль, далеко не завжди асинхронні операції ефективно підтримуються апаратурою та системним оточенням. Тому не варто дивуватися, якщо ефект від виконання обчислень на тлі пересилань виявиться нульовим або зовсім невеликим. Зроблені зауваження стосуються лише питань ефективності. Щодо наданої функціональності асинхронні операції виключно корисні, тому вони присутні практично в кожній реальній програмі.

Функція

```
int MPI_Isend (void *buf, int count,  MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

виконує деблокуючу посилку з буфера buf count елементів повідомлення типу datatype з ідентифікатором tag процесу dest комунікатора COMM. Повернення з процедури відбувається відразу після ініціалізації процесу передачі без очікування обробки всього повідомлення, що знаходиться в буфері buf. Це означає, що не можна повторно використовувати даний буфер для інших цілей без отримання додаткової інформації, яка б підтверджувала завершення даної посилки. Визначити той момент часу, коли можна повторно використовувати буфер buf без побоювання зіпсувати передане повідомлення, можна за допомогою параметра request, що повертається, і функцій сімейств MPI_Wait() і MPI_Test(). Параметр request має в мові С зумовлений тип MPI_Request і використовується для ідентифікації конкретної деблокуючої операції.

Аналогічно трьом модифікаціям функції MPI_Send(), передбачено три додаткових варіанти функції MPI_Isend():

- MPI_Ibsend() – деблокуюча передача повідомлення з буферизацією;
- MPI_Issend() – деблокуюча передача повідомлення з синхронізацією;
- MPI_Irsend() – деблокуюча передача повідомлення по готовності.

До викладеної вище семантики роботи цих функцій додається відсутність блокування.

Функція

```
int MPI_Irecv (void *buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm, MPI_Request *request)
```

виконує деблокуючий прийом в буфер buf не більше count елементів повідомлення типу datatype з ідентифікатором tag від процесу з номером source в комунікаторі COMM із заповненням змінної request. На відміну від блокуючого прийому, повернення з функції відбувається відразу після ініціалізації процесу прийому без

очікування отримання всього повідомлення і його запису в буфері buf. Закінчення процесу прийому можна визначити за допомогою параметру request і процедур сімейств MPI_Wait() і MPI_Test().

Повідомлення, відправлене будь-який з функцій MPI_Send(), MPI_Isend() і будь-який з трьох їхніх модифікацій, може бути прийнято будь-який з процедур MPI_Recv() або MPI_Irecv().

Звернемо особливу увагу на те, що до завершення деблокуючої операції не слід записувати в використовуваний масив даних!

Функція

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

забезпечує отримання в масиві status інформації про структуру очікуваного повідомлення з ідентифікатором tag від процесу з номером source в комунікаторі COMM без блокування. У параметрі flag повертається значення true, якщо повідомлення з відповідними атрибутами вже може бути прийнято (в цьому випадку дія процедури повністю аналогічно MPI_Probe()), і значення false, якщо повідомлення із зазначеними атрибутами ще немає.

Функція

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

очікує завершення асинхронної операції, асоційованої з ідентифікатором request і запущеної викликом функції MPI_Isend() або MPI_Irecv(). Поки асинхронна операція не буде завершена, процес, що виконував функцію MPI_Wait(), буде заблокований. Для операції неблокуючого прийому визначається параметр status. Після виконання функції ідентифікатор деблокуючої операції request встановлюється в значення MPI_REQUEST_NULL .

5. Колективні взаємодії процесів

В операціях колективної взаємодії процесів беруть участь всі

процеси комунікатора. Функції, які виконують колективні взаємодії, називаються колективними функціями. Колективна функція повинна бути викликана кожним процесом, можливо, зі своїм набором параметрів. Повернення з функції колективної взаємодії може відбутися в той момент, коли участь процесу в даній операції вже закінчено. Як і для блокуючих функцій, повернення означає те, що дозволено вільний доступ до буферу прийому або посилки. Асинхронних колективних операцій в MPI немає.

У колективних операціях можна використовувати ті ж комунікатори, що і були використані для операцій типу точка-точка. MPI гарантує, що повідомлення, викликані колективними операціями, ніяк не вплинуть на виконання інших операцій і не перетнуться з повідомленнями, що з'явилися в результаті індивідуальної взаємодії процесів.

Взагалі кажучи, не можна розраховувати на синхронізацію процесів за допомогою колективних операцій (крім функції `MPI_Barrier()`). Якщо якийсь процес завершив свою участь у колективній операції, то це не означає ні того, що дана операція завершена іншими процесами комунікатора, ні навіть того, що вона ними розпочата (якщо це можливо за змістом операції).

Колективні операції строго впорядковані відповідно до їх появи в тексті програми.

Функція

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

використовується для бар'єрної синхронізації процесів. Робота процесів блокується до тих пір, поки вся решта процесів комунікатора `COMM` не виконає цю функцію. Тільки після того, як останній процес комунікатора виконає цю функцію, всі процеси будуть розблоковані і продовжать виконання. Ця функція є колективною. Всі процеси повинні викликати `MPI_Barrier`(), хоча реально виконані різними процесами виклики комунікатора можуть бути розташовані в різних місцях програми.

Функція

```
int MPI_Bcast (void *buf, int count, MPI_Datatype
```

datatype, int root,MPI_Comm comm)
виконує розсилку count елементів даних типу datatype з масиву buf від процесу root всім процесам даного комунікатора COMM, включаючи сам процес, що розсилає. При поверненні з функції вміст буфера buf процесу root буде скопійовано в локальний буфер кожного процесу комунікатора comm. Значення параметрів count, datatype, root і comm повинні бути однаковими у всіх процесів.

Наступна схема ілюструє дію функції MPI_Bcast ():

Тут, так само як і в подальших схемах, по вертикалі зображуються різні процеси, які беруть участь у колективній операції, а по горизонталі – розташовані на них блоки даних.

Розглянемо приклад. Щоб переслати від процесу 2 всім іншим процесам програми масив buf із 100 цілочисельних елементів, потрібно, щоб в усіх процесах зустрівся наступний виклик:

```
MPI_Bcast (buf, 100, MPI_INT, 2, MPI_COMM_WORLD);
```

Функція

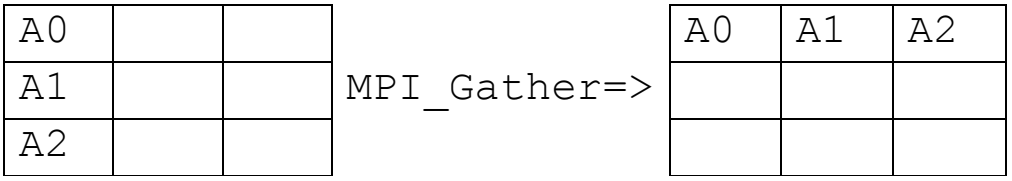
```
int MPI_Gather (void *sendbuf, int sendcount,MPI_Datatype sendtype, void *recvbuf,int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

збирає sendcount елементів даних типу sendtype з масивів sendbuf з усіх процесів комунікатора comm в буфері recvbuf процесу root. Кожен процес, включаючи root, посилає вміст свого буфера sendbuf процесу root. Збираючий процес зберігає дані в буфері recvbuf, розташовуючи їх у порядку зростання номерів процесів.

На процесі root істотними є значення всіх параметрів, а на

інших процесах – тільки значення параметрів `sendbuf`, `sendcount`, `sendtype`, `root` і `comm`. Значення параметрів `root` і `comm` повинні бути однаковими у всіх процесів. Параметр `recvcount` у процесі `root` позначає число елементів типу `resvtype`, що приймаються не від усіх процесів в сумі, а від кожного процесу.

Наступна схема ілюструє дію процедури MPI_Gather.



Розглянемо приклад. Для того, щоб процес 2 зібрав в масив `rbuf` по 10 цілочисельних елементів масивів `buf` з усіх процесів додатку, потрібно, щоб в усіх процесах зустрівся наступний виклик:

```
MPI_Gather (buf, 10, MPI_INT, rbuf, 10, MPI_INTEGER, 2, MPI_COMM_WORLD);
```

6. Глобальні операції

У паралельному програмуванні математичні операції над блоками даних, розподілених по процесорах, називають глобальними операціями редукції. У загальному випадку операцією редукції називається операція, аргументом якої є вектор, а результатом – скалярна величина, отримана застосуванням деякої математичної операції до всіх компонентів вектора. Зокрема, якщо компоненти вектора розташовані в адресних просторах процесів, що виконуються на різних процесорах, то в цьому випадку говорять про глобальну (паралельну) редукцію. Наприклад, нехай в адресному просторі всіх процесів деякої групи процесів є копії змінної `var` (не обов'язково вони мають одне і те ж значення), тоді застосування до неї операції обчислення глобальної суми або, іншими словами, операції редукції SUM, поверне одне значення, яке буде містити суму всіх локальних значень цієї змінної. Використання цих операцій є одним з основних засобів організації розподілених обчислень.

У MPI глобальні операції редукції представлені в декількох варіантах, наприклад:

– зі збереженням результату в адресному просторі одного процесу

(MPI_Reduce());

– зі збереженням результату в адресному просторі всіх процесів (функція MPI_Allreduce()).

Функція

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op,int root, MPI_Comm comm)
```

виконується наступним чином. Операція глобальної редукції, зазначена параметром op, виконується над першими елементами вхідного буфера, і результат надсилається в перший елемент буфера прийому процесу root. Потім те ж саме робиться для других елементів буфера і т. д.

Програміст може задати свою функцію для виконання глобальної операції за допомогою процедури MPI_Op_create().

Як операцію op можна використовувати або одну із зумовлених операцій, або операцію, сконструйовану користувачем. Всі зумовлені операції є асоціативними і комутативними. Сконструйована користувачем операція, принаймні, повинна бути асоціативною. Порядок редукції визначається номерами процесів в групі. Тип datatype елементів повинен бути сумісний з операцією op. У таблиці 2 подано перелік визначених операцій, які можуть бути використані у функціях редукції MPI.

Таблиця 2

Зумовлені операції у функціях редукції MPI

Назва	Операція	Дозволені типи
MPI_MAX MPI_MIN	максимум мінімум	C integer, Floating point
MPI_SUM MPI_PROD	сума добуток	C integer, Floating point, Complex
MPI_LAND MPI_LOR MPI_LXOR	логічне AND логічне OR логічне виключає OR	C integer, Logical
MPI_BAND MPI_BOR MPI_BXOR	порозрядне AND порозрядне OR порозрядне виключає OR	C integer, Byte

MPI_MAXLOC MPI_MINLOC	максимальне значення і його індекс мінімальне значення і його індекс	Спеціальні типи для цих функцій

У таблиці використовуються наступні позначення:

- C integer: `MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG;`
- Floating point: `MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE;`
- Logical: `MPI_LOGICAL;`
- Complex: `MPI_COMPLEX;`
- Byte: `MPI_BYTE.`

Як приклад використання глобальної операції розглянемо розв’язання наступного завдання. Необхідно впорядкувати масив `a` розмірності `N` невід'ємних цілих чисел. Максимальний елемент цього масиву набагато менше за `N`. Алгоритм рішення задачі полягає в наступному. Кожен паралельний процес формує допоміжний масив `a0`. Елемент `a0[i]` дорівнює кількості елементів, рівних `i`, які зустрічаються в частині масиву `a`, що оброблюється процесом. Потім паралельні процеси виконують глобальну операцію `MPI_SUM`. Після чого, в процесі з номером 0 елемент `a1[i]` – це сума елементів `a0[i]` всіх процесів. Нарешті, процес з номером 0 друкує упорядкований масив. Нижче приведена паралельна програма.

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

#define NN 20
#define MAX 5

int main (int argc, char **argv){
int a [NN] =
```

```
 {1,2,1,1,4,0,0,0,3,0,1,2,3,4,3,3,0,0,3,3};
    int a0 [MAX] = {0,0,0,0,0};
    int a1 [MAX] = {0,0,0,0,0};
    int i, j, size, ns, nf;
    int myrank, ranksize;
    MPI_Status status;
     MPI_Init (& argc, & argv);
      MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
      MPI_Comm_size (MPI_COMM_WORLD, &ranksize);
       size = NN/ranksize;
        ns = size*myrank;
         nf = ns + size;
          if (myrank == (size-1)) nf = NN;
           for (i = ns; i < nf; i++)
                a0[a[i]]++;
            MPI_Reduce  (a0,  a1,  MAX,  MPI_INT,
MPI_SUM, 0, MPI_COMM_WORLD);
             MPI_Barrier (MPI_COMM_WORLD);
              if (myrank == 0) {
                  printf ( "Sorted array: \n");
                   for (i = 0; i <MAX; i++)
                     for (j = 1; j <= a1[i]; j++)
                          printf ( "%d ", i);
             }
              MPI_Finalize ();
}
```

7. Похідні типи даних

Під повідомленням в MPI розуміється масив однотипних даних, розташованих в послідовних комірках пам'яті. Часто в програмах потрібні пересилання більш складних об'єктів даних, що складаються з різнотипних елементів або розташовані не в послідовних комірках пам'яті. В цьому випадку можна або посилати дані невеликими порціями розташованих підряд елементів одного типу, або використовувати копіювання даних перед відсиланням в певний

проміжний буфер. Обидва варіанти є незручними і вимагають додаткових витрат як часу, так і оперативної пам'яті.

Для пересилання різнотипних даних в MPI передбачені два спеціальних способи:

– похідні типи даних;
– упаковка даних.

Розглянемо перший з них. Похідні типи даних створюються під час виконання програми за допомогою функцій-конструкторів на основі існуючих на момент виклику конструктора типів даних.

Створення типу даних складається з двох етапів:

1. конструювання типу;
2. реєстрація типу.

Після реєстрації похідний тип даних можна використовувати поряд з зумовленими типами в операціях пересилання, в тому числі і в колективних операціях. Після завершення роботи з похідним типом даних його рекомендується анулювати. При цьому всі вироблені на його основі нові типи даних залишаються і можуть використовуватися далі.

Похідний тип даних характеризується послідовністю базових типів даних і набором цілочисельних значень зсуву елементів типу щодо початку буфера обміну. Зміщення можуть бути як додатними, так і від’ємними, не зобов'язані відрізнятися, не є потрібною їх упорядкованість. Таким чином, послідовність елементів даних в похідному типі може відрізнятися від послідовності вихідного типу, а один елемент даних може зустрічатися в конструйованому типі багатократно.

Функція

```
int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

створює новий тип даних `newtype`, що складається з `count` послідовно розташованих елементів базового типу даних `oldtype`. Фактично створюється тип даних представляє масив даних базового типу як окремий об'єкт. Розглянемо, наприклад, виклик

```
MPI_Type_contiguous(5, MPI_INT, &newtype);
```

Тут створюється новий тип даних newtype, який в подальшому (після реєстрації типу) може використовуватися для пересилання п'яти розташованих підряд цілих чисел.

Після створення новий тип даних треба зареєструвати в середовищі MPI функцією

```
int MPI_Type_commit (MPI_Datatype *datatype)
```

Тільки після реєстрації новий тип може використовуватися в комунікаційних операціях.

Після використання нового типу даних описувач нового типу даних треба видалити функцією

```
int MPI_Type_free (MPI_Datatype * datatype)
```

Функція MPI_Type_free() встановлює описувач типу в стан MPI_DATATYPE_NULL. Це не вплине на комунікаційні операції, що виконуються в даний момент з цим типом даних і на похідні типи, які раніше були визначені через знищений тип.

8. Створення глобальних операцій

Користувач може визначити свою глобальну операцію функцією

```
int MPI_Op_create (MPI_User_function *function,
                   int commute, MPI_Op *op)
```

Функція MPI_Op_create() пов'язує визначену користувачем глобальну операцію з покажчиком op, який згодом може бути використаний, наприклад, функцією MPI_Reduce().

Визначена користувачем операція передбачається асоціативною. Якщо commute=true, то операція повинна бути комутативною. Якщо commute=false, то порядок операндів фіксований, операнди розташовуються по зростанню номерів процесів, починаючи з нульового. Порядок виконання може бути змінений, щоб використовувати перевагу асоціативності операції. Якщо commute=true, то порядок виконання може бути змінений, щоб використовувати переваги комутативності й асоціативності. Параметр function – це визначена користувачем функція, яка повинна мати наступний прототип:

```
typedef void MPI_User_function(void *invec, void
```

*inoutvec, int *len, MPI_Datatype * datatype);

Параметр datatype – це дескриптор типу даних, що використовуються глобальної операцією, який вказується у виклику функції для виконання глобальної операції. Наприклад, в MPI_Reduce(). Користувацька функція редукції повинна бути написана так, щоб задовольняти наступному:

- позначимо u[0], … , u[len-1] – len елементів в буфері обміну, описаних аргументами invec, len і datatype під час виклику функції;
- позначимо v[0], … , v[len-1] – len елементів в буфері обміну, описаних аргументами, аргументами inoutvec, len і datatype під час виклику функції;
- позначимо w[0], … , w[len-1] – len елементів в буфері обміну, описаних параметрами inoutvec, len і datatype, коли функція повертає управління;

тоді w[i]=u[i]•v[i], для i = 0, ..., len-1, де • – операція редукції, яку обчислює функція.

Неформально, можна вважати inoutvec і invec масивами з len елементів, які комбінує MPI_User_function. Результат операції редукції перезаписує величини в inoutvec, що і визначає ім'я. Результатом кожного виклику функції є поточкова комбінація оператором редукції len елементів: тобто функція повертає в inoutvec[i] значення invec[i]•inoutvec[i] для i = 0,...,count-1, де • – операція комбінації, що реалізується функцією.

Функції обміну MPI не повинні викликатися з функції, призначеної для редукції. З цієї функції в разі помилки може бути викликана функція MPI_Abort().

Якщо створена глобальна операція більш не потрібна, описувач глобальної операції треба видалити функцією

int MPI_op_free (MPI_Op *op)

Функція встановлює значення стану описувача типу в

MPI_OP_NULL.

У наступній програмі наведено приклад створення користувацьких типів даних і глобальних операцій.

```
#include <stdio.h>
#include "mpi.h"
#define N 5
typedef struct {double r, i;} Complex;
MPI_User_function fcProd;
void fcProd (void *in, void *inout, int *len,
MPI_Datatype *dt) {
   int i;
   complex c, *in1, *inout1;
   double re,im, re_,im_,;
   in1    = (complex *) in;
   inout1 = (complex *) inout;
   for (i = 0; i < *len; ++i) {
     re  = inout1->r; im  = inout1->i;
     re_ = in1->r;    im_ = in1->i;
     c.r = re*re_ - im*im_;
     c.i = re*im_ + im*re_;
     *Inout1 = c;
     in1++; inout1++;
   }
 }
int main (int argc, char **argv){
    complex a[N], a1[N];
    MPI_Op cProd;
    MPI_Datatype MPI_COMPLEX;
    int myrank, i;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
    for (i = 0; i <N; i ++) {
            a[i].r =  myrank + i;
            a[i].i = (myrank * i)%N;
```

```
    }
    MPI_Type_contiguous       (2,       MPI_DOUBLE,
&MPI_COMPLEX);
    MPI_Type_commit (&MPI_COMPLEX);
    MPI_Op_create (fcProd, 1, &cProd);
    MPI_Reduce  (a,  a1,  N,  cType,  cProd,  0,
MPI_COMM_WORLD);
    MPI_Barrier (MPI_COMM_WORLD);
    if (myrank == 0) {
        printf ( "Result: \ n");
        for (i = 0; i <N; i++)
        printf ( "%f + i%f \n", a1[i].r, a1[i].i);
     }
     MPI_Type_free (&MPI_COMPLEX);
     MPI_Op_free (& cProd);
     MPI_Finalize ();
}
```

У програмі створюється новий тип даних `Ctype` для роботи з комплексними числами. Потім створюється глобальна операція `cProd` множення комплексних чисел. Кожен процес паралельної програми формує масив `a[]` з `N` комплексних чисел. В процесі з номером 0 в результаті виконання розподіленої операції `cProd` формується масив комплексних чисел `a1[]`, кожен елемент якого дорівнює добутку відповідних елементів масиву `a[]` у всіх процесах додатку.

9. Топологія процесів

Топологія – це механізм зіставлення процесів деякої альтернативної схеми адресації в новому комунікаторі. У MPI топології віртуальні, тобто вони не пов'язані з фізичною топологією комунікаційної мережі. Топологія використовується програмістом для більш зручного позначення процесів і, таким чином, наближення паралельної програми до структури алгоритму вирішення конкретного завдання. Крім того, топологія може використовуватися

системою для оптимізації розподілу процесів по фізичних процесорах паралельного комп'ютера за допомогою зміни порядку нумерації процесів всередині комунікатора.

У MPI передбачено два типи топологій:

- декартова топологія (мережа прямокутної решітки довільної розмірності);
- топологія графа.

Для визначення типу топології служить функція

```
MPI_Topo_test(MPI_Comm comm, int *status)
```

Функція повертає через змінну `status` топологію комунікатора `comm`. Можливі значення:

- `MPI_GRAPH` – топологія графа;
- `MPI_CART` – декартова топологія;
- `MPI_UNDEFINED` – топологія не задана.

Надалі будемо розглядати тільки декартову топологію. Для створення комунікатора з декартовою топологією використовується функція

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

Функція створює комунікатор `comm_cart`, що реалізує декартову топологію для процесів комунікатора `comm_old`. Параметр `ndims` задає розмірність одержуваної декартової решітки, `dims(i)` – число елементів в вимірі `i`, `i=0,...,ndims-1`. `periods` – логічний масив з `ndims` елементів, що визначає, чи є решітка періодичною (значення `true`) уздовж кожного виміру. `reorder` – логічний параметр, що визначає, що системі дозволено змінювати порядок нумерації процесів (при значенні `true`) для оптимізації розподілу процесів по фізичних процесорах паралельного комп'ютера. Функція є колективною, а значить, повинна бути викликана всіма процесами комунікатора `COMM`. Якщо кількість процесів у топології `comm_cart` менше числа процесів в комунікаторі `comm_old`, то деяким процесам може повернутися значення `MPI_COMM_NULL`, а значить, вони не

братимуть участі в створюваній топології. Якщо кількість процесів у топології, що задається, більше числа процесів в комунікаторі comm_old, то виклик буде помилковим.

У наступному прикладі створюється тривимірна топологія 4x3x2, кожний вимір якої є періодичним, крім того, дозволяється переупорядкування процесів. Даний фрагмент повинен виконуватися не менше, ніж на 24 процесах.

```
MPI_Comm *comm_cart;
int dims[3] = {4,3,2};
int periods [3] = {true, true, true};
 MPI_Cart_create    (MPI_COMM_WORLD,    3,    dims,
periods, true,  comm_cart);
```

У декартовій топології кожен процес визначається як номером (рангом) так і координатою в цій топології.

Функція

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

дозволяє визначити декартові координати процесу за його рангом `rank` в комунікаторі `COMM`. Координати повертаються в масиві `coords` розмірності `ndims`.

Функція

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

дозволяє визначити ранг `rank` процесу в комунікаторі `COMM` за його декартовими координатами `coords`. Для періодичних решіток координати поза допустимих інтервалів перераховуються, для неперіодичних решіток вони є помилковими.

Розглянемо приклад SPMD-обчислення на гіперкубі. Нехай $N = 2^n$ процесів розміщені в вершинах одиничного гіперкубу простору $R^n \quad (n > 2)$. Кожен процес може взаємодіяти тільки з *n* процесами по жорстко закріплених лініях зв'язку з номерами від 1 до *n* (лінії зв'язку є ребрами гіперкуба). Набір координат $u = (u_1, \cdots, u_n)$, де $u_i \in \{0,1\}$ для всіх *i,* довільної вершини гіперкуба задає ім'я процесу, що відповідає цій вершині.

Структуру зв'язків процесів у гіперкубі задають за допомогою функції *connect()*, яка визначається наступним чином. Процеси *u* і *v* з'єднані лінією зв'язку з номером *i*, що позначається

u = connect (v, i),

тоді і тільки тоді, коли набори координат цих процесів відрізняються тільки *i*-ю координатою.

Наведемо тепер алгоритм розв'язання наступної задачі.

Використовуючи $N = 2^n$ паралельних процесів з топологією *n*-вимірного гіперкуба, знайти найбільший спільний дільник 2^{n+1} цілого числа.

Алгоритм розв’язання задачі полягає в наступному. Кожен процес $u = (u_1, \cdots, u_n)$ виконує *k(u)* ітерацій циклу, де *k(u)* – номер першого зліва одиничного елемента в імені процесу, або *k(u) = n + 1*, якщо $u = (0, \cdots,)_n$. Наприклад, якщо *u = (0, 1, 1)* , то *k(u) = 2* , якщо *u = (1, 1, 1)* , то *k(u) = 1* , якщо *u = (0, 0, 0)* , то *k(u) = 4*. На *i*-й ітерації циклу (*i=1,...,n*) процес знаходить найбільший спільний дільник двох чисел, що містяться в змінних `v1` і `v2` за допомогою простого алгоритму:

```
while (v1 != v2)
  if (v1 > v2) v1 -= v2;else v2 -= v1;
```

Потім процес виконує наступне:

1. якщо *i* < *k(u)*, то процес *u* робить запит на прийом повідомлення в змінну `v2` від процесу *v = connect (u,i)*;
2. якщо *i = k(u)* ≤ *n* , то процес *u* робить запит на передачу значення змінної `v1` процесу *v = connect (u, i)* і після виконання передачі завершує свою роботу;
3. якщо *i = k(u) = n + 1*, то `v1 = v2` – шуканий найбільший спільний дільник і процес виводить значення змінної `v1` та завершує роботу паралельної програми.

Нижче наведено текст паралельної програми.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define N 16
```

```
int main (int argc, char **argv){
 int                    a[N]                    =
{16,24,100,100,40,20,80,80,32,200,16,24,32,44,32,3
2};
 MPI_Status status;
 MPI_Comm commCube;
 int myrank, rank;
 int ndims = 3;
 int dims[3] = {2,2,2};
 int periods[3] = {1,1,1};
 int reorder = 0;
 int coords[3], coords0[3];
 int i, j, ku, v1, v2;
 MPI_Init(&argc, &argv);
 MPI_Cart_create(MPI_COMM_WORLD,   ndims,   dims,
periods, reorder, &commCube);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 v1 = a[2*myrank];
 v2 = a[2*myrank + 1];
 MPI_Cart_coords   (commCube,   myrank,   ndims,
coords);
 i = 0;
 while ((i < ndims) && (coords [i] == 0))i++;
 ku = i + 1;
 for (i = 1; i<ku; i++) {
 while (v1 != v2)
 if (v1 > v2) v1 -= v2;   else    v2 -= v1;
 for (j = 0; j < ndims; j++)
   coords0[j] = coords[j];
   coords0 [i-1] = (coords0[i-1] + 1)% 2;
   MPI_Cart_rank(commCube, coords0, &rank);
    MPI_Recv (&v2, 1, MPI_INT, rank, 99, commCube,
     &status);
 }
 while(v1 != v2)
```

```
    if (v1 > v2) v1 -= v2;
    else  v2 -= v1;
 if (ku <= ndims) {
     for (j = 0; j < ndims; j++)
        coords0[j] = coords[j];
     coords0[i-1] = (coords0[i-1] + 1)%2;
 MPI_Cart_rank (commCube, coords0, &rank);
 MPI_Send (&v1, 1, MPI_INT, rank, 99, commCube);
 }
 else printf( "NOD = %d \n", v1);
  MPI_Finalize ();
}
```

Зауважимо, що програма написана для процесів, які утворюють топологію 3-куб, тому програма має виконуватися не менше ніж 8-ма процесами.

Завдання до лабораторних робіт

1. Написати паралельний додаток множення дійсної матриці розміру $m \times n$ на вектор розміру n. Кожен паралельний процес містить k рядків матриці.
2. Написати паралельний додаток множення дійсної матриці розміру $m \times n$ на вектор розміру n. Кожен паралельний процес містить k рядків матриці. Використовувати колективні функції.
3. Написати паралельний додаток множення дійсної матриці розміру $m \times n$ на вектор розміру n. Кожен паралельний процес містить k рядків матриці. Використовувати глобальні операції.
4. Написати паралельний додаток множення комплексної матриці розміру $m \times n$ на вектор розміру n. Кожен паралельний процес містить k рядків матриці. Використовувати глобальні операції.
5. Написати паралельний додаток для знаходження найбільшого за модулем елемента масивів комплексних чисел однакової розмірності, що зберігаються у паралельних процесах. Створити розподілену операцію для знаходження максимального за модулем комплексного числа.

6. Написати паралельний додаток, що моделює роботу клітинного автомата (КА). Поле КА являє собою двовимірний масив клітин, що поділяється між паралельними процесами. Кожна внутрішня клітина (в разі періодичного КА також і гранична) має 8 сусідів. Клітина може знаходитись в одному з двох станів: клітина вільна; клітина зайнята. КА переходить з поточного стану до наступного за такими правилами:
   - якщо у зайнятої клітини більше трьох сусідніх клітин зайняті, клітина звільняється;
   - якщо у вільної клітини хоча б дві сусідні клітини зайняті, клітина стає зайнятою.

   Для моделювання роботи КА використати топологію двовимірний масив (2×2).
7. Для моделювання роботи КА (див. завдання 6) використати топологію двовимірний масив (2×2) періодичний по першій координаті.
8. Для моделювання роботи КА (див. завдання 6) використати топологію двовимірний масив (2×2) періодичний по другій координаті.
9. Для моделювання роботи КА (див. завдання 6) використати топологію двовимірний масив (2×2) періодичний по обох координатах.
10. Для моделювання роботи КА (див. завдання 6) використовувати топологію кільця з 4-х процесорів.
11. 2^n паралельних процесів, що утворюють топологію n-куб, містять масиви цілих чисел. Спочатку всі процеси упорядковують масиви. Потім деякі з них передають упорядковані масиви іншим процесам і завершують роботу. Процес, який отримав упорядкований масив, об'єднує його зі своїм локальним масивом, зберігаючи впорядкованість. В кінці роботи паралельної програми один процес отримує упорядкований масив, що містить всі елементи масивів, які зберігались у паралельних процесах на початку роботи програми.
12. Написати паралельний додаток для знаходження номерів

нульових стовпців матриці розмірності $m \times n$, що розподілена по рядкам між p паралельними процесами. Використати розподілені операції.

14. Написати паралельний додаток для знаходження найбільшого елемента в кожному стовпці матриці розмірності $m \times n$, що розподілена по рядкам між p паралельними процесами. Використати розподілені операції.

Література

1. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press, 2014. 330 p.
2. MPI: A Message-Passing Interface Standard Version 4.0. Message Passing Interface Forum, June 9, 2021. 1139 p. (https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf)
3. Nielsen F. Introduction to HPC with MPI for Data Science. Springer International Publishing, 2016. 304 p.
4. Pacheco P. An Introduction to Parallel Programming / Elsevier, 2011. 392 p.
5. Rauber Th., Runger G. Parallel Programming For Multicore and Cluster Systems / Springer-Verlag Berlin Heidelberg, 2010. 463 p.
6. Robey R., Zamora Y. Parallel and High Performance Computing / Manning, Shelter Island, 2021. 705 p.
7. Schmidt B., González-Domínguez J., Hundt Ch., Schlarb M. Parallel Programming. Concepts and Practice / Publisher: Katey Birtcher, 2018. 405 p.
8. Trobec R., Slivnik B., Bulić P., Robič B. Introduction to Parallel Computing. From Algorithms to Programming on State-of-the-Art Platforms / Springer Nature Switzerland AG, 2018. 259 p.
9. Weinzierl T. Principles of Parallel Scientific Computing: A First Guide to Numerical Concepts and Programming Methods: Undergraduate Topics in Computer Science. / Springer Nature Switzerland AG, 2022. 314 p.

Інформаційні ресурси

1. The MPI Forum, which is the standardization forum for the Message Passing Interface (MPI). – Режим доступу: https://www.mpi-forum.org
2. Argonne National Laboratory, Center for Computational Science and Technology. – Режим доступу: http://www.mcs.anl.gov
3. CACR (Center of Advanced Computing Research). – Режим доступу: http://www.cacr.caltech.edu
4. Netlib is a collection of mathematical software, papers, and databases. – Режим доступу: http://netlib.org

Навчальне видання

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ MPI

ЕЛЕКТРОННІ МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

до лабораторних робіт студентам
факультету математики, фізики та інформаційних технологій

Електронне практичне видання

Укладачі:

Вербіцький Віктор Васильович

Максимов Артур Леонідович

В авторській редакції

Затвердж. авт. 01.11.2023. Шрифт Times New Roman.
Системні вимоги: операційна система сумісна з програмним забезпеченням для читання файлів формату PDF.
Обсяг 0,7 МБ. Зам. № 2688.

Видавець і виготовлювач
Одеський національний університет імені І. І. Мечникова
Свідоцтво суб'єкта видавничої справи ДК № 4215 від 22.11.2011 р.
65082, м. Одеса, вул. Єлісаветинська, 12, Україна
Тел.: (048) 723 28 39, e-mail: druk@onu.edu.ua