Тема №10: Бібліотеки паралельного програмування
Питання:

1. **Бібліотека паралельного програмування Pthreads**
2. **Бібліотека паралельного програмування OpenMP**
3. **Бібліотека паралельного програмування PVM**
4. **Бібліотека паралельного програмування MPI**

1. Бібліотека паралельного програмування Pthreads

Бібліотеки паралельного програмування являють собою набір підпрограм, що забезпечують створення процесів, управління ними, взаємодію і синхронізацію. Ці підпрограми, і особливо їх реалізація, залежать від того, який вид паралельності підтримує бібліотека — із змінними, що розділяються, або з обміном повідомленнями.

При створенні програм із змінними, що розділяються, на мовіС звичайно використовують стандартну бібліотеку Pthreads. При використанні обміну повідомленнями стандартними вважаються бібліотеки MPI і PVM; обидві вони мають широко використовувані загальнодоступні реалізації, які підтримують як С, так і Фортран. OpenMP є новим стандартом програмування із змінними, що розділяються, який реалізований основними виробниками швидкодіючих машин. На відміну від Pthreads, OpenMP є набором директив компілятора і підпрограм, має зв'язування, відповідне мові Фортран, і забезпечує підтримку обчислень, паралельних відносно даних.

Навчальний приклад: Pthreads

Механізми використання потоків і семафорів, підпрограми для блокування і умовних змінних можна використовувати і в програмі, що реалізовує метод ітерацій Якобі (лістинг 1) і одержаної безпосередньо з програми із розділеними змінними. Як завжди в програмах, що використовують Pthreads, головна підпрограма ініціалізувала атрибути потоку, читає аргументи з командного рядка, ініціалізує глобальні змінні і створює робочі процеси. Після того, як завершуються обчислення в робочих процесах, головна програма видає результати.

Лістинг 1. Метод ітерацій Якобі з використанням Pthreads

```
/* Метод ітерацій Якобі з використанням Pthreads */
#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXGRID 258 /* максимальний розмір сітки з межами */
#define MAXWORKERS 16 /* максимальне число робочих потоків */

void *Worker(void *);
void Barrier(int);

int gridSize, numWorkers, numIters, stripSize;
double maxDiff[MAXWORKERS];
double  grid[MAXGRID][MAXGRID], new[MAXGRID][MAXGRID];
//декларації інших глобальних змінних, наприклад бар'єрних прапорців ;

int main(int argc, char *argv[]){
       pthread_t workerid[MAXWORKERS]; /* індекси потоків і */
       pthread_attr_t attr;  /* їх атрибути */

       int i; double maxdiff = 0.0;

       /* установка глобальних атрибутів потоку */
       pthread_attr_init(&attr);
       pthread_attr_setscope(&attr PTHREAD_SCOPE_SYSTEM);

       /* зчитування аргументів з командної стрічки */
       /* передбачається, що gridSize кратне numWorkers */
       gridSize = atoi(argv[1]);
       numWorkers = atoi(argv[2]);
       numIters = atoi(argv[3]);
       stripSize = gridSize/numWorkers;
```

```
        //ініціалізація сіток і бар'єрних прапорців;

        /* створення робочих потоків і очікування їх завершення */
        for (i = 0; i < numWorkers; i++)
        pthread_create(&workerid[i] &attr, Worker,(void *) i);
        for (i = 0; i < numWorkers; i++)
        pthread_join(workerid[i], NULL);

        //обчислити maxdiff і вивести результати ;
}

void *Worker(void *arg) {
        int myid = (int) arg;
        double mydiff;
        int i, j, iters, firstRow, lastRow;

        /* визначити першу і останню рядки своєї смуги */
        firstRow = myid*stripSize + 1;
        lastRow = firstRow + stripSize - 1;
        for (iters = 1; iters <= numlters; iters++) {
        /* відновити свої крапки */
        for (i = firstRow; i<= lastRow; i++)
        for (j = 1; j <= gridSize; j++)
        new[i][j]= (grid[i-l][j]+ grid[i+l][j]+
        grid[i][j-l]+ grid[i][j+1])*0.25;
        Barrier(myid);

        /* знов відновити свої крапки */
        for (i = firstRow; i<= lastRow; i++)
        for (j = 1; j <= gridSize; j++)
        grid[i][j]  = (new[i-l][j]  +  new[i+l][j]  +
        new[i][j-1]  + new[i][j+l])*0.25;
        Barrier(myid);
        }

        //обчислити максимальну похибку на своїй смузі ;
        maxDiff[myid] = mydiff;
}

void Barrier (int workerid) {
/*  детально  не розглядається */
}
```

Кожен робочий потік відповідає за суцільну смугу в двох сітках. Щоб спростити задачу, оголошується фіксований максимальний розмір кожної сітки. Також передбачається, що розмір сіток кратний числу робочих потоків. Тіло бар'єру в програмі не записане; одна з Pthreads-реалізацій бар'єру за допомогою блокувань і умовних змінних. Проте, якщо кожен робочий потік виконується на своєму власному процесорі, набагато ефективніше використовувати бар'єр з розповсюдженням і активним очікуванням.

2. Бібліотека паралельного програмування OpenMP

OpenMP — це набір директив компілятора і бібліотечних підпрограм, використовуваних для представлення паралельності з розділенням пам'яті. Прикладні програмні інтерфейси (APIs) для OpenMP були розроблені групою, що представляла основних виробників швидкодіючого апаратного і програмного забезпечення. Інтерфейс Фортрану був визначений в кінці 1997 року, інтерфейс C/C++ — в кінці 1998, але стандартизація обох продовжується. Інтерфейси підтримують одні і ті ж функції, але виражаються по-різному із-за лінгвістичних відмінностей між Фортраном, C і C++.

Інтерфейс OpenMP в основному утворений набором директив компілятора. Програміст додає їх в послідовну програму, щоб вказати компілятору, які частини програми повинні виконуватися паралельно, і задати точки синхронізації. Директиви можна додавати поступово, тому OpenMP забезпечує розпаралелювання існуючого програмного забезпечення. Ці

властивості OpenMP відрізняють її від бібліотек Pthread і MPI, які містять підпрограми, що викликаються з послідовної програми і компоновані з нею, і вимагають від програміста уручну розподіляти роботу між процесами.

Нижче описано і проілюстровано використання OpenMP для програм Фортрану. Спочатку представлена послідовна програма для методу ітерацій Якобі. Потім в неї додані директиви OpenMP, що виражають паралельність. В кінці розділу стисло описані додаткові директиви і інтерфейс C/C++.

У лістингу 1 представлений ескіз послідовної програми для методу ітерацій Якобі. Її синтаксис своєрідний, оскільки програма написана з використанням вимог Фортрану по представленню даних з фіксованою крапкою. Рядки з коментарями починаються з букви *c* в першій колонці, а декларації і оператори — з колонки 7. Додаткові коментарі починаються символом `!`. Всі коментарі продовжуються до кінця рядка.

Лістинг 1 Послідовний метод ітерацій Якобі на Фортрані

```
      program main               ! головна програма
      integer n,maxiters         ! загальні дані
      common  /idat/  n,maxiters
      зчитування значень n і maxiters (не показано)
      call jacobi()
      stop
      end

      subroutine jacobi()        ! реалізує метод ітерацій Якобі
      integer n,maxiters         ! повторне оголошення змінних
      common /idat/ n,maxiters
      integer i,j,iters
      double precision grid(n,n), new(n,n)
      double precision maxdiff, tempdiff
      ініціалізувати grid і new (див. текст)

с головний цикл: відновити  сітки maxiters разів
      do iters = 1,maxiters, 2   ! цикл від 1 до maxiters з кроком 2
            do j = 2,n-1         ! відновити точки new
                  do i  =  2,n-1
                        new(i,j)= (grid(i-1,j)+ grid(i+1,j)+
                        grid(i,j-1)+ grid(i,j+1)) * 0.25
                  enddo
            enddo
      do j = 2,n-1         ! відновити точки grid
                  do i = 2,n-1
                        grid(i,j)= (new(i-1,j)+ new(i+1,j)+ new(i,j-1)+
                        new(i,j+1)) * 0.25
                  enddo
            enddo
      enddo

      с обчислення максимальної похибки
      maxdiff =0.0
      do j = 2,n-1
            do i = 2,n-1
                  tempdiff = abs(grid(i,j) -new(i,j)) maxdiff = max(maxdiff,tempdiff)
            enddo
      enddo
      return
      end
```

Послідовна програма складається з двох підпрограм: `main` і `jacobi`. У підпрограмі `main` прочитуються значення `n` (розмір сітки з межами) і `maxiters` (максимальне число ітерацій), а потім викликається підпрограма `jacobi`. Значення даних зберігаються в загальній області пам'яті і, отже, неявно передаються з `main` в `jacobi`. Це дозволяє `jacobi` розподіляти пам'ять для масивів `grid` і `new` динамічно.

У підпрограмі `jacobi` реалізований послідовний алгоритм. Основна відмінність між програмами обумовлена синтаксичною відмінністю псевдо-С від Фортрану. У Фортрані нижня межа кожної розмірності масиву рівна 1, тому індекси внутрішніх точок матриць по рядках і

стовпцях приймають значення від 2 до n-1. Крім того, Фортран зберігає матриці в пам'яті машини по стовпцях, тому у вкладених циклах do спочатку виконуються ітерації по стовпцях, а потім по рядках.

У OpenMP використовується модель виконання "розгалуження-злиття" (fork-join). Спочатку існує один потік виконання. Зустрівши одну з директив parallel, компілятор вставляє код, щоб розділити один потік на декілька підпотоків. Разом головний потік і підпотоки утворюють так звану безліч робочих потоків. Дійсна кількість робочих потоків встановлюється компілятором (по замовчуванню) або визначається користувачем — або статично за допомогою змінних середовища (environment), або динамічно за допомогою виклику підпрограми з бібліотеки OpenMP.

Щоб розпаралелювати програму за допомогою OpenMP, програміст спочатку визначає частини програми, які можуть виконуватися паралельно, наприклад цикли, і оточує їх директивами parallel і end parallel. Кожен робочий потік виконує цей код, обробляючи різні підмножини в просторі ітерацій (для циклів, паралельних до даних) або викликаючи різні підпрограми (для програм, паралельних по задачах). Потім в програму додаються додаткові директиви для синхронізації потоків під час виконання. Таким чином, компілятор відповідає за розділення потоків і розподіл роботи між ними (у циклах), а програміст повинен забезпечити достатню синхронізацію.

Як конкретний приклад розглянемо наступний послідовний код, в якому внутрішні точки grid і new ініціалізувалися нулями.

```
do j = 2,n-1
       do i = 2,n-1
              grid(i,j) = 0.0
              new(i,j) = 0.0
       enddo
enddo
```

Щоб розпаралелювати цей код, додамо в нього три директиви компілятора OpenM.

```
!$omp parallel do
!$omp& shared(n,grid,new), private(i,j)
       do j = 2,n-1
              do i = 2,n-1
                     grid(i,j) =0.0 new(i,j)= 0.0
              enddo
       enddo
!$omp end parallel do
```

Кожна директива компілятора починається з !$omp. Перша визначає початок паралельного циклу do. Друга доповнює першу, що позначено додаванням символу & до !$omp. У другій директиві повідомляється, що у всіх робочих потоках n, grid і new є змінними, що розділяються, а i і j — локальними. Остання директива вказує на кінець паралельного циклу do і встановлює точку неявної бар'єрної синхронізації.

У даному прикладі компілятор розділить ітерації зовнішнього циклу do (по j) і призначить їх робочим процесам деяким способом, залежним від реалізації. Щоб управляти призначенням, програміст може додати пропозицію schedule. У OpenMP підтримуються різні види призначення, зокрема по блоках, по смугах (циклічно) і динамічно (портфель завдань). Кожен робочий потік виконуватиме внутрішній цикл do (по i) для призначених йому стовпців.

У лістингу 4 представлений один із способів розпаралелювання тіла підпрограми jacobi з використанням директив OpenMP. Основний потік розділяється на робочі потоки для ініціалізації сіток, як було показано вище. Проте maxdif ініціалізується в основному потоці. Ініціалізація maxdiff перенесена, оскільки її бажано виконати в одному потоці до початку обчислень максимальної похибки. (Натомість можна було б використовувати директиву single, що обговорюється нижче.)

Після ініціалізації змінних, що розділяються, слідує директива `parallel`, що розділяє основний потік на декілька робочих. У наступних двох пропозиціях вказано, які змінні є загальними, а котрі — локальними. Кожен робочий виконує головний цикл. У цикл додані директиви `do` для вказівки, що ітерації зовнішніх циклів, оновлюючі `grid` і `new`, повинні бути розділені між робочими. Закінчення цих циклів позначені директивами `end do`, які також забезпечують неявні бар'єри.

Після головного циклу (який завершується одночасно всіма робочими) використовується ще одна директива `do`, щоб максимальна похибка обчислювалася паралельно. У цьому розділі `maxdiff` використовується як змінна редукція, тому до директиви `do` додано пропозицію `reduction`. Семантика змінної редукції така, що кожне оновлення є неподільним (у даному прикладі за допомогою функції `max`). Насправді OpenMP реалізує змінну редукції, використовуючи приховані змінні в кожному робочому потоці; значення цих змінних "зливаються" неподільним чином в одне на неявному бар'єрі в кінці розпаралелюючого циклу.

Програма в лістингу 2 ілюструє найбільш важливі директиви OpenMP. Бібліотека містить декілька додаткових директив для розпаралелювання, синхронізації і управління робочим середовищем (data environment). Наприклад, для забезпечення повнішого управління синхронізацією операторів можна використовувати наступні директиви.

`critical`	Виконати блок операторів як критичну секцію.
`atomic`	Виконати одного оператора неподільним чином.
`single`	В одному робочому потоці виконати блок операторів.
`barrier`	Виконати бар'єр, встановлений для всіх робочих потоків.

У OpenMP є декілька бібліотечних підпрограм для запитів до робочого середовища і управління ним. Наприклад, є підпрограми установки числа робочих потоків і його динамічної зміни, а також визначення ідентифікатора потоку.

Лістинг 2 Паралельний метод ітерацій Якобі з використанням OpenMP

```
      subroutine jacobi()
      оголошення загальних, розділяємих і локальних змінних
      паралельна ініціалізація grid і new (див. текст)
      maxdiff = 0.0       ! ініціалізація в основному потоці
с старт робочих потоків; кожен виконує головний цикл
!$omp parallel
!$отр& shared(n,maxiters,grid,new,maxdiff)
!$omp& private(i,j,iters,tempdiff)
      do iters = l,maxiters,2
!$omp do                  !  розділення ітерацій зовнішнього циклу
      do  j =  2,n-l
            do i  = 2,n-l
                  new(i,j)  = (grid(i-l,j)  +  grid(i+l,j)  +
                        grid(i,j-l)  +grid(i,j+1)) * 0.25
            enddo
      enddo
!$omp end do              ! неявний бар'єр
!$omp do                  ! розділення ітерацій зовнішнього циклу
      do  j =  2,n-l
            do i =  2,n-l
                  grid(i,j)  = (new(i-l,j) +new(i+l,j) +
                   new(i,j-l)  +new(i,j+l)) * 0.2 5
            enddo
      enddo
!$omp enddo               ! неявний бар'єр
enddo                     ! кінець головного циклу

с обчислення максимальної похибки в змінній редукції
!$omp do                  ! розділення ітерацій зовнішнього циклу
!$omp$ reduction (max:  maxdiff) ! використовується змінна редукції
      do  j =  2,n-l
```

```
                do  i  =  2,n-1
                      tempdiff  =  abs(grid(i,j) -new(i,j))
                      maxdiff  = max(maxdiff, tempdiff) ! неподільне оновлення
                enddo
          enddo
!$omp end do              ! неявний бар'єр
!$omp end parallel        ! кінець паралельного розділу
          return
          end
```

Інтерфейс OpenMP для C/C++ забезпечує ті ж функції, що і інтерфейс для Фортрану. Різниця між ними обумовлена лінгвістичними відмінностями C/C++ від Фортрану. Наприклад, директива паралельності `parallel` має наступний вигляд:

```
pragma omp parallel  clauses
```

Ключове слово `pragma` означає директиву компілятора. Оскільки в С замість циклів `do` для визначеної кількості ітерацій використовують цикл `for`, еквівалентом директиви `do` в С являється:

```
pragma omp for calasses
```

В інтерфейсі C/C++ немає директиви `end`. Замість неї частини коду програми встановлюються у фігурні скобки, котрі визначають зону дії директив.

3. Бібліотека паралельного програмування PVM

PVM містить безліч підпрограм для глобальної і локальної взаємодії. Деякі з них використані в програмі для методу ітерацій Якобі (лістинг 3).

Лістинг 2. Метод ітерацій Якобі з використанням PVM

```
/* Метод ітерацій Якобі з використанням MPI */
#include < PVM.h>
#include <stdio.h>
#define MAXGRID 258 /* максимальний розмір сітки з межами */
#define COORDINATOR 0 /* номер процесу, що управляє */
#define TAG 0  /* не використовується */

static void Coordinator(int,int,int);
static void Worker(int,int,int,int,int);

int main(int argc, char *argv[]){
       int myid, numIters;
       int numWorkers, gridSize; /* передбачається, що */
       int stripSize;  /* gridSize кратне numWorkers */

       PVM_Init(&argc &argv); /* ініціалізація PVM */
       PVM_Comm_rank(PVM_COMM_WORLD, &myid);
       PVM_Comm_size(PVM_COMM_WORLD, &numWorkers);
       numWorkers--;  /* один керівник, інші - робітники */

       //прочитати gridSize u numIters; обчислити stripSize;

       if (myid == COORDINATOR)
       Coordinator(numWorkers, stripSize, gridSize);
       else
       Worker(myid,numWorkers,stripSize,gridSize,numIters);
       PVM_Finalize(); /* закінчення роботи PVM */
}

static void Coordinator(int numWorkers, int stripSize, int gridSize){
       double grid[MAXGRID]{MAXGRID];
       double mydiff = 0.0, maxdiff = 0.0;
       int i, worker, startrow, endrow; PVM_Status status;

       /* отримати остаточні значення в сітці від Workers */
       for (worker = 1; worker <= numWorkers; worker++) {
       startrow = (worker-1)*stripSize + 1;
```

```
        endrow = startrow +stripSize -1;
        for (i = startrow; i<= endrow; i++)
        PVM_Recv(&grid[i] [1], gridSize PVM_DOUBLE, worker, TAG PVM_COMM_WORLD, &status);
        }

        /* редукція похибок від Workers */
        PVM_Reduce(&mydiff &maxdiff, 1 PVM_DOUBLE, PVM_MAX, COORDINATOR PVM_COMM_WORLD);

        //вивести результати;
}

static void Worker(int myid, int numWorkers, int stripSize, int gridSize, int numIters){
        double grid [2][MAXGRID][MAXGRID];
        double mydiff, maxdiff;
        int i, j, iters;
        int current = 0, next = 1; /* поточна і наступна сітки */
        int left, right; /* сусідні робітники */
        PVM_
tatus status;

        //ініціалізувати матриці; визначити сусідів left і right;

        for (iters = 1; iters <=  numIters; iters++)  {
        /*  обмін  межами з  сусідами  */
        if (right !=  0) PVM_Send(&grid[next][stripSize][1],gridSize PVM_DOUBLE, right, TAG
PVM_COMM_WORLD); if (left != 0) PVM_Send(&grid[next][1][1],gridSize,PVM_DOUBLE, left, TAG
PVM_COMM_WORLD);
        if (left != 0) PVM_Recv(&grid[next][0][1],gridSize PVM_DOUBLE, left, TAG
PVM_COMM_WORLD, &status);
        if (right != 0) PVM_Recv(&grid[next][stripSize+1][1], gridSize PVM_DOUBLE, right, TAG
PVM_COMM_WORLD, &Status) ;
        /* відновити свої крапки */
        for (i = 1; i <= StripSize; i++)
        for (j = 1; j <= gridSize; j++)
        gridfnext] [i] [j] = (grid[current] [i-1] [j]+ grid[current][i+1][j]+ grid[current]
[i] [j-1]+ grid[current][i][j+1]) * 0.25;
        current = next; next = 1-next; /* поміняти місцями сітки */
        }

        /* відправити рядки остаточної сітки процесу, що управляє */
        for (i = 1; i <= stripSize; i++) {
        PVM_Send(&grid[current][i][1], gridSize PVM_DOUBLE, COORDINATOR, TAG PVM_COMM_WORLD);

        }

        //обчислити mydiff;

        /* редукція mydiff в процесі, що управляє */
        PVM_Reduce(&mydiff &maxdiff, 1 PVM_DOUBLE, PVM_MAX, COORDINATOR PVM_COMM_WORLD);

        }
```

Програма в лістингу 3 містить три функції: `main`, `Coordinator` і `Worker`. Передбачається, що виконуються всі `numWorkers+1` екземплярів програми. (Вони запускаються за допомогою команд, специфічних для конкретної версії PVM.) Кожен екземпляр починається з виконання підпрограми main, яка ініціалізувала PVM і прочитує аргументи командного рядка. Потім залежно від номера (ідентифікатора) екземпляра з `main` викликається або процес `Coordinator`, або робітник `Worker`, що управляє.

Кожен процес `Worker` відповідає за смугу крапок. Спочатку він ініціалізував обидві свої сітки і визначає своїх сусідів, `left` і `right`. Потім робітники багато разів обмінюються з сусідами краями своїх смуг і обновляють свої точки. Після numIters циклів обмена- оновлення кожен робітник відправляє рядки своєї смуги процесу, що управляє, вы-числяет максимальну різницю між парами крапок на своїй смузі і, нарешті, викликає PVM_Reduce, щоб відправити mydiff процесу, що управляє.

Процес `Coordinator` просто збирає результати, що відправляються робочими процесами. Спочатку він одержує рядки остаточної сітки від всіх робітників. Потім викликає підпрограму `PVM_Reduce`, щоб одержати і скоротити максимальні різниці, обчислені кожним робочим процесом. Помітимо, що аргументи у викликах `PVM_Reduce` однакові і в робітниках, і в керівнику процесах. Передостанній аргумент `COORDINATOR` задає, що редукція повинна відбуватися в процесі, що управляє.

4. Бібліотека паралельного програмування MPI

Найбільш поширеною технологією програмування паралельних комп'ютерів з розподіленою пам'яттю в даний час є MPI. Основним способом взаємодії паралельних процесів в таких системах є передача повідомлень один одному. Це і відображено в назві даної технології — Message Passing Interface. Стандарт MPI фіксує інтерфейс, який повинні дотримувати як система програмування MPI на кожній обчислювальній системі, так і користувач при створенні своїх програм. Сучасні реалізації найчастіше відповідають стандарту MPI версії 1.1. У 1997—1998 роках з'явився стандарт MPI-2.0, що значно розширив функціональність попередньої версії. Проте дотепер цей варіант MPI не набув широкого поширення. Якщо версію стандарту явно не вказано, то мається на увазі, що ми маємо справу із стандартом 1.1.

MPI підтримує роботу з мовами C і Fortran. Всі приклади і описи всіх функцій подані з використанням мови C. Однак це абсолютно не є принциповим, оскільки основні ідеї MPI і правила оформлення окремих конструкцій для цих мов багато в чому схожі.

Повна версія інтерфейсу містить опис більше 120 функцій. Якщо описувати його повністю, то цьому потрібно присвячувати цілу книгу. Наше завдання — пояснити ідею технології і допомогти освоїти необхідні на практиці компоненти. Найбільш вживані функції бідіть описані в даному розділі

Інтерфейс підтримує створення паралельних програм в стилі MIMD, що має на увазі об'єднання процесів з різними текстами програм. Проте на практиці програмісти набагато частіше використовують SPMD-модель, в рамках якої для всіх паралельних процесів використовується один і той же код. В даний час все більше і більше реалізацій MPI підтримують роботу з нитками.

Всі додаткові об'єкти: імена функцій, константи, зумовлені типи даних і т. п., використовувані в MPI, мають префікс `MPI_`. Наприклад, функція відправлення повідомлення від одного процесу іншому має ім'я `MPI_Send`. Якщо користувач не використовуватиме в програмі імен з таким префіксом, то конфліктів з об'єктами MPI свідомо не буде. Всі описи інтерфейсу MPI зібрані у файлі mpi.h, тому на початку MPI-програми повинна стояти директива `#inciude <mpi. h>`.

MPI-программа — це безліч паралельних взаємодіючих процесів. Всі процеси породжуються один раз, утворюючи паралельну частину програми. В ході виконання MPI-програми породження додаткових процесів або знищення тих, що існують не допускається.

Кожен процес працює в своєму адресному просторі, ніяких загальних змінних або даних в MPI немає. Основним способом взаємодії між процесами є явна посилка повідомлень.

Для локалізації взаємодії паралельних процесів програми можна створювати групи процесів, надаючи їм окреме середовище для спілкування — коммуникатор. Склад утворюваних груп довільний. Групи можуть повністю входити одна в іншу, не перетинатися або перетинатися частково. При старті програми завжди вважається, що всі породжені процеси працюють в рамках всеосяжногокоммуникатора, що має зумовлене ім'я mpi_comm_world. Цей коммуникатор існує завжди і служить для взаємодії всіх процесів MPI-програми.

Кожен процес MPI-програми має унікальний атрибут номер процесу, який є цілим ненегативним числом. За допомогою цього атрибуту відбувається значна частина взаємодії процесів між собою. Ясно, що в одному і тому ж комунікаторі всі процеси мають різні номери. Але оскільки процес може одночасно входити в різні комунікатори, то його номер в одному

комунікаторі може відрізнятися від його номера в іншому. Звідси два основні атрибути процесу: комунікатор і номер в комунікаторі.

Якщо група містить *n* процесів, то номер будь-якого процесу в даній групі лежить в межах від *0* до *n — 1*. Подібна лінійна нумерація не завжди адекватно відображає логічний взаємозв'язок процесів програми. Наприклад, згідно завдання, процеси можуть розташовуватися у вузлах прямокутних решіток і взаємодіяти тільки з своїми безпосередніми сусідами. Таку ситуацію користувач може легко відобразити в своїй програмі, описавши відповідну віртуальну топологію процесів. Ця інформація може виявитись корисною при відображенні процесів програми на фізичні процесори обчислювальної системи. Сам процес відображення в MPI ніяк не специфікується, проте система підтримки MPI у деяких випадках може значно зменшити комунікаційні накладні витрати, якщо скористається знанням віртуальної топології.

Основним способом спілкування процесів між собою є посилка повідомлень. Повідомлення — це набір даних деякого типу. Кожне повідомлення має декілька атрибутів, зокрема, номер процесу-відправника, номер процесу-одержувача, ідентифікатор повідомлення та ін. Одним з важливих атрибутів повідомлення є його ідентифікатор або тег. По ідентифікатору процес, що приймає повідомлення, наприклад, може розрізнити два повідомлення, що прийшли йому від одного і того ж процесу. Сам ідентифікатор повідомлення є цілим невід'ємним числом в діапазоні від 0 до 32 767. Для роботи з атрибутами повідомлень введена структура `MPI_Status`, поля якої дають доступ до значень атрибутів.

На практиці повідомлення найчастіше являється набором однотипних даних, розташованих підряд один за одним в деякому буфері. Таке повідомлення може полягати, наприклад, з двохсот цілих чисел, які користувач розмістив у відповідному цілочисельному векторі. Це типова ситуація, на неї орієнтовано більшість функцій MPI, проте така ситуація має, принаймні, два обмеження. По-перше, іноді необхідно скласти повідомлення з різнотипних даних. Звичайно ж, можна окремим повідомленням послати кількість дійсних чисел, що містяться в подальшому повідомленні, але це може бути і незручно програмісту, і не так ефективно. По-друге, не завжди дані для посилки займають безперервну область в пам'яті. Якщо в Fortran елементи стовпців матриці розташовані в пам'яті один за одним, то елементи рядків вже йдуть з деяким кроком. Щоб послати рядок, дані потрібно спочатку упакувати, передати, а потім знов розпакувати.

Щоб зняти вказані обмеження, в MPI передбачений механізм для вводу похідних типів даних (derived datatypes). Описавши склад і схему розміщення в пам'яті даних для посилки, користувач надалі працює з такими типами так само, як і із стандартними типами даних MPI. Оскільки власні типи даних і віртуальні топології процесів використовуються на практиці не дуже часто, тому їх не описують детально.

Вправи і завдання до теми №10

1. Сформулюйте характерні особливості моделі передачі повідомлень.
2. Чи допускає OpenMP зміну кількості паралельних ниток по ходу роботи програми?
3. Чи можна автоматично конвертувати DVM – програму в програму на OpenMP.
4. Спробуйте виділити найсильніші і найслабші боки кожної з технологій OpenMP і DVM.

ЛІТЕРАТУРА

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб: БХВ-Петербург, 2002.

2. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем. М.:Мир, 1991.

3. Программирование на параллельных вычислительных системах: Пер с англ./Под ред. Р.Бэбба.М.:Мир, 1991.

4. Бройнль Т. Паралельне програмування: Початковий курс: Навчальний посібник. – К.:Вища школа.,1997.

5. Воеводин В.В. Математические основы параллельных вычислений.- М.: Изд-во МГУ, 1991.

6. Векторизация программ: теория, методы, реализация: Пер. с англ. и нем. /Под ред. Г.Д.Чинина. - М:. Мир, 1991.

7. Корнеев В.В. Параллельные вычислительные системы. М.: Нолидж, 1999

8. С. Немнюгин, О.Стесик Параллельное программирование для многопроцессорных вычислительных систем. – СПб: БХВ-Петербург, 2002.

9. Pacheco P. Parallel Programming With MPI (див. www.parallel.ru).

10. Gropp W., Lusk E., Skjellum A. Using MPI (див. www.parallel.ru).

11. Питерсон Дж. Теория сетей Петри і моделирования систем: Пер. с англ. -М.: Мир, 1984. -264 с., ил.

12. Internet-сайти

13. С.Д. Погорілий, Ю.В. Бойко, Д.Б. Грязнов, О.Д. Ломакін, В.А. Мар'яновський, 2008 ISSN 1727-4907. Проблеми програмування. 2008. № 2-3. Спеціальний випуск

Ресурси Інтернет стосовно паралельних обчислень.

1. http://www.globus.org – Побудова метакомп'ютера.
2. http://www.gridforum.org– Побудова метакомп'ютера.
3. http://www.top500.org – Характеристики 500 найпотужніших комп'ютерів в світі.
4. http://www.mpiforum.org – Повний варіант описів стандартів MPI.
5. http://www.keldysh.ru.norma – Опис системи програмування НОРМА.
6. http://www.citforum.ru – Сервер інформаційних технологій.
7. http://www.parallel.ru – Інформаційно-аналітичний центр з паралельних обчислень.
8. http://www.csa.ru – Інститут високопродуктивних обчислень і баз даних.
9. http://www.hpc.nw.ru – Високопродуктивні обчислення.
10. http://www.epm.ornl.gov/pvm/ - інформація про PVM.
11. http://www.beowulf.org – Інформація про кластери.