МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ПУЛЮЯ

Кафедра комп'ютерних наук

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ

ЛАБОРАТОРНИХ РОБІТ

з дисципліни

«Розподілені системи моніторингу та керування»

для студентів освітнього рівня «бакалавр»
спеціальності 125 «Кібербезпека»

Тернопіль
2016

УДК 681.3+378
ББК 32.97+74
М54

Укладачі:
Шимчук Г.В., асистент,
Маєвський О.В., ст. викладач,
Назаревич О.Б., канд. техн. наук, асистент.

Рецензент:
М.М. Касянчук, канд. фіз.-мат. наук, доцент.

Методичні вказівки розглянуто й затверджено на засіданні
кафедри компютерних наук
Тернопільського національного технічного університету імені Івана Пулюя
протокол № 2 від 09 вересня 2015 р.

Схвалено та рекомендовано до друку на засіданні методичної комісії факультету
комп'ютерно-інформаційних систем та програмної інженерії Тернопільського
національного технічного університету імені Івана Пулюя
протокол № 2 від 25 вересня 2015 р.

М54 Методичні вказівки до виконання лабораторних робіт з дисципліни «Розподілені системи моніторингу та керування» для студентів освітнього рівня «бакалавр» спеціальності 125 – «Кібербезпека» / Укладачі : Шимчук Г.В., Маєвський О.В., Назаревич О.Б. – Тернопіль : Вид-во ТНТУ імені Івана Пулюя, 2016 – 124 с.

УДК 681.3+378
ББК 32.97+74

Методичні вказівки до виконання лабораторних робіт призначені для полегшення засвоєння дисципліни «Розподілені системи моніторингу та керування». Складається з урахуванням модульної системи навчання.

Вказівки складені з урахуванням матеріалів літературних джерел, названих у списку.

Відповідальний за випуск: *М.В. Приймак*, докт. техн. наук, професор

ЗМІСТ

ВСТУП

Мета курсу полягає у підготовці майбутніх фахівців для ефективного використання сучасних обчислювальних систем у процесі виконання своїх професійних обов’язків.

Завдання дисципліни – навчитися розробляти паралельне програмне забезпечення для розв'язування прикладних задач з використанням сучасних технологій: NET, MPI, OpenMP та GRID. Навчитися обґрунтовувати продуктивність та ефективність використання технологій паралельних та розподілених обчислень.

Вивчення дисципліни "Розподілені системи моніторингу та керування" дасть можливість для студента отримати ґрунтовну підготовку в областях архітектури та стандартів компонентних моделей, комунікаційних засобів і розподілених обчислень, уміння розв'язувати проблеми масштабованості, підтримки віддалених компонентів і взаємодії різних програмних платформ в розподілених корпоративних інформаційних системах рівня підприємства, дасть можливість базові використовувати знання в галузі інформатики й сучасних інформаційних технологій, навички використання програмних засобів і навички роботи в комп'ютерних мережах, уміння створювати бази даних і використовувати Інтернет-ресурс.

Дані методичні вказівки призначені для виконання лабораторних робіт у 5 семестрі вивчення дисципліни “Розподілені системи моніторингу та керування” для студентів освітнього рівня бакалавр спеціальності 125 «Кібербезпека».

Виконання лабораторних робіт повинно забезпечити закріплення теоретичних знань, отриманих при вивченні лекційного матеріалу та практичної частини дисципліни.

В методичних вказівках розглянуті основні теоретичні питання, пов’язані з виконанням лабораторних робіт.

ЛАБОРАТОРНА РОБОТА № 1

Тема: Організація багатозадачності в середовищі ОС Windows за допомогою процесів і потоків

Мета: Ознайомлення з можливостями багатозадачності ОС Windows та робота з процесами і потоками, використовуючи Visual C++.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ

Вступ

32-розрядна платформа Windows завдяки своїй структурі надає програмістам нові можливості, що захоплюють дух, – можливості ***багатозадачності***. Перш за все важливо з'ясувати для себе, коли слід подумати про її використання у своєму застосуванні. Відповідь так само очевидна, як і визначення терміну "багатозадачність" – вона потрібна тоді, коли ви хочете, щоб декілька ділянок коду виконувалося одночасно. Наприклад, ви хочете, щоб якісь дії виконувалися у фоновому режимі, або щоб протягом ресурсоємних обчислень, що виконуються вашою програмою, вона продовжувала реагувати на дії користувача. При цьому важливо розібратися зосновними поняттями, які забезпечують цю багатозадачність, – процеси і потоки.

Процеси

Процесом (process) зазвичай називають екземпляр виконуваної програми.

Хоча на перший погляд здається що програма і процес поняття практично однакові, вони фундаментально відрізняються один від одного. ***Програма*** є статичним набором команд, а процес це набір ресурсів і даних, що використовуються при виконанні програми. Процес в Windows складається з наступних компонентів:

- Структура даних, що містить всю інформацію про процес, зокрема список відкритих дескрипторів різних системних ресурсів, унікальний ідентифікатор процесу, різну статистичну інформацію і т.д.;
- Адресний простір – діапазон адрес віртуальної пам'яті, яким може користуватися процес;
- Виконувана програма і дані, що проектуються на віртуальний адресний простір процесу.

Потоки

Процеси інертні. Відповідають же за виконання коду, що міститься в адресному просторі процесу, потоки. ***Потік*** (thread) – це певна сутність усередині процесу, що отримує процесорний час для свого виконання. У кожному процесі є мінімум один потік. Цей ***первинний потік*** створюється системою автоматично при створенні процесу. Далі цей потік може породити інші потоки, ті у свою чергу нові і т.д. Таким чином, один процес може володіти декількома потоками, і тоді вони одночасно виконують код в адресному просторі процесу. Кожен потік має:

- Унікальний ***ідентифікатор потоку***;

- Вміст набору ***регістрі процесора***, що відображають стан процесора;
- Два ***стеки***, один з яких використовується потоком при виконанні в режимі ядра, а інший – в призначеному для користувача режимі;
- Закриту область пам'яті, що називається ***локальною пам'яттю потоку*** (thread local storage, TLS) і використовується підсистемами, run-time бібліотеками і DLL.

Планування потоків

Щоб всі потоки працювали, операційна система відводить кожному з них певний ***процесорний час***. Тим самим створюється ілюзія одночасного виконання потоків (зрозуміло, що для багатопроцесорних комп'ютерів можливий істинний паралелізм). У Windows реалізована система ***витісняючого планування на основі пріоритетів***, в якій завжди виконується потік з найбільшим пріоритетом, що готовий до виконання. Вибраний для виконання потік працює протягом деякого періоду, званого ***квантом***. Квант визначає, скільки часу виконуватиметься потік, поки операційна система не перерве його. Після закінчення кванта операційна система перевіряє, чи готовий до виконання інший потік з таким же (або великим) рівнем пріоритету. Якщо таких потоків не виявилось, поточному потоку виділяється ще один квант. Проте потік може не повністю використовувати свій квант. Як тільки інший потік з вищим пріоритетом готовий до виконання, поточний потік витісняється, навіть якщо його квант ще не закінчився.

Квант не вимірюється в яких би то не було одиницях часу, а виражається цілим числом. Для кожного потоку зберігається поточне значення його кванта. Коли потоку виділяється квант процесорного часу, це означає, що його квант встановлюється в початкове значення. Воно залежить від операційної системи. Наприклад, для Win2000 Professional початкове значення кванта рівне 6, а для Win2000 Server – 36.

Усякий раз, коли виникає переривання від таймера, з кванта потоку віднімається 3, і так до тих пір, поки він не досягне нуля. Частота спрацьовування таймера залежить від апаратної платформи. Наприклад, для більшості однопроцесорних x86 систем він складає 10мс, а на більшості багатопроцесорних x86 систем – 15мс.

У будь-якому випадку операційна система повинна визначити, який потік виконувати наступним. Вибравши новий потік, операційна система перемикає контекст. Ця операція полягає в збереженні параметрів виконуваного потоку (регістри процесора, покажчики на стек ядра і призначений для користувача стек, покажчик на адресний простір, в якому виконується потік і ін.), і завантаженні аналогічних параметрів для іншого потоку, після чого починається виконання нового потоку.

Планування в Windows здійснюється на рівні потоків, а не процесів. Це здається зрозумілим, оскільки самі процеси не виконуються, а лише надають ресурси і контекст для виконання потоків. Тому при плануванні потоків, система не звертає уваги на те, якому процесу вони належать. Наприклад, якщо процес А має 10 готових до виконання потоків, а процес В – два, і всі 12 потоків мають однаковий пріоритет, кожний з потоків отримає 1/12 процесорного часу.

Пріоритети процесів та потоків

У Windows існує 32 рівні ***пріоритету***, від 0 до 31. Вони групуються так: 31 – 16 рівні реального часу; 15 – 1 динамічні рівні; 0 – системний рівень, зарезервований для потоку обнулення сторінок (zero-page thread).

При створенні процесу, йому призначається один з шести ***класів пріоритетів***:

- ***Real time class*** (значення 24)
- ***High class*** (значення 13)
- ***Above normal class*** (значення 10)
- ***Normal class*** (значення 8)
- ***Below normal class*** (значення 6)
- ***Idle class*** (значення 4).

У Windows NT/2000/XP можна подивитися пріоритет процесу в Task Manager.

Пріоритет кожного потоку (***базовий пріоритет потоку***) складається з пріоритету його процесу і відносного пріоритету самого потоку. Є сім відносних пріоритетів потоків:

- ***Normal***: такий же як і у процесу;
- ***Above normal***: +1 до пріоритету процесу;
- ***Below normal***: -1;
- ***Highest***: +2;
- ***Lowest***: -2;
- ***Time critical***: встановлює базовий пріоритет потоку для Real time класу в 31, для решти класів в 15.
- ***Idle***: встановлює базовий пріоритет потоку для Real time класу в 16, для решти класів в 1.

Прив'язка до процесорів

Якщо операційна система виконується на машині, де встановлено більш за один процесор, то за умовчанням, потік виконується на будь-якому доступному процесорі. Проте в деяких випадках, набір процесорів, на яких потік може працювати, може бути обмежений. Це явище називається ***прив'язкою до процесорів*** (processor affinity). Можна змінити прив'язку до процесорів програмно, через Win32-функції планування.

Пам'ять для процесів та потоків

Кожному процесу в Win32 доступний лінійний 4-гігабайтний (232 = 4 294 967 296) ***віртуальний адресний простір***. Зазвичай верхня половина цього простору резервується за операційною системою, а друга половина доступна процесу.

Віртуальний адресний простір процесу доступний всім потокам цього процесу. Іншими словами, всі потоки одного процесу виконуються в єдиному адресному просторі.

З іншого боку, механізм віртуальної пам'яті дозволяє ізолювати процеси один від одного. Потоки одного процесу не можуть посилатися на адресний простір іншого процесу.

Віртуальна пам'ять може зовсім не відповідати структурі фізичної пам'яті. Диспетчер пам'яті транслює віртуальні адреси на фізичні, по яких реально зберігаються дані. Оскільки далеко не всякий комп'ютер в змозі виділити по 4 Гбайт фізичної пам'яті на кожен процес, використовується ***механізм підкачки*** (swapping).

Створення процесів

Створення Win32 процесу здійснюється викликом однієї з таких функцій, як ***CreateProcess***, CreateProcessAsUser (для Win NT/2000), CreateProcessWithLogonW і відбувається у декілька етапів:

1. Створюється об'єкт Win32 "процес".
2. Створюється первинний потік (стек, контекст і об'єкт "потік").
3. Підсистема Win32 повідомляється про створення нового процесу і потоку.
4. Починається виконання первинного потоку.
5. У контексті нового процесу і потоку ініціалізувався адресний простір (наприклад, завантажуються необхідні DLL) і починається виконання програми.

Завершення процесів

Процес завершується якщо:

- Вхідна функція первинного потоку повернула управління.
- Один з потоків процесу викликав функцію ***ExitProcess***.
- Потік іншого процесу викликав функцію ***TerminateProcess***.

Коли процес завершується, всі User- і GDI-об'єкти, створені процесом, знищуються, об'єкти ядра закриваються (якщо їх не використовує інший процес), адресний простір процесу знищується.

Приклад 1. Програма створює процес "Калькулятор".

```
#include <windows.h>
int main(int argc, char* argv[])
{
     STARTUPINFO si;
     PROCESS_INFORMATION pi;
     ZeroMemory( &si, sizeof(si));
     si.cb = sizeof(si);
     ZeroMemory( &pi, sizeof(pi));
     if( !CreateProcess( NULL, "c:/windows/calc.exe", NULL, NULL,
FALSE
     0, NULL, NULL &si, &pi))
     return 0;
     // Close process and thread handles.
     CloseHandle( pi.hProcess );
     CloseHandle( pi.hThread );
     return 0;
}
```

Створення потоків

Первинний потік створюється автоматично при створенні процесу. Решта потоків створюється функціями Win32 API ***CreateThread*** і CreateRemoteThread (тільки у Win NT/2000/XP). Якщо використовувати надбудову MFC (для Visual C++), то потік можна створити за допомогою функції ***AfxBeginThread***.

Завершення потоків

Потік завершується якщо:

- Функція потоку повертає управління.
- Потік самознищується, викликавши ***ExitThread***.
- Інший потік даного або стороннього процесу викликає ***TerminateThread***.
- Завершується процес, що містить даний потік.

Wait функції

Як можна призупинити роботу потоку? Існує багато способів. Ось деякі з них.

Функція ***Sleep()*** припиняє роботу потоку на задане число мілісекунд. Якщо як аргумент ви вкажете 0 ms, то відбудеться наступне. Потік відмовиться від свого кванта процесорного часу, проте тут же з'явиться в списку потоків готових до виконання. Іншими словами відбудеться навмисне перемикання потоків.

Функція ***WaitForSingleObject()*** припиняє виконання потоку до тих пір, поки не відбудеться одна з двох подій:

- закінчиться таймаут очікування;
- очікуваний об'єкт перейде в сигнальний (signaled) стан.

По повертаному значенню можна зрозуміти, яка з двох подій відбулася. Чекати за допомогою wait-функцій можна більшість об'єктів ядра, наприклад, об'єкт "процес" або "потік", щоб визначити, коли вони завершать свою роботу.

Функції ***WaitForMultipleObjects*** передається відразу масив об'єктів. Можна чекати спрацьовування відразу всіх об'єктів або якогось одного з них.

Приклад 2. Програма створює два однакові потоки і чекає їх завершення.

Потоки просто виводять текстове повідомлення, яке передане їм при ініціалізації.

```
#include <windows.h>
#include <process.h>

unsigned __stdcall ThreadFunc( void * arg) // Функція потоку
{
     char ** str = (char**)arg;
     MessageBox(0,str[0],str[1],0);
     _endthreadex( 0 );
     return 0;
};
int main(int argc, char* argv[])
{
     char * InitStr1[2]= {"First thread running!","11111"};//
рядок для першого потоку
     char * InitStr2[2]= {"Second thread running!","22222"};//
рядок для другого потоку
     unsigned uThreadIDs[2];

     HANDLE hThreads[2];
     hThreads[0]= (HANDLE)_beginthreadex( NULL, 0 &ThreadFunc,
InitStr1, 0,&uThreadIDs[0]);
     hThreads[1]= (HANDLE)_beginthreadex( NULL, 0 &ThreadFunc,
InitStr2, 0,&uThreadIDs[1]);

     // Чекаємо, поки потоки не завершать свою роботу
     WaitForMultipleObjects(2, hThreads, TRUE, INFINITE ); // Set
no time-out
     // Закриваємо дескриптори
     CloseHandle( hThreads[0]);
     CloseHandle( hThreads[1]);
     return 0;
}
```

Створення і управління потоками за допомогою MFC

Microsoft Foundation Class Library (MFC) забезпечує підтримку для багатопотокових застосувань. Усі потоки в застосуваннях MFC представлені об'єктами класу ***CWinThread***.

У MFC розглядаються два види потоків:

- Потоки з інтерфейсом для користувача (user-interface threads)
- Робочі потоки (worker threads)

Потоки з інтерфейсом для користувача обробляють повідомлення та реагують на події, що генеруються користувачем. Робочі потоки використовуються для виконання якогось завдання, яке не потребує втручання користувача.

MFC управляє потоками з інтерфейсом для користувача шляхом забезпечення черги повідомлень для подій. Об'єкт класу ***CWinApp*** є прикладом потоку з інтерфейсом для користувача, так як він породжений від CWinThread і обробляє події та повідомлення, що генеруються користувачем.

Для створення нового потоку викликається функція ***AfxBeginThread***, яка створює об'єкт класу ***CWinThread*** і, по замовчуванню, запускає цей потік:

```
CWinThread* AfxBeginThread( AFX_THREADPROC pfnThreadProc, LPVOID pParam, int nPriority = THREAD_PRIORITY_NORMAL, UINT nStackSize = 0, DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );

CWinThread* AfxBeginThread( CRuntimeClass* pThreadClass, int nPriority = THREAD_PRIORITY_NORMAL, UINT nStackSize = 0, DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

Значення, що повертається (Return Value)

Вказівник на новостаоренний об'єкт потоку.

Параметри

pfnThreadProc

Вказує на управляючу функцію робочого потоку.

pThreadClass

RUNTIME_CLASS об'єкта, що породжений від CWinThread.

pParam

Параметр, що передається в управляючу функцію.

nPriority

Пріоритет потоку.

dwCreateFlags

Додатковий прапорець, який контролює створення потоку:

- CREATE_SUSPENDED створює об'єкт, але не запускає його, поки не виконано функцію ResumeThread.
- 0 запускає потік одразу ж після створення.

Перша форма AfxBeginThread створює робочий потік, а друга – з інтерфейсом для користувача.

Щоб завершити виконання потоку, потрібно або викликати з нього ***AfxEndThread***, або дочекатися поки виконається управляюча функція потоку.

Після запуску потока, управління передається управляючій функції ***CWinThread::Run***. Тому користувач повинен переписати дану функцію, щоб потік виконав поставлене завдання. Вона повертає ціле значення, яке можна отримати за допомогою функції ***GetExitCodeThread***.

2 ХІД РОБОТИ

1. Ознайомитися з теоретичними відомостями.
2. За допомогою програми “Img2Arr” перетворити файл зображення згідно варіанту в матрицю.
3. Відкрити файл матриці “output.arr” за допомогою “Блокноту”, знищити останню пусту стрічку і табуляцію, переконатися, що кількість рядків матриці є кратною 2, 4 і 8.
4. Відкрити проект “ArrManip” у середовищі Visual Studio, завантаживши файл “ArrManip.sln”. Знайти у файлі “ArrManipDlg.cpp” управляючу функцію потоку (DWORD WINAPI ThreadProc(LPVOID lpParam)) і запрограмувати операцію над матрицею згідно варіанту. Наприклад, наступний фрагмент коду управляючої функції дозволяє обчислити суму елементів матриці:

```
    res = 0;  //  Сума  елементів  фрагменту  матриці,  яку  знаходить
кожний потік
          for(r=pData->startRow; r<pData->endRow; r++)
               for(c=0; c<cols; c++)
                    res += pArr[r][c];

          EnterCriticalSection(&pDlg->CriticalSection);    //   Початок
критичної секції

          *pArrRes += res;    // Сума елементів всієї матриці

          LeaveCriticalSection(&pDlg->CriticalSection);    //    Кінець
критичної секції
```

5. Відкомпілювати проект (клавіша F7) і, запустити на виконання файл “ArrManip.exe”. Виконати операцію над матрицею використовуючи 1, 2, 4 і 8 потоків. Завантажити диспетчер задач і переглянути кількість потоків для процесу “ArrManip”.
6. Відключити вивід повідомлення з результатами обчислень для кожного потоку в управляючій функції за допомогою коментарів і перекомпілювати проект.

```
    /*
    CString msg, title;
    msg.Format("Результат  =  %.2lf\nРядки  =  %u-%u",  res,  pData-
>startRow, pData->endRow);
    title.Format("Потік #%u", pData->thrId);
    ::MessageBox(NULL, msg, title, MB_OK);
    */
```

7. Виконати операцію над матрицею використовуючи 1, 2, 4 і 8 потоків по три рази для кожної конфігурації та відобразити за допомогою діаграми Excel залежність середнього часу виконання від кількості потоків.

3 ЗМІСТ ЗВІТУ

1. Короткі теоретичні відомості по лабораторній роботі.
2. Зображення згідно варіанту.
3. Лістинг зміненої згідно варіанту управляючої функції потоку.
4. Копія екрану з результатами виконання програми для 1, 2, 4 і 8 потоків (з виводом повідомлень для кожного потоку).
5. Копія екрану з результатами виконання програми для 1, 2, 4 і 8 потоків (без виводу повідомлень для кожного потоку).
6. Діаграма залежності середнього часу виконання від кількості потоків.
7. Висновки.

4 КОНТРОЛЬНІ ЗАПИТАННЯ

1. Процес, потік, програма
2. Структура процесу
3. Структура потоку
4. Первинний потік
5. Суть витісняючого планування на основі пріоритетів
6. Пріоритети процесів та потоків
7. Адресний простір процесів та потоків
8. Функції управління процесами
9. Функції управління потоками
10. Wait-функції
11. Види потоків в MFC

5 ВИКОРИСТАНА ЛІТЕРАТУРА

1. Соломон, Руссинович. Внутреннее устройство MS Windows 2000. ISBN 5-7502-0136-8.
2. Jeffrey Richter. Programming Applications for MicrosoftR Windows. ISBN 1-57231-996-8.
3. Platform SDK / Windows Base Services / Executables / Processes and Threads.
4. Platform SDK / Windows Base Services / Interprocess Communication / Synchronization.
5. Periodicals 1996 / MSJ / December / First Aid For Thread-impaired: Using Multiple Threads with MFC.
6. Periodicals 1996 / MSJ / March / Win32 Q&A.
7. Periodicals 1997 / MSJ / July / C++ Q&A.
8. Periodicals 1997 / MSJ / January / Win32 Q&A.
9. MSDN.

6 ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

№ варіанту	Файл зображення	Операція з матрицею
1.	1.jpg	Сума елементів, більших за 100
2.	2.jpg	Максимальне значення
3.	3.jpg	Кількість нулів
4.	4.jpg	Сума непарних стовпчиків
5.	5.jpg	Кількість рядків, в яких є 0
6.	6.jpg	Кількість елементів, більших за 100
7.	7.jpg	Сума діагональних елементів
8.	8.jpg	Максимальне значення непарних рядків
9.	9.jpg	Максимальне значення діагоналі
10.	10.jpg	Сума елементів, кратних 10
11.	11.jpg	Добуток елементів, що >0 і <5
12.	12.jpg	Кількість елементів, що >10 і <100
13.	13.jpg	Мінімальне значення
14.	14.jpg	Сума парних рядків
15.	15.jpg	Мінімальне значення діагоналі
16.	16.jpg	Сума парних стовпчиків
17.	17.jpg	Сума непарних елементів
18.	18.jpg	Максимальне значення парн.рядків
19.	19.jpg	Сума елементів
20.	20.jpg	Кількість рядків, сума яких <1000
21.	21.jpg	Сума парних елементів
22.	22.jpg	Сума елементів, що >10 і <100
23.	23.jpg	Кількість непарних елементів
24.	24.jpg	Сума непарних рядків
25.	25.jpg	Кількість елементів, кратних 5
26.	26.jpg	Сума елементів, поділених на 10
27.	27.jpg	Середнє значення
28.	28.jpg	Сума елементів, менших за 10
29.	29.jpg	Кількість елементів, кратних 10
30.	30.jpg	Кількість парних елементів

ЛАБОРАТОРНА РОБОТА № 2

Тема: Організація колективних комунікаційних операцій
Мета: Одержати практичні навички організації колективних обмінів

ТЕОРЕТИЧНІ ВІДОМОСТІ

Огляд колективних операцій

Набір операцій типу точка-крапка є достатнім для програмування будь-яких алгоритмів, однак MPI навряд чи б завоював таку популярність, якби обмежувався тільки цим набором комунікаційних операцій. Однієї з найбільш привабливих сторін MPI є наявність широкого набору колективних операцій, що беруть на себе виконання найбільш часте дій, що зустрічаються при програмуванні. Наприклад, часто виникає потреба розіслати деякий перемінну чи масив з одного процесора всім іншим. Кожен програміст може написати таку процедуру з використанням операцій Send/Recv, однак набагато зручніше скористатися колективною операцією MPI_Bcast. Причому гарантовано, що ця операція буде виконуватися набагато ефективніше, оскільки MPI-функція реалізована з використанням внутрішніх можливостей комунікаційного середовища. Головна відмінність колективних операцій від операцій типу точка-крапка полягає в тому, що в них завжди беруть участь усі процеси, зв'язані з деяким комунікатором. Недотримання цього правила приводить або до аварійного завершення задачі, або до ще більш неприємного зависання задачі.

Набір колективних операцій включає:

Синхронізацію всіх процесів за допомогою бар'єрів (MPI_Barrier):

- Колективні комунікаційні операції, у число яких входять;
- розсилання інформації від одного процесу всім іншим членам деякої області зв'язку (MPI_Bcast);
- зборка (gather) розподіленого по процесах масиву в один масив зі збереженням його в адресному просторі виділеного (root) процесу (MPI_Gather, MPI_Gatherv);
- зборка (gather) розподіленого масиву в один масив з розсиланням його всім процесам деякої області зв'язку (MPI_Allgather, MPI_Allgatherv);
- розбивка масиву і розсилання його фрагментів (scatter) усім процесам області зв'язку (MPI_Scatter, MPI_Scatterv);
- сполучена операція Scatter/Gather (All-to-All), кожен процес поділяє дані зі свого буфера передачі і розкидає фрагменти всім іншим процесам, одночасно збираючи фрагменти, послані іншими процесами у свій буфер прийому (MPI_Alltoall, MPI_Alltoallv).
- Глобальні обчислювальні операції (sum, min, max і ін.) над даними, розташованими в адресних просторах різних процесів:
- с збереженням результату в адресному просторі одного процесу (MPI_Reduce);
- с розсиланням результату всім процесам (MPI_Allreduce);
- сполучена операція Reduce/Scatter (MPI_Reduce_scatter);
- префіксна редукція (MPI_Scan).

Усі комунікаційні підпрограми, за винятком MPI_Bcast, представлені в двох варіантах:

- простий варіант, коли всі частини переданого повідомлення мають однакову довжину і займають суміжні області в адресному просторі процесів;
- "векторний" варіант надає більш широкі можливості по організації колективних комунікацій, знімаючи обмеження, властивому простому варіанту, як у частині довжин блоків, так і в частині розміщення даних в адресному просторі процесів. Векторні варіанти відрізняються додатковим символом "v" наприкінці імені функції.

Відмінні риси колективних операцій:

- Колективні комунікації не взаємодіють з комунікаціями типу точка-крапка.
- Колективні комунікації виконуються в режимі з блокуванням. Повернення з підпрограми в кожнім процесі відбувається тоді, коли його участь у колективній операції завершилося, однак це не означає, що інші процеси завершили операцію.
- Кількість одержуваних даних повинна бути дорівнює кількості посланих даних.
- Типи елементів що посилаються й одержуваних повідомлень повинні збігатися.
- Повідомлення не мають ідентифікаторів.

Примітка: У даному розділі часто будуть використовуватися поняття *буфер обміну, буфер передачі, буфер прийому*. Не слід розуміти ці поняття в буквальному значенні – як деяку спеціальну область пам'яті, куди містяться дані перед викликом комунікаційної функції. Насправді, це, як правило, звичайні масиви, використовувані в програмі, і які безпосередньо можуть брати участь у комунікаційних операціях. У викликах підпрограм передається адреса початку безупинної області пам'яті, що буде брати участь в операції обміну.

Вивчення колективних операцій почнемо з розгляду двох функцій, що коштують особняком: MPI_Barrier і MPI_Bcast.

Функція синхронізації процесів MPI_Barrier блокує роботу її процесу, що викликав, доти, поки всі інші процеси групи також не викликають цю функцію. Завершення роботи цієї функції можливо тільки всіма процесами одночасно (усі процеси "переборюють бар'єр" одночасно).

int MPI_Barrier(MPI_Comm comm)

IN comm – комунікатор.

Синхронізація за допомогою бар'єрів використовується, наприклад, для завершення всіма процесами деякого етапу рішення задачі, результати якого будуть використовуватися на наступному етапі. Використання бар'єра гарантує, що жоден з процесів не приступить завчасно до виконання наступного етапу, поки результат роботи попереднього не буде остаточно сформований. Неявну синхронізацію процесів виконує будь-як колективна функція.

Широкомовне розсилання даних виконується за допомогою функції *MPI_Bcast*. Процес з номером root розсилає повідомлення зі свого буфера передачі всім процесам області зв'язку комунікатора comm.

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

INOUT Buffer – адреса початку розташування в пам'яті даних, що розсилається;
IN count – число елементів, що посилаються;
IN datatype – тип елементів, що посилаються;
IN root – номер процесу-відправника;
IN comm – комунікатор.

Після завершення підпрограми кожен процес в області зв'язку комунікатора comm, включаючи і самого відправника, одержить копію повідомлення від процесу-відправника root.

На Рис. 2.1 представлена графічна інтерпретація операції Bcast.

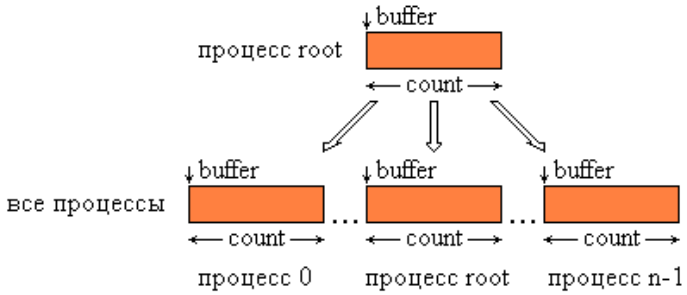


Рисунок 2.1 – Графічна інтерпретація операції Bcast

Приклад використання функції MPI_Bcast. 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. У результаті, у кожного не *root* процесу в буфері виявляється саме це повідомлення.

```
# 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;
}
```

Функції збору блоків даних від усіх процесів групи

Сімейство функцій збору блоків даних від усіх процесів групи складається з чотирьох підпрограм: MPI_Gather, MPI_Allgather, MPI_Gatherv, MPI_Allgatherv. Кожна з зазначених підпрограм розширює функціональні можливості попередніх.

Функція MPI_Gather робить зборку блоків даних, що посилаються всіма процесами групи, в один масив процесу з номером root. Довжина блоків передбачається однакової. Об'єднання відбувається в порядку збільшення номерів процесів-відправників. Тобто дані, послані процесом і зі свого буфера sendbuf, містяться в і-ю порцію буфера recvbuf процесу root. Довжина масиву, у який збираються дані, повинна бути достатньої для їхнього розміщення.

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

IN sendbuf – адреса початку розміщення даних, що посилається;

IN sendcount – число елементів, що посилаються;

IN sendtype – тип елементів, що посилаються;

OUT recvbuf – адреса початку буфера прийому (використовується тільки в процесі-одержувачі root);

IN recvcount – число елементів, одержуваних від кожного процесу (використовується тільки в процесі-одержувачі root);

IN recvtype – тип одержуваних елементів;

IN root – номер процесу-одержувача;

IN comm – Комунікатор.

Тип елементів, що посилаються, sendtype повинний збігатися з типом recvtype одержуваних елементів, а число sendcount повинне дорівнювати числу recvcount. Тобто, recvcount у виклику з процесу root – це число елементів, що збираються від кожного процесу, а не загальна кількість зібраних елементів. Графічна інтерпретація операції Gather представлена на рис. 2.2.

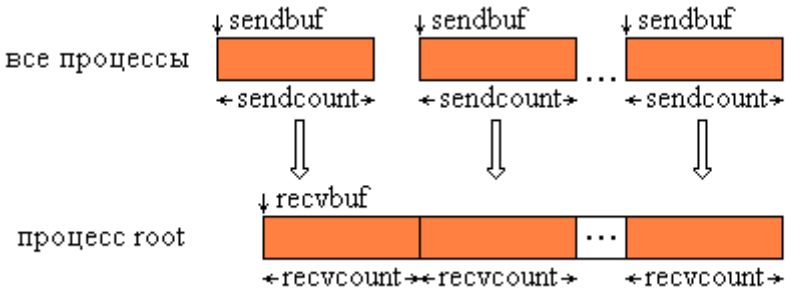


Рисунок 2.2 – Графічна інтерпретація операції Gather.

Функції розподілу блоків даних по всіх процесах групи

Сімейство функцій розподілу блоків даних по всіх процесах групи складається з двох підпрограм: MPI_Scatter і MPI_Scaterv.

Функція MPI_Scatter розбиває повідомлення з буфера посилки процесу root на рівні частини розміром sendcount і посилає і-ю частину в буфер прийому процесу з номером і (у тому числі і самому собі). Процес root використовує обох буферів (посилки і прийому), тому у викликуваній їм підпрограмі всі параметри є істотними. Інші процеси групи з комунікатором comm є тільки одержувачами, тому для них параметри, специфицирующие буфер посилки, не істотні.

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

IN	sendbuf	–	адреса початку розміщення блоків розподілених даних (використовується тільки в процесі-відправнику root);
IN	sendcount	–	число елементів, що посилаються кожному процесу;
IN	sendtype	–	тип елементів, що посилаються;
OUT	recvbuf	–	адреса початку буфера прийому;
IN	recvcount	–	число одержуваних елементів;
IN	recvtype	–	тип одержуваних елементів;
IN	root	–	номер процесу-відправника;
IN	comm	–	Комунікатор.

Тип елементів, що посилаються, sendtype повинний збігатися з типом recvtype одержуваних елементів, а число елементів, що посилаються, sendcount повинне дорівнювати числу прийнятих recvcount. Варто звернути увагу, що значення sendcount у виклику з процесу root – це число елементів, що посилаються кожному процесу, а не загальна їхня кількість. Операція Scatter є зворотної стосовно Gather. На рис. 2.3 представлена графічна інтерпретація операції Scatter.

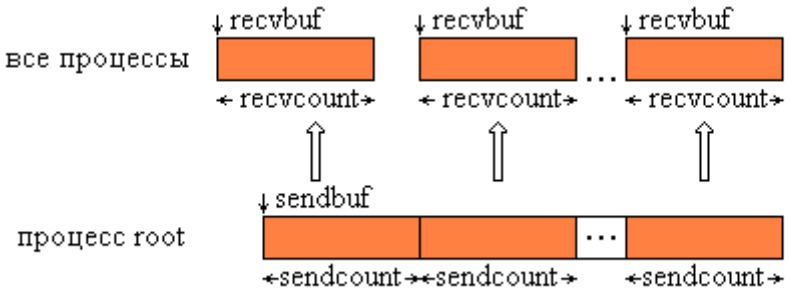


Рисунок 2.3 – Графічна інтерпретація операції Scatter

Приклад використання функції MPI_Scatter.

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.

```
# 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();
}
```

Сполучені колективні операції

Функція MPI_Alltoall сполучає в собі операції Scatter і Gather і є по суті дела розширенням операції Allgather, коли кожен процес посилає різні дані різним одержувачам. Процес і посилає j-ий блок свого буфера sendbuf процесу j, що поміщає його в і-ий блок свого буфера recvbuf. Кількість посланих даних повинна бути дорівнює кількості отриманих даних для кожної пари процесів.

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm).

IN	Sendbuf	–	адреса початку буфера посилки;
IN	Sendcount	–	число елементів, що посилаються;
IN	sendtype	–	тип елементів, що посилаються;
OUT	recvbuf	–	адреса початку буфера прийому;
IN	recvcount	–	число елементів, одержуваних від кожного процесу;
IN	recvtype	–	тип одержуваних елементів;
IN	comm	–	Комунікатор.

Схема виконання операції Alltoall представлена на рис. 2.4.

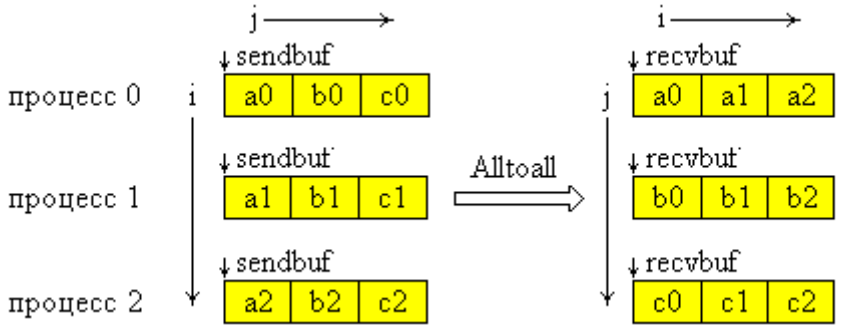


Рисунок 2.4 – Схема виконання операції Alltoall

Функція MPI_Alltoallv реалізує векторний варіант операції Alltoall, що допускає передачу і прийом блоків різної довжини з більш гнучким розміщенням переданих і прийнятих даних.

Глобальні обчислювальні операції над розподіленими даними

У рівнобіжному 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 (рівнобіжної) редукції. Наприклад, нехай в адресному просторі всіх процесів деякої групи процесів маються копії перемінної var (необов'язково мають те саме значення), тоді застосування до неї операції обчислення глобальної чи суми, іншими словами, операції редукції SUM поверне одне значення, що буде містити суму всіх локальних значень цієї перемінної. Використання цих операцій є одним з основних засобів організації розподілених обчислень.

У MPI глобальні операції редукції представлені в декількох варіантах:

- Зі збереженням результату в адресному просторі одного процесу (MPI_Reduce).
- Зі збереженням результату в адресному просторі всіх процесів (MPI_Allreduce).
- Префіксна операція редукції, що як результат операції повертає вектор. і-я компонента цього вектора є результатом редукції перших і компонент розподіленого вектора (MPI_Scan).
- Сполучена операція Reduce/Scatter (MPI_Reduce_scatter).

Функція MPI_Reduce виконується в такий спосіб. Операція глобальної редукції, зазначена параметром op, виконується над першими елементами вхідного буфера, і результат посилається в перший елемент буфера прийому процесу root. Потім тих же саме робиться для других елементів буфера і т.д.

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

IN	Sendbuf	–	адреса початку вхідного буфера;
OUT	Recvbuf	–	адреса початку буфера результатів (використовується тільки в процесі-одержувачі root);
IN	Count	–	число елементів у вхідному буфері;
IN	Datatype	–	тип елементів у вхідному буфері;
IN	Op	–	операція, по якій виконується редукція;
IN	Root	–	номер процесу-одержувача результату операції;
IN	Comm	–	коммуникатор.

На рис. 2.5 представлена графічна інтерпретація операції Reduce. На даній схемі операція "+" означає будь-яку припустиму операцію редукції.

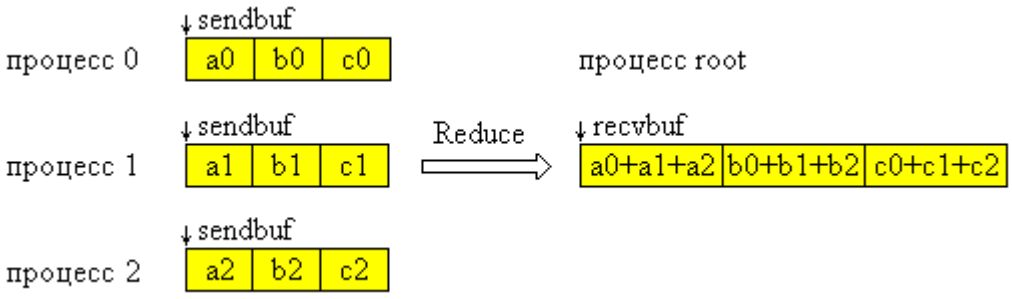


Рисунок 2.5 – Графічна інтерпретація операції Reduce

Як операцію op можна використовувати або одну з визначених операцій, або операцію, сконструйовану користувачем. Усі визначені операції є асоціативними і комутативними. Сконструйована користувачем операція, принаймні, повинна бути асоціативної. Порядок редукції визначається номерами процесів у групі. Тип datatype елементів повинний бути сполучимо з операцією op. У таблиці 6

представлений перелік визначених операцій, що можуть бути використані у функціях редукції MPI.

Таблиця 2.1. Визначені операції у функціях редукції MPI

Назва	Операція	Дозволені типи
MPI_MAX MPI_MIN	Максимум Мінімум	C integer
MPI_SUM MPI_PROD	Сума Добуток	C integer
MPI_LAND MPI_LOR MPI_LXOR	Логічне AND Логічне OR Логічне що виключає OR	C integer
MPI_BAND MPI_BOR MPI_BXOR	Порозрядне AND Порозрядне OR Порозрядне що виключає OR	C integer
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
FORTRAN integer:	MPI_INTEGER
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
Logical:	MPI_LOGICAL
Complex:	MPI_COMPLEX
Byte:	MPI_BYTE

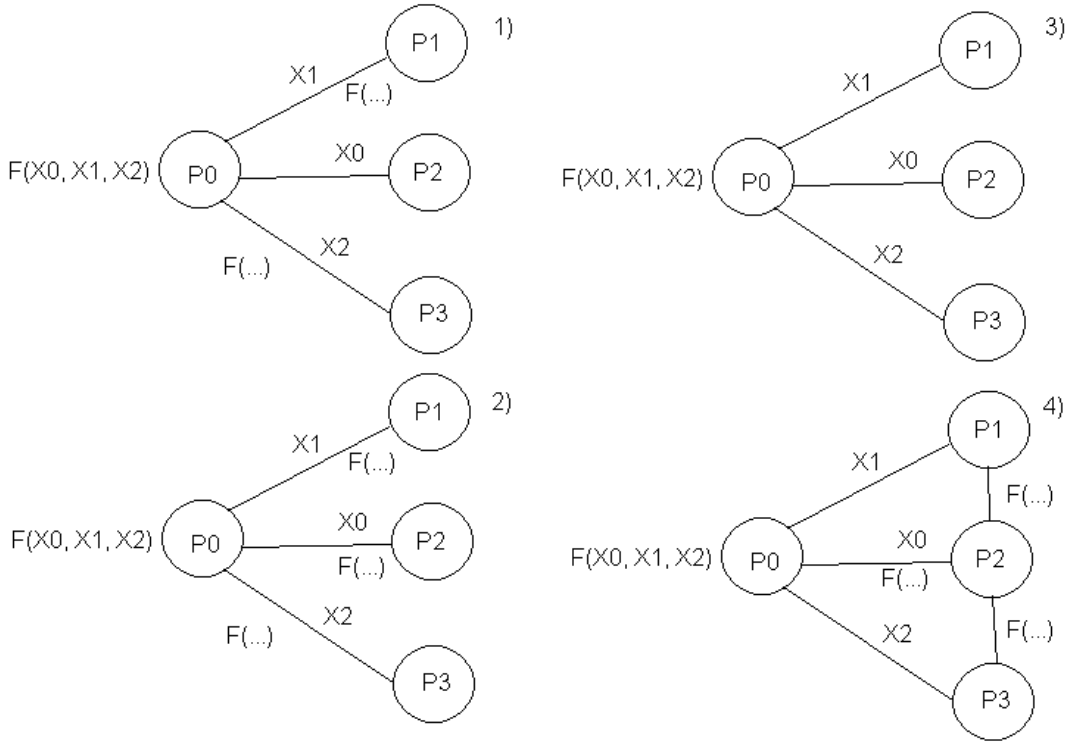
Операції MAXLOC і MINLOC виконуються над спеціальними парними типами, кожен елемент яких зберігає двох величин: значення, по яких шукається чи максимум мінімум, і індекс елемента. У MPI мається 9 таких визначених типів.

MPI_FLOAT_INT	float	and	int
MPI_DOUBLE_INT	double	and	int
MPI_LONG_INT	long	and	int
MPI_2INT	int	and	int
MPI_SHORT_INT	short	and	int
MPI_LONG_DOUBLE_INT	long double	and	int

Завдання до лабораторної роботи

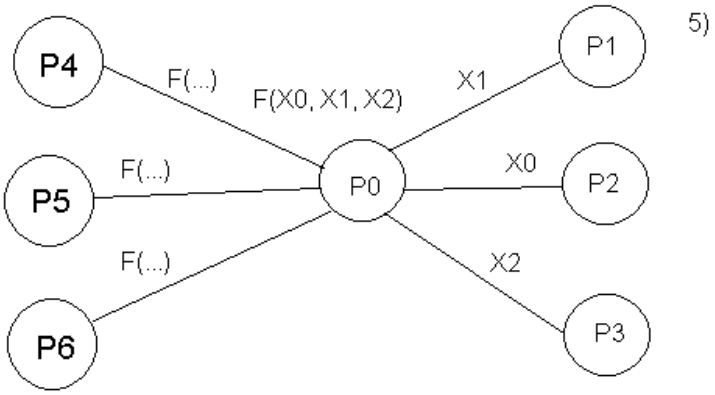
Розробити програму мовою С з використанням функцій колективного обміну та операцій передачі повідомлень типу «крапка – крапка» бібліотеки MPI, що здійснює обчислення функції F(X1, …, Xn), відповідно до варіанта завдання.

Варіанти завдань

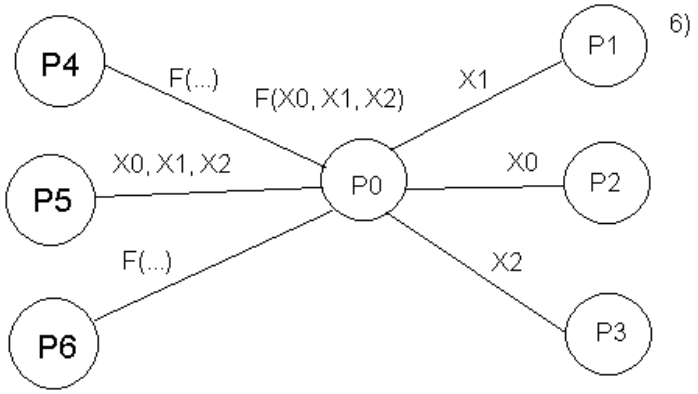


F(X0, X1, X2) = X1 + X0*sin(X0*X2)

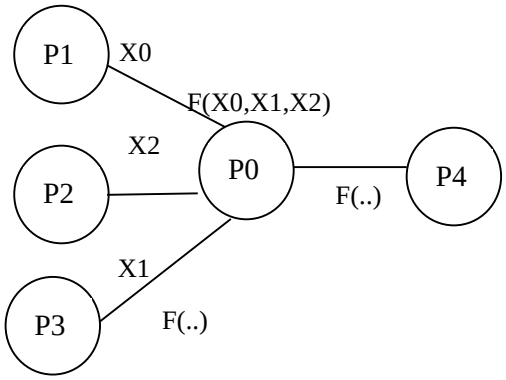
5)


P4
F(...)
F(X0, X1, X2)
X1
P1
P5
F(...)
P0
X0
P2
F(...)
X2
P6
P3

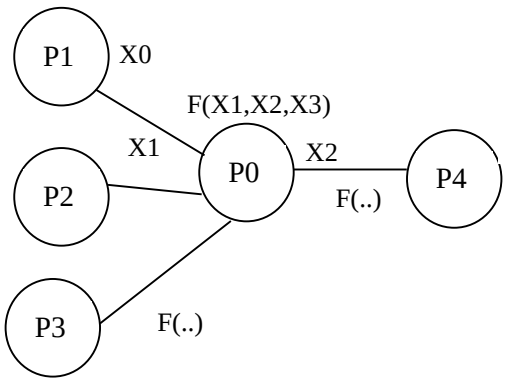
6)


P4
F(...)
F(X0, X1, X2)
X1
P1
X0, X1, X2
P5
P0
X0
P2
F(...)
X2
P6
P3

7)


P1
X0
F(X0,X1,X2)
X2
P2
P0
P4
F(..)
X1
P3
F(..)

8)


P1
X0
F(X1,X2,X3)
X1
P2
P0
X2
P4
F(..)
P3
F(..)

ЛАБОРАТОРНА РОБОТА № 3

Тема: Лабораторна робота
Мета: Лабораторна робота

ТЕОРЕТИЧНІ ВІДОМОСТІ

Grid портали

Веб-орієнтовані обчислювальні портали (Grid -портали) зарекомендували себе як ефективний інструмент для забезпечення користувачів обчислювальних Grid простим і інтуїтивно-зрозумілим інтерфейсом для доступу до інформації і використовування Grid- ресурсів. В даний момент Grid-портали розробляються і розгортаються на великих Grid системах, таких як EGEE, Partnership for Advanced Computational Infrastructure (PACI) TeraGrid, NASA Information Power Grid, National Institute Health (NIH) Biomedical Informatics Research Network.

Grid-портали роблять розподілені обчислювальні середовища і середовища даних Grid більш доступними для користувачів і вчених шляхом використовування загальних для веб і інтерфейсу ПГО для користувача правил (рис.1). Grid-портали і інші вэб-портали надають можливість розробникам і користувачам змінювати зміст інформації і засіб його подачі (розмітка сторінки, рівень деталізації) для наявних інструментів і сервісів. Grid-портали здатні автоматично виконати специфічні додатки, надавати точні посилання за спеціалізованими колекціями даних, об'єднувати (і приховувати) потік оброблюваних даних між додатками, автоматизувати створення наборів вихідних файлів. Портали також надають можливість доступу до нижнього обчислювального шару, генерують звіти про доступність ресурсів, статус виконуваних задач і поточне завантаження Grid-ресурсів.

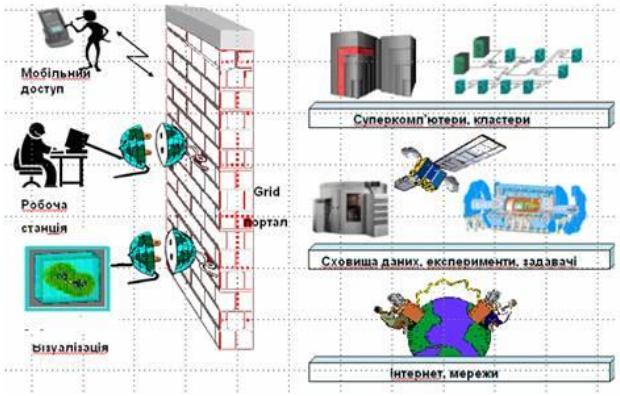


Рисунок 3.1 – Призначення Grid порталу

Програмне забезпечення, що використовується для створення Grid-порталів, повинне взаємодіяти з ПГО проміжного рівня і в деяких випадках повинне компенсувати бракуючу функціональність. ПО порталу повинне бути сумісним із загальновідомими вэб-серверами і браузерами. Існує декілька засобів розробки Grid-порталів які дають можливість значно спростити задачу розробника з інтеграції комплексних Grid-технологій з Grid-сервісами за допомогою зручного у

використовуванні Веб-інтерфейсу. З розвитком і появою нових веб-технологій почали розроблятися стандарти і протоколи доступу до Grid-ресурсів. Веб-технології значно спрощують використовування Grid і Grid-технологій, що дозволяє використовувати більш узагальнені Grid-додатки. Завдяки стандартизації веб-сервісів і стандартів, а також тому, що засоби розробки Grid-порталів є достатньо зрілими продуктами, розробка, упровадження, підтримка і використовування Grid-порталів стає все більш і більш зручною.

ПО для розробки Grid-порталів повинне містити широкий спектр різних програмних і апаратних систем. Grid-портал грає унікальну у своєму роді роль на рівні ПГО проміжного рівня Grid – він об'єднує ПО нижніх рівнів з різними програмними засобами взаємодії.

Користувачі і розробники GRID-порталів

Grid знаходиться поки що на початковій стадії переходу до широкого розгортання у суспільстві, у зв'язку з чим на даний момент кількість його користувачів обмежена. У міру того як Grid просувається у промисловість, освіту і бізнес, його користувачі потребуватимуть більшій допомозі в своїх спробах розробити додатки для експлуатації. В науковому середовищі існує три основні класи потенційних користувачів Grid. По-перше, це кінцеві користувачі, які працюватимуть тільки з вже заздалегідь готовими додатками, що запускаються через Unix оболонку на супер- комп’ютерних ресурсах (див. Додаток 4). Можливо пристосування цих додатків до Grid з впровадженням простого веб-інтерфейсу для надання конкретних для кожного додатку параметрів і інформації по структурі виконуємої задачі . У зв'язку з цим ця група як раз може буди найбільш просто підготовленою для використовування Grid, хоч в той же самий час значна частина роботи лягає на плечі розробникам Grid порталів. Користувачі першої групи звичайно володіють невеликими знаннями про ті суперкомп’ютери, що використовуються ними, але ціх знань проте досить для завантаження ввідної інформації, запускупрограми, опрацювання файлів інформації, що виводиться. Також ця група включає користувачів, що працюють з додатками і моделями, доступними для широкого кола користувачів (GAMESS, NASTRAN, і т.д.) Велика частина користувачів цієї групи, можливо, немає знань про паралельні комп'ютери і засоби виконання ними задач. Для користувачів першої групи, програма додатку є віртуальною лабораторією. Крім цього, існують частина клієнтів, далека від світу суперкомп’ютерних обчислень, оскільки вона знаходить навіть цей найскромніший з рівнів знань про них або гнітючим, або занадто трудомістким. Проте з допомогою інтуїтивного і досконалого порталу доступу ці дослідники зможуть запустити на Grid необхідні додатки. Ефективні Grid портали доступу можуть також бути корисними для користувачів, що обізнані з суперкомп’ютерними обчисленнями, але тільки розпочинають свою роботу з Grid. Grid портали забезпечують іх простими і ефективними механізмами для прозорої роботи і досягнення поставленої дослідницької мети. Друга група складається з дослідників, які є більш досвідченими користувачами суперкомп’ютерних обчислень і мають облікові записи на кількох таких системах. Як учені вони віддають перевагу проведенню моделюння з існуючими додатками з ціллю розробки нового . додатку для паралельного обчислення , хоч ця задача і містить деякі складнощі. Ця група

користувачів подібна першій, описаній вище, оскільки пріоритетний інтерес для неї лежить в рішенні наукових проблем. Для цієї групи необхідний сумісний портал доступу , який забезпечує користувачів інформацією з всіх індивідуальних Grid систем, на яких вони мають облікові записи. Це дозволяє користувачам з'ясувати, як працювати з кожною системою, використовувати доступні системи і давати оцінку цим системам, самим формуючи завдання для проведення наступних симуляцій. Користувачі цієї групи вже мають досвід роботи з Unix і командами, пов'язаними з тими супер-комп'ютерними системами, на які вони вже встановлювали свої програми. Портал повинен дозволити їм проводити роботу, зберегати і одержувати інформацію, а також управляти файлами в будь-якій системі з одного робочого місця. Для цих користувачів такий портал може збільшити ефективність використовування існуючих систем для підтримки своїх досліджень.

Третя група Grid користувачів включає дослідників, які є експертами з обчислень і займаються оцінкою і використуванням самих останніх обчислювальних технологій. Група складається, як правило, з програмістів, які тільки почали вивчати Grid технології. Користувачі цієї групи отримують переваги від використовування Grid порталу, безпосередньо інтегруючи компоненти Grid в прикладні додатки, які були ними розроблені завдогіль для суперкомп'ютерних застосувань. Додатково в цю групу також входять розробники порталів, бібліотек і додатків.

Філософія, на яку спирається архітектура Grid порталу доступу , грунтується на наступній думці: безліч потенційних користувачів і розробників Grid одержують значні переваги від порталів, якщо вони надають легкий і швидкий доступ до ресурсів, не вимагаючи при цьому значних трудовитрат з боку розробників. Від користувачів Grid порталів, як правило, не вимагається встановлювати додаткове програмне забезпечення. Все що їм потрібно для доступу до сервісів і ресурсів Grid – це всього лише веб-браузер. Якби кожному розробнику Grid порталів доступу для створення порталу б було необхідне встановлювати веб-додатки, ПО порталу і ПГО проміжного рівня, то це породило б величезну кількість повторних зусиль і зайву складність мкркжі підключених систем.

Grid портал доступу намагається розв'язати подібні задачі шляхом визначення декількох ключових моментів:

• *Універсальний доступ*: веб-орієнтовані портали здатні працювати де завгодно і в який завгодно момент часу, не вимагаючі ніякого додаткового ПО і працюючі навіть з «старими браузерами» (підтримка XML на стороні клієнта).

• *Надійні інформаційні сервіси*: надають порталам і їх користувачам централізований доступ до всеосяжної і точної інформації зGrid ресурсів.

• *Загальні Grid технології і стандарти*: мінімізують навантаження на адміністраторів ресурсів, тому що не вимагають власного демона Grid порталу на суперкомп'ютерних ресурсах.

• *Інфраструктура, що масштабується і гнучка*: полегшує додавання або відключення додатків порталів, програмних систем Grid, обчислювальних ресурсів, сервісів, користувачів, задач і т.д.

• Безпека: використовування GSI, підтримка HTTPS/SSL (Secure Sockets Layer) криптування на всіх рівнях.

• *Єдина система аутентифікації*: дозволяє використовувати один обліковий запис для доступу до різних Grid ресурсів.

• *Обмін технологіями*: інструментарій, дозволяючий розробнику з легкістю встановити і використати портал.

• Стандарти *Global Grid Forum*: прив'язка до прийнятих стандартів і конвенцій.

• Підтримка розподілених клієнтських додатків і порталів: дозволяє ученим створювати свої власні портали додатків і використовувати існуючі для загальних інфраструктурних задач.

Враховуючи всі ці ключові моменти і виходячи з висновків, отриманих в процесі розробки Grid порталів, був створений інструментарій для розроблення Grid-порталів з архітектурою, заснованої на веб-орієнтованих сервісах, коли клієнти містять додатки порталів на локальних системах і дістають через них доступ до розподілених Grid-сервісів.

В останні роки збільшився наголос на співпраці між великими командами вчених, на використанні ними досконалих методів обрабки наукових даних , які включають як засоби об'єднання ресурсів даних з різними механізмами підтримки (такими, як реляційні бази даних, бази даних XML чи системи неструктурованих файлів), так і засоби уніфікованого доступу до них, їх модернізації, компресування (також декомпресування) та віалізації їх через веб-сервіси.

Рисунок 3.2 – Трьохрівнева архітектура Grid сервісів

Розпочалася розробка так званих ***Семантичних Grid***, у яких користувачі, програмні компоненти й обчислювальні ресурси (всі можуть належати різним власникам) об'єднані безперервно. При цьому необхідний високий ступінь автоматизації, щоб підтримувати гнучке співробітництво й обчислення в глобальному масштабі. Крім того, це середовище має бути індивідуалізована кожним учасникам і має запропонувати прямий простий доступ як до програмних компонентів, так і до інших користувачів.

Вищезгадані можливості характеризують Grid інфраструктуру, яка складається із трьох концептуальних рівнів:

• *Сервіси даних/обчислень*: Цей рівень містить інформацію, яким чином розташовані обчислювальні ресурси, коли заплановано виконувати обчислення та шляхи передачі даних між різними обчислювальними ресурсами. Рівень має справу з великими об'ємами даних, забезпечуючи швидкі мережі й представляючи різноманітні ресурси як єдиний метакомп'ютер.

• *Інформаційні сервісчи*: Цей рівень вказує, яким чином інформація представляється, зберігається, хто і яким чином має до неї доступ. Тут інформація зрозуміла як дані зі значенням. Наприклад, виявлення цілого числа як подання температури процесу реакції, розпізнавання, що рядок – ім'я людини.

• *Сервіси Знань*: Цей рівень надає спосіб, яким знання придбане, використовується, знайдено, опубліковано, щоб допомогти користувачам досягати своїх специфічних цілей. Отут знання представляються як інформація, що застосована для досягнення мети, вирішення проблеми або вказання рішення. Прикладом може бути процедура розпізнавання оператором підприємства моменту часу, коли температура реакції вимагає завершення виконання процесу.

Треба визначити, що всі Grid системи , які уже побудовані або будуть побудовані, несуть в собі деякі елементи всіх трьох рівнів. Ступінь важливості використання цих рівнів буде вирішуватися користувачем. Таким чином, у деяких випадках обробка величезних об'ємів даних буде домінуючим завданням, у той час як в інших випадках обслуговування знання буде основною проблемою. Дотепер більшість науково-дослідних робіт в галузі Grid концентрувалося на рівні даних/обчислень й на інформаційному рівні. У той час все ще багато невирішених проблем, що стосуються керування широкомасштабними розподіленими обчисленнями та ефективного доступу і розповсюдження інформації з гетерогенних джерел. Вважається, що повний потенціал Grid обчислень може бути досягнутий тільки завдяки повної експлуатації функціональних можливостей та можливостей, що надаються рівнем знання. Тому цей рівень необхідний для автоматизованого прямого простого доступу до операцій і взаємодій.

Одна із стандартних методик, що використовуються в управлінні знаннями, – це розробка **порталу знань**. Стандартне визначення для нього таке: портал – це працюючий на базі Веб додаток, що забезпечує засоби для накопичення, пристосування та персоналізації даних.

Аби наблизитися до питання розробки Grid порталу знань, були переглянуті існуючі Grid-портали та визначені їх три головні категорії:

- проектні сайти, що збирають інформацію про окремий проект,
- портали знань, які збирають інформацію (з різних джерел) про наукову діяльність, наукові статті, та ін.;
- портали доступу,які полегшують доступ до ресурсів Grid.

Очевидно, деякі портали забезпечують сервіси, що представлять собою суміш цих базових категорій.

Grid-портал доступу

а допомогою простого Веб-браузера, який у наш час встановлений на кожному комп'ютері. Зараз він розглядаються у світі як найбільш перспективний шлях подальшого використання Grid. Портальний інтерфейс також дуже важливий, оскільки за допомогою його користувач абсолютно будь-якої підготовки і рівня знань може без проблем почати працювати в Grid.

Розглянемо для прикладу повнофункціональний Grid-портал GENIUS (Grid Enabled web eNvironment for site Independent User job Submission), розроблений в італійському інституті INFN (https://genius.ct.infn.it), який в даний час активно

використовується в рамках проекту GILDA (Grid Infn Laboratory for Dissemination Activities), що є віртуальною лабораторією для демонстрації

Рисунок 3.3 – Початкова сторінка порталу

GILDA можливостей технології Grid (рис. 3.3.).

Проект GILDA (https://gilda.ct.infn.it/) складається з декількох частин:

- GILDA Testbed – набір сайтів зі встановленим ПО LCG.
- Grid Demonstrator – веб-інтерфейс GENIUS, що дозволяє працювати з певним набором додатків.
- GILDA CA – центр сертифікації, що видає 14-денні сертифікати для роботи з GILDA.
- GILDA VO – віртуальна організація, об'єднуюча всіх користувачів GILDA.
- Grid Tutor – веб-інтерфейс GENIUS, що використовується для демонстрації можливостей технології Grid.

Grid Demonstrator є демо-версією, яка дозволяє отримати перший досвід роботи в Grid-середовищі. При цьому від користувача не потрібна ні наявність сертифікату, ні попередня реєстрація на сайті – використовуються приречені ім'я/пароль. Правда через демо-режим можлива робота тільки з наперед підготовленими задачами і сервісами неповної функціональності. Можливості GILDA Grid Demonstrator (рис.3.4.):

- Проглядання файлів в каталозі демо-користувача
- Проглядання змінних оточення
- Навігація в межах каталога віртуальної організації GILDA (перегляд і викачування файлів)
- Отримання інформації про проксі-сертифікаті
- Посилання на веб-інтерфейси деяких проектів, встановлених в GILDA: BLAST (біоінформатика) і Sonification (перетворення тексту в графічну або візуальну форму).

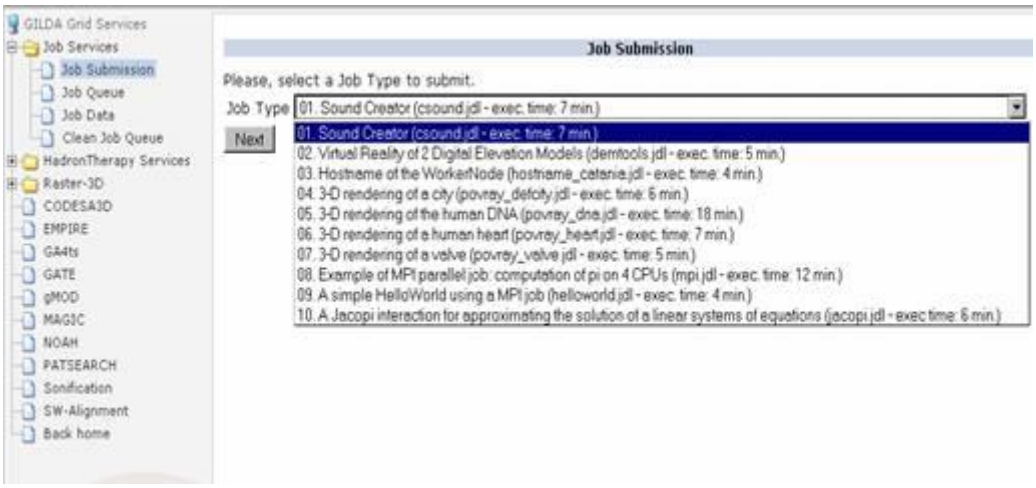


Рисунок 3.4. Робота в GILDA Grid Demonstrator

Основний набір додатків і засобів управління завданнями стає доступний в пункті меню VO Services (рис.3.5.).

Рисунок 3.5 – Відкрите меню VO Services

Можна вибрати один з додатків, встановлених в GILDA і, використовуючи веб- інтерфейс для відповідного додатку, запустити завдання і отримати результати його виконання. Як альтернатива можна відкрити пункт меню Job Services (рис. 24) і вибрати одне завдання із списку доступних. Зупинимося більш детально на другому варіанті.

Виберемо одне із завдань. Для визначеності хай це буде ***06. 3-Drendering a human heart***, результатом виконання якого повинна бути побудова тривимірної моделі людського серця. Після натиснення кнопки Next з'явиться наступна сторінка,

де можна вибрати обчислювальний елемент, на якому виконуватиметься задача. Його можна вибрати явно із списку або дозволити брокеру задач самому вирішити це за Вас. Залишимо вибраним другий варіант.

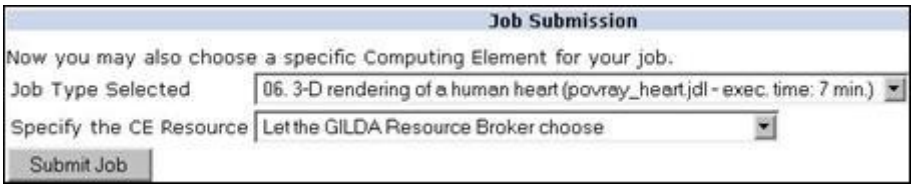


Рисунок 3.6 – Вибір обчислювального елемента для виконання завдання

Після натиснення кнопки Submit Job на екрані з'явиться інформація про успішне створіння завдання і призначення йому унікального ідентифікатора.

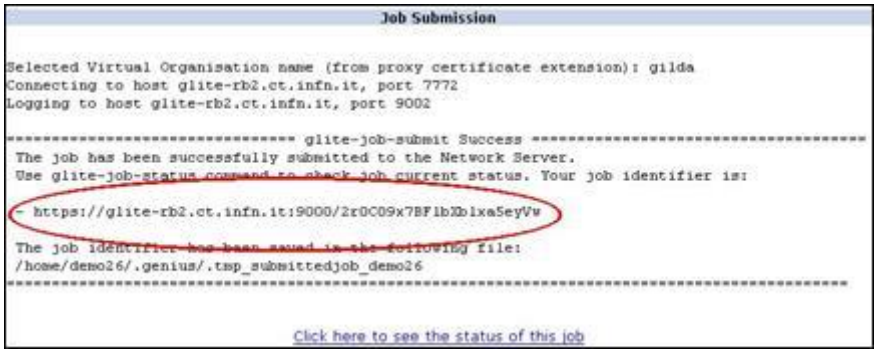


Рисунок 3.7 – Завдання створено і послано в Grid

За станом виконання завдання можна прослідити, натискуючи посилання внизу або вибравши пункт меню Job Queue.

Рисунок 3.8 – Панель відображення статусу виконання завдання.

На рис.3.8 – показані ідентифікатор завдання, час запуску, статус виконання, ресурс, де виконується завдання. Коли завдання буде виконано, в полі Status з'явиться значення Done. Для того, щоб отримати результати виконання завдання, у випадаючому списку Status виберіть GET OUTPUT. Після попередження з'явиться сторінка, на якій потрібно вибрати ідентифікатор завдання, результати виконання якого необхідно отримати.

Після вибору певного завдання з'явиться список файлів, які створені в результаті виконання. В даному випадку:

- povray_heart.out – вихідний текстовий файл

- povray_heart.err – повідомлення про помилки і попередження
- heart.png – графічний файл, що представляє тривимірну модель людського серця.

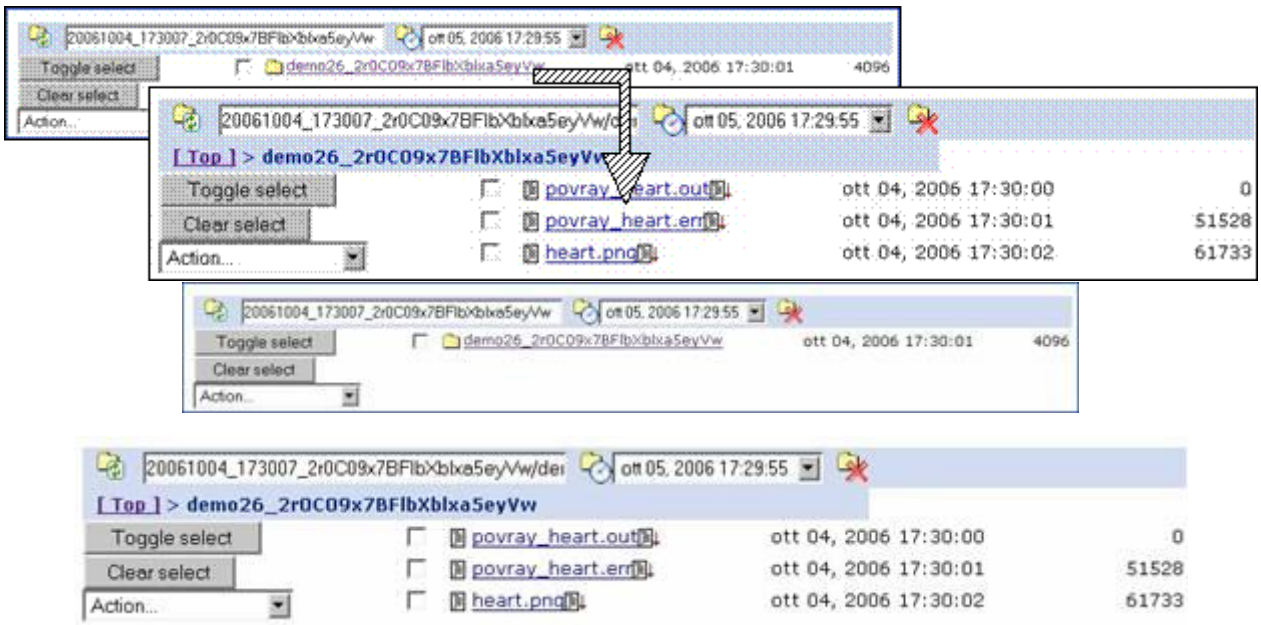


Рисунок 3.9 – Вибір вихідних файлів

Користуючись можливостями браузера, можна прямо в браузере проглянути результати виконання завдання або зберегти вихідні файли на локальному комп'ютері.

Таким чином, використовуючи тільки графічний інтерфейс GILDA Demonstrator, можна вибирати задачі для запуску, вибирати обчислювальний елемент, на якому виконуватиметься завдання, контролювати процес виконання, проглядати результати виконання завдання і зберігати вихідні файли на комп'ютері користувача.

Інший сервіс проекту GILDA – Grid Tutor дозволяє запускати в GILDA Testbed довільні завдання користувача, але в цьому випадку потрібні отримання сертифікату від GILDA CA і включення у GILDA VO. Частіше за все Grid Tutor використовується для проведення учбових курсів EGEE по роботі в Grid. В цьому випадку всі необхідні попередні дії по отриманню сертифікатів, членства у VO, отримання доступу до UI GILDA Testbed і т.і. виконуються службою підтримки GILDA.Можна використовувати GILDA Grid Tutor і для роботи в повнофункціональній Grid-інфраструктурі GILDA Tested. Інструкції можна знайти на сайті GILDA – https://gilda.ct.infn.it/ users.html.

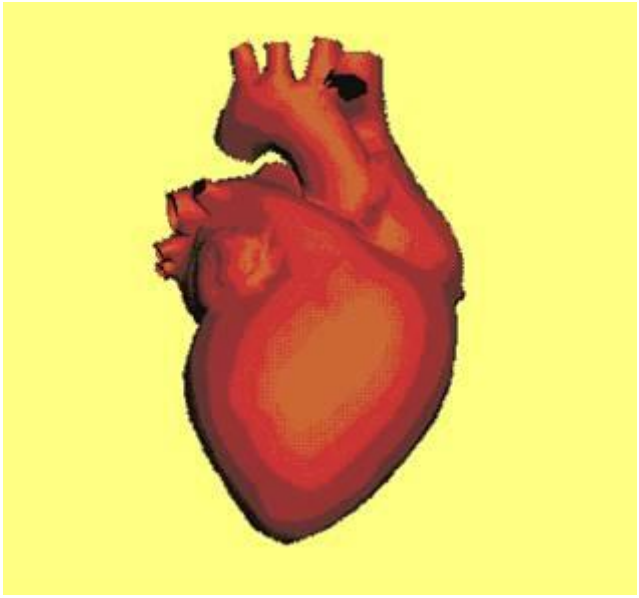


Рисунок 3.10 – Графічний файл з трьохвимірною моделлю людського серця

Портал знань

У рамках проекту UNGI планується застосувати удосконалений портал, який би вібрав в себе найкращі риси існуючих порталів Grid. На шляху до цієї цілі планується, крім всього іншого:

- включити "*об'єднання типів інформації*", притаманених іншим порталам; наприклад, включити (див. також табл.2.):
  - технічні доповіді та офіційні папери;
  - посилання на наукові статті;
  - загальнодоступне сховище компонентів та додатків (з документацією);
  - навчальні модулі (сумісний із SCROM 2004) та ін.
- забезпечити засоби підтримки співпраці, такі як форуми, Вікіпедія, FAQ, та ін.;
- розповсюджувати інформацію, використовуючи механізми керованих станцій з дистанційним управлінням;
- відвести важливе місце транслюванню навчальної діяльності, чатам з експертами (своїми чи запрошеними), які можуть бути записані та збережені для подальшого повторного використання;
- забезпечити доступ до інфраструктури UNGI шляхом адаптації існуючих порталів доступу (наприклад, порталу P-Grade);
- забезпечити інтерактивну інформаційну експертну систему, що пов'язана з процесом отримання знань (див. наступний підрозділ).

Слід зазначити, що, притримуючись нинішньої тенденції впровадження штучного інтелекту в системи IT, портал UNGI буде використовувати найновіші семантичні технології .Наприклад, всі об'єкти (ресурси), накопичені у рамках порталу знань, будуть онтологічно розмежовані. Для цього проводиться оцінка найновітніших з онтологій Grid, яка дасть можливість відібрати для використання в проекті з них ту, що найбільш стійка до збоїв Ця онтологія повинна бути достатньо

гнучкою, щоб грати свою роль у трьох взаємопов'язаних функціях: (а) забезпечувати легкий засіб інтеграції прикладних доменних онтологій, що будуть використовуватися для розмежування компонентів, які належать до різних додатків; (б) забезпечувати належний Grid-орієнтований опис компонентів та додатків, що зберігаються в сховищі компонентів/додатків, та (в) підтримувати управління знаннями на мета-рівні, включаючи інформаційну експертну систему. Таким чином, всі функції в рамках порталу, такі як пошук, співставлення, перегляд інформації, та ін., будуть базуватися на онтологічному підґрунті. Нарешті, планується застосувати існуючі знання про системи рекомендацій та розділені моделі для *виділення персоналізованих ресурсів користувачам порталу*. Зокрема, для постійних користувачів порталу будуть розроблені їх персональні профайли та використані виявлені відповідності, щоб забезпечити їх інформацією про ресурси, які можуть їх цікавити. Пропонуємий портал знань буде розроблятися за допомогою однієї з існуючих загальнодоступних технологій. Допоміжні функції, безпосередньо пов'язані з функціями управління знаннями і передбачені в порталі UNGI говорюються нижче.

Web 2.0/3.0 та GRID

Зараз ми є свідками переходу веб-сайтів від ізольнованої разрозненої інформації до комбінування в них функціональності різних програмних інтерфейсів і змісту даних різних джерел (цей перехід називають «Web 2.0»). Веб-сайти стають комп'ютерними платформами, що обслуговують веб-додатки кінцевих користувачів. Web 2.0 – це підхід до розробки нової функціональності додатків, що базується на сервіс-орієнтованій архітектурі (service oriented architecture, SOA). Раніше термін «Web 2.0» вживали як синонім до семантичної веб- мережі , але зараз йдеться про набагато ширший набір можливостей: Веб як платформа; інновації у сукупностях систем та сайтів, сформованих шляхом збирання від розподілених, незалежних розробників; полегшені бізнес-моделі, забезпечені об'єднанням змісту та сервісу; легкість зчитування інформації тими, хто вперше використовує сайти Web 2.0. Останні можуть підтримувати семантичну мову HTML, використання мікро форматів і агрегування даних у RSS/Atom; використання програмного забезпечення wiki, XACML поверх SOAP для управління доступом між організаціями та доменами; розміщення блогів, машапів, REST чи XML Web-service APIs, використання дружелюбних користувацьких систем керування змістом та інш.

Машап (від англ. mash-up – «змішувати») – це веб-додаток, що поєднує дані більш ніж з одного джерела і будується комбінуванням функціональності різних програмних інтерфейсів і джерел даних. У нього багато спільного з нинішніми Grid-сервісами, заснованими на WSRF. Машапи вже існують як:

1. сервіси агрегування: збирають інформацію з різних джерел та розміщують їх в одному місці;
2. машапи даних: збирають дані з різних джерел, щоб створити новий сервіс (тобто агрегування);
3. відслідковують та фільтрують зміст: відслідковують, фільтрують, аналізують та дозволяють пошук сервісів;

4. сервісні машапи.

Інформаційні потоки та машапи мають багато спільного: фактично інформаційний потік – це різновид машапу. Web 2.0 і Grid можуть забезпечити нові рішення для е-науки. Web 2.0 має особливі переваги в тому, що дозволяє фахівцям легко співпрацювати, використовуючи існуючі ресурси. При цьому Grid виступає у ролі провайдера сервісу, стійкого до помилок, а Web 2.0 – користувацького інтерфейсу (API).

Семантичний Web – частина глобальної концепції розвитку мережі Інтернет, метою якої є реалізація можливості машинної обробки інформації, доступної у Всесвітній павутині. Основний акцент концепції робиться на роботі з метаданими, що однозначно характеризують властивості та зміст ресурсів Всесвітньої павутини, замість того, щоб здійснювати текстовий аналіз документів. Ця концепція була прийнята і просувалась Консорціумом W3C. Для її впровадження пропонується створення мережі документів, що містять метадані про ресурси Всесвітньої павутини і існує паралельно з ними. Тоді як самі ресурси призначені для сприйняття людиною, метадані використовуються пошуковими роботами та іншими інтелектуальними агентами для проведення однозначних логічних висновків про властивості цих ресурсів.

На сьогодні дослідження з семантичного Grid доводять, що все ще існує технологічний розрив між нинішніми можливостями Grid та вимогами е-науки Комбінуючи переваги Web 2.0 та Grid, є шанс зменшити цей розрив. Слід зазначити, що зв'язок між Grid та Web 2.0 – тема зовсім нова. Тому рішення та технології, такі як інтелектуальне ПЗ для обробки семантичних даних, в даний час підлягають експериментальному дослідженню для розроблення найбільш ефективного засобу управління даними та інформацією в рамках Web 3.0, що очікується у 2010 році.

Автоматична композиція сервісу

В проекті UNGI едбачено створення системи високої гнучкості, яка зможе використовувати різні коди у формі веб-сервісів, що походять з різних джерел. З іншого боку, UGrid – це не тільки "технологічна площадка", але також і система для підтримки процесів вилучення знань, тому вона повинна бути зручною у використанні перспективих додатків.

Композиція сервісу може бути визначена як процес відкриття, інтегрування та виконання набору пов'язаних сервісів у належному порядку для формування змістовного та комплексного сервісу Наскільки це стосується сучасного рівня розвитку Grid, композиція сервісів – поле для активних досліджень. За останні роки багато зусиль в дослідницькій діяльності припадало як раз на композицію веб-сервісу. Багато мов запропонували академічні та промислові дослідницькі групи: WSFL та XLANG від Microsoft були двома самими ранніми мовами для визначення стандартів композиції веб-сервісів. Обидві мови розширили WSDL. BPEL4WS було запропоновано як специфікацію, що представляє собою злиття WSFL та XLANG. У додаток до цих комерційних стандартів, що працюють на XML, дослідники розробили унікальну мову композиції веб-сервісу – OWL-S, що передбачає більш широке (також більш семантичне) описання композицій веб-сервісу.

Багато робіт з композиції веб-сервісу впроваджують орієнтований на людину підхід. Розробник вручну генерує план композиції, що публікується у відкритому реєстрі як новий сервіс і виконується цетралізованим механізмом. Коли йде запит до складеного веб-сервісу, механізм виконує план та викликає веб-сервіс, згаданий у плані. Цей тип композиції не масштабуємий, коли число веб-сервісів зростає. Крім того, розробнику треба володіти базовими знаннями про вибрані технології. Виконання не буде здійснене, якщо хоча б один сервіс, згаданий у плані, недоступний. Може застосовуватися полуавтоматичний підхід. Тут розробник опрацьовує інфраструктуру, яка шляхом семантичного аналізу доступних сервісів спроможна допомогти йому відібрати компоненти, які необхідні в процесі композиції. Крок за кроком розробник вибирає сервіс із списку. інфраструктура підтримує погодження між вибраним сервісом та вимогами від компонентів., яке базується на врахуванні функціональних та не функціональних властивостях. Новий складений сервіс зберігається як комплексний процес DAML-S. Цей підхід все ще не є масштабуємим. Існує підхід, який є повністю автоматичним. Він повинен підтримуватися семантичним описом забезпечених сервісних можливостей, щоб розпізнати, чи є щось спільне між доступними можливостями та необхідними функціями, що виконуються. Описання сервісу повинне надавати інформацію про сервісні вимоги, протокол взаємодії та механізми низького рівня, що використовуються також для активізації. Тобто автоматична композиція потребує повну семантичну підтримку, а не тільки синтаксичну спроможність до взаємодії. Деякі стандарти описують, як велику кількість веб-сервісів можна скомпонувати разом, щоб створити більш комплексний веб-сервіс, але вони тільки розглядають композицію на синтаксичному рівні. Таким чином, автоматична композиція сервісу матиме справу у UNGI з двома різними питаннями:

1. Відкриття набору сервісів, чия поведінка на високому рівні відома, який, вірно скомпонований, забезпечує необхідний результат. Цьому подходу слідують у семантичній веб-мережі.
2. Описання взаємозв’язків між компонентами та спосіб їх виконувати. Цей підхід адресує композицію веб-сервісу як проблему управління інформаційним потоком.

Складений інформаційний потік може бути виконаний відповідно до підходів оркестровки та хореографії. Оркестровка веде до централізованої архітектури, у якій механізм оркестровки контролює виконання завдання. Хореографія базується на співробітницькому підході, де кожна група грає свою власну роль, виконуючи дії, задані лише їй. *Сервісна оркестровка* – дуже популярний термін в управлінні бізнес-процесами та пов’язане з гнучким визначенням з сервісним порядком виконання. Тут процес складається з дій, що виконуються в передбачуваному порядку. Ці порядки виконання можуть змінюватися швидко чи – у випадку включення механізмів бізнес-правил – можуть навіть мати багато виконуваних гілок, визначених одночасно. Процеси, які використовують засоби моделювання та концепції, повинні бути відображені на існуючих сервісах, щоб забезпечити автоматичне їх виконання у випадку перерозподілу процесів. На технологічному рівні визначення бізнес-процесу перетворюється у порядок виконання сервісу. Мови моделювання для цього типу сервісів і їх завдань включають BPEL, XLANG, XPDS

чи ASAP/WF-XML jPDL, JSR207, ebXML чи CPPA. Тут онтологія використовується для того, щоб фільтрувати доступні веб-сервіси та щоб вибирати для композиції ті, що підходять. Інформаційні потоки визначаються статично, в той час як їх виконання оркеструється. *Хореографія* може бути тісно прив'язана до використання агентних систем. Тут сущності визначаються через свої ролі та спроби змусити їх співпрацювати одна з одною. Співробітництво включає, окрім всього іншого, включення у набір стандартизованих протоколів взаємодії (наприклад, контрактного мережного протоколу FIPA). Крім того, взаємодія між компонентами (агентами) досягається через обмін повідомленнями (наприклад, використовуючи мову взаємодії агентів FIPA (FIPA Agent Communication Language – ACL). Крім того, спроможність системи до адаптації досягається через зміну схем (патернів) взаємодії між "компонентами." У випадку взаємодіючих компонентів (агентів) онтології використовуються напряму "усередині" повідомлень, якими обмінюються (наприклад, застосовуючи семантичну мову FIPA (FIPA Semantic Language – FIPA SL), щоб передавати точну природу дії, на яку був запит).

Онтології та GRID

Онтології вважаються одним з найбільш багатообіцяючих підходів для вводу семантики (значення) у автоматичну обробку інформації. У випадку Grid передбачається, що використання онтологій покращить якість та споживчі якості інформації про програмне забезпечення, програмне забезпечення та дані Grid; оскільки вони дозволяють ділитися спільними знаннями про доменні концепції, роблять можливим повторне використання доменних знань, чітке визначення доменних визначень, підвищення спроможності до взаємодії тощо. Онтологія – це засіб опису семантики проблемної області за допомогою словника і підібраної специфікації існуючих в ній відношень та обмежень, що забезпечують інтеграцію словника.Інформаційні онтології створюються завжди з конкретною метою – рішення конструкторських задач; вони оцінюються більше з точки зору використання, ніж повноти.

Онтології – це фундаментальні блоки для будівництва семантичної Grid. Їх визначення: "розширення існуючої Grid, де інформації та сервісам надаються добре визначене значення, покращені можливості для об'єднаної роботи людей та комп'ютерів". W3C- рекомендовані онтологічні мови, такі як OWL та RDF, базуються на класичній парадигмі, яка представляє собою модельну парадигму, що базується на наборі характеристик, які відповідають відкритому середовищу семантичної веб-мережі. Більш того, проекти W3C та OASIS мають результатом серію специфікацій, що робить можливими співробітництво та композицію динамічних сервісів. Тут найбільш відомими є WS-Transaction, WS-Coordination, WSCI, BPML та BPEL4WS. У той же самий час у спільноті Grid було витрачено багато зусиль до того, щоб розробити автоматичне управління ресурсами Grid та брокерські рішення. В цьому контексті важливе розділення між значенням, поведінкою та реалізацією компонентів додатку Grid. Можна підбирати специфікацію додатку високого рівня під оптимальну комбінацію доступних компонентів, у той час як можна також перетворювати абстрактний інформаційний

потік у конкретний. Інфраструктура вибору ресурсів GrADS розглядає відкриття та конфігурацію фізичних ресурсів, що відповідають вимогам прикладних програм. Існує опис співставлення ресурсів Grid з використанням семантичних веб-технологій (співставлення матеріальних ресурсів з використанням онтологій, фундаментальні знання та правил). Він придає велике значення потребі у семантичному описі ресурсів Grid та співставленню ресурсів, але не розглядає питання їх виконання та ефективності. Проект CrossGrid забезпечує розподілений реєстр компонентів з технологією P2P та підтримує внутрішньореєстровий зв'язок для виконання табличної когерентності.

Фундаментальна операція в управлінні онтологіями – співставлення, що бере дві онтології як входи та здійснює перетворення даних між елементами двох онтологій, що семантично відповідають один одному. Співставлення грає головну роль у великому числі додатків, таких як Веб-орієнтована інтеграція даних, електронна комерція, інтеграція схем, еволюція та міграція схем, еволюція прикладної системи, технологія інформаційних сховищ, розробка баз даних, створення веб-сторінок та управління ними, та компонентно-орієнтоване середовище. Співставлення схеми вручну – втомлюючий, довгий, схильний до помилок, і тому дорогий процес. Тому потрібне автоматичне співставлення схеми. Аби визначити оператор співставлення (*match*), треба вибрати представлення їх вхідних схем та вихідного відображення. На приктиці повине бути обране особливе представлення, таке як модель зв’язку сущностей (entity-relationship, ER) , об’єктно-орієнтовані (ОО) моделі та моделі БД, XML чи направлені графи. В кожному з цих випадків існує природна відповідність між складовими блоками представлення та поняттями елементів та структури: сущністі та зв’язки у моделях ER; об’єкти та зв’язки у моделях ОО; елементи, субелементи та IDREF у XML; та вузли та ребра у графах. Співставлення – бінарна операція, що визначає пари відповідних елементів з їх вхідних операндів та виконує операції над метаданими (елементами схеми), і елемент в результаті співставлення може бути пов'язаний з великим числом елементів з обох входів. Було розроблено багато систем для підгонки схем. Вони базувалися на одному чи більше (часто комбінації) методів, описаних вище. Визначні приклади – S-Match , Anchor-Prompt, Cupid, QOM, COMA . SUN розробило прототипний засіб, який виконує формування семантичних веб-сервісів, що базується на семантичномій та структурномій підгонці схеми; він підгоняє вхідну онтологію, описує запит сервісу та описи веб-сервісів на “синтаксичному рівні” через WSDL, чи на семантичному рівні через сервісні онтології, включені в OWL-S, WSMO, SWSF and WSDL-S. Як було сказано вище, існуючі Grid- та доменні технології можуть використовуватися у трьох взаємопов’язаних контекстах: (а) доменні онтології для розмежування предметних компонентів, (б) загальна онтологія Grid для розмежування Grid-пов’язаних аспектів ресурсів, які збираються та над якими здійснюються операції у рамках інфраструктури Grid, та (в) розширена

онтологія Grid для забезпечення розмежування ресурсу мета-рівня, таким чином підтримуючи функції, пов’язані з управлінням знаннями.

Щодо інструментів, що можуть бути використані для збору знань, тут існує можливість вибору. Система PCPACK забезпечує підтримку для таких кроків: (1) створення бази знань; (2) аналіз тексту (використовуючи інструмент протоколу); (3) структурування знань (використовуючи інструмент ланцюгової схеми / сходинок); (4) анотація сторінок (використовуючи інструмент анотації); (5) публікація бази знань (використовуючи інструмент публікації); (6) прогляд веб-сайту (використовуючи стандартний браузер). Інший інструмент, що може бути дуже корисним протягом процесу розробки інформаційної системи – SPHINX (інструмент AtechSPHINX, www.aitech.pt). Пакет містить наступні інструменти: експертна система-оболонка (PC-Shell); нейронний пристрій для моделювання мережі (Neuronix); виконуюча система представлення знань (CAKE); інтелектуальний розробник прикладних програм (HybRex); система прогнозування (Predyctor); ведуча система навчання (deTreex). Аналіз буде виконано на основі аналізу вимог. В проекті UNGI будуть широко використовуватися онтології, інтеграція інструментів вибору управління знаннями з іншими базованими на онтологіях технологіями, де інструменти є надзвичайно важливими.

Сервіси аналізу данних

Маючи набір зібраних даних, можна використовувати деякі методики аналізу даних, аби збільшити кількість знань, зосереджених у порталі знань, і щоб розв’язати нові задачі. ***Data Mining*** – це процес виділення, дослідження і моделювання великих об'ємів даних для виявлення невідомих до цього структур (patterns) з метою досягнення переваг. Це процес, мета якого – знайти нові значущі кореляції, образи і тенденції в результаті просівання великого об'єму збережених даних з використанням методик розпізнавання образів плюс застосування статистичних і математичних методів .Термін Data Miningчасто переводиться як здобич даних, витягання інформації, інтелектуальний аналіз даних, засоб пошуку закономірностей, розкопка знань в базах даних, "промивання" даних.

Виділяють наступні стадії цього процесу вилучення даних:

- стадія виявлення закономірностей (вільний пошук);
- стадія використовування виявлених закономірностей для прогнозу невідомих значень (прогнозуюче моделювання). На додаток до цієї стадії іноді вводять стадію валідації;
- стадія аналізу виключень, яка призначена для виявлення і пояснення аномалій, знайдених в закономірностях.

В технології Data Mining гармонійно об'єдналися строго формалізовані методи і методи неформального аналізу, тобто кількісний і якісний аналіз даних. Всі методи Data Mining підрозділяються на дві великі групи за принципом роботи з початковими даними: чи зберігаються дані після Data Mining або вони дистилюються для подальшого використовування.

До статистичних методів обробки даних відносять:

- дескриптивний аналіз і опис початкових даних;
- аналіз зв'язків (кореляційний і регресійний аналіз, аналіз чинника, дисперсійний аналіз);
- багатовимірний статистичний аналіз (компонентний аналіз, дискримінтальний аналіз, багатовимірний регресійний аналіз, канонічні кореляції і ін.);
- аналіз тимчасових рядів (динамічні моделі і прогнозування).

До кібернетичних засобів обробки даних належать:

- штучні нейроні мережі (розпізнавання, кластеризація, прогноз);
- еволюційне програмування (у тому числі алгоритми методу групового обліку аргументів);
- генетичні алгоритми (оптимізація);
- асоціативна пам'ять (пошук аналогів, прототипів);
- нечітка логіка;
- дерева рішень;
- системи обробки експертних знань.

Серед методів Data Mining виділяють також методи рішення задачі *сегментації* (тобто задачі класифікації і кластеризації): і *прогнозуючі* методи. До першої групи відносяться: ітеративні методи кластерного аналізу, у тому числі: алгоритм к-середніх, к-медіан, ієрархічні методи кластерного аналізу, карти Кохонена, методи крос- табличної візуалізації, що самоорганізуються, різні методи візуалізації і інші. Описові методи служать для знаходження шаблонів або образів, що описують дані, які піддаються інтерпретації з погляду аналітика.

Прогнозуючі методи використовують значення одних змінних для прогнозування невідомих (пропущених) або майбутніх значень інших (цільових) змінних. До методів, направлених на отримання прогнозуючих результатів, відносяться такі методи: нейроні мережи, дерева рішень, лінійна регресія, метод найближчого сусіда, метод опорних векторів і д.

Технології Data Mining використовуються для наукових досліджень: в медицині, біологіяї молекулярній генетиці і генній інженерії, біоінформатиці, астрономії, прикладній хімії, дослідженнях, що стосуються наркотичної залежності, і інші; для вирішення Веб-задач: пошукові машини (search engines); для вирішення бізнес-задач (банківська справа, фінанси, страхування, CRM, виробництво, телекомунікації, електронна комерція, маркетинг, фондовий ринок і інші); для вирішення задач державного рівня пошук осіб, що ухиляються від податків; засоби в боротьбі з тероризмом тощо.

Слід зазначити, що в рамках проекту UGrid планується вбудувати у Grid певне число засобів інтелектуального аналізу даних. Вони будуть старанно відібрані та адаптовані, враховуючи обсяг та типи даних, що зберігаються у Світовому Центрі Даних , при цьому всеохоплююче використання онтологій вплине на вибір методів аналізу даних.

Приклади порталів

В табл. 3.1 приведені короткі описання деяких сучасних ключових порталів Grid.

Таблиця 3.1

Приклади порталів

Портал	*Опис*	*Тип*
База знань E-IRG	Розроблений, щоб підтримувати роботу e-IRG та забезпечувати інформацію про e-Інфраструктури, доступні і в країнах ЄС (національні), і на європейському рівні (перш за все, Grid та суперкомп'ютери). Поєднує інформацію з багатьох джерел. Доступні ресурси включають кілька тематичних карт: принципи e-Інфрастуктур; Національна та регіональна Grid; системи e-Інфраструктури; проекти Grid; документи; класифікація сфер застосування Grid; TOP500; країни; делегати E-IRG. URL: http:// www.e-irg.org/	Портал знань
EGEE	Завдання EGEE не обмежується ядерною фізикою і полягає в тому, щоб реалізувати потенціал Grid і для багатьох інших науково-технологічних галузей. Так, у найближчих планах керівництва проекту – створення окремого біоінформаційного «Grid-блоку» URL: http://egee.cesnet.cz/en/	потужня Європейська система
GILDA	GILDA (Grid INFN Laboratory for Dissemination Activities) – це віртуальна лабораторія, створена для того, щоб демонструвати / розповсюджувати можливості Grid. Портал GILDA P-Grade – це середовище розробки та виконання інформаційного потоку Grid, що базується на останній версії Grid-порталу P-Grade. URL: https://gilda.ct.infn.it	Доступ до Grid плюс допоміжні знання
OMII	Включає: інформацію по проекту; документацію по нинішній версії (керівництво з установлення та запуску, керівництво користувача, навчальне керівництво, Wiki). URL: http://www.omii.ac.uk	Портал сумісного доступу
Портал P-Grade	Заснований на Веб середовищі для розробки, виконання та моніторингу інформаційних потоків на різних платформах Grid (Globus Toolkit 2, Globus Toolkit 4, LCG та gLite Grid middlewares). Включає програмне забезпечення та документацію. URL: http://www.lpds.sztaki.hu/pgrade/	Портал сумісного доступу

Migrating Desktop Platform	Забезпечує доступ користувачів до ресурсів Grid,виконує моніторинг і візуалізацію, виконання інтерактивних додатків, керування файлами даних. Розроблений Познанським суперомп'ютерним центром і використовується в проекті BalticGrid URL: http://desktpo.psnc.pl	Портал доступу
GENIUS	Точка доступу до Grid, що базується на інтерфейсі командної строки (Command line interface, CLI). Доступний «звідки завгодно», наприклад, з настольного ПК, ноутбуку, персонального електронного асистенту чи мобільного телефону. Оснований на мові описання завдання (Job Description Language, JSDL). GENIUS сумісний з багатьма версіями програмного забезпечення проміжного шару EGEE, в тому числі з gLite. URL: http://egee.cesnet.cz/en/user/genius.html	Портал доступу
SDSC HotPage	HotPage – це набір скриптів cgi та html-сторінок, що дають користувачам змогу отримати доступ до інформації про віддалені комп'ютери з веб-браузеру. Веб-браузер періодично поновлює кадри стану за допомогою Java-скрипту. Наступна версія HotPage планує використовувати програмне забезпечення проміжного шару Globus. URL: http://www.growl.org.uk/ukhec_portal/node6.html	Портал доступу, обмежені можливості

ЛАБОРАТОРНА РОБОТА № 4

Тема: Загальна організація mpi. Організація комунікаційних операцій типу "крапка – крапка".

Мета: ознайомитися з бібліотекою паралельного програмування MPI, одержати практичні навички програмування мовою С з використанням бібліотеки MPI, організації обмінів типу «крапка – крапка».

ТЕОРЕТИЧНІ ВІДОМОСТІ

Загальна організація MPI.

MPI-програма являє собою набір незалежних процесів, кожний з який виконує свою власну програму (не обов'язково ту саму), написану мовою С чи FORTRAN. З'явилися реалізації MPI для C++, однак розроблювачі стандарту MPI за них відповідальності не несуть. Процеси MPI-програми взаємодіють один з одним за допомогою виклику комунікаційних процедур. Як правило, кожен процес виконується у своєму власному адресному просторі, однак допускається і режим поділу пам'яті. MPI не специфікує модель виконання процесу – це може бути як послідовний процес, так і багатопотоковий. MPI не надає ніяких засобів для розподілу процесів по обчислювальних вузлах і для запуску їх на виконання. Ці функції покладаються або на операційну систему, або на програміста. Зокрема, на кластерах – спеціальний командний файл (скріпт) mpirun, що припускає, что здійсненні модулі вже якимсь чином розподілені по комп'ютерах кластера. MPI не накладає жодних обмежень на те, як процеси будуть розподілені по процесорах, зокрема, можливий запуск MPI програми з декількома процесами на звичайній однопроцесорній системі.

Для ідентифікації наборів процесів вводиться поняття *групи*, що поєднує всі чи якусь частина процесів. Кожна група утворює *область зв'язку*, з яким зв'язується спеціальний об'єкт – *комунікатор області зв'язку*. Процеси усередині групи нумеруються цілим числом у діапазоні 0..groupsize-1. Усі комунікаційні операції з деяким комунікатором будуть виконуватися тільки усередині області зв'язку, описуваної цим комунікатором. При ініціалізації MPI створюється визначена область зв'язку, що містить усі процеси MPI-програми, з яким зв'язується визначений комунікатор MPI_COMM_WORLD. У більшості випадків на кожнім процесорі запускається один окремий процес, і тоді терміни процес і процесор стають синонімами, а величина groupsize стає рівної NPROCS – числу процесорів, виділених задачі. Отже, MPI – це бібліотека функцій, що забезпечує взаємодію рівнобіжних процесів за допомогою механізму передачі повідомлень. Це досить об'ємна і складна бібліотека, що складається приблизно з 130 функцій, у число яких входять:

- функції ініціалізації і закриття MPI процесів;
- функції, що реалізують комунікаційні операції типу крапка – крапка;
- функції, що реалізують колективні операції;
- функції для роботи з групами процесів і комунікаторами;
- функції для роботи із структурами даних;

- функції формування топології процесів.

Набір функцій бібліотеки MPI далеко виходить за рамки набору функцій, мінімально необхідного для підтримки механізму передачі повідомлень. Користувачу належить право самому вирішувати, які засоби з наданого арсеналу використовувати, а які ні. У принципі, будь-яка паралельна програма може бути написана з використанням всього 6 MPI функцій, а досить повне і зручне середовище програмування складає набір з 24 функцій.

Кожна з MPI функцій характеризується способом виконання:

1. Локальна функція – виконується усередині процесу, що її визвав. Її завершення не вимагає комунікацій.
2. Нелокальна функція – для її завершення потрібно виконання MPI процедури іншим процесом.
3. Глобальна функція – процедуру повинні виконувати всі процеси групи. Недотримання цієї умови може приводити до зависання задачі.
4. Функція, що блокує – повернення керування з процедури гарантує можливість повторного використання параметрів, що беруть участь у виклику. Ніяких змін у стані процесу, що визвали запит, що блокує, до виходу з процедури не може відбуватися.
5. Функція, що неблокує – повернення з процедури відбувається негайно, без чекання закінчення операції і до того, як буде дозволене повторне використання параметрів, що беруть участь у запиті. Завершення операцій, що не блокують, здійснюється спеціальними функціями.

Використання бібліотеки MPI має деякі відмінності в мовах C і FORTRAN.

У мові C усі процедури є функціями, і більшість з них повертає код помилки. При використанні імен підпрограм і іменованих констант необхідно строго дотримувати регістр символів. Масиви індексуються з 0. Логічні перемінні представляються типом int (true відповідає 1, a false – 0). Визначення всіх іменованих констант, прототипів функцій і визначення типів виконується підключенням файлу mpi.h. Уведення власних типів у MPI було продиктовано тим обставиною, що стандартні типи мов на різних платформах мають різне представлення. MPI допускає можливість запуску процесів паралельної програми на комп'ютерах різних платформ, забезпечуючи при цьому автоматичне перетворення даних при пересиланнях. У таблиці 4.1 приведена відповідність визначених у MPI типів стандартним типам мови C.

Таблиця 4.1.

Відповідність між MPI-типами 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	
MPI_PACKED	

Будемо розглядати тільки С – реалізацію бібліотеки.

У таблиці 4.1 перерахований обов'язковий мінімум підтримуваних стандартних типів, однак, якщо в базовій системі представлені й інші типи, то їхню підтримку буде здійснювати і MPI, наприклад, якщо в системі є підтримка комплексних перемінних подвійної точності DOUBLE COMPLEX, то буде присутній тип MPI_DOUBLE_COMPLEX. Типи MPI_BYTE і MPI_PACKED використовується для передачі двоїчної інформації без якого-небудь перетворення. Крім того, програмісту даються засоби створення власних типів на базі стандартних (див. нижче).

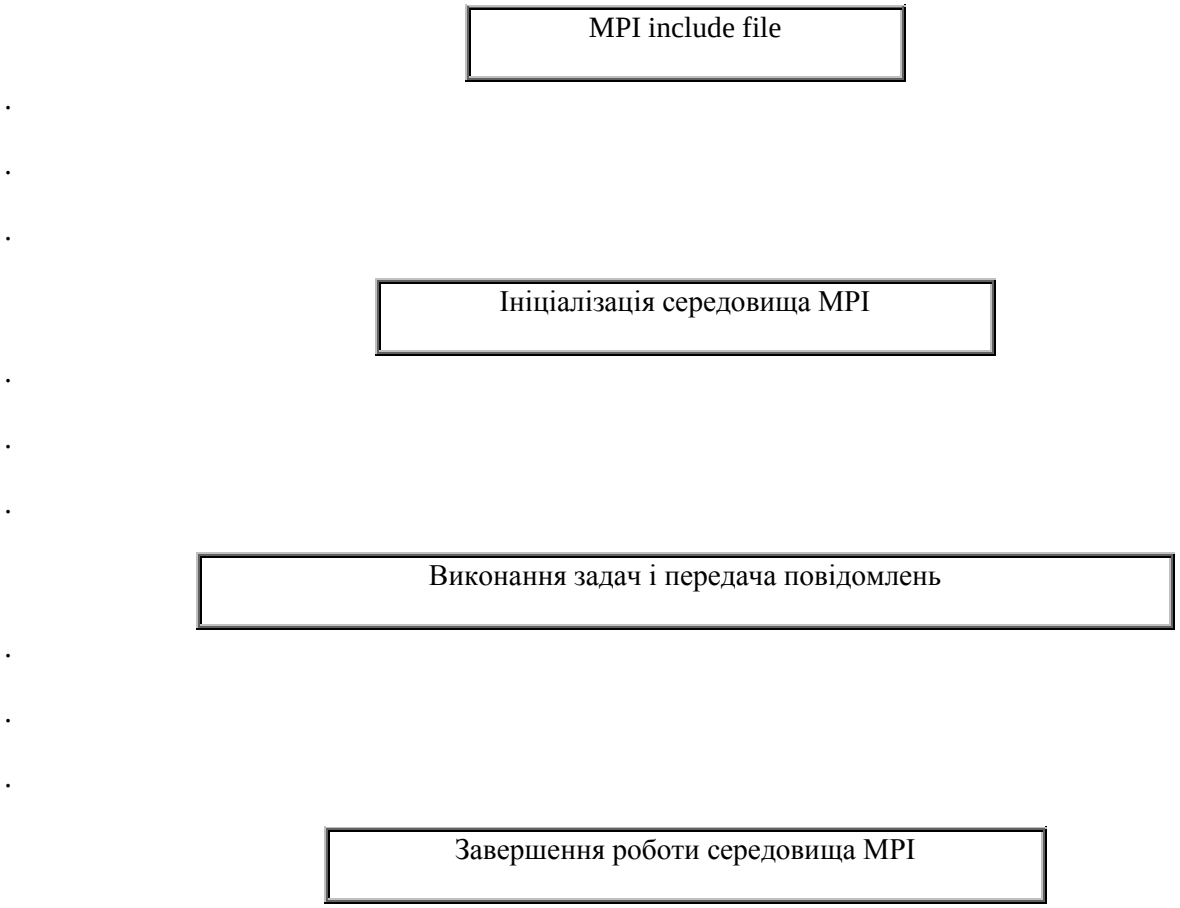
Вивчення MPI почнемо з розгляду базового набору з 6 функцій, що утворять мінімально повний набір, достатній для написання найпростіших програм. Під час обговорення параметрів процедур символами IN будемо указувати вхідні параметри процедур, символами OUT вихідні, а INOUT – вхідні параметри, що модифікуються процедурою.

Базові функції MPI

Основні конструкції MPI, необхідні в будь-якій змістовній програмі описані в заголовному файлі mpi.h.

include <mpi.h>

Програма на MPI звичайно має наступну структуру:



Будь-яка прикладна MPI-програма повинна починатися з виклику функції ініціалізації MPI (функція MPI_Init). У результаті виконання цієї функції створюється група процесів, у яку містяться всі процеси, і створюється область зв'язку, описуваний визначеним комунікатором MPI_COMM_WORLD. Ця область зв'язку поєднує всі процеси. Процеси в групі упорядковані і пронумеровані від 0 до groupsize-1, де groupsize дорівнює числу процесів у групі. Крім цього, створюється визначений комунікатор MPI_COMM_SELF, що описує свою область зв'язку для кожного окремого процесу.

Для визначення які набори процесів можуть один з одним спілкуватися в MPI використовуються так звані групи і коміунікатори. Більшість функцій MPI вимагають визначення комунікатора як аргумента.

Для простоти можна спочатку використовувати визначений комунікатор **MPI_COMM_WORLD**, що включає усі ваші MPI процеси.

Усередині комунікатора кожен процес має свій унікальний цілочисельний ідентифікатор – ранг, видаваний системою при ініціалізації процесу. Ранг звичайно називають "process ID". Ранг видається безупинно, починаючи з нуля.

При передачі повідомлень ранг використовується для визначення адрес.

Функції керування середовищем MPI.

Синтаксис *функції ініціалізації MPI_Init*:

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

У програмах на С кожному процесу при ініціалізації передаються аргументи функції main, отримані з командного рядка.

Функція завершення MPI програм MPI_Finalize.

int MPI_Finalize(void)

Функція закриває всі MPI-процеси і ліквідує всі області зв'язку.

Функція визначення числа процесів в області зв'язку MPI_Comm_size.

int MPI_Comm_size(MPI_Comm comm, int *size)

IN comm – комунікатор;

OUT size – число процесів в області зв'язку комунікатора comm.

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

До створення явно груп і зв'язаних з ними комунікаторов єдино можливими значеннями параметра COMM є MPI_COMM_WORLD і MPI_COMM_SELF, що створюються автоматично при ініціалізації MPI. Підпрограма є локальної.

Функція визначення номера процесу MPI_Comm_rank.

int MPI_Comm_rank(MPI_Comm comm, int *rank)

IN comm – комунікатор;

OUT rank – номер процесу, що визвав функцію.

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

int MPI_Abort(MPI_Comm comm, int errcode)

Знищує всі MPI процеси, що асоціюються з комунікатором comm.

int MPI_Get_processor_name(char *name, int *len)

Визначає ім'я процесора, на якому виконується дана команда. Також визначає довжину імені процесора. Буфер name повинний бути як мінімум розміром у MPI_MAX_PROCESSOR_NAME символів. Ім'я, що повертається, залежить від апаратного забезпечення і може не збігатися з ім'ям, що повертається shell-командою hostname.

int MPI_Initialized(int *flag)

Перевіряє чи було ініціалізовано середовище MPI.

Тепер знаючи все це можна розглянути перший приклад:

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

int main(int argc, char** argv)
{
      int    numtasks, rank, rc;

      rc = MPI_Init(&argc, &argv);
      if (rc != 0) {
            printf     ("Error     starting     MPI     program.
Terminating.\n");
            MPI_Abort(MPI_COMM_WORLD, rc);
      }

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

```
        printf ("Number of tasks= %d My rank= %d\n", numtasks,
rank);

        /******* do some work *******/
        MPI_Finalize();
    }
```

У мінімальний набір варто включити також дві функції передачі і прийому повідомлень.

Функція передачі повідомлення MPI_Send.

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

IN buf – адреса початку розташування даних, що пересилаються;
IN count – число елементів, що пересилаються;
IN datatype – тип елементів, що посилаються;
IN dest – номер процесу-одержувача в групі, зв'язаної з комунікатором comm;
IN tag – ідентифікатор повідомлення
IN comm – Комунікатор області зв'язку.

Функція виконує посилку count елементів типу datatype повідомлення з ідентифікатором tag процесу dest в області зв'язку комунікатора comm. Перемінна buf – це, як правило, масив чи скалярна перемінна. В останньому випадку значення count = 1.

Функція прийому повідомлення MPI_Recv.

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

OUT buf – адреса початку розташування прийнятого повідомлення;
IN count – максимальне число прийнятих елементів;
IN datatype – тип елементів прийнятого повідомлення;
IN source – номер процесу-відправника;
IN tag – ідентифікатор повідомлення;
IN comm – Комунікатор області зв'язку;
OUT status – Атрибути прийнятого повідомлення.

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

Розглянемо функцію, що не входить в обкреслений мінімум, але важлива для розробки ефективних програм. Мова йде про функцію одержання відліку часу – таймері. Досвід роботи з різними операційними системами показує, що при переносі додатків з однієї платформи на другу перше (а іноді і єдине), що приходиться переробляти, це звертання до функцій обліку часу. Тому розроблювачі MPI, домагаючись повної незалежності програм від операційного середовища, ввели і свої функції відліку часу.

Функція відліку часу (таймер) MPI_Wtime.

double MPI_Wtime(void)

Функція повертає астрономічний час у секундах, що пройшла з деякого моменту в минулому (крапки відліку). Гарантується, що ця крапка відліку не буде змінена протягом життя процесу. Для хронометрування ділянки програми виклик

функції робиться на початку і кінці ділянки і визначається різниця між показаннями таймера.

```
{
  double starttime, endtime;
  starttime = MPI_Wtime();
  ... хронометруєма ділянка ...
  endtime = MPI_Wtime();
 printf("Виконання зайняло %f секунд\n", endtime-starttime);
}
```

Функція MPI_Wtick, що має точно такий же синтаксис, повертає мінімальне значення кванта часу.

Огляд комунікаційних операцій типу крапка – крапка

До операцій цього типу відносяться дві представлені в попередньому розділі комунікаційні процедури. У комунікаційних операціях типу крапка – крапка завжди беруть участь не більш двох процесів: передавальний і приймаючий. У MPI мається безліч функцій, що реалізують такий тип обмінів. Різноманіття порозумівається можливістю організації таких обмінів безліччю способів. Описані в попередньому розділі функції реалізують *стандартний режим із блокуванням.*

Функції, що блокують, мають на увазі повне закінчення операції після виходу з процедури, тобто процес, що визиває блокується, поки операція не буде довершена. Для функції посилки повідомлення це означає, що всі дані, що пересилаються, поміщені в буфер (для різних реалізацій MPI це може бути або якийсь проміжний системний буфер, або буфер одержувача). Для функції прийому повідомлення блокується виконання інших операцій, поки все дані з буфера не будуть поміщені в адресний простір приймаючого процесу.

Функції, що не блокують, мають на увазі сполучення операцій обміну з іншими операціями, тому що не блокують функції передачі і прийому, по суті є функціями ініціалізації відповідних операцій. Для опитування закінчення операції (і завершення) введені додаткові функції.

Як для операцій, що блокують, так і для тих, що не блокують, MPI підтримує чотири режими виконання. Ці режими стосуються тільки функцій передачі даних, тому що для операцій що блокують і не блокують є по чотирьох функцій посилки повідомлення. У таблиці 4.2 перераховані імена базових комунікаційних функцій типу крапка – крапка, що є в бібліотеці MPI.

Таблиця 4.2

Список комунікаційних функцій типу крапка – крапка

Спосіб зв'язку	З блокуванням	Без блокування
Стандартна посилка	MPI_Send	MPI_Isend
Синхронна посилка	MPI_Ssend	MPI_Issend
Буферізована посилка	MPI_Bsend	MPI_Ibsend
Погоджена посилка	MPI_Rsend	MPI_Irsend
Прийом інформації	MPI_Recv	MPI_Irecv

З таблиці добре видний принцип формування імен функцій. До імен базових функцій Send/Recv додаються різні префікси.

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

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

Префікс R (ready) – погоджений чи підготовлений режим передачі даних. Операція передачі даних починається тільки тоді, коли приймаючий процесор виставив ознаку готовності прийому даних. Функція нелокальна.

Префікс I (immediate) – відноситься до операцій, що не блокують.

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

Комунікаційні операції, що блокують

Синтаксис базових комунікаційних функцій MPI_Send і MPI_Recv був розглянутий вище, тому тут ми розглянемо тільки семантику цих операцій.

У стандартному режимі виконання операції обміну включає три етапи:

1. Передавальна сторона формує пакет повідомлення, у який крім переданої інформації упаковуються адресу відправника (source), адреса одержувача (dest), ідентифікатор повідомлення (tag) і комунікатор (comm). Цей пакет передається відправником у буфер, і на цьому функція посилки повідомлення закінчується.
2. Повідомлення системними засобами передається адресату.
3. Приймаючий процесор витягає повідомлення із системного буфера, коли в нього з'явиться потреба в цих даних. Змістовна частина повідомлення міститься в адресний простір приймаючого процесу (параметр buf), а службова в параметр status.

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

- комунікатора (comm), оскільки кожен процес може одночасно входити в кілька областей зв'язку;
- номера відправника в цій області зв'язку (source);

- ідентифікатора повідомлення (tag), що використовується для взаємної прив'язки конкретної пари операцій посилки і прийому повідомлень.

Параметр count (кількість прийнятих елементів повідомлення) у процедурі прийому повідомлення повинний бути не менше, ніж довжина прийнятого повідомлення. При цьому реально буде прийматися стільки елементів, скільки знаходиться в буфері. Така реалізація операції читання зв'язана з тим, що MPI допускає використання розширених запитів для ідентифікаторів повідомлень (MPI_ANY_TAG – читати повідомлення з будь-яким ідентифікатором) і для адрес відправника (MPI_ANY_SOURCE – читати повідомлення від будь-якого відправника). Не допускається розширених запитів для комунікаторів. Розширені запити можливі тільки в операціях читання. У цьому відбивається фундаментальна властивість механізму передачі повідомлень – асиметрія операцій передачі і прийому повідомлень, зв'язана з тим, що ініціатива в організації обміну належить передавальній стороні.

Таким чином, після читання повідомлення деякі параметри можуть виявитися невідомими, а саме: число лічених елементів, ідентифікатор повідомлення й адреса відправника. Цю інформацію можна одержати за допомогою параметра status. Перемінні status повинні бути явно оголошені в MPI програмі. У мові C status – це структура типу MPI_Status із трьома полями MPI_SOURCE, MPI_TAG, MPI_ERROR. У мові FORTRAN status – масив типу INTEGER розміру MPI_STATUS_SIZE. Константи MPI_SOURCE, MPI_TAG і MPI_ERROR визначають індекси елементів. Призначення полів перемінної status представлено в таблиці 4.3.

Таблиця 4.3

Призначення полів перемінної status

Поля status	C
Процес-відправник	status.MPI_SOURCE
Ідентифікатора повідомлення	status.MPI_TAG
Код помилки	status.MPI_ERROR

Як видно з таблиці 4.3, кількість лічених елементів у перемінну status не заноситься. Для визначення числа фактично отриманих елементів повідомлення необхідно використовувати спеціальну *функцію MPI_Get_count*:

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

IN status – атрибути прийнятого повідомлення;
IN datatype – тип елементів прийнятого повідомлення;
OUT count – число отриманих елементів.

Підпрограма MPI_Get_count може бути викликана або після читання повідомлення (функціями MPI_Recv, MPI_Irecv), або після опитування факту надходження повідомлення (функціями MPI_Probe, MPI_Iprobe). Операція читання безповоротно знищує інформацію в буфері прийому. При цьому спроба вважати повідомлення з параметром count менше, ніж число елементів у буфері, приводить до втрати повідомлення. *Визначити параметри отриманого повідомлення* без його читання можна за допомогою *функції MPI_Probe*.

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

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

Підпрограма MPI_Probe виконується з блокуванням, тому завершиться вона лише тоді, коли повідомлення з придатним ідентифікатором і номером процесу-відправника буде доступно для одержання. Атрибути цього повідомлення повертаються в перемінної status. Наступний за MPI_Probe виклик MPI_Recv з тими ж атрибутами повідомлення (номером процесу-відправника, ідентифікатором повідомлення і комунікатором) помістить у буфер прийому саме те повідомлення, наявність якого було опитано підпрограмою MPI_Probe.

ЛАБОРАТОРНА РОБОТА № 5

Тема: Дослідження механізмів колективної взаємодії. Реалізація механізмів блокувань та бар'єрів між паралельними процесами.

Мета: дослідити механізми колективної взаємодії між паралельними обчислювальними процесами у середовищі MPI, навчитися програмувати паралельні задачі з використанням механізмів блокувань та бар'єрів між паралельними процесами.

ЗМІСТ

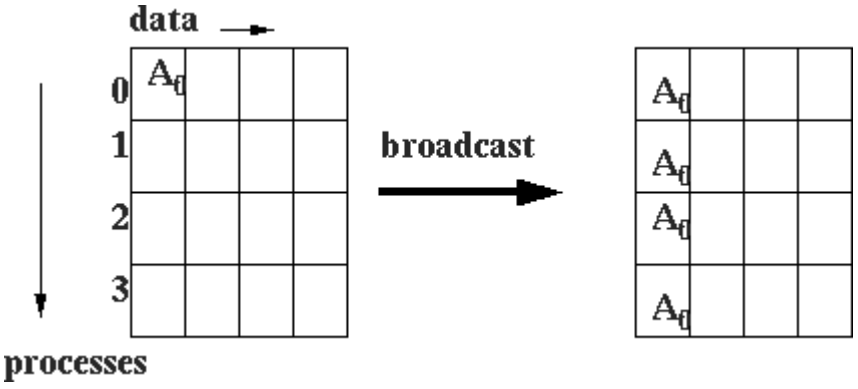
1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Колективні взаємодії процесів

В операціях колективної взаємодії процесів беруть участь всі процеси комунікатора. Відповідна процедура повинна бути викликана кожним процесом, можливо, зі своїм набором параметрів. Повернення із процедури колективної взаємодії може відбутися в той момент, коли участь процесу в даній операції вже закінчено. Як і для процедур, що блокують, повернення означає те, що дозволено вільний доступ до буфера прийому або посилки, але не означає ні того, що операція завершена іншими процесами, ні навіть того, що вона ними почата (якщо це можливо за змістом операції).

int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)

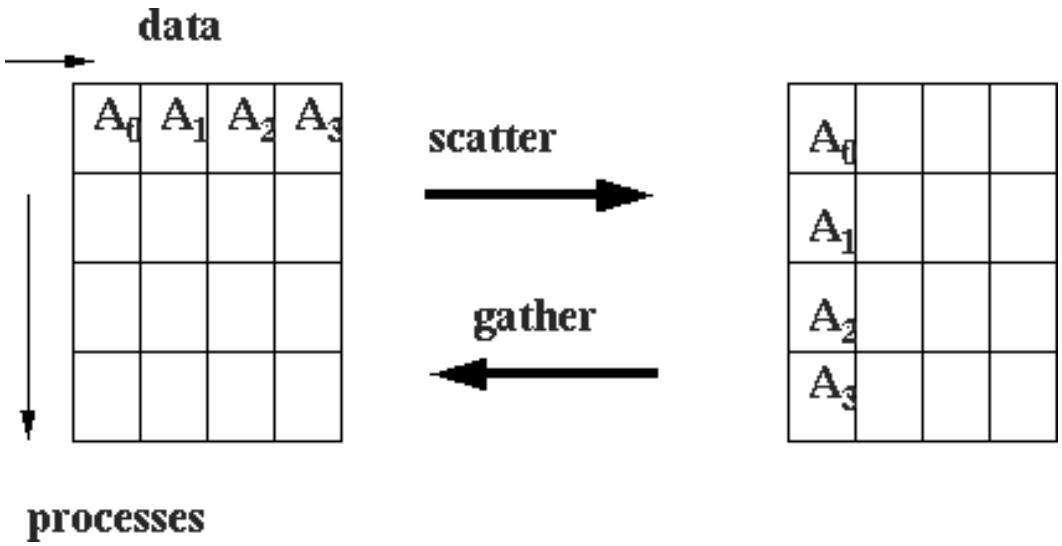
- OUT *buf* – адреса початку буфера посилки повідомлення
- *count* – число переданих елементів у повідомленні
- *datatype* – тип переданих елементів
- *source* – номер процесу, що розсилає
- *comm* – ідентифікатор групи



Розсилання повідомлення від процесу *source* всім процесам, включаючи процес, що розсилає. При поверненні із процедури вміст буфера *buf* процесу *source* буде скопійовано в локальний буфер процесу. Значення параметрів *count*, *datatype* і *source* повинні бути однаковими у всіх процесів.

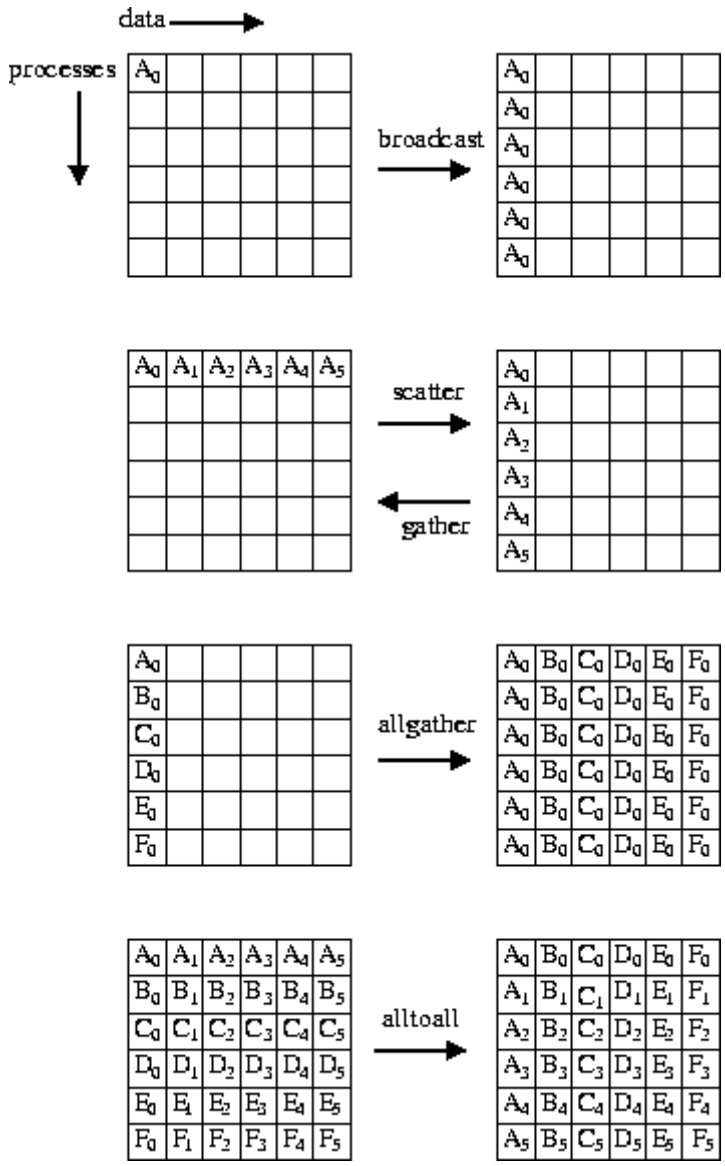
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int dest, MPI_Comm comm)

- *sbuf* – адреса початку буфера посилки;
- *scount* – число елементів що посилаються;
- *stype* – тип елементів повідомлення, що відсилає;
- OUT *rbuf* – адреса початку буфера збору даних;
- *rcount* – число елементів у прийнятому повідомленні;
- *rtype* – тип елементів прийнятого повідомлення;
- *dest* – номер процесу, на якому відбувається збір даних;
- *comm* – ідентифікатор групи;
- OUT *ierror* – код помилки.



Збір даних із всіх процесів у буфері *rbuf* процесу *dest*. Кожний процес, включаючи *dest*, посилає вміст свого буфера *sbuf* процесу *dest*. Збираючий процес зберігає дані в *буфері* rbuf, розташовуючи їх у порядку зростання номерів процесів. Параметр *rbuf* має значення тільки на процесі, що збирає, і на інших ігнорується, значення *параметрів* count, datatype *і* dest повинні бути однаковими у всіх процесів.

MPI_Scatter – аналогічна функція, що розсилає дані з буферу *rbuf* процесу *dest* усім іншим процесам.



int MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- *sbuf* – адреса початку буфера для аргументів
- OUT *rbuf* – адреса початку буфера для результату
- *count* – число аргументів у кожного процесу
- *datatype* – тип аргументів
- *op* – ідентифікатор глобальної операції
- *comm* – ідентифікатор групи

Виконання *count* глобальних операцій *op* з поверненням *count* результатів у всіх процесах у буфері *rbuf*. Операція виконується незалежно над відповідними аргументами всіх процесів. Значення параметрів *count* і *datatype* у всіх процесів повинні бути однаковими. З міркувань ефективності реалізації передбачається, що операція *op* має властивості асоціативності й комунікативності.

int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

- *sbuf* – адреса початку буфера для аргументів

- OUT *rbuf* – адреса початку буфера для результату
- *count* – число аргументів у кожного процесу
- *datatype* – тип аргументів
- *op* – ідентифікатор глобальної операції
- *root* – процес-одержувач результату
- *comm* – ідентифікатор групи

Функція аналогічна попередній, але результат буде записаний у буфер *rbuf* тільки в процесу *root*.

Колективні операції MPI_Op:

MPI_SUM – сума
MPI_MIN – мінімальне
MPI_MAX – максимальне
MPI_PROD – добуток

1.2 Бар’єрна синхронізація процесів

int MPI_Barrier(MPI_Comm comm)

- *comm* – ідентифікатор групи

Блокує роботу процесів, що викликали дану процедуру, доти, поки всі процеси, що залишилися, *групи* comm також не виконають цю процедуру.

Синхронізація за допомогою бар'єрів використовується, наприклад, для завершення всіма процесами деякого етапу рішення завдання, результати якого будуть використовуватися на наступному етапі. Використання бар'єра гарантує, що жоден із процесів не розпочнеться завчасно до виконання наступного етапу, поки результат роботи попереднього не буде остаточно сформований. Неявну синхронізацію процесів виконує будь-яка колективна функція.

1.3 Бібліотека MPE

Бібліотека MPE (Multi-Processing Environment) містить процедури, які полегшують написання, налагодження й оцінку ефективності MPI-Програм.

Одним з найпоширеніших засобів для аналізу характеристик паралельних програм є **реєстрація** (logging), що формує файл траси відзначених у часі подій – логфайл (logfile). Бібліотека MPE створює можливість легко одержати такий файл у кожному процесі й зібрати їх разом по закінченню роботи. Вона також автоматично обробляє неузгодженість і дрейф часу на множині процесорів, якщо система не забезпечує синхронізацію часу. Ця бібліотека призначена для користувача, що бажає створити свої власні події й програмні стани.

Аналіз результатів реєстрації виробляється після виконання обчислень. Засоби реєстрації й аналізу включають ряд профілюючих бібліотек, утилітних програм і ряд графічних засобів.

Бібліотечні ключі забезпечують збори процедур, які створюють логфайли. Ці логфайли можуть бути створені вручну шляхом розміщення в програмі MPI

звертань до MPE, або автоматично при установленні зв'язку з відповідними MPE-Бібліотеками, або комбінацією цих двох методів.

Logging Libraries (бібліотека реєстрації) – самі корисні й широко використовувані профілюючі бібліотеки в MPE. Вони формують базис для генерації логфайлів з користувальницьких програм. Зараз є три різних формати логфайлів, припустимих в MPE. За замовчуванням використовується формат CLOG. Він містить сукупність подій з єдиним відмітчиком часу. Формат ALOG більше не розвивається й підтримується для забезпечення сумісності з ранніми програмами. І найбільш потужним є формат – SLOG (для Scalable Logfile), що може бути конвертований із уже наявного CLOG- файлу або отриманий прямо з виконуваної програми (для цього необхідно встановити змінну середовища MPE_LOG_FORMAT в SLOG).

Набір утилітних програм в MPE включає конвертори форматів (наприклад, clog2slog), друк логфайлів (slog_print), оболонки засобів візуалізації логфайлів, які вибирають коректні графічні засоби для подання логфайлів відповідно до їх розширень.

Результати розрахунку часу дають деяке подання про ефективність програми. Але в більшості випадків потрібно докладно довідатися, яка була послідовність подій, скільки часу був витрачений на кожну стадію обчислення й скільки часу займає кожна окрема операція передачі. Щоб полегшити їхнє сприйняття, потрібно представити їх у графічній формі. Але для цього спочатку потрібно створити файли подій зі зв'язаними тимчасовими оцінками, потім досліджувати їх після закінчення програми й тільки потім інтерпретувати їх графічно на робочій станції. Здатність автоматично генерувати логфайли є важливим компонентом всіх засобів для аналізу ефективності паралельних програм.

Бібліотека для створення логфайлов відділена від бібліотеки обміну повідомленнями MPI. Перегляд логфайлов незалежний від їхнього створення, і тому можуть використовуватися різні інструментальні засоби. Бібліотека для створення логфайлов MPE розроблена таким чином, щоб співіснувати з будь-який MPI-Реалізацією й поширюється поряд з модельною версією MPI.

Щоб створити файл реєстрації, необхідно викликати процедуру **MPE_Log_event**. Крім того, кожний процес повинен викликати процедуру **MPE_Init_log**, щоб приготуватися до реєстрації, і **MPE_Finish_log**, щоб об'єднати файли, що зберігаються локально при кожному процесі в єдиний логфайл. **MPE_Stop_log** використовується, щоб призупинити реєстрацію, хоча таймер продовжує працювати.

MPE_Start_log починає реєстрацію. Програміст вибирає будь-які ненегативні цілі числа, бажані для типів подій; система не надає ніяких приватних значень типам подій. Події розглядаються як такі, що не мають тривалості. Щоб виміряти тривалість стану програми, необхідно, щоб пара подій відзначила початок і закінчення стану. Стан визначається процедурою **MPE_Describe_state**, що описує початок і закінчення типів подій. Процедура **MPE_Describe_state** також додає назву стану і його колір на графічному виводі.

Відповідна процедура **MPE_Describe_event** забезпечує опис події кожного типу. Використовуючи ці процедури, приведемо приклад обчислення числа? Для цього оснастимо програму обчислення числа π відповідними операторами. Важливо,

щоб реєстрація події не створювала більших накладних витрат. **MPE_Log_event** зберігає невелика кількість інформації у швидкій пам'яті. Під час виконання **MPE_Log_event** ці буфери поєднуються паралельно й кінцевий буфер, відсортований по тимчасовим міткам, записується процесом 0.

Після виконання програми **MPI**, що містить процедури MPE для реєстрації подій, директорій, де вона виконувалася, містить файл подій, відсортованих за часом, причому час скоректований з обліком відхилення частоти генераторів. Можна написати багато програм для аналізу цього файлу й подання інформації.

Наприклад, одна з реалізацій MPE містить коротку програму, **states**. Якщо ми виконуємо її з логфайлом, що описали вище, ми одержимо:

Стан: Час:
Broadcast 0.000184
Compute 4.980584
Reduce 0.000248
Sync 0.000095
Сума: 4.981111

Така підсумкова інформація є досить поширеною, але грубою формою профілювання; вона повідомляє тільки, де програма витрачає час. Значно інформативніше графічне подання, забезпечуване спеціалізованими програмами, наприклад, **upshot** і **jampshot**.

Вхідні процедури MPE використовуються, щоб створити логфайли (звіти) про події, які мали місце при виконанні паралельної програми. Ці процедури дозволяють користувачеві включати тільки ті події, які йому цікаві в даній програмі. Базові процедури − **MPE_Init_log**, **MPE_Log_event** і **MPE_Finish_log**.

MPE_Init_log повинна викликатися всіма процесами, щоб ініціювати необхідні структури даних. **MPE_Finish_log** збирає звіти із всіх процесів, поєднує їх, вирівнює по загальній шкалі часу. Потім процес із номером 0 у комунікаторі **MPI_COMM_WORLD** записує звіт у файл, ім'я якого зазначено в аргументі. Одиночна подія встановлюється процедурою **MPE_Log_event**, що задає тип події (вибирає користувач), ціле число й рядок для даних. Щоб розмістити логфайл, що буде потрібний для аналізу або для програми візуалізації (подібної upshot), процедура **MPE_Describe_state** дозволяє додати події й описувані стани, указати крапку старту й закінчення для кожного стану. При бажанні для візуалізації звіту можна використовувати колір. **MPE_Stop_log** і **MPE_Start_log** призначені для того, щоб динамічно включати й виключати створення звіту.

int **MPE_Init_log** (void)
int **MPE_Start_log** (void)
int **MPE_Stop_log** (void)
int **MPE_Finish_log** (char *logfilename)
int **MPE_Describe_state** (int start, int end, char *name, char *color)
int **MPE_Describe_event** (int event, char *name)
int **MPE_Log_event** (int event, int intdata char *chardata)

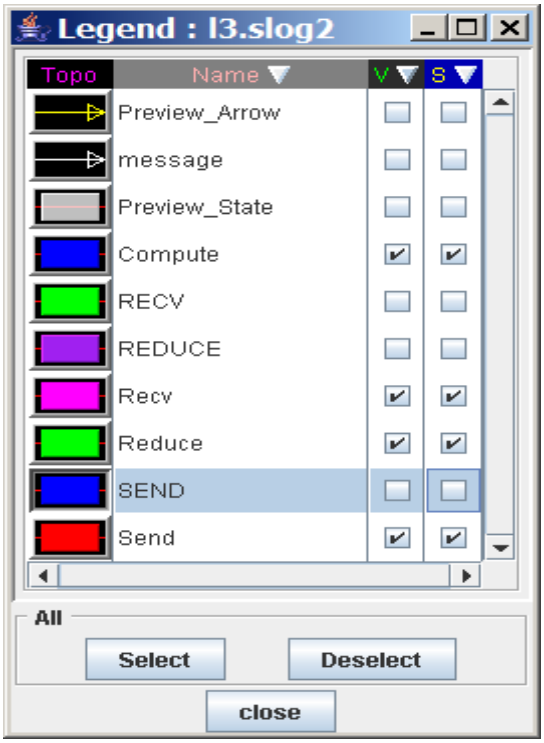
2. ХІД РОБОТИ

2.1 Уважно ознайомтеся із теоретичною частиною лабораторної роботи.

2.2 Розглянемо приклад програми, що містить у собі функцію колективної взаємодії процесів **MPI_Reduce** та дослідимо її. А також використаємо для синхронізації процесів функцію **MPI_Barrier**. **MPI_Reduce** виконує задану кількість глобальних операцій (сумування, мінімум, максимум тощо) з поверненням результатів у всіх процесах у буфері *rbuf*. Операція виконується незалежно над відповідними аргументами всіх процесів. Подана нижче програма демонструє приклад порядкового розподілу двовимірного масиву на вектори, які розсилаються різним процесам для обробки. Кожен процес виконує над своїм вектором певні математичні операції, після чого кінцевий результат (сумування, мінімум, максимум тощо) формується за допомогою колективної функції **MPI_Reduce** та повертається кожному процесу. Ознайомтеся з текстом програми та дослідіть її. Програма також містить функції відлагодження бібліотеки **MPE.**

2.3 Відповідно до Вашого варіанту завдання змініть код програми поданої нижче, відкомпілюйте її з використанням ключів бібліотеки **MPE**. Запустіть програму на виконання, а результат занотуйте у звіт. Помістіть модифіковану програму у звіт по лабораторній роботі як додаток.

2.4 Проаналізуйте створений програмою *логфайл* за допомогою засобу **Jumpshot**. Під час аналізу залишіть відмітки лише на заданих вами подіях, як на малюнку поданому нижче. Результат збережіть у звіт.



Приклад програми колективної взаємодії процесів та їх синхронізації:

```
  #include "mpi.h"
#include "mpe.h"
#include <stdio.h>
#define SIZE1 2
#define SIZE2 8

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i, j;
float a[SIZE1][SIZE2] =
  {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 b[SIZE1];
float res, globalres;
double startwtime = 0.0, endwtime;
int event1a, event1b, event2a, event2b,
event3a, event3b, event4a, event4b,
event5a, event5b;

MPI_Status stat;
MPI_Datatype rowtype;

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

MPI_Type_contiguous(SIZE2, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

/*  Get event ID from MPE, user should NOT assign event ID  */
event1a = MPE_Log_get_event_number();
event1b = MPE_Log_get_event_number();
event2a = MPE_Log_get_event_number();
event2b = MPE_Log_get_event_number();
event3a = MPE_Log_get_event_number();
event3b = MPE_Log_get_event_number();
event4a = MPE_Log_get_event_number();
event4b = MPE_Log_get_event_number();
event5a = MPE_Log_get_event_number();
event5b = MPE_Log_get_event_number();

if (numtasks == SIZE1) {
   /*------- for master process ------------------------------*/
  if (rank == 0) {

   MPE_Describe_state(event1a, event1b, "Send", "red");
   MPE_Describe_state(event2a, event2b, "Compute", "blue");
   MPE_Describe_state(event3a, event3b, "Reduce", "green");
   MPE_Describe_state(event4a, event4b, "Sync", "orange");
   MPE_Describe_state(event5a, event5b, "Recv", "magenta");
```

```
        startwtime = MPI_Wtime();

        /* print matrix on screen */
        printf("\t\t");
        for (i =0 ; i<SIZE2;i++) printf("%d\t",i);
        printf("\n");
        printf("\n");
        for (i = 0; i<SIZE1; i++) {
         printf("%d\t\t",i);
            for (j=0;j<SIZE2;j++) {
             printf("%3.1f\t", a[i][j]);
         }
        printf("\n");
        }

        MPE_Start_log();

        /* send vectors to other processes*/
        MPE_Log_event(event1a, 0, "Start Send");
        for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
        MPE_Log_event(event1b, 0, "Finish Send");
        }

        /*------  for other processes ------------------------------
----*/
        /* receiving vectors and print on screen */
        MPE_Log_event(event5a, 0, "Start Recv");
        MPI_Recv(b,  SIZE2,  MPI_FLOAT,  source,  tag,  MPI_COMM_WORLD,
&stat);
        MPE_Log_event(event5b, 0, "Finish Recv");
        printf("process= %d  ", rank);
        for (i=0; i<SIZE2; i++)
        printf("%3.1f ", b[i]); printf("\n");

        /* synchronization */
        fflush(stdout);
        MPE_Log_event(event4a, 0, "Start Compute");
        MPI_Barrier(MPI_COMM_WORLD);
        MPE_Log_event(event4b, 0, "Finish Compute");

        /* processing the part of matrix - sum of vectors */
        MPE_Log_event(event2a, 0, "Start Process");
        for(i=0;i<SIZE2;i++){
        res=res+b[i];

        MPE_Log_event(event2b, 0, "Finish Process");
        }

        printf("process= %d   result= %3.1f \n", rank, res);

        /* synchronization */
        MPE_Log_event(event4a, 0, "Start Sync");
```

```
        MPI_Barrier(MPI_COMM_WORLD);
        MPE_Log_event(event4b, 0, "Finish Sync");

        /* reduce global result */
        MPE_Log_event(event3a, 0, "Start Redude");
        MPI_Reduce(&res,&globalres,1,                    MPI_FLOAT,
MPI_SUM,0,MPI_COMM_WORLD);
        MPE_Log_event(event3b, 0, "Finish Reduce");

        if (rank==0){

        printf("process= %d   MPI_SUM = %3.1f \n", rank, globalres);
        endwtime = MPI_Wtime();
        printf("wall clock time = %f\n", endwtime-startwtime);
        }

        }
     else
        printf("Must specify %d processors. Terminating.\n",SIZE1);

    MPI_Finalize();
    }
```

Варіанти завдань

Варіант	Матриця a[SIZE1xSIZE2]	Кількість елементів в матриці	Тип елементів в матриці	Розмірність матриці SIZE1xSIZE2	Кількість паралельних процесів	Значення змінної tag	Операція над вектором b[i]	Колективна операція функції Reduce
1	4.3, 3.1, 9.4, 5.9, 5.7, 12.5, 12.3, 6.4, 8.1, 12.0	10	float	2x5	2	1	Пошук максимального	MPI_SUM
2	10, 11, 8, 9, 5, 9, 7, 12, 11, 5, 4, 2	12	int	4x3	4	2	Добуток вектора	MPI_MIN
3	9.7, 1.8, 10.4, 2.8, 7.8, 8.5, 9.6, 8.5, 3.2, 10.8, 10.0, 10.6, 5.4, 4.1, 3.5, 2.8	16	float	4x4	4	3	Сума вектора	MPI_MAX
4	5, 9, 7, 1, 9, 9, 7, 4, 7, 10, 8, 8, 2, 5, 6, 1, 7, 3	18	int	2x9	2	4	Пошук мінімального	MPI_PROD
5	6, 9, 2, 8, 4, 5, 9, 6, 3, 4, 4, 10, 7, 8, 6, 6, 1, 5, 5, 9	20	int	2x10	2	5	Пошук максимального	MPI_SUM
6	8, 9, 10, 7, 6, 8, 6, 8, 6, 6, 10, 9, 7, 10, 9, 3, 1, 8, 9, 6, 3, 6, 5, 1	24	int	4x6	4	6	Сума вектора	MPI_MIN
7	6, 5, 2, 2, 6, 4, 10, 4, 2, 7, 2, 2, 11, 6, 5, 3, 4, 7, 6, 5, 9, 11, 6, 6, 7, 11, 4, 9	28	int	2x14	2	7	Добуток вектора	MPI_MAX
8	2, 4, 9, 6, 4, 9, 2, 7, 3, 10, 7, 5, 7, 2, 11, 6, 7, 2, 3, 8, 3, 8, 8, 11, 6, 3, 6, 6, 8, 7, 4, 7	32	int	4x8	4	8	Пошук мінімального	MPI_PROD
9	2.3, 4.0, 1.1, 10.9, 3.3, 2.0, 6.9, 10.7, 8.3, 9.5	10	float	2x5	2	9	Пошук максимального	MPI_SUM
10	10, 5, 8, 7, 1, 4, 8, 5, 8, 6, 7, 4	12	int	4x3	4	10	Сума вектора	MPI_MIN
11	10.0, 6.3, 4.0, 7.5, 8.7, 8.8, 9.7, 3.3, 1.4, 9.8, 10.2, 2.5, 6.0, 5.9, 1.0, 8.5	16	float	2x8	2	11	Добуток вектора	MPI_MAX
12	4, 2, 5, 8, 10, 5, 11, 2, 2, 1, 10, 4, 3, 10, 8, 5, 7, 11	18	int	2x9	2	12	Пошук мінімального	MPI_PROD

Варіант	Матриця a[SIZE1xSIZE2]	Кількість елементів в матриці	Тип елементів в матриці	Розмірність матриці SIZE1xSIZE2	Кількість паралельних процесів	Значення змінної tag	Операція над вектором b[i]	Колективна операція функції Reduce
13	10, 2, 5, 1, 6, 2, 9, 1, 8, 11, 6, 9, 9, 3, 4, 9, 3, 10, 6, 3	20	int	2x10	2	13	Добуток вектора	MPI_SUM
14	11, 9, 10, 10, 10, 4, 4, 9, 10, 1, 5, 8, 4, 9, 3, 2, 2, 3, 4, 3, 7, 3, 11, 11	24	int	4x6	4	14	Пошук максимального	MPI_MIN
15	4.0, 3.1, 1.2, 6.8, 10.2, 7.1, 10.9, 8.8, 9.2, 7.0, 6.5, 5.6, 1.6, 10.4, 9.5, 10.6, 10.4, 1.9, 4.3, 10.4, 1.4, 7.0, 6.6, 8.8, 2.5, 8.7, 3.0, 9.2	28	float	2x14	2	15	Сума вектора	MPI_MAX
16	6, 10, 2, 8, 9, 8, 3, 4, 10, 5, 1, 8, 4, 11, 4, 8, 1, 2, 9, 1, 5, 10, 10, 7, 4, 8, 4, 10, 7, 5, 7, 2	32	int	8x4	8	16	Пошук мінімального	MPI_PROD
17	9, 3, 8, 4, 1, 4, 8, 4, 4, 10	10	int	2x5	2	17	Пошук максимального	MPI_SUM
18	7.6, 5.4, 8.8, 8.0, 8.1, 8.4, 8.3, 3.4, 3.8, 5.9, 8.0, 4.2	12	float	4x3	4	18	Сума вектора	MPI_MIN
19	7, 11, 3, 2, 2, 6, 10, 5, 9, 3, 8, 2, 7, 4, 2, 10	16	int	4x4	4	19	Добуток вектора	MPI_MAX
20	4, 1, 8, 4, 8, 9, 3, 4, 8, 3, 9, 11, 10, 4, 4, 9, 10, 4	18	int	2x9	2	20	Пошук мінімального	MPI_PROD
21	3.7, 10.1, 5.6, 1.6, 6.5, 2.6, 2.3, 11.0, 10.4, 5.7, 8.0, 9.1, 4.7, 8.0, 7.1, 3.2, 2.8, 6.6, 4.7, 9.6	20	float	2x10	2	21	Пошук максимального	MPI_SUM
22	11, 4, 10, 11, 1, 9, 5, 2, 3, 7, 11, 3, 11, 1, 10, 10, 10, 11, 11, 2, 3, 9, 10, 7	24	int	4x6	4	22	Добуток вектора	MPI_MIN
23	5, 3, 11, 4, 9, 11, 8, 10, 3, 4, 9, 11, 10, 2, 10, 3, 6, 3, 2, 7, 2, 1, 3, 2, 7, 6, 4, 10	28	int	2x14	2	23	Сума вектора	MPI_MAX
24	10.3, 7.3, 2.1, 5.6, 5.4, 4.5, 7.4, 7.6, 1.9, 4.4, 10.0, 3.5, 10.1, 5.3, 7.9, 4.7, 2.9, 9.2, 7.3, 6.0, 2.4, 10.7, 9.9, 11.0, 2.2, 1.5, 7.9, 6.9, 5.3, 5.2, 1.0, 11.0	32	float	8x4	8	24	Пошук мінімального	MPI_PROD
25	3, 9, 3, 7, 9, 7, 3, 4, 2, 7	10	int	2x5	2	25	Пошук максимального	MPI_SUM
26	5, 11, 9, 11, 6, 10, 10, 8, 9, 6, 11, 2	12	int	4x3	4	26	Добуток вектора	MPI_MIN
27	4, 7, 8, 6, 4, 3, 3, 3, 3, 3, 3, 10, 3, 10, 1, 8	16	int	2x8	2	27	Сума вектора	MPI_MAX
28	10.4, 8.8, 9.1, 1.5, 8.1, 5.8, 9.7, 3.9, 1.7, 8.4, 6.7, 1.2, 2.0, 5.0, 6.7, 8.8, 2.5, 5.3	18	float	2x9	2	28	Пошук мінімального	MPI_PROD
29	3, 2, 5, 5, 3, 6, 10, 9, 2, 5, 4, 2, 11, 6, 9, 5, 5, 11, 3, 4	20	int	2x10	2	29	Пошук максимального	MPI_SUM
30	7, 6, 8, 1, 8, 7, 2, 10, 4, 9, 9, 6, 9, 6, 5, 10, 4, 2, 4, 2, 9, 3, 8, 10	24	int	6x4	6	30	Сума вектора	MPI_MIN
31	9, 5, 2, 2, 7, 8, 6, 7, 9, 10, 1, 6, 4, 10, 6, 2, 8, 1, 5, 6, 1, 10, 3, 6, 10, 8, 6, 4	28	int	2x14	2	31	Добуток вектора	MPI_MAX
32	10, 10, 8, 1, 6, 2, 4, 7, 10, 6, 7, 1, 3, 6, 8, 8, 5, 6, 5, 9, 2, 4, 7, 9, 5, 8, 1, 5, 2, 7, 8, 7	32	int	4x8	4	32	Пошук мінімального	MPI_PROD

3. ЗМІСТ ЗВІТУ

3.1 Тема, мета лабораторної роботи
3.2 Короткі теоретичні відомості
3.3 Опис виконання лабораторної роботи згідно пунктів 2.2-2.4
3.4 Відповідь на одне контрольне запитання згідно номера варіанту (порядковий номер у списку журналу групи)
3.5 Висновки

4. КОНТРОЛЬНІ ПИТАННЯ

4.1 Які ви знаєте операції колективної взаємодії процесів?
4.2 Призначення функції ***MPI_Bcast?***
4.3 Що таке MPI?
4.4 Опишіть параметри функції ***MPI_Bcast***
4.5 Для чого призначена бібліотека MPI?
4.6 Опишіть функцію ***MPI_Gather та MPI_Scatter,*** яка між ними різниця.
4.7 Опишіть параметри функції ***MPI_Reduce?***
4.8 Для яких мов програмування реалізована бібліотека MPI?
4.9 Які ви знаєте колективні операції MPI_Op?
4.10 Що таке бібліотека MPE і для чого вона призначена?
4.11 Бар'єрна синхронізація процесів у MPI.
4.12 Перерахуйте категорії MPE процедур?
4.13 Що забезпечують процедури Паралельної графіки?
4.14 Які особливості процедур реєстрації **(**Logging**)**?
4.15 Назвіть три профілюючі бібліотеки MPE.
4.16 Які існують формати логфайлів?
4.17 Як можна проаналізувати логфайл?
4.18 Для чого призначена програма **states**?
4.19 Які Ви знаєте програми для графічного подання інформації профілювання?
4.20 Яке призначення команди **top**?
4.21 Які Ви знаєте програми для віддаленого доступу до Linux-систем?
4.22 Наведіть приклад компілювання MPI програм з бібліотекою MPE.
4.23 Наведіть приклад запуску на виконання MPI програм.
4.24 Які ключі компілювання вказують на використання бібліотек MPE?

5. СПИСОК ЛІТЕРАТУРИ

5.1 Камерон Хьюз, Трейси Хьюз. Параллельное и распределенное программирование с использованием C++. : Пер. с англ. – М. : Издательский дом "Вильямс", 2004. – 627 с.

5.2 Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 512 с.

5.3 Воеводин В.В. Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. – 608 с.

5.4 А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону. Издательство ООО «ЦВВР», 2003, 208 с.

5.5 Джин Бэкон, Тим Харрис. Операционные системы. Параллельные и распределенные системы. – Питер: Издательская группа BHV, 2004. – 800 с.

5.6 www.mpi-forum.org

5.7 www.parallel.ru

ЛАБОРАТОРНА РОБОТА № 6

Тема: Розпаралелення програми за допомогою технології OpenMP

Мета: Ознайомлення з можливостями розпаралелення послідовних програм засобами OpenMP, використовуючи Visual C++.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ

Введення

Одним зі стандартів для програмування систем з загальною пам'яттю є інтерфейс (API) OpenMP (Open Multi-Processing). Цей стандарт реалізовано для мов програмування C, C++ та Fortran на великій кількості комп'ютерних архітектур, включаючи платформи Unix та Microsoft Windows. OpenMP API складається з набору директив компілятора pragma (прагм), функцій та змінних середовища, що впливають на поведінку паралельної програми під час її виконання. Коли прагми OpenMP використовуються в програмі, вони дають вказівки компілятору створити виконуваний модуль, який буде виконуватись паралельно з використанням декількох потоків.

В OpenMP застосовується модель паралельного виконання, що отримала назву «розгалуження-злиття». Така програма починається як один потік виконання, який називають начальним потоком. Коли потік зустрічає паралельну конструкцію, він створює нову групу потоків, що складається з цього потоку та позитивного числа допоміжних потоків, і стає головним в новій групі. Всі члени нової групи виконують код в межах паралельної конструкції. В кінці такої конструкції є неявний бар'єр. Після її виконання код програми продовжує лише головний потік.

Прагми OpenMP

Для активації підтримки прагм OpenMP в компіляторі необхідно застосовувати додаткові параметри або флаги компіляції. Для Visual C++ таким параметром є /openmp, для gcc – fopenmp. Також у Visual C++ увімкнути підтримку OpenMP можна за допомогою відповідного параметру в діалозі налаштування проекту: вибра-ти Configuration Properties, C/C++, Language і змінити значення параметру «OpenMP Support» на yes. Для використання функцій OpenMP необхідно підключити до проекту файл vcomp.lib (або vcompd.lib в режимі відлагодження).

Прагми OpenMP починаються зі слів #pragma omp і мають наступний формат:

```
#pragma omp <директива> [список параметрів]
```

OpenMP підтримує наступні директиви: atomic, barrier, critical, flush, for, master, ordered, parallel, parallel for, section, sections та single, що визначають або механізми розділення коду, або конструції синхронізації. Необов'язкові параметри директиви уточнюють її поведінку.

Найбільш уживана директива – parallel, яка має наступний синтаксис:

```
#pragma omp parallel [список параметрів]
```

Структурований блок

Вона інформує компілятор, що структурований блок (складений оператор) має виконуватись паралельно в кількох потоках. Як правило, кількість потоків дорівнює кількості процесорів в системі. В якості прикладу розглянемо варіант класичної програми «Hello, world!».

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

int main ()
{
#pragma omp parallel
{
printf (" Hello , OpenMP!\n") ;
} / * #pragma omp parallel * /
return 0;
} / * int main () * /
```

На двопроцесорній (або двоядерній) системі результат роботи цієї програми може бути наступним:

```
Hello, OpenMP!
Hello, OpenMP!
```

Функції та змінні середовища OpenMP

OpenMP пердбачає також набір функцій, що дозволяють:

• під час виконання програми отримувати та встановлювати різноманітні параметри, що визначають її поведінку, наприклад, кількість потоків, можливість вкладеного паралелізму.

• застосовувати сінхронізацію на основі замків (locks).

Прототипи функцій знаходяться в файлі omp.h. Розглянемо набір функцій OpenMP, що використовуються найчастіше.

Функція omp_get_thread_num() повертає номер потоку, в якому була викликана. Головний потік паралельного блоку має номер 0. Функція має наступний прототип:

```
int omp_get_thread_num( void )
```

Функція omp_set_num_threads() має наступний прототип:

```
void omp_set_num_threads( int num_threads)
```

і використовується для встановлення кількості потоків, що будуть виконуватись в на-ступному паралельному блоці.Для визначення поточної кількості паралельних потоків застосовується функція omp_get_num_threads() з наступним прототипом:

```
int omp_get_num_threads( void )
```

В лістінгу нижче наведено текст програми, що демонструє використання розглянутих функцій

```
# include <stdio .h>
# include <omp.h>
int main ()
{
omp_set_num_threads (5) ;
#pragma omp parallel
{
printf ( " Hello , OpenMP!( thread num=%d)\n" ,
omp_get_thread_num () ) ;
} / * #pragma omp parallel * /
return 0;
} / * int main ( ) * /
```

В стандарті OpenMP визначено ряд змінних середовища операційної системи, які контролюють поведінку OpenMP-програм. Зокрема, змінна OMP_NUM_THREADS визначає максимальну кількість потоків, що будуть виконуватись в паралельній про-грамі. Наприклад, в командному рядку Windows необхідно виконати наступну команду:

```
set OMP_NUM_THREADS=4
```

Таким чином, за допомогою функції omp_set_num_threads() або змінної середови-ща OMP_NUM_THREADS можна встановити довільну кількість потоків, що будуть створюватись під час виконання OpenMP-програм.

Директива for

Однією з директив, що застосовується досить часто, є директива for, яка належить до директив розподілення роботи (work-sharing directive). Ця директива інформує ком-пілятор, що при виконанні циклу for в паралельному блоці ітерації циклу мають бути розподілені між потоками групи.

```
#include <stdio.h>
#include <omp.h>
int main ()
{
int i ;
#pragma omp parallel
{
#pragma omp for
for ( i = 0; i < 5; i++)
printf ( " thread N%d i=%d\n" , omp_get_thread_num () , i ) ;
} /* #pragma omp parallel */
return 0;
}
```

Результат роботи програми наведено нижче:

```
thread N1 i=3
thread N1 i=4
thread N0 i=0
thread N0 i=1
thread N0 i=2
```

Порядок появи рядків на екрані може бути довільним і змінюватись при кожному запуску програми. Якщо з тексту програми вилучити директиву #pragma omp for, то кожний потік буде виконувати повний цикл for.

У зв'язку з тим, що розпаралелювання коду найчастіше виконується саме в циклічних конструкціях, OpenMP має скорочений варіант запису комбінації директив

```
#pragma omp parallel i #pragma omp for:
#pragma omp parallel for
for(i = 0; i < 5; ++i)
...
```

Слід підкреслити, що розпаралелюватись можуть лише такі циклі, в яких ітерації не мають залежностей одна від одної. Якщо цикл не має залежностей, компілятор має змогу виконати цикл в любому порядку, навіть паралельно. Також цикли, які OpenMP може розпаралелювати, мають відповідати наступному формату:

```
for(цілий тип i = інваріант циклу1;
i {<,>,=,!=,<=,>=} інваріант циклу2;
```
i {+,-}= інваріант циклу3)

Ці вимоги введені для того, щоб OpenMP міг визначити кількість ітерацій циклу.

При розробці паралельних програм необхідно враховувати, які змінні є спільними (shared), а які приватними (private). Спільні змінні доступні всім потокам в групі, тому зміна значеннь таких змінних в одному потоці стає видимою в інших потоках. Що стосується приватних зміних, то кожний потік має окремі їх копії, тому їх зміна в одному потоці не відображається на інших потоках.

За замовчуванням всі змінні в паралельному блоці є спільними. Лише в трьох випадках змінні є приватними. По-перше, приватними є індекси паралельних циклів. По-друге, приватними є локальні змінні паралельних блоків. По-третє, приватними стають змінні, що вказані в параметрах private, firstprivate, lastprivate и reduction прагми for.

Параметр private вказує, що для кожного потоку має бути створена локальна копія кожної змінної, що вказана в списку. Приватні копії будут ініціалізуватись значенням за замовчанням, при можливості буде застосовуватись конструктор за замовчанням.

Параметр firstprivate виконує аналогічні дії, однак перед виконанням паралельного блоку він копіює значення приватної змінної в кожний потік, застосовуючи при необхідності конструктор копій.

Параметр lastprivate теж подібний до параметра private, однак після виконання останньої ітерації циклу або конструкції паралельного блоку значення змінних, що вказані в списку цього параметра, привласнюються змінним основного потоку. При цьому може застосовуватись оператор привласнювання копій.

Параметр reduction в якості аргументів приймає змінну й оператор. Оператори, що підтримуються reduction, наведені в таблиці нижче, а змінна має бути скалярною

(наприклад, int або float). Змінна параметру reduction ініціалізується значенням, що вказане в таблиці. Після завершення паралельного блоку вказаний оператор застосовується до кожної приватної копії і початкового значенні змінної.

Оператор reduction	Початкове значення змінної
+	0
*	1
-	0
&	~0 (усі біти в змінній встановлені)
|	0
^	0
&&	0
||	0

В лістінгу наведено приклад програми, що використовує розглянуті параметри.

```
#include <stdio.h>
#include <omp.h>
int main ()
{
int i , j ;
int array [] = {1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9};
int sum;
#pragma omp parallel
{
#pragma omp for firstprivate ( j ) \
lastprivate ( i ) \
reduction (+: sum)
for ( i = 0; i<sizeof ( array )/ sizeof ( array [0]) ; i++)
{
long private = i ;
sum += array [ i ];
} / * for ( i = 0 ; i < 5 ; i ++) * /
} / * #pragma omp parallel * /
printf ( "%d\n" , sum) ;
return 0;
}
```

В цьому лістінгу змінна i є приватною тому, що вона є змінною циклу for, а також тому, що вказана в параметрі lastprivate. Змінна j примусово зроблена приватною за допомогою firstprivate. Змінна private також є приватною через те, що вона оголошена в паралельному блоці. В кожному потоці екземпляр приватної змінної sum неявно ініціалізується нулем.

2 ХІД РОБОТИ

1. Ознайомитися з теоретичними відомостями.
2. За допомогою програми “Img2Arr” перетворити файл зображення згідно варіанту в матрицю.

3. Відкрити файл матриці “output.arr” за допомогою “Блокноту”, знищити останню пусту стрічку і табуляцію, переконатися, що кількість рядків матриці є кратною 2, 4 і 8.

4. Відкрити проект “ArrManip” у середовищі Visual Studio (2005 і вище), завантаживши файл “ArrManip.sln”. Знайти у файлі “ArrManipDlg.cpp” функцію void CArrManipDlg::GetArrResult(int nThrs) і запрограмувати операцію над матрицею згідно варіанту. Наприклад, наступний фрагмент коду дозволяє обчислити суму елементів матриці:

```
#pragma omp parallel
{
#pragma omp for reduction(+: res)
for(r=0; r<rows; r++)
for(c=0; c<cols; c++)
res = res + pArr[r][c];
}

ArrRes = res;
```

5. Відкомпілювати проект (клавіша F7) і, запустити на виконання файл “ArrManip.exe”.

6. Виконати операцію над матрицею використовуючи 1, 2, 4 і 8 потоків по три рази для кожної конфігурації та відобразити за допомогою діаграми залежність середнього часу виконання від кількості потоків.

3 ЗМІСТ ЗВІТУ

1. Короткі теоретичні відомості по лабораторній роботі.
2. Зображення згідно варіанту.
3. Лістинг зміненої згідно варіанту функції GetArrResult.
4. Копія екрану з результатами виконання програми для 1, 2, 4 і 8 потоків (без виводу повідомлень для кожного потоку).
5. Діаграма залежності середнього часу виконання від кількості потоків.
6. Висновки.

4 КОНТРОЛЬНІ ЗАПИТАННЯ

1. Для програмування яких обчислювальних систем використовується стандарт OpenMP?
2. Складові частини стандарту OpenMP
3. Прагми OpenMP
4. Функції та змінні середовища OpenMP
5. Директива for
6. Спільні і приватні змінні в паралельних блоках
7. Параметр private
8. Параметр firstprivate
9. Параметр lastprivate
10. Параметр reduction

5 ВИКОРИСТАНА ЛІТЕРАТУРА

[1] Openmp: Simple, portable, scalable smp programming. http://www.openmp.org/drupal/.

[2] Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования.: Пер. с англ. – М.: Издательский дом “Вильямс”, 2003. – 512 с.

[3] Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 2002. – 400 с.

6 ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

№ варіанту	Файл зображення	Операція з матрицею
1.	11.jpg	Добуток елементів, що >0 і <5
2.	6.jpg	Кількість елементів, більших за 100
3.	29.jpg	Кількість елементів, кратних 10
4.	25.jpg	Кількість елементів, кратних 5
5.	12.jpg	Кількість елементів, що >10 і <100
6.	23.jpg	Кількість непарних елементів
7.	3.jpg	Кількість нулів
8.	30.jpg	Кількість парних елементів
9.	5.jpg	Кількість рядків, в яких є 0
10.	20.jpg	Кількість рядків, сума яких <1000
11.	2.jpg	Максимальне значення
12.	9.jpg	Максимальне значення діагоналі
13.	8.jpg	Максимальне значення непарн.рядків
14.	18.jpg	Максимальне значення парн.рядків
15.	13.jpg	Мінімальне значення
16.	15.jpg	Мінімальне значення діагоналі
17.	27.jpg	Середнє значення
18.	7.jpg	Сума діагональних елементів
19.	19.jpg	Сума елементів
20.	1.jpg	Сума елементів, більших за 100
21.	10.jpg	Сума елементів, кратних 10
22.	28.jpg	Сума елементів, менших за 10
23.	26.jpg	Сума елементів, поділених на 10
24.	22.jpg	Сума елементів, що >10 і <100
25.	17.jpg	Сума непарних елементів
26.	24.jpg	Сума непарних рядків
27.	4.jpg	Сума непарних стовпчиків
28.	21.jpg	Сума парних елементів
29.	14.jpg	Сума парних рядків
30.	16.jpg	Сума парних стовпчиків

ЛАБОРАТОРНА РОБОТА № 7

Тема: Розпаралелення секцій програми за допомогою технології OpenMP

Мета: Розпаралелення блоків і циклів послідовних програм засобами OpenMP, використовуючи Visual C++.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ

Директива sections

Як правило, OpenMP застосовується для розпаралелювання циклів. Однак, в OpenMP є також засоби для підтримки паралелізму на рівні функцій. Цей механізм називається секціями OpenMP (OpenMP sections). Для створення паралельного блоку секцій застосовується директива #pragma omp sections, або, аналогічно директиві #pragma omp parallel for, її скорочений варіант #pragma omp parallel sections . Секції в блоці створюються директивой #pragma omp section . Кожній секції ставиться у відповідність один потік, і всі секції виконуються паралельно.

Фрагмент програми, що демонструє розпаралелювання на рівні функцій наведено нижче.

```
void quickSort ( int L, int R)
{
int i ;
if (R > L)
{
i = partition (L, R);
#pragma omp parallel sections
{
#pragma omp section
quickSort (L, i – 1);

#pragma omp section
quickSort ( i + 1 , R);
} /* #pragma omp parallel sections * /
} /* if (R > l ) * /
} /* void quickSort ( int l , int r ) * /
```

2 ХІД РОБОТИ

7. Ознайомитися з теоретичними відомостями.
8. Створити проект консольного застосування з назвою “OMPPi” і скопіювати в нього програму з додатку А. Дана програма призначена для знаходження числа Пі.
9. Відкомпілювати проект, виконати і знайти середній час виконання.
10. Розпаралелити послідовну програму за допомогою OpenMP:

– Включити для даного проекту підтримку OpenMP: меню Project\OMPPi Properties, вкладка Configuration Properties\C/C++\Language, для властивості OpenMP support вказати значення Yes (/openmp).

– У вікні файлу OMPPi.cpp підключити відповідну бібліотеку: #include "omp.h".

– У текст програми вставити необхідні прагми OpenMP.

11. Відкомпілювати проект, виконати і знайти середній час виконання.

12. Створити проект консольного застосування з назвою “OMPSections” і скопіювати в нього програму з додатку Б. Дана програма здійснює дві секції обчислень над масивами.

13. Виконати кроки 3-5 для проекту “OMPSections”. При розпаралеленні даної програми необхідно, щоб випадкова ініціалізація масивів відбувалася паралельно, обидві секції обчислень (Section 1, Section 2) теж виконувалися одночасно. При оцінці часу виконання послідовної і розпаралеленої версій програми доцільно відключати вивід наступних повідомлень:

```
printf("Thread %d: C[%d]= %f\n", tid, i, C[i]);
printf("Thread %d: D[%d]= %f\n", tid, i, D[i]);
```

14. Результати виконання лабораторної роботи оформити у вигляді звіту.

3 ЗМІСТ ЗВІТУ

7. Короткі теоретичні відомості по лабораторній роботі.
8. Лістинги розпаралелених програм OMPPi та OMPSections.
9. Копії екрану з результатами виконання послідовних і паралельних версій програм OMPPi та OMPSections.
10. Діаграми або таблиці з порівнянням швидкодії послідовних і паралельних версій програм OMPPi та OMPSections.
11. Висновки.

4 КОНТРОЛЬНІ ЗАПИТАННЯ

11. Якими директивами виконується розпаралелювання циклів в OpenMP?
12. Які змінні в паралельних блоках є спільними, а які — приватними?
13. Яким чином діє параметр private?
14. Яким чином діє параметр firstprivate?
15. Яким чином діє параметр lastprivate?
16. Яким чином діє параметр reduction?
17. Які оператори можуть застосовуватись з параметром reduction?
18. Яким чином виконується розпаралелювання коду на рівні функцій?

5 ВИКОРИСТАНА ЛІТЕРАТУРА

[1] Openmp: Simple, portable, scalable smp programming. http://www.openmp.org/drupal/.

[2] Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования.: Пер. с англ. – М.: Издательский дом “Вильямс”, 2003. – 512 с.

[3] Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 2002. – 400 с.

6 ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

№ варіанту	ОМРРі
1.	num_steps = 1000000000 кількість потоків = 2
2.	num_steps = 500000000 кількість потоків задається динамічно
3.	num_steps = 1000000000 кількість потоків = 4
4.	num_steps = 100000000 кількість потоків задається динамічно
5.	num_steps = 1000000000 кількість потоків задається динамічно
6.	num_steps = 500000000 кількість потоків = 2
7.	num_steps = 1000000000 кількість потоків задається динамічно
8.	num_steps = 100000000 кількість потоків задається динамічно
9.	num_steps = 1000000000 кількість потоків = 8
10.	num_steps = 500000000 кількість потоків задається динамічно
11.	num_steps = 1000000000 кількість потоків задається динамічно
12.	num_steps = 100000000 кількість потоків задається динамічно
13.	num_steps = 1000000000 кількість потоків = 2
14.	num_steps = 500000000 кількість потоків задається динамічно
15.	num_steps = 1000000000
16.	num_steps = 100000000 кількість потоків задається динамічно
17.	num_steps = 1000000000
18.	num_steps = 500000000 кількість потоків = 4
19.	num_steps = 1000000000 кількість потоків задається динамічно
20.	num_steps = 100000000 кількість потоків задається динамічно
21.	num_steps = 1000000000 кількість потоків задається динамічно
22.	num_steps = 500000000 кількість потоків задається динамічно
23.	num_steps = 1000000000 кількість потоків задається динамічно
24.	num_steps = 100000000 кількість потоків задається динамічно
25.	num_steps = 1000000000 кількість потоків = 3
26.	num_steps = 500000000 кількість потоків задається динамічно
27.	num_steps = 1000000000 кількість потоків задається динамічно
28.	num_steps = 100000000 кількість потоків = 2
29.	num_steps = 1000000000 кількість потоків задається динамічно
30.	num_steps = 500000000 кількість потоків задається динамічно

Додаток А
Програма для розрахунку числа Пі

```
// OMPPi.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#include "conio.h"

int main()
{
 long long num_steps = 1000000000;
 double step;
 clock_t start, stop;
 double x, pi, sum=0.0;
 int i;

 step = 1./(double)num_steps;
 start = clock();

 for (i=0; i<num_steps; i++)
 {
      x = (i + .5)*step;
      sum = sum + 4.0/(1.+ x*x);
 }

 pi = sum*step;
 stop = clock();

 printf("The value of PI is %15.12lf\n", pi);
 printf("The   time   to   calculate   PI   was   %lf   seconds\n",
((double)(stop - start)/1000.0));

 printf("Press any key...");
 _getch();

 return 0;
}
```

Додаток Б
Програма з обчисленнями над масивом

```
// OMPSections.cpp : Defines the entry point for the console application.
//

    #include "stdafx.h"
    #include <stdio.h>
    #include <stdlib.h>
    #include "time.h"
    #include "conio.h"
    #include "math.h"

    int main ()
    {
     const int N = 30000;

     int i, nthreads, tid;
     float A[N], B[N], C[N], D[N];
     clock_t start, stop;

     srand(time(0));

     // Random initializations for arrays
     for (i=0; i<N; i++)
     {
          A[i] = (float) rand() / RAND_MAX;
          B[i] = (float) rand() / RAND_MAX;
          C[i] = (float) rand() / RAND_MAX;
     }

     start = clock();

     tid = omp_get_thread_num();
     if (tid == 0)
     {
          nthreads = omp_get_num_threads();
          printf("Number of threads = %d\n", nthreads);
     }
     printf("Thread %d starting...\n", tid);


     // Section 1

     printf("Thread %d doing section 1\n", tid);
     for (i=0; i<N; i++)
     {
          C[i] = cos(A[i]) + sin(B[i]) - sqrt(fabs(A[i] + B[i]));
          printf("Thread %d: C[%d]= %f\n", tid, i, C[i]);
     }

     // Section 2
```

```
    printf("Thread %d doing section 2\n", tid);
    for (i=0; i<N; i++)
    {
        D[i] = sqrt(fabs(A[i] * B[i])) - sin(A[i]*A[i]) / cos (B[i]);
        printf("Thread %d: D[%d]= %f\n", tid, i, D[i]);
    }


    stop = clock();

    printf("Thread %d done.\n", tid);


    printf("The computing time is %.2lf ms\n\n", ((double)(stop -
start)));

    printf("Press any key...");
    _getch();
   }
```

ЛАБОРАТОРНА РОБОТА № 8

Тема: Ознайомлення з бібліотеками паралельного програмування MPI та MPE в середовищі Linux

Мета: навчитися віддалено працювати на багатопроцесорній ЕОМ під керуванням ОС Linux, виконувати на ній паралельні задачі реалізовані за допомогою MPI, дослідити процес виконання паралельних програм використовуючи засіб MPE.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 SIMD- і MIMD- системи як апаратна основа паралельних середовищ

Класифікація ЕОМ, запропонована Флінном, розподіляє світ комп'ютерів на чотири групи:

- **SISD** (Single Instruction Single Data) (Одна Інструкція Одні Дані) – класичні одно-процесорні ЕОМ
- **SIMD** (Single Instruction Multiple Data) (Одна Інструкція Багато Даних) – векторні й матричні ЕОМ
- **MIMD** (Multiple Instruction Multiple Data) (Багато Інструкцій Багато Даних) – багатопроцесорні ЕОМ
- **MISD** (Multiple Instruction Single Data) (Багато Інструкцій Одні Дані) – конвеєрні ЕОМ

SISD- системи являють собою класичні ЕОМ за принципом фон-Неймана: вони виконують операцію, що подана на шину команд, над операндами, що знаходяться на шинах даних.

MISD- системи виконують кілька операцій над даними: наприклад, скласти два числа, зсунути результат на один розряд праворуч та помножити на третє число. Такі системи дозволяють отримати деякий виграш у виконанні, але є досить спеціалізованими системами та не отримали значного поширення. Деякі підходи до таких систем використовуються при конвеєризації SISD- структур.

SIMD- системи мають іншу структуру: вони складаються з тисяч процесорних елементів, що можуть виконати деяку спільну інструкцію над даними, що в них містяться. Дуже часто виникають ситуації, коли треба виконати якусь операцію над багатьма даними (наприклад, скласти дві матриці 100000 x 100000); у таких випадках SIMD- система дає суттєвий приріст у швидкодійності.

MIMD- системи являють подальший розвиток обчислювальної техніки: вони складаються з кількох процесорів, кожен з котрих функціонує за власною програмою. Така структура призначена для серверних завдань (дозволяє кільком користувачам одночасно підключатися до системи та запускати на виконання кожен власну програму), складних задач, коли потрібне розпаралелювання частин програми, або для використання її як SIMD- системи.

Основні характеристики MIMD:

- тісний зв'язок через загальну пам'ять – багатопроцесорна ЕОМ
- розподілена обчислювальна система з не тісним зв'язком через мережу комутацій та обміном повідомленнями

З розвитком обчислювальної техніки з метою підвищення потужності найбільш розповсюджених серед звичайних користувачів SISD- систем до них

починають використовувати підходи конвеєризації та розпаралелювання, характерні для MIMD- систем. Нові системи складаються з кількох (найчастіше – двох) процесорів та обладують високошвидкісними механізмами внутрішньої комунікації між системними компонентами (пам’ять, накопичувачі, пристрої вводу/виводу)

1.2 MPI- стандарт програмування у паралельних середовищах

Головний недолік розроблених паралельних мов – необхідність розробки компілятора для кожної нової платформи, тому на зміну декількох мов прийшов єдиний стандарт MPI для MIMD систем, так як SIMD системи були вже практично витіснені.

MPI являє собою бібліотеку функцій для розробки програм, що виконуються на декількох процесорах. Підхід до стандарту як саме до бібліотеки функцій, а не нової мови, зробив стандарт популярним: замість розробки нового (або портування існуючого) компілятора для системи потрібно реалізувати лише бібліотеку відповідних функцій (для мов програмування C та Fortran). Згідно зі стандартом, кілька копій програми виконується кожна на власному процесорі та окремо проводять обчислення. Копії виконують комунікацію між собою шляхом посилання повідомлень. Існують також механізми редукції та розповсюдження даних між процесами.

Інший момент пов’язаний з можливістю побудови багатопроцесорних кластерів з кількох SISD машин, так як комутація між програмами здійснюється завдяки remote services (механізм виконання команд на віддалених системах). Якщо з’єднати кілька машин, що находяться на відстані, можна одержати розподілену обчислювальну мережу.

Нові версії стандарту крім власне механізмів комунікації пропонують також прискорений файловий ввід/вивід (Fast I/O), засоби для відлагодження програми, аналізу їх виконання (MPE) та ін.

Існує велика кількість вільно розповсюджуваних реалізацій MPI, що портовані на багато платформ. Виробник нової MIMD системи або пише власну реалізацію стандарту, або використовує існуючу – у будь-якому випадку модель можна без змін використовувати на різних платформах.

Номер процесу – ціле додатнє число, що є унікальним атрибутом кожного процесу.

Атрибути повідомлення – номер процесу-відправника, номер процесу-одержувача й ідентифікатор повідомлення. Для них заведена структура *MPI_Status*, що містить три поля: *MPI_Source* (номер процесу відправника), *MPI_Tag* (ідентифікатор повідомлення), *MPI_Error* (код помилки); можуть бути й додаткові поля.

Ідентифікатор повідомлення (msgtag) – атрибут повідомлення, що є цілим додатнім числом, що лежить у діапазоні від 0 до 32767. Процеси поєднуються в *групи*, можуть бути вкладені групи. Усередині групи всі процеси пронумеровані. З кожною групою асоційований свій *комунікатор*. Тому при здійсненні пересилання необхідно вказати ідентифікатор групи, усередині якої здійснюється це пересилання. Всі процеси знаходяться в групі з визначеним ідентифікатором *MPI_COMM_WORLD*.

При описі процедур MPI будемо користуватися словом OUT для позначення "вихідних" параметрів, тобто таких параметрів, через які процедура повертає результати.

Загальні процедури MPI

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

MPI_Init – ініціалізація паралельної частини програми. Реальна ініціалізація для кожного додатка виконується не більше одного разу, а якщо MPI уже був ініційований, те ніякі дії не виконуються й відбувається негайне повернення з підпрограми. Всі MPI-процедури, що залишилися, можуть бути викликані тільки після *виклику MPI_*Init.

Повертається: у випадку успішного виконання – *MPI_SUCCESS*, інакше – код помилки. (Теж саме повертають майже всі інші функції).

int MPI_Finalize(void)

MPI_Finalize – завершення паралельної частини програми. Всі наступні звертання до будь-яких MPI-процедур, у тому числі до *MPI_Init*, заборонені. До моменту виклику *MPI_Finalize* деяким процесом всі дії, що вимагають його участі в обміні повідомленнями, повинні бути завершені. Складний тип аргументів *MPI_Init* передбачений для того, щоб передавати всім процесам аргументи *main*:

```
int main(int argc, char** argv)
{
      MPI_Init(&argc, &argv);
          ...
      MPI_Finalize();
}
```

int MPI_Comm_size(MPI_Comm comm, int* size)

Визначення загального числа паралельних процесів у групі *comm*.

- *comm* – ідентифікатор групи
- OUT *size* – розмір групи

int MPI_Comm_rank(MPI_Comm comm, int* rank)

Визначення номера процесу в групі *comm*. Значення, що повертає за адресою *&rank*, лежить у діапазоні від 0 до *size_of_group-1*.

- *comm* – ідентифікатор групи
- OUT *rank* – номер викликаючого процесу в групі *comm*

double MPI_Wtime(void)

Функція повертає астрономічний час у секундах (речовинне число), що пройшло з деякого моменту в минулому. Гарантується, що цей момент не буде змінений за час існування процесу.

1.3 Засоби для відлагодження MPI програм, аналізу їх виконання

Бібліотека MPE (Multi-Processing Environment) містить процедури, які полегшують написання, налагодження й оцінку ефективності MPI-Програм. MPE-процедури діляться на кілька категорій.

Паралельна графіка (Parallel X graphics**).** Ці процедури забезпечують доступ всім процесам до поділюваного X-дисплею. Вони створюють зручний вивід для паралельних програм, дозволяють виводити текст, прямокутники, кола, лінії й так далі.

Реєстрація (Logging**).** Одним з найпоширеніших засобів для аналізу характеристик паралельних програм є файл траси відзначених у часі подій – **логфайл** (logfile). Бібліотека MPE створює можливість легко одержати такий файл у кожному процесі й зібрати їх разом по закінченню роботи. Вона також автоматично обробляє неузгодженість і дрейф часу на множині процесорів, якщо система не забезпечує синхронізацію часу. Ця бібліотека призначена для користувача, що бажає створити свої власні події й програмні стани.

Послідовні секції (Sequential Sections**).** Іноді секція коду, що виконується на ряді процесів, повинна бути виконана тільки по одному разу на процесах у порядку номерів цих процесів. MPE має функції для такої роботи.

Обробка помилок (Error Handling). MPI має механізм, що дозволяє користувачеві управляти реакцією реалізації на помилки часу виконання, включаючи можливість створення свого власного оброблювача помилок.

Засоби реєстрації (logging) для аналізу ефективності паралельних програм. Аналіз результатів реєстрації виробляється після виконання обчислень. Засоби реєстрації й аналізу включають ряд профілюючих бібліотек, утилітних програм і ряд графічних засобів.

Перша група засобів профілювання. Бібліотечні ключі забезпечують збори процедур, які створюють логфайли. Ці логфайли можуть бути створені вручну шляхом розміщення в програмі MPI звертань до MPE, або автоматично при установленні зв'язку з відповідними MPE-Бібліотеками, або комбінацією цих двох методів. У цей час MPE пропонує наступні три профілюючі бібліотеки:

1. **Tracing Library** (бібліотека трасування) – трасує всі MPI-Виклики. Кожний виклик випереджається рядком, що містить номер зухвалого процесу в **MPI_COMM_WORLD** і супроводжується іншим рядком, який вказує, що виклик завершився. Більшість процедур **send** і **receive** також вказують значення **count**, **tag** і імена процесів, які посилають або приймають дані.

2. **Animation Libraries (**анімаційна бібліотека**)** – проста форма програмної анімації в реальному часі, що вимагає процедур X-Вікон.

3. **Logging Libraries** (бібліотека реєстрації) – самі корисні й широко використовувані профілюючі бібліотеки в MPE. Вони формують базис для генерації логфайлів з користувальницьких програм. Зараз є три різних формати логфайлів, припустимих в MPE. За замовчуванням використовується формат CLOG. Він містить сукупність подій з єдиним відмітчиком часу. Формат ALOG більше не розвивається й підтримується для забезпечення сумісності з ранніми програмами. І найбільш потужним є формат – SLOG (для Scalable Logfile), що може бути конвертований із уже наявного CLOG- файлу або отриманий прямо з виконуваної програми (для цього необхідно встановити змінну середовища MPE_LOG_FORMAT в SLOG).

Набір утилітних програм в MPE включає конвертори форматів (наприклад, clog2slog), друк логфайлів (slog_print), оболонки засобів візуалізації логфайлів, які вибирають коректні графічні засоби для подання логфайлів відповідно до їх розширень.

Бібліотеки реєстрації **Logging Libraries**. Результати розрахунку часу дають деяке подання про ефективність програми. Але в більшості випадків потрібно докладно довідатися, яка була послідовність подій, скільки часу був витрачений на

кожну стадію обчислення й скільки часу займає кожна окрема операція передачі. Щоб полегшити їхнє сприйняття, потрібно представити їх у графічній формі. Але для цього спочатку потрібно створити файли подій зі зв'язаними тимчасовими оцінками, потім досліджувати їх після закінчення програми й тільки потім інтерпретувати їх графічно на робочій станції. Здатність автоматично генерувати логфайли є важливим компонентом всіх засобів для аналізу ефективності паралельних програм.

Бібліотека для створення логфайлов відділена від бібліотеки обміну повідомленнями MPI. Перегляд логфайлов незалежний від їхнього створення, і тому можуть використовуватися різні інструментальні засоби. Бібліотека для створення логфайлов MPE розроблена таким чином, щоб співіснувати з будь-який MPI-Реалізацією й поширюється поряд з модельною версією MPI.

Щоб створити файл реєстрації, необхідно викликати процедуру **MPE_Log_event**. Крім того, кожний процес повинен викликати процедуру **MPE_Init_log**, щоб приготуватися до реєстрації, і **MPE_Finish_log**, щоб об'єднати файли, що зберігаються локально при кожному процесі в єдиний логфайл. **MPE_Stop_log** використовується, щоб призупинити реєстрацію, хоча таймер продовжує працювати.

MPE_Start_log починає реєстрацію. Програміст вибирає будь-які ненегативні цілі числа, бажані для типів подій; система не надає ніяких приватних значень типам подій. Події розглядаються як такі, що не мають тривалості. Щоб виміряти тривалість стану програми, необхідно, щоб пара подій відзначила початок і закінчення стану. Стан визначається процедурою **MPE_Describe_state**, що описує початок і закінчення типів подій. Процедура **MPE_Describe_state** також додає назву стану і його колір на графічному виводі.

Відповідна процедура **MPE_Describe_event** забезпечує опис події кожного типу. Використовуючи ці процедури, приведемо приклад обчислення числа ?. Для цього оснастимо програму обчислення числа π відповідними операторами. Важливо, щоб реєстрація події не створювала більших накладних витрат. **MPE_Log_event** зберігає невелика кількість інформації у швидкій пам'яті. Під час виконання **MPE_Log_event** ці буфери поєднуються паралельно й кінцевий буфер, відсортований по тимчасовим міткам, записується процесом 0.

Аналіз логфайлов. Після виконання програми **MPI**, що містить процедури MPE для реєстрації подій, директорій, де вона виконувалася, містить файл подій, відсортованих за часом, причому час скоректований з обліком відхилення частоти генераторів. Можна написати багато програм для аналізу цього файлу й подання інформації.

Наприклад, одна з реалізацій MPE містить коротку програму, **states**. Якщо ми виконуємо її з логфайлом, що описали вище, ми одержимо:

Стан: Час:
Broadcast 0.000184
Compute 4.980584
Reduce 0.000248
Sync 0.000095
Сума: 4.981111

Така підсумкова інформація є досить поширеною, але грубою формою профілювання; вона повідомляє тільки, де програма витрачає час. Значно інформативніше графічне подання, забезпечуване спеціалізованими програмами, наприклад, **upshot** і **jampshot**.

Вхідні процедури MPE використовуються, щоб створити логфайли (звіти) про події, які мали місце при виконанні паралельної програми.

Ці процедури дозволяють користувачеві включати тільки ті події, які йому цікаві в даній програмі. Базові процедури − **MPE_Init_log**, **MPE_Log_event** і **MPE_Finish_log**.

MPE_Init_log повинна викликатися всіма процесами, щоб ініціювати необхідні структури даних. **MPE_Finish_log** збирає звіти із всіх процесів, поєднує їх, вирівнює по загальній шкалі часу. Потім процес із номером 0 у комунікаторі **MPI_COMM_WORLD** записує звіт у файл, ім'я якого зазначено в аргументі. Одиночна подія встановлюється процедурою **MPE_Log_event**, що задає тип події (вибирає користувач), ціле число й рядок для даних. Щоб розмістити логфайл, що буде потрібний для аналізу або для програми візуалізації (подібної upshot), процедура **MPE_Describe_state** дозволяє додати події й описувані стани, указати крапку старту й закінчення для кожного стану. При бажанні для візуалізації звіту можна використовувати колір. **MPE_Stop_log** і **MPE_Start_log** призначені для того, щоб динамічно включати й виключати створення звіту.

Ці процедури використовуються в одній із профілюючих бібліотек, що поставляються з дистрибутивом для автоматичного запуску подій бібліотечних викликів MPI.

Два змінні середовища **TMPDIR** і **MPE LOG FORMAT** потрібні користувачеві для установки деяких параметрів перед генерацією логфайлів.

MPE LOG FORMAT – визначає формат логфайла, отриманого після виконання додатка, пов'язаного з MPE-Бібліотекою. **MPE LOG FORMAT** може приймати тільки значення **CLOG**, **SLOG** і **ALOG**. Коли **MPE LOG FORMAT** установлений в **NOT**, передбачається формат **CLOG**.

TMPDIR – описує директорій, що використовується як тимчасова пам'ять для кожного процесу. За замовчуванням, коли

TMPDIR є **NOT**, буде використовуватися "**/tmp**". Коли користувачеві потрібно одержати дуже довгий логфайл для дуже довгої MPI-Роботи, користувач повинен переконатися, що **TMPDIR** досить великий, щоб зберігати тимчасовий логфайл, що буде вилучено, якщо об'єднаний файл буде створений успішно.

1.4 Засоби перегляду логфайлів

Існує чотири графічних засоби візуалізації, розповсюджуваних разом з MPE: upshot, nupshot, Jumpshot-2 і Jumpshot-3. Із цих чотирьох переглядачів логфайлів тільки три побудовані за допомогою MPE. Це upshot, Jumpshot-2 і Jumpshot-3.

Upshot і Nupshot. Один з використовуваних інструментів називається upshot. Найпростіший вид екрана Upshot показаний на мал. 2.2. Результат представлений у вигляді смуг, по одній смузі на кожний процес. Кожна смуга складається з ділянок різного кольору (для чорно-білих моніторів колір заміняється різними штрихуваннями). Кожному кольору відповідає стан процесу певного типу. Унизу

наведені оцінки часу, що починаються від нуля.

Таке подання дозволяє візуально визначити стан будь-якого процесу в будь-який момент часу.

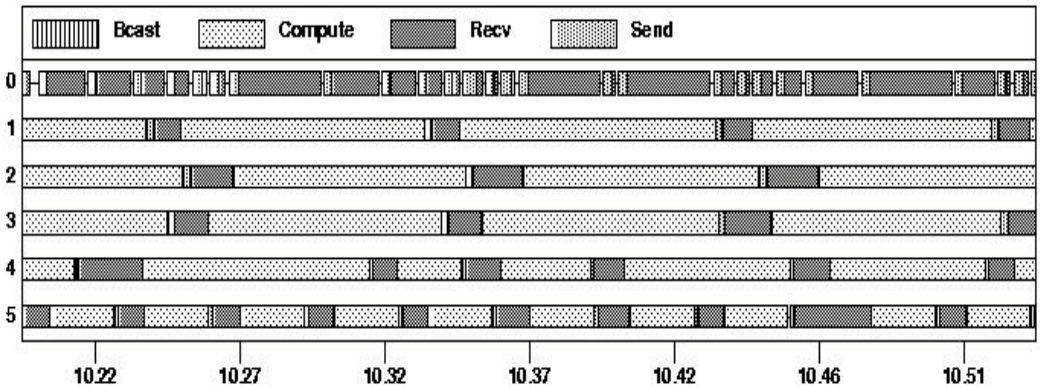


Рисунок 8.1 – Подання результатів профілювання за допомогою upshot

Існують і більше складні зображення вікон, які дозволяють міняти розмір зображення по горизонталі або вертикалі, центрувати на будь-якій крапці дисплея, обраною мишею.

Jumpshot-3 і Jumpshot-4. Існує ще дві версії, що поставляються разом з MPE. Це Jumpshot-3 і Jumpshot-4, що розвилися з Upshot і Nupshot. Обидві написані на Java і є графічними засобами візуалізації для інтерпретації двійкових файлів трас, які показують їх на дисплеї.

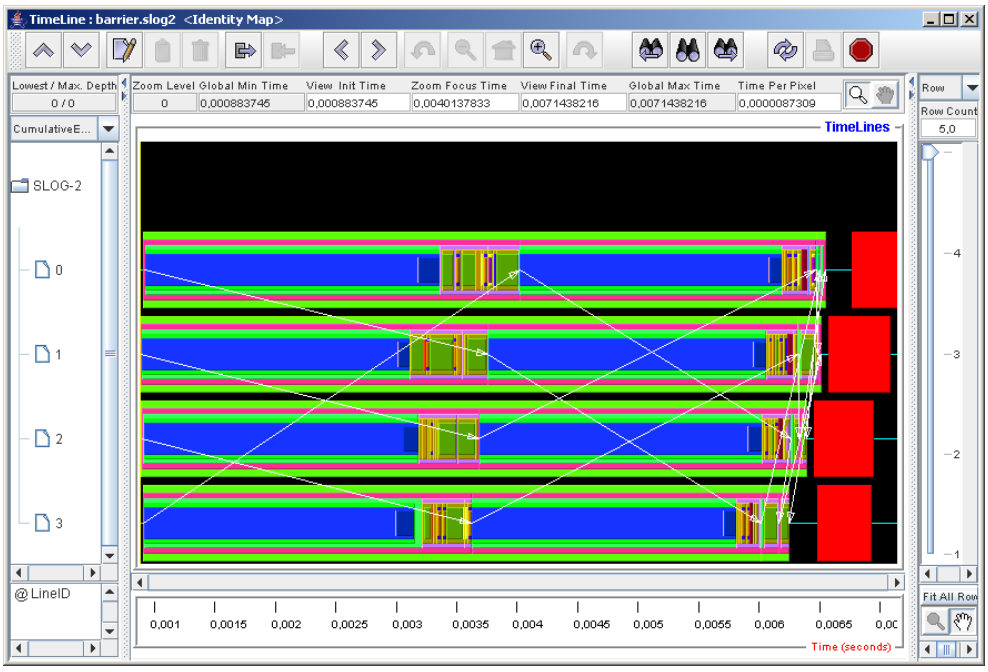


Рисунок 8.2 – Подання результатів профілювання за допомогою jumpshot-4

Накінець, існує команда яку ви можете використати для виводу інформації що поновлюється з перебігом часу. Ця команда називається top і запускається як:

$ **top**

Останнє виведе повний екран різноманітної інформації про активні процеси, також деякі додаткові дані про систему. Це включає середнє навантаження системи, кількість процесів, стан процесора, пам'яті, інформацію про, власне, процеси, включаючи ідентифікаційний номер (PID), пріоритет, скільки вони споживають процесора і пам'яті, час активності і назву програми.

```
 6:47pm  up 1 day,  18:01,  1 user,  load average: 0.02, 0.07, 0.02
61 processes: 59 sleeping, 2 running, 0 zombie, 0 stopped
CPU states:  2.8% user,  3.1% system,  0.0% nice, 93.9% idle
Mem:   257992K av,  249672K used,    8320K free,   51628K shrd,  78248K buff
Swap:   32764K av,     136K used,   32628K free,                 82600K cached
PID USER    PRI  NI  SIZE  RSS  SHARE  STAT  LIB %CPU %MEM  TIME COMMAND
 112 root     12   0 19376  18M   2468  R       0  3.7  7.5 55:53 X
4946 root     12   0  1040 1040    836  R       0  1.5  0.4  0:00 top
 121 david     4   0   796  796    644  S       0  1.1  0.3 25:37 wmSMPmon
 115 david     3   0  2180 2180   1452  S       0  0.3  0.8  1:35 wmaker
4948 david    16   0   776  776    648  S       0  0.3  0.3  0:00 xwd
   1 root      1   0   176  176    148  S       0  0.1  0.0  0:13 init
 189 david     1   0  6256 6156   4352  S       0  0.1  2.4  3:16 licq
4734 david     0   0  1164 1164    916  S       0  0.1  0.4  0:00 rxvt
   2 root      0   0     0    0      0  SW      0  0.0  0.0  0:08 kflushd
   3 root      0   0     0    0      0  SW      0  0.0  0.0  0:06 kupdate
   4 root      0   0     0    0      0  SW      0  0.0  0.0  0:00 kpiod
   5 root      0   0     0    0      0  SW      0  0.0  0.0  0:04 kswapd
  31 root      0   0   340  340    248  S       0  0.0  0.1  0:00 kerneld
  51 root      0   0    48   48     32  S       0  0.0  0.0  0:00 dhcpcd
  53 bin       0   0   316  316    236  S       0  0.0  0.1  0:00 rpc.portmap
  57 root      0   0   588  588    488  S       0  0.0  0.2  0:01 syslogd
```

Сама назва програми **top** вказує на те, що найбільш ресурсоємкі процеси будуть перечислені зверху. **top** досить ефективна для виявлення які програми погано поводяться і тому повинні бути зупиненими.

Якщо вас цікавлять лише процеси що належать певному користувачу, ви можете використати прапорець -u (user) з ім'ям користувача або його ідентифікаційним номером (UID):

$ **top -u st**

top також підтримує відслідковування процесів за їхнім ідентифікаційним номером (PID), ігнорування пасивних (idle) і зомбі (zombie) процесів і ще багато додаткових опцій, про які ви можете дізнатися зі сторінки посібника

1.5 Структура апаратних та системних програмних засобів

Доступ до обчислювальних ресурсів паралельного комп’ютера (науково-дослідної лабораторії Тернопільського державного економічного університету) відбувається з робочого місця (РМ) через локальну мережу.

Характеристики паралельної – системи:

– кількість процесорів: 4 центральних процесора

– підтримка стандартів паралельного програмування: Message Passing Interface

– операційна система: Linux Debian 6.0

До системних програмних засобів відноситься усе програмне забезпечення, що дозволяє системі функціонувати.

Існує кілька засобів копіювання даних між віддаленими системами, але частіше використовуються ftp. При використанні ftp на комп'ютери встановлюються ftp-сервер (ftpd) та ftp-клієнт (ftp). Клієнтська програма ініціює зв'язок з сервером та отримає від нього дозвіл на роботу з файловими даними на серверу (читання/запис).

Для операцій упорядковування даних використовуються класичні команди системи UNIX (mkdir, cp, rm і т.д.), дивіться Додаток 1.

Для одержання можливості виконати якусь команду на віддаленій системі необхідно спочатку підключитись та війти до системи. Це може бути здійснене завдяки використанню особливих програмних засобів у Windows: telneat або pytty та у Linux – ssh. Telneat та pytty дозволяють виконати повний вхід на віддалену систему, як на локальну, а ssh – запускає командну оболонку на віддаленій системі з шифруванням сесії. Telnet вимагає наявності на віддаленій системі telnet- сервера (telnetd), а ssh – ssh- сервера (sshd).

До засобів побудови програмного забезпечення відносяться компілятори C/C++, з підтримкою MPI (команди mpicc, mpiCC, mpirun).

Команди управління процесами у Linux-середовищі наведено у Додатку 2.

2. ХІД РОБОТИ

2.1 Уважно ознайомтеся із теоретичною частиною лабораторної роботи.

2.2 За допомогою програми віддаленого доступу Putty зайти на суперкомп'ютер, використавши наступні дані:

IP: 193.104.213.19
Port: 22
Connection type: SSH
Window\Translation: UTF-8

Login: отримати у викладача
Password: отримати у викладача

2.3 Щоб проглянути вміст каталогу введіть команду ls. Скопіюйте у свою домашню папку файли prg1.c, prg2.c, prg4.c. Це можна зробити за допомогою програми WinSCP.

Примітка:

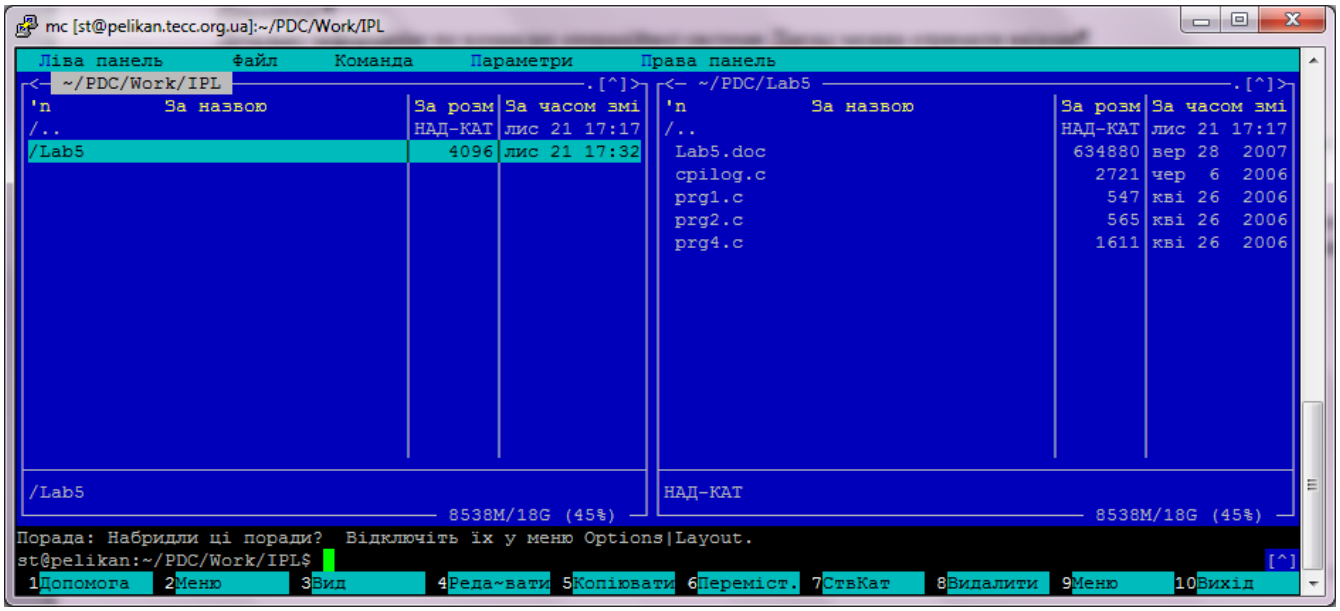
Детальну інформацію по командах операційної системи Лінукс можна отримати ввівши

<ім'я команди> --help | less

Буде виведена довідка, клавіші зі стрілками „вгору" і „вниз" служать для прокрутки, вихід з довідки клавіша **q**

2.4 Запустіть файловий менеджер Midnight Commander (mc). Навігація у ньому здійснюється аналогічно як і у Far та Total Commander.

2.5 Розглянемо просту паралельну програму з використанням MPI, у якій кожен паралельний процес виводитиме на екран свій номер та загальну кількість процесів. Ознайомтеся з текстом програми prg1.c (F3).



3.1 Відкомпілюйте та запустіть програму **prg1.c**

Приклад компілювання програми:

mpicc -o prg1 prg1.c

Приклад виконання:

mpirun -np 2 ./prg1

де 2 – це кількість паралельних процесів

Примітка: якщо при виклику mpirun виникає помилка, то потрібно здійснити наступні кроки:

– Створити в домашній папці файл з іменем .mpd.conf. У даному файлі має бути стрічка:

MPD_SECRETWORD=<будь-яка послідовність символів>

– Змінити права доступу до файлу .mpd.conf, щоб тільки власник мав право зчитувати і записувати.

– Виконати в консолі команду mpdboot (завантажує менеджер процесів MPI).

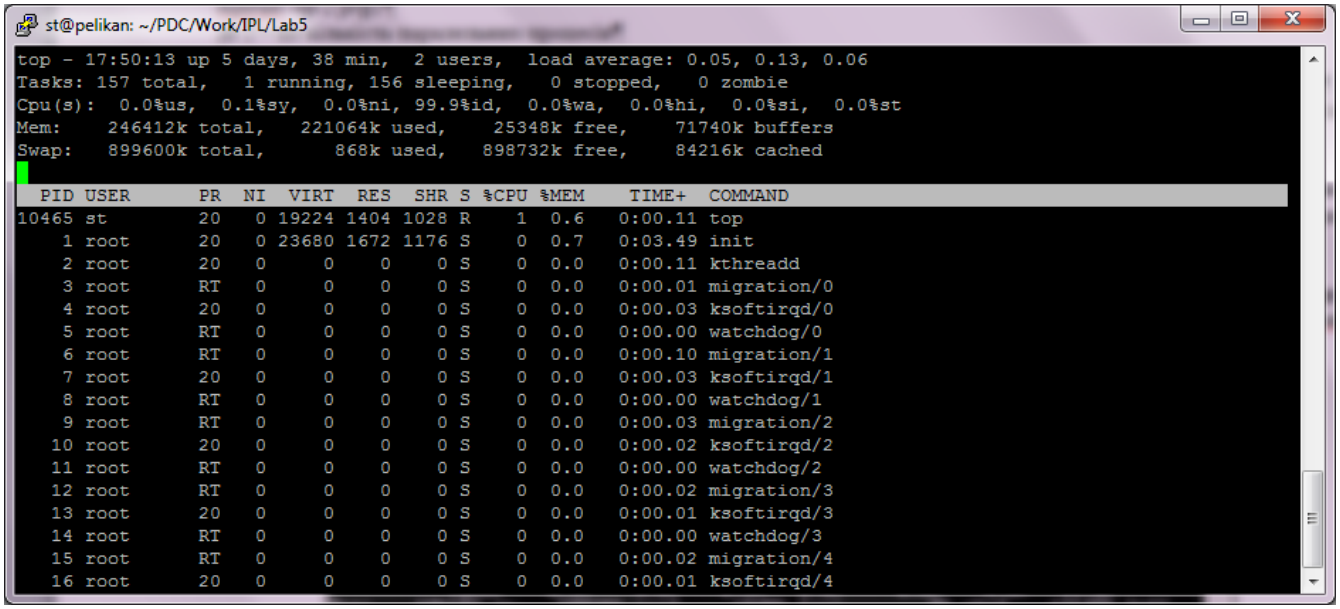
– Спробувати виконати програму за допомогою mpirun.

Отже, дана програма виведе кількість паралельних процесів, номер кожного процесу та ім’я кожного процесора.

Запускайте програму **prg1** на двох і чотирьох процесорах. Зверніть увагу, що повідомлення від процесів виводитися у довільному порядку.

3.2 Скопіюйте у свою папку, відкомпілюйте та виконайте програму **prg2.c** з каталогу **/home/st/Work/Ваше прізвище/Lab5/.** Запускайте програму **prg2** на двох процесорах. Змініть код програми (див. вище Редагування), ввівши замість тексту **Hello, MPI** своє прізвище латинськими літерами, відкомпілюйте її і запустіть знову.

3.3 Для того, щоб переглянути завантаженість віддаленого сервера можна скористатися програмою **top**. Виконайте та проаналізуйте її, чи завантажені процесори у даний момент часу і на скільки відсотків. Вихід з програми **top** – клавіша **q.** Команда **top** відображає стан процесів і їх активність "в реальному режимі часу".



- USER – ім'я власника процесу;
- PID – ідентифікатор процесу в системі;
- PPID – ідентифікатор батьківського процесу;
- %CPU – частка часу центрального процесора (у відсотках), виділеного даному процесу;
- %MEM – частка реальної пам'яті (у відсотках), використовувана даним процесом;
- RSS – розмір резидентного набору (кількість 1K-страниц в пам'яті);
- STAT – статус процесу;
- PRI – пріоритет планування;
- NI – значення стартового пріоритету;
- TIME – скільки часу центрального процесора зайняв даний процес;
- COMMAND – командний рядок запуску програми, виконуваної даним процесом.

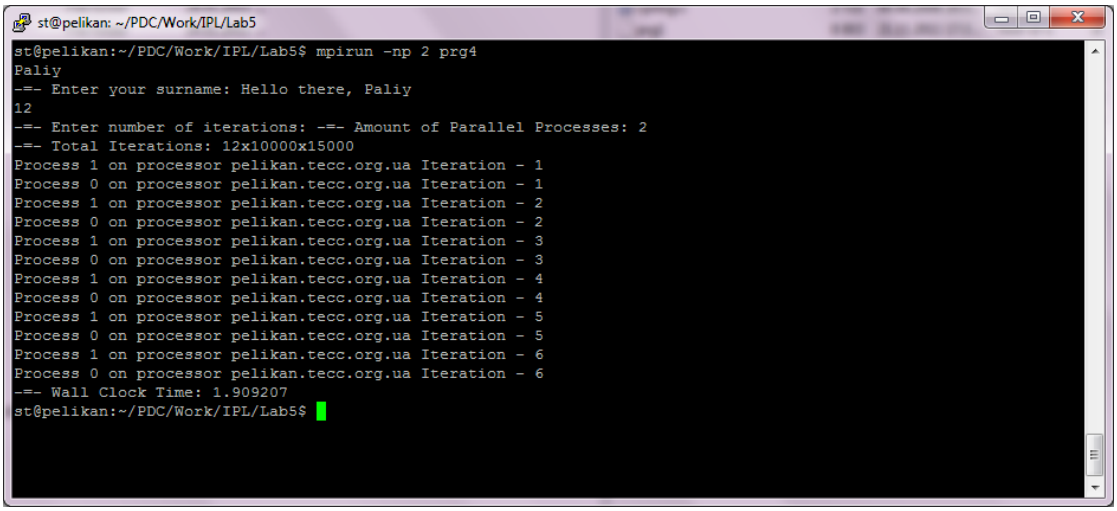
Як бачите, у верхній частині вікна відображається астрономічний час, час, що пройшов з моменту запуску системи, число користувачів в системі, число запущених процесів і число процесів, що знаходяться в різних станах, дані про використання ЦП, пам'яті і файлу підкачки. А далі йде таблиця, що характеризує окремі процеси. Число рядків, що відображаються в цій таблиці, визначається розміром вікна: скільки рядків поміщається, стільки і виводиться. Вміст вікна обновляється кожні 5 секунд. Список процесів може бути відсортований по використовуваному часу ЦП (за умовчанням), по використанню пам'яті, по PID, за часом виконання. Перемикати режими відображення можна за допомогою команд,

які програма top сприймає. Це наступні команди (просто натискайте відповідні клавіші, тільки з урахуванням регістра, тобто разом з клавішею Shift):

- <Shift>+<N> – сортування по PID;
- <Shift>+<A> – сортувати процеси по віку;
- <Shift>+<P> – сортувати процеси по використанню ЦП;
- <Shift>+<M> – сортувати процеси по використанню пам'яті;
- <Shift>+<T> – сортування за часом виконання.

3.4 Відкомпілюйте та виконайте програму **prg4.c**

Дана програма виконує три вкладені цикли і розпаралелює верхній цикл. Запустіть програму **prg4** на одному, двох, трьох чи більше процесорах. При цьому Вам необхідно ввести **своє прізвище** (латинськими літерами) та **кількість ітерацій верхнього циклу**, для цього використайте **свій порядковий номер** у списку журналу групи. Час виконання циклів, що виводитиметься програмою при різній кількості процесів буде різним. Проаналізуйте та занотуйте у звіт отримані результати, зокрема час роботи програми на одному, двох, чотирьох чи більше процесорах, що необхідно оформити у вигляді таблиці. Базуючись на отриманих значеннях часу виконання паралельної програми потрібно намалювати графік залежності прискорення розпаралелення програми від кількості процесорів паралельної машини. Наприклад:



Для того, щоб побачити як ваша програма завантажує сервер (коли працюють один чи обидва процесори) скористайтеся командою **top**, для цього запустіть ще одну копію програми **Putty** і увійшовши на сервер, виконайте команду **top.** Проаналізуйте завантаженість сервера процесами.

3. ЗМІСТ ЗВІТУ

3.1 Тема, мета лабораторної роботи
3.2 Короткі теоретичні відомості
3.3 Опис виконання лабораторної роботи згідно пунктів 2.2-2.10
3.4 Відповідь на одне контрольне запитання згідно номера варіанту (порядковий номер у списку журналу групи)
3.5 Висновки

4. КОНТРОЛЬНІ ПИТАННЯ

4.1 Що таке MPI?
4.2 Для чого призначена бібліотека MPI?
4.3 Перерахувати типи ЕОМ за Флінном.
4.4 Що таке SISD комп'ютер?
4.5 Охарактеризуйте векторні й матричні ЕОМ.
4.6 Для яких мов програмування реалізована бібліотека MPI?
4.7 Охарактеризуйте MIMD та MISD ЕОМ.
4.8 Що таке бібліотека MPE і для чого вона призначена?
4.9 Яке призначення команди **top**?
4.10 Які Ви знаєте програми для віддаленого доступу до Linux-систем?
4.11 Наведіть приклад компілювання MPI програм.
4.12 Наведіть приклад запуску на виконання MPI програм.

5. СПИСОК ЛІТЕРАТУРИ

5.1 Камерон Хьюз, Трейси Хьюз. Параллельное и распределенное программирование с использованием C++. : Пер. с англ. – М. : Издательский дом "Вильямс", 2004. – 627 с.

5.2 Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 512 с.

5.3 Воеводин В.В. Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. – 608 с.

5.4 А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону. Издательство ООО «ЦВВР», 2003, 208 с.

5.5 Джин Бэкон, Тим Харрис. Операционные системы. Параллельные и распределенные системы. – Питер: Издательская группа BHV, 2004. – 800 с.

5.6 www.mpi-forum.org

5.7 www.parallel.ru

ДОДАТОК 1.
Управління файлами у linux

Навігація: ls, cd, pwd

ls

Команда ls(1) перечислює файли у певній директорії. Якщо ви більш знайомі із DOS командною лінією, то ls дещо подібна до dir команди. Сама по собі, без аргументів, ls перечислить файли у поточній директорії. Щоб побачити що знаходиться у вашому кореневому каталозі (/), наприклад, ви можете використати ці команди:

```
$ cd /
$ ls
bin  cdr   dev home lost+found proc sbin  tmp  var
boot cdrom etc lib  mnt        root suncd usr  vmlinuz
```

Проблема з таким виводом може полягати у тому що у ньому важко відрізнити звичайні файли від каталогів. Деякі користувачі надають перевагу вживанню команди із опціями що допомагають ідентифікувати тип файла, наприклад:

```
$ ls -FC
bin/  cdr/   dev/ home/ lost+found/ proc/ sbin/  tmp/  var/
boot/ cdrom/ etc/ lib/  mnt/        root/ suncd/ usr/  vmlinuz
```

Завдяки -FC опції, у виводі каталоги отримують слеш на кінці нзазви, виконуючі файли – зірочку.

Дуже популярною опцією являється також --color, або --color=auto. Ця опція дозволяє отримати кольоровий вивід. Так назви каталогів будуть синього кольору, виконуючі файли – зеленого, символічні посилання – світло-блакитного, тощо.

- *Ви можете закріпити за ls командою --color опцію, якщо створите відповідний синонім у .bashrc файлі у домашньому каталозі, ви просто повинні добавити alias ls="ls --color=auto" до ~/.bashrc.*

Ми можемо також перечислити що знаходяться у відмінній від поточної директорії. Для цього аргументом ls повинен бути каталог який нас цікавить. Так, ми можемо перечислити що знаходиться у /boot каталозі виконавши

```
$ ls /boot
```

ls також може вивести додаткові інформацію про файли, таку як дата створення, власник і права доступу, для цього слід добавти -l (*long*) іпцію:

```
$ ls -l /
drwxr-xr-x   2 root     bin         4096 May  7 09:11 bin/
drwxr-xr-x   2 root     root        4096 Feb 24 03:55 boot/
drwxr-xr-x   2 root     root        4096 Feb 18 01:10 cdr/
drwxr-xr-x  14 root     root        6144 Oct 23 18:37 cdrom/
drwxr-xr-x   4 root     root       28672 Mar  5 18:01 dev/
drwxr-xr-x  10 root     root        4096 Mar  8 03:32 etc/
drwxr-xr-x   8 root     root        4096 Mar  8 03:31 home/
drwxr-xr-x   3 root     root        4096 Jan 23 21:29 lib/
drwxr-xr-x   2 root     root       16384 Nov  1 08:53 lost+found/
drwxr-xr-x   2 root     root        4096 Oct  6 12:47 mnt/
```

```
dr-xr-xr-x  62 root     root          0 Mar  4 15:32 proc/
drwxr-x--x  12 root     root       4096 Feb 26 02:06 root/
drwxr-xr-x   2 root     bin        4096 Feb 17 02:02 sbin/
drwxr-xr-x   5 root     root       2048 Oct 25 10:51 suncd/
drwxrwxrwt   4 root     root     487424 Mar  7 20:42 tmp/
drwxr-xr-x  21 root     root       4096 Aug 24 03:04 usr/
drwxr-xr-x  18 root     root       4096 Mar  8 03:32 var/
```

Щоб побачити також приховані файли і каталоги (ті чия назва починається з крапки), використайте -a (*all*) прапорець (~, тильда є скороченням для домашнього каталогу):

```
$ ls -a ~
.Xauthority      .bash_profile    .mc          .vimrc
.Xdefaults       .bashrc          .mozilla     docs
.bash_history    .inputrc         .viminfo     www
```

Крім вище вказаних опцій також уснують багато інших. Обов'язково передивіться сторінку посібника man для ls.

cd

cd(1) команда використовується щоб поміняти директорію у якій ви знаходитесь. Просто введіть cd із шляхом до каталогу у який ви хочете перейти:

```
darkstar:~$ cd /bin
darkstar:/bin$ cd usr
bash: cd: usr: No such file or directory
darkstar:/bin$ cd /usr
darkstar:/usr$ ls
bin
darkstar:/usr$ cd bin
darkstar:/usr/bin$
```

Зауважте що якщо назва шляху не починається з слешу, cd перенесе нас у підкаталог поточного каталогу із такою назвою, тобто якщо ми вже знаходимся у /usr і ми вказали cd команді bin каталог, то ми опинимось у /usr/bin. Це демонструє важливий концепт *абсолютних шляхів* і *відносних шляхів*. Назви шляхів що починаються зі слешу, тобто вказується повний шлях, починаючи з кореневого каталогу, називаються абсолютним шляхом. У відносних шляхах, в свою чергу, вказується шлях до файла по відношенню до поточного каталогу. До відносних шляхів ми також можемо долучити дві крапки (..), що означає "одним каталогом вище поточного". Так, якщо ми знаходимось у /usr/bin, то cd .. перенесе нас у /usr.

cd команда без жодних аргументів перенесе нас у наш домашній каталог.

cd команда відрізняється від багатьох інших команд тим що вона є *вбудованою командою оболонки*. Вбудовані оболонкові команди розглянуті у Розділі 8.3.1. Це, можливо, не є важливим у даний момент для вас, все що вас повинно цікавити, це те що cd описане у man 1 bash і отримати допомогу по цій команді можна завдяки

```
$ help cd
```

pwd

Команда pwd(1) (*print working directory*) використовується для того щоб показати ваше поточне місцезнаходження. Просто введіть pwd на командній лінії:

```
$ cd /bin
$ pwd
/bin
$ cd /usr
$ cd bin
$ pwd
/usr/bin
```

Переглядачі: more, less, most

more

Програма more(1), це текстовий переглядач (*пейджер*). Вона вживається часто коли текст файла (або вивід команди) не вміщається на екрані тож ми повинні переглядати його по частинах. more розбиває вивід на окремі частини і очікує притиску важіля пробілу перед тим як відобразити наступну сторінку тексту. Притиск Enter клавіші перемістить вивід на одну лінію. Команда візьме як аргумент назви файлів які ви хочете переглянути:

```
$ more /etc/X11/xorg.conf
```

Також more часто вживається у конвеєрі (введення до конвеєрів ви знайдете у Розділі 8.2.3), наприклад:

```
$ ls /usr/bin | more
```

Як і в більшості переглядачах, ви можете здійснювати пошук по тексту. Для цього під час роботи програми, введіть слеш за яким слідує текстовий рядок який вас цікавить. more перенесе вас до першого ж місця у тексті де цей рядок присутній.

Як тільки more дійшла до кінця тексту, вона припиняє свою роботу. Прочитайте обов'язково сторінку посібника more(1) для додаткових опцій і ключів навігації. Під час роботи програми **h** клавіша (*help*) викличе сторінку допомоги, **q** клавіша (*quit*) припинить роботу переглядача.

less

Хоча more є досить зручною програмою, вона має один невеликий недолік – ви не можете прокрутити текст назад. Саме тому, напевне, більше вживають less(1). less дуже подібна на попередню програму, ви так само викликаєте less і можете вживати її в конвеєрі:

```
$ less /var/log/messages
$ dmesg | less
```

Пошук по тексту так само працює з less. Сторінку допомоги ви теж отримаєте з допомогою **h** клавіші. Ключі навігації описані у lesskey(1) сторінці посібника. *q* ключ завершує роботу переглядача.

view

На додаток до вищезгаданих програм перегляду тексту варто згадати view(1) переглядач, який є частиною vi(1) редактора. Це фактично vi, запущений у режимі *тільки для читання*. view зручний тим що надає стандартну для vi підсвітку синтаксису, що допомагає у перегляді різноманітного коду. Також view успадковує розвинутий пошук по тексту, характерний для vi. Для припинення огляду, введіть **:q** команду.

Простий вивід: cat, echo

cat

cat(1) (скорочення від *concatenate*) спершу була задумана як програма що поєднує декілька файлів у один, але може використовуватись і для інших цілей.

Для того щоб поєднати декілька файлів у один більший, перечисліть необхідні файли після cat команди, після чого перенаправте вивід у інший файл (перенаправлення описане у Розділі 8.2.3):

$ cat file1 file2 file3 > bigfile

Простіший способів використання cat, це для відображення змісту файла на стандартному виводі. Для цього просто введіть:

$ cat file1

Ви також можете вживати cat у поєднанні з переглядачами, такими як more або less, якщо пропустите вивід cat команди через конвеєр (хоча простіше просто виконати more file1):

$ cat file1 | more

Інший поширений спосіб використання cat – для копіювання файлів (навіть бінарних):

$ cat /bin/bash > ~/mybash

Останнє скопіює bash програму у ваш домашній каталог, назвавши її mybash.

Також типовим використанням є долучення змісту одного файлу до іншого за домомогою подвійного оператора перенаправлення ">>":

$ cat file1 >> file2

В останньому випадку, file2 збереже свій зміст, до нього лише буде додано те що знаходиться у file1.

Універсальність cat робить її однією з найчастіше вживаних команд. Завдяки тому що cat широко використовує стандартний ввід і стандартний вивід, вона ідеальна для оболонкових скриптів і як частина інших, складніших команд.

echo

echo(1) команда виводить наданий їй як аргумент рядок тексту на екран.

$ echo Hello there...

За замовчуванням echo виведе рядок і добавить нову лінію у кінці. Ви можете запобігти цьому якщо добавите -n прапорець до echo. Опція -e заставить echo брати до уваги любі екрановані знаки такі як \t (таб), \a (дзвоник), \e (знак екранації), тощо. Дуже радимо прочитати echo(1) і bash(1) сторінки посібника man для додаткової інформації.

Створення файлів і каталогів: touch, mkdir

touch

touch(1) використовується для зміни часу доступу або часу модифікації файла. Це у випадку якщо файл вже існує, якщо вказаного файла немає, тоді touch команда створить його. Отже, щоб створити новий порожній файл, виконайте:

$ touch file1

Існує декілька опцій touch, включаючи опції щодо який саме час змінити, модифікації (-m) чи доступу (-a), який саме час вказати (-t 200509170923) (рік, місяць, дата, година), тощо. Ви знайдете деталі у touch(1) сторінці посібника.

mkdir

mkdir(1) команда створить новий каталог. Ви просто повинні вказати ім'я каталогу або каталогів які ви хочете створити. Наприклад, щоб створити новий каталог або каталоги у поточній директорії, виконайте:

$ mkdir documents

$ mkdir charts images

Ви також можете вказати шлях до каталогу який ви хочете створити, приміром

$ mkdir /tmp/temporary

mkdir команда не має багато опцій. Важливими є -m, опція що вказуватиме права доступу до каталогу, і -p опція що дозволяє створювати директорії рекурсивно (розміщені одна в одній):

$ mkdir -m 700 noshare

$ mkdir -p www/images/banners

Перша команда створить каталог noshare із правами на читання, запис і виконання тільки користувачем, тоді як друга утворить одночасно декілька розміщених один в одному каталогів. Якщо не надати -p (*parent*) прапорець в останній команді, це призведе до помилки, якщо ні www ні images попередньо не існували.

Копіювання і зміна назви.

cp

cp(1) команда копіює файли і каталоги якщо надано -r або (краще) -R (*recursively*) прапорець. Наприклад:

$ cp processes.txt /tmp

$ cp -r charts /tmp

Багато користувачів віддають перевагу збереженню значення часу створення або модифікації а також прав виконання з допомогою -a (всерівно що -dpR) або -p опції:

$ cp -a file1 /tmp

$ cp -pR noshare /tmp

$ ls -ld /tmp/file1 /tmp/noshare

cp має ще декілька опцій. Радимо вам прочитати сторінку посібника для cp(1).

mv

mv(1) команда переносить файл з одного місця у інше. mv також дуже часто використовується для зміни назви файла. Власне, це єдина команда яка дозволяє це здійснити у один крок.

$ mv file1 /tmp/fileX

$ mv /tmp/fileX /tmp/fileY

Якщо ви заглянете у сторінку посібника для mv(1), ви знайдете декілька додаткових опцій цієї команди. На практиці, мало хто їх використовує.

Усунення: rm, rmdir

rm

Команда rm(1) видаляє файли і директорії. До цієї команди портібно ставитись обережно, оскільки у Лінуксі не існує легкого засобу відтворення видаленого файлу, тобто те що видалено буде втрачено незворотньо. Звичайно, ви можете видалити лише ті файли на які ви маєте права запису.

Щоб видалити один або більше файлів, ви просто перечисляєте їхні назви за rm командою:

$ rm file2 /tmp/fileY

Якщо ви не маєте прав на запис, ви отримаєте "Permission denied" помилку. Це запобігає видаленню звичайними користувачами важливих системних файлів, наприклад. Серед різноманітних опцій rm команди, варто звернути увагу на -f (force) яка запобігає підтвердження наміру видалення і звісток про помилки у випадку невдачі, також -r або -R (*recursively*), що дозволяє рекурсивно видаляти ієрархії директорії:

$ rm -Rf /tmp/noshare

rm це доволі потужна команда, особливо якщо вона використовується root користувачем. Так, щоб видалити всю кореневу ієрархію, необхідно лише видати rm -Rf / (звичайно ми не радимо робити цього). Також будьте особливо уважними у вживанні rm із шаблонами. Якщо не впевнені, завжди краще вживати rm із -i (*interactive*) прапорцем, що вимагатиме підтвердження (**y** або **n**) вашого наміру видалити кожний файл.

rmdir

rmdir(1) як ви можливо здогадуєтесь, видаляє каталоги з файлової системи. Каталог обов'язково повинен бути пустим перед тим як його можна буде видалити за допомогою rmdir(1). Синтаксис команди досить простий – сама команда, за якою слідує один або більше каталогів що належать усуненню:

$ rmdir /tmp/charts

Якщо charts не є порожнім, ви отримаєте помилку "rmdir: `charts': Directory not empty". В такому випадку варто або випорожнити charts каталог перед усуненням, або вжити -p прапорець (або скористатися з уже відомої нам rm -Rf):

$ rmdir -p /tmp/charts

Остання команда може бути вжита лише звичайним користувачем, оскільки вона по ідеї повинна видалити всі каталоги із /tmp включно, але оскільки звичайні користувачі не мають права видалити /tmp, цей каталог буде збережено. У випадку root користувача, /tmp теж буде усунено (що небажано).

Жорсткі і символічні посилання: ln

Посилання, це просто вказівники на інший файл. Вказівник і файл на який він вказує можуть знаходитись у різних каталогах файлової системи. Існує два типи посилань – жорсткі посилання і символічні посилання. Програма ln(1) бере на себе завдання по створенню посилань.

Жорсткі посилання, це просто інша назва для того самого файла (один файл може мати різні назви). Жорсткі посилання можуть знаходитись лише у межах однієї файлової системи і бути видаленими лише коли остання назва файла видалена з системи. Ось приклад створення жорсткого посилання, мається на увазі що file1 вже існує і fileH буде жорстким посиланням до першого. Ми використовуємо ln без жодних прапорців у цьому випадку:

$ ln file1 fileH

Жорсткі посилання бувають корисними у деяких випадках, але більшість користувачів, напевне, нададуть перевагу універсальнішим символічним посиланням.

Символічні посилання можуть вказувати на файл по-за файлвої системи, навіть на вже неіснуючий файл. Це власне, невеличкий файл що утримує необхідну інформацію. Ви можете додавати і усувати символічі посилання без впливу на справжній файл. Оскільки символічні посилання, це лише файл із власною інформацією, він може вказувати також на каталоги. Це, власне, дуже поширене, мати /var/tmp як символічне посилання на /tmp каталог.

Щоб сворити символічне посилання, додайте -s до ln команди:

$ ln -s file1 fileS

$ ls -l fileS

lrwxrwxrwx 1 nabis users 5 Sep 20 23:16 fileS -> file1

Як бачите, в обох випадках, жорсткого і символічного посилань, першим аргументом ln стоїть справжній файл (file1), а потім йде назва файла-посилання що буде створено (fileH та fileS).

ДОДАТОК 2.
Контроль над процесами

Кожна розпочата програма називається процесом. Ці процеси сягають від Віконної Системи X до системних програм (*демонів*), запущених під час старту системи. Кожний процес діє від імені певного користувача, ті що започатковано під час завантаження системи, як правило, запущені від імені root або nobody. Процеси, які ви розпочали, будуть числитись під вашим іменем.

Вам надається повний контроль над процесами, що ви започаткували, root користувачу — над всіма процесами, інших користувачів включно. Контроль над процесами і спостереження здійснюється за допомогою декількох програм і команд оболонки, розглянутих нижче.

Фонові процеси

Програми, запущені з командного рядка, як правило, залишаються на передньому плані. Це дозволяє бачити весь вивід програми і взаємодіяти з нею. Тим не менш, часто буває, що нам необхідно розпочати програму таким чином, щоб вона не блокувала нашого терміналу, тобто у фоновому режимі. Існує декілька способів добитися цього.

Під час запуску програми можна добавити знак кон'юнкції (&) в кінці командного рядка. Наприклад, ви вирішили використати програму amp(1) для того щоб прослухати каталог із mp3-файлами, але, одночасно, вам потрібен командний рядок для виконання інших завдань. Наступна команда запустить amp у фоновому режимі:

$ **amp *.mp3 &**

Іншим способом помістити програму у фоновий режим, це запустити її звичайно на передньому плані, тобто у пріоритетному режимі, потім притиснути комбінацію клавіш **Ctrl+Z**, яка призупиняє процес. Хоча процес і призупинено, його завжди можна знову запустити з того самого місця де його було перервано. Існує дві команди для цього: fg (скорочення для *foreground*) і bg (*background*). Щоб перезапустити процес у фоновому режимі, ми повинні виконати

[1]+ Stopped amp *.mp3

$ **bg**

Тепер колишній процес переднього плану буде виконуватись у фоновому режимі.

Пріоритетні процеси

Як ми вже сказали, запуск програми без амперсанду (знака кон'юкції) залишить її на передньому плані. Якщо ж ви загнали процес у фоновий режим за допомогою одного з способів, перелічених у попередньому розділі, то можете видобути його на передній план командою

$ **fg**

Якщо процес дійсний, він візьме контроль над вашим терміналом. Іноді програми завершують свою роботу у фоновому режимі, у таких випадках ви отримаєте повідомлення на зразок:

[1]+ Done amp *.mp3

Цілком можливо мати декілька процесів, запущених у фоновому режимі, одночасно. У таких випадках потрібно знати, який саме процес вивести назовні, сама по собі команда fg виведе на передній план лише останню команду, що було загнано у фоновий режим, це може бути не тим, що нас цікавить. Нам потрібно спочатку перечислити всі процеси у підгрунтті за допомогою вбудованої bash-команди jobs:

```
$ jobs
[1]   Stopped                 vim
[2]-  Stopped                 amp
[3]+  Stopped                 man ps
```

У цьому прикладі показано перелік процесів у фоновому режимі. Як ви бачите, вони всі позначені як "Stopped", тобто призупинені. Числа у квадратних дужках, це своєрідний ідентифікаційний номер процесів. Процес із знаком плюс (+) буде саме тим процесом, що звичайна fg виведе назовні. Щоб вивести якийсь інший з перечислених процесів, добавте ідентифікаційний номер після команди fg, наприклад для vim:

```
$ fg 1
```

Запуск програм у фоновому режимі дуже зручний у випадку ssh-з'єднань або використанню терміналів через даялап з'єднання, тощо. Ви можете запустити декілька програм одночасно і переходити від однієї до іншої у разі потреби.

ps

Крім процесів, запущених вами з командного рядка, існують також багато додаткових, включаючи постійні системні процеси. Команда ps(1) допоможе перелічити всі процеси, включаючи ті що не належать вам. Ця команда має багато додаткових опцій, ми розглянемо найважливіші. Для повного списку загляніть у сторінку посібника ps(1).

Сама по собі <tt>ps виведе список програм, запущених у поточному терміналі, включаючи пріоритетні процеси (разом із оболонкою, яку ви використовуєте і, звичайно, саму ps), фонові процеси також будуть перечислені. Частіше, це буде доволі коротким списком:

```
$ ps
  PID TTY          TIME CMD
 7923 ttyp0    00:00:00 bash
 8059 ttyp0    00:00:00 ps
```

Останнє є доволі типовим виводом команди ps. Розглянемо, що ці поля виводу означають.

PID (*process ID*) – кожному запущеному процесові надається унікальне ідентифікаційне число у діапазоні між 1 і 32767. Кожний процес, що започатковано отримає наступний вільний PID-номер. Якщо процес припиняє свою роботу або його вбито (дивіться kill команду у наступному підрозділі), він звільняє свій PID-номер. Коли досягнуто максимальної кількості номерів PID, призначення починається з нижчих звільнених чисел.

TTY-стовпчик вказує на якому терміналі запущено процес. Звичайна команда ps перечислить лише процеси, запущені на поточному терміналі, тож всі процеси

містять ту саму інформацію у нашому прикладі у стовпчику TTY, тобто ttyp0 (цей термінал призначено для віддалений під'єднань або X-емуляторів терміналу).

TIME-стовпчик вказує скільки CPU-часу процес спожив. Цей час відрізнятиметься від фізичного часу. Пам'ятаймо, що Лінукс – це багатозадачна операційна система, багато процесів запущено водночас, кожному з яких може бути виділена лише маленька частка процесорного часу. Тож TIME-колонка покаже дійсний час, виділений процесором для окремих завдань. Якщо час у цій колонці перевищуватиме декілька хвилин, це може означати, що щось негаразд.

Накінець, CMD-колонка вказує на саму програму. Виводиться лише базова назва програми, без командних опцій і аргументів. Для того щоб отримати цю додаткову інформацію, вам необхідно буде вжити деякі додаткові прапорці ps, розглянуті нижче.

Ви можете отримати повний список процесів, запущених на вашій системі, за допомогою додаткових -ax або ax (без дефісу) опцій до ps, як показано у наступному прикладі (ми дещо скоротили вивід):

```
$ ps -ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:03 init [3]
    2 ?        SW     0:13 [kflushd]
    3 ?        SW     0:14 [kupdate]
    4 ?        SW     0:00 [kpiod]
    5 ?        SW     0:17 [kswapd]
   11 ?        S      0:00 /sbin/kerneld
   30 ?        SW     0:01 [cardmgr]
   50 ?        S      0:00 /sbin/rpc.portmap
   54 ?        S      0:00 /usr/sbin/syslogd
   57 ?        S      0:00 /usr/sbin/klogd -c 3
   59 ?        S      0:00 /usr/sbin/inetd
   61 ?        S      0:04 /usr/local/sbin/sshd
   63 ?        S      0:00 /usr/sbin/rpc.mountd
   65 ?        S      0:00 /usr/sbin/rpc.nfsd
   67 ?        S      0:00 /usr/sbin/crond -l10
   69 ?        S      0:00 /usr/sbin/atd -b 15 -l 1
   77 ?        S      0:00 /usr/sbin/apmd
   79 ?        S      0:01 gpm -m /dev/mouse -t ps2
  106 tty1     S      0:08 -bash
  108 tty3     SW     0:00 [agetty]
  109 tty4     SW     0:00 [agetty]
  110 tty5     SW     0:00 [agetty]
  111 tty6     SW     0:00 [agetty]
...
```

Ваш вивід може відрізнятися, в залежності від того які сервіси у вас запущені. Як бачите, деякі процеси показані з опціями командного рядка, тоді як інші мають опції прихованими (завдяки ptrace латці ядра, що приховує командні опції певних процесів).

Як ви, напевне, зауважили, для багатьох процесів у TTY-стовпчику вказано "?". Це означає що ці процеси не прив'язано до жодного терміналу. Це характерно для демонів, — сервісів, що виконуються у фоновому режимі. Найпоширенішими демонами є sendmail, BIND, Apache (httpd), NFS, cron, тощо. Як правило, останні очікують запитів від клієнтів і повертають характерну інформацію якщо запит здійснено.

Тут ми побачимо додаткову колонку під назвою STAT. STAT є скороченням від *status* (статус). Умовними позначеннями для статусів процесів є:

diS – для сплячих (*sleeping*) процесів, що очікують якоїсь дії для активації.

Z – для зомбованих процесів (*zombie*), чиїх батьківський процес припинив свою роботу, залишивши за собою випадковий дочірній процес. Це, загалом, вважається поганою поведінкою програм.

D – позначатиме процеси що увійшли у стан сплячки без можливості переривання. Часто такі процеси відмовляються припинити своє існування, навіть після сигналу SIGKILL(описаному у наступному підрозділі).

W – означає?поділ пам'яті на сторінки? (*paging*).

X – позначатиме мертвий процес.

T – процес, що відслідковується, трасується (*traced*) або зупинено.

R – означає що процес активний (*runable*).

Якщо ви хочете отримати навіть більше інформації про процеси, спробуйте наступне:

$ **ps -aux**

```
 USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
 root         1  0.0  0.0    344    80 ?        S    Mar02   0:03 init [3]
 root         2  0.0  0.0      0     0 ?        SW   Mar02   0:13 [kflushd]
 root         3  0.0  0.0      0     0 ?        SW   Mar02   0:14 [kupdate]
 root         4  0.0  0.0      0     0 ?        SW   Mar02   0:00 [kpiod]
 root         5  0.0  0.0      0     0 ?        SW   Mar02   0:17 [kswapd]
 root        11  0.0  0.0   1044    44 ?        S    Mar02   0:00 /sbin/kerneld
 root        30  0.0  0.0   1160     0 ?        SW   Mar02   0:01 [cardmgr]
 bin         50  0.0  0.0   1076   120 ?        S    Mar02   0:00 /sbin/rpc.port
 root        54  0.0  0.1   1360   192 ?        S    Mar02   0:00 /usr/sbin/sysl
 root        57  0.0  0.1   1276   152 ?        S    Mar02   0:00 /usr/sbin/klog
 root        59  0.0  0.0   1332    60 ?        S    Mar02   0:00 /usr/sbin/inet
 root        61  0.0  0.2   1540   312 ?        S    Mar02   0:04 /usr/local/sbi
 root        63  0.0  0.0   1796    72 ?        S    Mar02   0:00 /usr/sbin/rpc.
 root        65  0.0  0.0   1812    68 ?        S    Mar02   0:00 /usr/sbin/rpc.
 root        67  0.0  0.2   1172   260 ?        S    Mar02   0:00 /usr/sbin/cron
 root        77  0.0  0.2   1048   316 ?        S    Mar02   0:00 /usr/sbin/apmd
```

```
root       79  0.0  0.1  1100  152 ?        S    Mar02   0:01 gpm
chris     106  0.0  0.5  1820  680 tty1     S    Mar02   0:08 -bash
root      108  0.0  0.0  1048    0 tty3     SW   Mar02   0:00 [agetty]
root      109  0.0  0.0  1048    0 tty4     SW   Mar02   0:00 [agetty]
root      110  0.0  0.0  1048    0 tty5     SW   Mar02   0:00 [agetty]
root      111  0.0  0.0  1048    0 tty6     SW   Mar02   0:00 [agetty]
...
```

Останнє додасть також інформацію про те, який саме користувач започаткував процес, яку долю системних ресурсів процес використовує (%CPU, %MEM, VSZ та RSS колонки), також дату коли процес було започатковано. Очевидно, що це достатньо інформації, цікавої системному адміністраторові. Може статися, що виведені рядки будуть занадто довгі і вийдуть за рамки екрану, в такому випадку можна вжити також ключ -w (*wrap*) для завертання довгих ліній.

Ви знайдете ще декілька опцій ps у сторінці посібника до цієї програми.

kill

Зрідка, програми поводяться не так як належить і вам необхідно якимось чином вплинути на них. Програмою для таких завдань являється kill(1), яка може впливати на перебіг процесів у декілька способів. Найбільш очевидне завдання kill – це знищити процес – буває необхідним коли останній зациклено і споживає всі системні ресурси.

Для того щоб знищити процес, вам необхідно знати PID, тобто ідентифікаційний номер процесу. Використайте команду ps, яку ми пройшли у попередньому розділі, для цього. Так, наприклад, щоб вбити процес під номером 4747, виконайте наступне:

$ **kill 4747**

Звичайно, ви повинні бути володарем цього процесу для того щоб його знищити. Це одна з рис безпеки системи, тож користувачі не можуть переривати процеси які їм не належать. root-користувач, як і очікується, може перервати будь-який процес.

Одним з різновидів kill є програма killall(1). Остання зупинить всі процеси з певною, наданою вами, назвою. Наприклад, якщо би ви хотіли вбити всі процеси під назвою mozilla-bin, командою було б:

$ **killall mozilla-bin**

Любі, запущені вами програми mozilla-bin, будуть припинені. Ця сама команда, виконана root-користувачем, вб'є всі процеси mozilla-bin усіх користувачів. Останнє приводить до цікавого способу викинути тимчасово всіх користувачів з системи, включаючи самого root-користувача, командою:

killall bash

Іноді буває деякі програми відмовляються завершити свою роботу при використанні звичайної команди kill. В такому разі, необхідно використати агресивнішу форму kill:

$ **kill -9 4747**

Обидві kill і killall візьмуть як аргумент номер, що відповідає сигналу який можна послати процесові. Звичайна kill посилає більш цивілізований сигнал SIGTERM, що вказує процесові завершити роботу, очистити пам'ять і вийти, тоді як останньому прикладі -9 означатиме потужніший SIGKILL, що нагально перериває процес. Загалом, SIGKILL використовується лише в крайніх випадках, оскільки він може призвести до псування даних.

Ви можете отримати список всіх можливих сигналів за допомогою прапорця -l (англійська "л") (*list*):

$ **kill -l**

```
 1) SIGHUP     2) SIGINT    3) SIGQUIT   4) SIGILL
 5) SIGTRAP    6) SIGABRT   7) SIGBUS    8) SIGFPE
 9) SIGKILL   10) SIGUSR1  11) SIGSEGV  12) SIGUSR2
13) SIGPIPE   14) SIGALRM  15) SIGTERM  17) SIGCHLD
18) SIGCONT   19) SIGSTOP  20) SIGTSTP  21) SIGTTIN
22) SIGTTOU   23) SIGURG   24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF  28) SIGWINCH 29) SIGIO
30) SIGPWR
```

Крім номерів сигналів, також можна використовувати їхні назви без префіксу "SIG", наприклад:

$ **killall -KILL xedit**

Ще одним застосуванням kill буде рестарт процесів. Якщо ви пошлете сигнал SIGHUP програмі, вона рестартує, перечитавши власний конфігураційний файл, якщо такий є. Це особливо корисно для того щоб заставити системним процесам перечитати файли конфігурації після їхнього редагування.

ДОДАТОК 3

Приклад програми пересилки повідомлень між паралельними процесами

```
#include "mpi.h"
main (argc, argv)
int argc;
char **argv;
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
   if (myrank==0)    /* code for process zero */
   {
     strcpy (message, "Hello, MPI");
      MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
   }
   else   /* code for process one */
   {
     MPI_Recv (message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
     printf ("received :%s:\n", message);
   }
  MPI_Finalize();
}
```

ДОДАТОК 4

Програма, що виводить кількість паралельних процесів та їх номери

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

int main(int argc, char** argv){
 int allproc, procid, namelen;
 char processor_name[MPI_MAX_PROCESSOR_NAME];
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &allproc);
 MPI_Comm_rank(MPI_COMM_WORLD, &procid);
 MPI_Get_processor_name(processor_name,&namelen);
 if (procid == 0)
 {        printf("All processes count is: %d \r\n", allproc);
 }
 printf("Process %d on %s\n", procid, processor_name);
 MPI_Finalize();
 return 0;
}
```

ДОДАТОК 5

Приклад розпаралелення ітераційного процесу

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define BUFLEN 512

int main(argc,argv)
int argc;
char *argv[];
{
int             i, myid, numprocs, next, rc, namelen;
unsigned long   j1, j2, sum=0;
char            buffer[BUFLEN],
processor_name[MPI_MAX_PROCESSOR_NAME];
double          startwtime = 0.0, endwtime;
MPI_Status      status;
int             iter=10, iter2=15000, iter3=25000, piter;
char studName[20]="";

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Get_processor_name(processor_name,&namelen);

if (myid == 0)
{
 printf("-=- Enter your surname: "); scanf("%20s", studName);
 printf("-=- Enter number of iterations: "); scanf("%u", &iter);
 printf("-=- Amount of Parallel Processes: %d\n-=- Total Iterations: %ux%dx%d\n",
numprocs, iter, iter2, iter3);
```

```
 for (i=0;i<numprocs;i++)
 MPI_Send(&iter, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
}

MPI_Recv(&iter, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
MPI_Barrier(MPI_COMM_WORLD); /* synchronization */

if (myid == 0) startwtime = MPI_Wtime(); /* time stamp of start*/

piter=iter/numprocs; /* partial iterations */

if (myid == 0) if (iter%numprocs > 0) piter=piter+(iter%numprocs);
for (i=1;i<=piter;i++)
{
 for (j1=1;j1<=iter2;j1++)
 {
 for (j2=1;j2<=iter3;j2++) sum = sum + 1;
 }
 printf("Process %d on processor %s Iteration – %d \n",myid,processor_name,i);
}
MPI_Barrier(MPI_COMM_WORLD);

if (myid == 0) endwtime = MPI_Wtime();

MPI_Finalize();

if (myid == 0) printf("-=- Wall Clock Time: %f\n", endwtime-startwtime);
}
```

ЛАБОРАТОРНА РОБОТА № 9

Тема: Асинхронні та синхронні процедури передачі MPI повідомлень між паралельними обчислювальними процесами у середовищі ОС Linux.

Мета: ознайомитися з асинхронними та синхронними процедурами передачі повідомлень, навчитися програмувати та виконувати паралельні процеси з використанням синхронних та асинхронних функцій MPI на багатопроцесорних ЕОМ під керуванням ОС Linux.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Термінологія і позначення

MPI – message passing interface – бібліотека функцій, призначена для підтримки роботи паралельних процесів у термінах передачі повідомлень.

Номер процесу – ціле додатнє число, що є унікальним атрибутом кожного процесу.

Атрибути повідомлення – номер процесу-відправника, номер процесу-одержувача й ідентифікатор повідомлення. Для них заведена структура *MPI_Status*, що містить три поля: *MPI_Source* (номер процесу відправника), *MPI_Tag* (ідентифікатор повідомлення), *MPI_Error* (код помилки); можуть бути й додаткові поля.

Ідентифікатор повідомлення (msgtag) – атрибут повідомлення, що є цілим додатнім числом, що лежить у діапазоні від 0 до 32767. Процеси поєднуються в *групи*, можуть бути вкладені групи. Усередині групи всі процеси пронумеровані. З кожною групою асоційований свій *комунікатор*. Тому при здійсненні пересилання необхідно вказати ідентифікатор групи, усередині якої здійснюється це пересилання. Всі процеси знаходяться в групі з визначеним ідентифікатором *MPI_COMM_WORLD*.

При описі процедур MPI будемо користуватися словом OUT для позначення "вихідних" параметрів, тобто таких параметрів, через які процедура повертає результати.

1.2 Загальні процедури MPI

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

MPI_Init – ініціалізація паралельної частини програми. Реальна ініціалізація для кожного додатка виконується не більше одного разу, а якщо MPI уже був ініційований, те ніякі дії не виконуються й відбувається негайне повернення з підпрограми. Всі MPI-процедури, що залишилися, можуть бути викликані тільки після *виклику MPI_*Init.

Повертається: у випадку успішного виконання – *MPI_SUCCESS*, інакше – код помилки. (Теж саме повертають майже всі інші функції).

int MPI_Finalize(void)

MPI_Finalize – завершення паралельної частини програми. Всі наступні звертання до будь-яких MPI-процедур, у тому числі до *MPI_Init*, заборонені. До моменту виклику *MPI_Finalize* деяким процесом всі дії, що вимагають його участі в

обміні повідомленнями, повинні бути завершені. Складний тип аргументів *MPI_Init* передбачений для того, щоб передавати всім процесам аргументи *main*:

```
int main(int argc, char** argv)
{
      MPI_Init(&argc, &argv);
          ...
      MPI_Finalize();
}
```

int MPI_Comm_size(MPI_Comm comm, int* size)

Визначення загального числа паралельних процесів у групі *comm*.

- *comm* – ідентифікатор групи
- OUT *size* – розмір групи

int MPI_Comm_rank(MPI_Comm comm, int* rank)

Визначення номера процесу в групі *comm*. Значення, що повертає за адресою *&rank*, лежить у діапазоні від 0 до *size_of_group-1*.

- *comm* – ідентифікатор групи
- OUT *rank* – номер викликаючого процесу в групі *comm*

double MPI_Wtime(void)

Функція повертає астрономічний час у секундах (речовинне число), що пройшло з деякого моменту в минулому. Гарантується, що цей момент не буде змінений за час існування процесу.

1.3 Прийом/передача повідомлень із блокуванням (синхронні процедури)

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

- *buf* – адреса початку буфера посилки повідомлення
- *count* – число переданих елементів у повідомленні
- *datatype* – тип переданих елементів
- *dest* – номер процесу-одержувача
- *msgtag* – ідентифікатор повідомлення
- *comm* – ідентифікатор групи

Блокуюча посилка повідомлення з *ідентифікатором* msgtag, що складається *з* count елементів *типу* datatype, процесу з *номером* dest. Всі елементи повідомлення розташовані підряд у буфері *buf*. Значення *count* може бути нулем. Тип переданих елементів *datatype* повинен вказуватися за допомогою визначених констант типу. Дозволяється передавати повідомлення самому собі.

Блокування гарантує коректність повторного використання всіх параметрів після повернення з підпрограми. Вибір способу здійснення цієї гарантії: копіювання в проміжний буфер або безпосередньо передача процесу *dest*, залишається за MPI. Слід зазначити, що повернення з підпрограми *MPI_Send* не означає ні того, що повідомлення вже передане процесу *dest*, ні того, що повідомлення покинуло процесорний елемент, на якому виконується процес, що виконав *MPI_Send*.

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Status *status)

- OUT *buf* – адреса початку буфера прийому повідомлення
- *count* – максимальне число елементів у прийнятому повідомленні
- *datatype* – тип елементів прийнятого повідомлення
- *source* – номер процесу-відправника
- *msgtag* – ідентифікатор прийнятого повідомлення
- *comm* – ідентифікатор групи
- OUT *status* – параметри прийнятого повідомлення

Прийом повідомлення з ідентифікатором *msgtag* від процесу *source* із блокуванням. Число елементів у прийнятому повідомленні не повинне перевершувати значення *count*. Якщо число прийнятих елементів менше значення *count*, то гарантується, що в буфері *buf* зміняться тільки елементи, що відповідають елементам прийнятого повідомлення. Якщо потрібно довідатися точне число елементів у повідомленні, то можна скористатися підпрограмою *MPI_Probe*.

Блокування гарантує, що після повернення з підпрограми всі елементи повідомлення прийняті й розташовані в буфері *buf*.

Як номер процесу-відправника можна вказати визначену константу *MPI_ANY_SOURCE* – ознака того, що прийматимуться повідомлення від будь-якого процесу. Як ідентифікатор прийнятого повідомлення можна вказати константу *MPI_ANY_TAG* – ознака того, що прийматимуться повідомлення з будь-яким ідентифікатором.

Якщо процес посилає два повідомлення іншому процесу й обоє ці повідомлення відповідають тому самому виклику *MPI_Recv*, те першим буде прийняте те повідомлення, що було відправлено раніше.

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

- *status* – параметри прийнятого повідомлення
- *datatype* – тип елементів прийнятого повідомлення
- OUT *count* – число елементів повідомлення

За значенням параметра *status* дана підпрограма визначає число вже прийнятих (після звертання до *MPI_Recv*) або прийнятого (після звертання до *MPI_Probe* або *MPI_Iprobe*) елементів повідомлення типу *datatype*.

int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status)

- *source* – номер процесу-відправника або *MPI_ANY_SOURCE*
- *msgtag* – ідентифікатор очікуваного повідомлення або *MPI_ANY_TAG*
- *comm* – ідентифікатор групи
- OUT *status* – параметри виявленого повідомлення

Одержання інформації про структуру очікуваного повідомлення із блокуванням. Повернення з підпрограми не відбудеться доти, поки повідомлення з потрібним ідентифікатором і номером процесу-відправника не буде доступно для одержання. Атрибути доступного повідомлення можна визначити звичайним образом за допомогою параметра *status*. Варто звернути увагу, що підпрограма визначає тільки факт приходу повідомлення, але реально його не приймає.

1.4 Прийом/передача повідомлень без блокування (асинхронні процедури)

Перевага функцій прийому/передачі без блокування особливо помітна, коли час передачі даних між паралельними процесами є досить значним і передача/прийом повідомлень чергується з обчислювальними операціями над цими даними.

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

- *buf* – адреса початку буфера посилки повідомлення
- *count* – число переданих елементів у повідомленні
- *datatype* – тип переданих елементів
- *dest* – номер процесу-одержувача
- *msgtag* – ідентифікатор повідомлення
- *comm* – ідентифікатор групи
- OUT *request* – ідентифікатор асинхронної передачі

Передача повідомлення, аналогічна *MPI_Send*, однак повернення з підпрограми відбувається відразу після ініціалізації процесу передачі без очікування обробки всього повідомлення, що перебуває в буфері *buf*. Це означає, що не можна повторно використати даний буфер для інших цілей без одержання додаткової інформації про завершення даної посилки. Закінчення процесу передачі (тобто того моменту, коли можна перевикористати буфер *buf* без побоювання зіпсувати передане повідомлення) можна визначити за допомогою параметра *request* і процедур *MPI_Wait* і *MPI_Test*.

Повідомлення, відправлене кожною із процедур *MPI_Send* і *MPI_Isend*, може бути прийнято кожною із процедур *MPI_Recv* і *MPI_Irecv*.

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

- OUT *buf* – адреса початку буфера прийому повідомлення
- *count* – максимальне число елементів у прийнятому повідомленні
- *datatype* – тип елементів прийнятого повідомлення
- *source* – номер процесу-відправника
- *msgtag* – ідентифікатор прийнятого повідомлення
- *comm* – ідентифікатор групи
- OUT *request* – ідентифікатор асинхронного прийому повідомлення

Прийом повідомлення, аналогічний *MPI_Recv*, однак повернення з підпрограми відбувається відразу після ініціалізації процесу прийому без очікування одержання повідомлення в буфері *buf*. Закінчення процесу прийому можна визначити за допомогою параметра *request* і процедур *MPI_Wait* і *MPI_Test*.

int MPI_Wait(MPI_Request *request, MPI_Status *status)

- *request* – ідентифікатор асинхронного прийому або передачі
- OUT *status* – параметри повідомлення

Очікування завершення асинхронних процедур *MPI_Isend* або *MPI_Irecv*, асоційованих з ідентифікатором *request*. У випадку прийому, атрибути й довжину отриманого повідомлення можна визначити звичайним образом за допомогою параметра *status*.

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)

- *count* – число ідентифікаторів
- *requests* – масив ідентифікаторів асинхронного прийому або передачі
- OUT *statuses* – параметри повідомлень

Виконання процесу блокується доти, поки всі операції обміну, асоційовані із зазначеними ідентифікаторами, не будуть завершені. Якщо під час однієї або декількох операцій обміну виникли помилки, то поле помилки в елементах масиву *statuses* буде встановлено у відповідне значення.

int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status)

- *count* – число ідентифікаторів
- *requests* – масив ідентифікаторів асинхронного прийому або передачі
- OUT *index* – номер завершеної операції обміну
- OUT *status* – параметри повідомлень

Виконання процесу блокується доти, поки яка-небудь операція обміну, асоційована із зазначеними ідентифікаторами, не буде завершена. Якщо кілька операцій можуть бути завершені, то випадковим чином вибирається одна з них. Параметр *index* містить номер елемента в масиві *requests*, що містить ідентифікатор завершеної операції.

int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

- *incount* – число ідентифікаторів
- *requests* – масив ідентифікаторів асинхронного прийому або передачі
- OUT *outcount* – число ідентифікаторів операцій, що завершилися, обміну
- OUT *indexes* – масив номерів операції, що завершилася, обміну
- OUT *statuses* – параметри повідомлень, що завершилися

Виконання процесу блокується доти, поки принаймні одна з операцій обміну, асоційованих із зазначеними ідентифікаторами, не буде завершена. Параметр *outcount* містить число завершених операцій, а перші *outcount* елементів масиву *indexes* містять номера елементів масиву *requests* з їхніми ідентифікаторами. Перші *outcount* елементів масиву *statuses* містять параметри завершених операцій.

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

- *request* – ідентифікатор асинхронного прийому або передачі
- OUT *flag* – ознака завершення операції обміну
- OUT *status* – параметри повідомлення

Перевірка завершення асинхронних процедур *MPI_Isend* або *MPI_Irecv*, асоційованих з ідентифікатором *request*. У параметрі *flag* повертає значення 1, якщо відповідна операція завершена, і значення 0 у противному випадку. Якщо завершено процедуру прийому, то атрибути й довжину отриманого повідомлення можна визначити звичайним чином за допомогою параметра *status*.

int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses)

- *count* – число ідентифікаторів
- *requests* – масив ідентифікаторів асинхронного прийому або передачі
- OUT *flag* – ознака завершення операцій обміну
- OUT *statuses* – параметри повідомлень

У параметрі *flag* повертає значення *1*, якщо всі операції, асоційовані із зазначеними ідентифікаторами, завершені (із вказівкою параметрів повідомлень у масиві *statuses*). У іншому випадку повертається *0*, а елементи масиву *statuses* невизначені.

int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)

- *count* – число ідентифікаторів
- *requests* – масив ідентифікаторів асинхронного прийому або передачі
- OUT *index* – номер завершеної операції обміну
- OUT *flag* – ознака завершення операції обміну
- OUT *status* – параметри повідомлення

Якщо до моменту виклику підпрограми хоча б одна з операцій обміну завершилася, то в параметрі *flag* повертається значення *1*, *index* містить номер відповідного елемента в масиві *requests*, а *status* – параметри повідомлення.

int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

- *incount* – число ідентифікаторів
- *requests* – масив ідентифікаторів асинхронного прийому або передачі
- OUT *outcount* – число ідентифікаторів операцій, що завершилися, обміну
- OUT *indexes* – масив номерів операції, що завершилася, обміну
- OUT *statuses* – параметри операцій, що завершилися

Дана підпрограма працює так само, як і *MPI_Waitsome*, за винятком того, що повернення відбувається негайно. Якщо жодна із зазначених операцій не завершилася, то значення *outcount* буде дорівнює нулю.

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

- *source* – номер процесу-відправника або *MPI_ANY_SOURCE*
- *msgtag* – ідентифікатор очікуваного повідомлення або *MPI_ANY_TAG*
- *comm* – ідентифікатор групи
- OUT *flag* – ознака завершення операції обміну
- OUT *status* – параметри виявленого повідомлення

Одержання інформації про надходження й структуру очікуваного повідомлення без блокування. У параметр *flag* повертає значення *1*, якщо повідомлення з потрібними атрибутами вже може бути прийнято (у цьому випадку її дія повністю аналогічна *MPI_Probe*), і значення *0*, якщо повідомлення із зазначеними атрибутами ще немає.

1.4 Визначені константи типу елементів повідомлень

Константи MPI	Тип в C
MPI_CHAR	signed char
MPI_SHORT	signed int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Інші визначені типи

MPI_Status – структура; атрибути повідомлень; містить три обов'язкові поля:

- *MPI_Source* (номер процесу відправника)
- *MPI_Tag* (ідентифікатор повідомлення)
- *MPI_Error* (код помилки)

MPI_Request – системний тип; ідентифікатор операції посилки-прийому повідомлення

MPI_Comm – системний тип; ідентифікатор групи (комунікатора)

- *MPI_COMM_WORLD* – зарезервований ідентифікатор групи, що складає їхніх всіх процесів програми

Константи-заглушки

- *MPI_COMM_NULL*
- *MPI_DATATYPE_NULL*
- *MPI_REQUEST_NULL*

Константа невизначеного значення

- *MPI_UNDEFINED*

Глобальні операції

MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD

Будь-який процес/ідентифікатор

MPI_ANY_SOURCE
MPI_ANY_TAG

Код успішного завершення процедури
MPI_SUCCESS

2. ХІД РОБОТИ

2.1 Уважно ознайомтеся із теоретичною частиною лабораторної роботи.

2.2 Уважно ознайомтеся з наступним кодом програми, щоб зрозуміти спосіб пересилки повідомлень між процесами. Наберіть код програми без коментарів у текстовому редакторі. відкомпілюйте та виконайте на віддаленому комп'ютері на двох процесорах, так як було показано у Лабораторній роботі №8. Результат виконання програми скопіюйте у звіт.

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


  /* Ідентифікатори повідомлень */
#define tagFloatData  1
#define tagDoubleData 2

  /* Цей макрос створений для зручності, */
  /* він дозволяє вказувати довжину масиву в кількості комірок*/
#define ELEMS(x)  ( sizeof(x) / sizeof(x[0]) )

int main( int argc, char **argv )
{
    int size, rank, count;
    float floatData[10];
    double doubleData[20];
    MPI_Status status;

      /* Ініціюємо бібліотеку */
    MPI_Init( &argc, &argv );

      /* Визначаємо кількість завдань у запущеному додатку */
    MPI_Comm_size( MPI_COMM_WORLD, &size );

      /* ... і свій власний номер: від 0 до (size-1) */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

      /* користувач повинен запустити рівно два завдання, інакше буде помилка */
    if( size != 2 ) {

        /* завдання з номером 0 повідомляють користувача про помилку */
```

```
    if( rank==0 )
        printf("Error: two processes required instead of %d, abort\n",
            size );

      /* Всі завдання-абоненти коммунікатора MPI_COMM_WORLD
       * будуть очыкувати, поки завдання 0 не виведе повідомлення.
       */
    MPI_Barrier( MPI_COMM_WORLD );
      /* Без крапки синхронізації може виявитися, що одне із завдань
       * викличе MPI_Abort раніше, ніж встигне виконатись printf()
       * у завданні 0, MPI_Abort негайно примусово завершить
       * всі завдання й повідомлення виведене не буде
       */
      /* всі завданння аварійно завершують роботу */
    MPI_Abort(
        MPI_COMM_WORLD,  /* Комунікатор області зв'язку, на яку */
                         /*    поширюється дія помилки */
        MPI_ERR_OTHER ); /* Цілочисельний код помилки */
    return -1;
}

if( rank==0 ) {
  /* Завдання 0 щось таке передає завданню 1 */

    MPI_Send(
        floatData,        /* 1) адреса переданого масиву */
        5,       /* 2) скільки: 5 осередків, тобто floatData[0]..floatData[4] */
        MPI_FLOAT,        /* 3) тип комірки */
        1,                /* 4) кому: завданню 1 */
        tagFloatData,     /* 5) ідентифікатор повідомлення */
        MPI_COMM_WORLD ); /* 6) комунікатор області зв'язку, через яку */
                          /*    відбувається передача */

      /* і ще одна передача: дані іншого типу */
    MPI_Send( doubleData, 6, MPI_DOUBLE, 1, tagDoubleData, MPI_COMM_WORLD );

} else {
  /* Завдання 1 щось таке приймає від завдання 0 */

      /* чекаємо повідомлення й поміщаємо дані, що прийшли, у буфер */
    MPI_Recv(
        doubleData,     /* 1) адреса масиву, куди записується прийняте */
```

```
            ELEMS( doubleData ), /* 2) фактична довжина прийнятих даних */
                                 /*    масиву в числі комірок */
            MPI_DOUBLE,      /* 3) повідомляємо MPI, що повідомлення, що прийшло, */
                             /*    складається із чисел типу 'double' */
            0,               /* 4) від кого: від завдання 0 */
            tagDoubleData,   /* 5) очікуємо повідомлення з таким ідентифікатором */
            MPI_COMM_WORLD, /* 6) комунікатор області зв'язку, через яку */
                             /*    очікується прийом повідомлення */
            &status );       /* 7) сюди буде записаний статус завершення прийому */

          /* Обчислюємо фактично прийнята кількість даних */
        MPI_Get_count(
            &status,         /* статус завершення */
            MPI_DOUBLE,      /* повідомляємо MPI, що повідомлення, що прийшло, */
                             /*    складається із чисел типу 'double' */
            &count );        /* сюди буде записаний результат */

          /* Виводимо фактичну довжину прийнятого на екран */
        printf("[DOUBLE] Received %d elems\n", count );

          /* Аналогічно приймаємо повідомлення з даними типу float
           * Зверніть увагу: завдання-приймач має можливість
           * приймати повідомлення не в тім порядку, у якому вони
           * відправлялися, якщо ці повідомлення мають різні ідентифікатори
           */
        MPI_Recv( floatData, ELEMS( floatData ), MPI_FLOAT,
            0, tagFloatData, MPI_COMM_WORLD, &status );
        MPI_Get_count( &status, MPI_FLOAT, &count );
        printf("[FLOAT] Received %d elems\n", count );

      }

      /* Обидва завдання завершують виконання */
    MPI_Finalize();
    return 0;
}
```

Результат виконання програми

```
[DOUBLE] Received 6 elems
[FLOAT] Received 5 elems
```

2.3 Розглянемо тепер дві програми, що виконують одну і ту ж задачу пересилки числового значення від одного процесу до іншого, однак перша програма

з використанням синхронних процедур send/receive, а інша – асинхронних процедур isend/ireceive.

Наберіть код обох програм у текстовому редакторі та збережіть у вигляді вихідних програм на С. Змініть у програмах значення змінних tag=1234 (ідентифікатор повідомлення) та buffer=5678 (значення, що пересилається) відповідно до вашого варіанту. Врахуйте в програмі тип даних що передаються. Відкомпілюйте ці обидві програми та виконайте на двох процесорах.

Результат виконання обох програм скопіюйте у звіт. Проаналізуйте відмінності роботи кожної програми та занотуйте у звіт.

а) Пересилка повідомлення з використанням синхронних процедур.

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

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;
    if(myid == source){
      buffer=5678;
      MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
      printf("synchronous: processor %d  sent %d\n",myid,buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("synchronous: processor %d  got %d , message tag is %d\n", myid,
buffer, tag);
    }
    MPI_Finalize();
}
```

б) Пересилка повідомлення з використанням асинхронних процедур.

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

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;
    request=MPI_REQUEST_NULL;
    if(myid == source){
      buffer=5678;

MPI_Isend(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD,&request);
    }
    if(myid == destination){
        MPI_Irecv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&request);
    }
    MPI_Wait(&request,&status);
    if(myid == source){
      printf("asynchronous: processor %d  sent %d\n",myid,buffer);
    }
    if(myid == destination){
      printf("asynchronous:  processor  %d    got  %d,  message  tag  is
%d\n",myid,buffer,tag);
    }
    MPI_Finalize();
}
```

3. ЗМІСТ ЗВІТУ

3.1 Тема, мета лабораторної роботи
3.2 Короткі теоретичні відомості
3.3 Опис виконання лабораторної роботи згідно пунктів 2.2-2.3
3.4 Відповідь на одне контрольне запитання згідно номера варіанту (порядковий номер у списку журналу групи)
3.5 Висновки

4. КОНТРОЛЬНІ ПИТАННЯ

4.1 Що таке MPI?
4.2 Які атрибути повідомлення MPI?
4.3 Якою функцією завершується MPI-програма?
4.4 Яка функція повертає номер процесу?
4.5 Напишіть MPI програму з мінімальним набором коду, необхідного для її роботи.
4.6 Як ідентифікуються повідомлення та процеси MPI?
4.7 Як визначити кількість процесів у групі?
4.8 Що означає константа *MPI_ANY_SOURCE*?
4.9 Яка функція часу використовується у MPI?
4.10 Які функції прийому передачі наявні в MPI?
4.11 Опишіть функцію *MPI_Send* та її параметри.
4.12 Чому функція *MPI_ISend* називається асинхронною функцією?
4.13 Опишіть функцію *MPI_IRecv* та її параметри. Як вона використовується?
4.14 Чому функція *MPI_Recv* називається блокуючою функцією?
4.15 Скільки разів можна виконувати ініціалізацію MPI-програми?
4.16 Опишіть функції *MPI_Waitall* та *MPI_Waitany*.
4.17 Що означає константа *MPI_ANY_TAG*?
4.18 Опишіть команду компілювання MPI програм.
4.19 Чому функція *MPI_Send* називається блокуючою функцією?
4.20 Які є типи даних в MPI і як формується їх назва?
4.21 Опишіть функцію та *MPI_Test, MPI_Testal, MPI_Testany, MPI_Testsome*.
4.22 Опишіть функцію *MPI_Get_count* та *MPI_Probe*.
4.23 Які є константи в MPI для позначення будь-якого процесу та ідентифікатора?
4.24 Опишіть команду запуску MPI програм.
4.25 Опишіть функцію *MPI_Wait* та *MPI_Waitsome* її параметри.
4.26 Який буде результат виконання функції MPI_Get_processor_name(processor_name,&namelen) ?
4.27 Що робить функція MPI_Finalize() ?
4.28 Який буде результат виконання функції MPI_Comm_rank (MPI_COMM_WORLD, &myrank) ?
4.29 Опишіть структуру *MPI_Status.*
4.30 Який буде результат виконання функції MPI_Comm_size(MPI_COMM_WORLD, &allproc) ?

Варіанти завдань

№ в журналі	buffer (значення, що пересилається)	Тип даних	tag (ідентифікатор повідомлення)
1	"a"	MPI_CHAR	908
2	8373	MPI_SHORT	458
3	5977	MPI_INT	460
4	753922	MPI_LONG	541
5	253	MPI_UNSIGNED_CHAR	744
6	50949	MPI_UNSIGNED_SHORT	657
7	100	MPI_UNSIGNED	611
8	200	MPI_UNSIGNED_LONG	991
9	82,96	MPI_FLOAT	171
10	19,7	MPI_DOUBLE	789
11	580,3	MPI_LONG_DOUBLE	685
12	"b"	MPI_CHAR	567
13	1475	MPI_SHORT	399
14	7931	MPI_INT	574
15	7669441	MPI_LONG	261
16	215	MPI_UNSIGNED_CHAR	358
17	47684	MPI_UNSIGNED_SHORT	406
18	90	MPI_UNSIGNED	784
19	120	MPI_UNSIGNED_LONG	569
20	69,26	MPI_FLOAT	922
21	442,4	MPI_DOUBLE	993
22	9053,2	MPI_LONG_DOUBLE	752
23	"c"	MPI_CHAR	575
24	1964	MPI_SHORT	759
25	2459	MPI_INT	761
26	339017	MPI_LONG	188
27	244	MPI_UNSIGNED_CHAR	234
28	33525	MPI_UNSIGNED_SHORT	248
29	95	MPI_UNSIGNED	621
30	138	MPI_UNSIGNED_LONG	144

5. СПИСОК ЛІТЕРАТУРИ

5.1 Камерон Хьюз, Трейси Хьюз. Параллельное и распределенное программирование с использованием C++. : Пер. с англ. – М. : Издательский дом "Вильямс", 2004. – 627 с.

5.2 Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 512 с.

5.3 Воеводин В.В. Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. – 608 с.

5.4 А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону. Издательство ООО «ЦВВР», 2003, 208 с.

5.5 Джин Бэкон, Тим Харрис. Операционные системы. Параллельные и распределенные системы. – Питер: Издательская группа BHV, 2004. – 800 с.

5.6 www.mpi-forum.org

5.7 www.parallel.ru

Навчально-методична література

Шимчук Г.В., Маєвський О.В., Назаревич О.Б.

Методичні вказівки до виконання
лабораторних робіт
з дисципліни

«Розподілені системи моніторингу та керування»

для студентів освітнього рівня «бакалавр»
спеціальності 125 «Кібербезпека»

Компютерне макетування *А.П. Катрич*

Формат 60х90/16. Обл. вид. арк. 5,77. Тираж 10 прим. Зам. № 2697

Видавництво Тернопільського національного
технічного університету імені Івана Пулюя.
46001, м. Тернопіль, вул. Руська, 56.
Свідоцтво суб'єкта видавничої справи ДК № 4226 від 08.12.11.