

MPI. ТЕОРЕТИЧНІ ВІДОМОСТІ. – 1с

ПРОСТИЙ ПРИКЛАД ПРОГРАМИ З ВИКОРИСТАННЯМ MPI – 7с

РОЗПАРАЛЕЛЕННЯ МНОЖЕННЯ МАТРИЦЬ – 8с

ПРИКЛАД ПРОГРАМИ МНОЖЕННЯ МАТРИЦІ НА ВЕКТОР З ВИКОРИСТАННЯМ MPI – 11с

ІНСТАЛЯЦІЯ MPICH ДЛЯ ВИКОРИСТАННЯ MPI – 16с

MPI. ТЕОРЕТИЧНІ ВІДОМОСТІ.

У обчислювальних системах з розподіленою пам'яттю процесори працюють незалежно один від одного. Для організації паралельних обчислень в таких умовах необхідно мати можливість розподіляти обчислювальне навантаження і організувати інформаційну взаємодію (передачу даних) між процесорами. Одним з способів взаємодії між паралельними процесами є передача повідомлень між ними, що відображено в самій назві технології MPI (message passing interface) – інтерфейс передачі повідомлень.

Для розподілу обчислень між процесорами необхідно проаналізувати алгоритм розв'язку задачі, виділити інформаційно незалежні фрагменти обчислень, виконати їх програмну реалізацію і розмістити отримані частини програми на різних процесорах. В MPI використовуються простіший підхід - для виконання завдання розробляється одна програма, яка запускається одночасно на виконання на всіх наявних процесорах. При цьому для того, щоб уникнути ідентичності обчислень на різних процесорах, можна, по-перше, підставляти різні дані для програми на різних процесорах, а по-друге, використовувати наявні в MPI засоби для ідентифікації процесора, на якому виконується програма (тим самим надається можливість організувати відмінності в обчисленнях залежно від використовуваного програмою процесора).

Інтерфейс MPI підтримує реалізацію програм для MIMD систем (Multiple Instructions Multiple Data), проте відлагодження таких програм є не тривіальною задачею. Тому на практиці для написання програм в більшості випадків застосовується SPMD (single program multiple processes) модель паралельного програмування - "одна програма безліч процесів".

1. MPI: основні поняття і визначення

MPI - це стандарт, якому повинні задовольняти засоби організації передачі повідомлень. Крім того MPI - це програмні засоби, які забезпечують можливість передачі повідомлень і при цьому відповідають всім вимогам стандарту MPI. Згідно стандарту, ці програмні засоби повинні бути організовані у вигляді бібліотек програмних функцій (бібліотеки MPI) і повинні бути доступні для найширше використовуваних алгоритмічних мов C і Fortran. Подібну "подвійність" MPI слід враховувати при використанні термінології. Як правило, аббревіатура MPI застосовується при згадці стандарту, а поєднання "бібліотека MPI" указує на ту або іншу програмну реалізацію стандарту. Оскільки достатньо часто скорочено позначення MPI використовується і для бібліотек MPI, тому для правильної інтерпретації терміну, слід враховувати контекст.

1.1. Поняття паралельної програми. Під паралельною програмою в MPI розуміється множина одночасно виконуваних процесів. Процеси можуть виконуватися як на різних процесорах, так і на одному процесорі можуть виконуватися і декілька процесів (в цьому випадку їх виконання здійснюється в режимі розділення часу). У граничному випадку для виконання паралельної програми може використовуватися один процесор - як правило, такий спосіб застосовується для початкової перевірки правильності паралельної програми.

Кожен процес паралельної програми породжується на основі копії одного і того ж програмного коду (модель SPMP). Даний програмний код, представлений у вигляді виконуваної програми, повинен бути доступний у момент запуску паралельної програми на всіх використовуваних процесорах. Початковий програмний код для виконуваної програми

розробляється на алгоритмічних мовах C або Fortran із застосуванням тієї або іншої реалізації бібліотеки MPI.

Кількість процесів і використовуваних процесорів визначається у момент запуску паралельної програми засобами середовища виконання MPI-програм і в ході обчислень не може змінюватися без застосування спеціальних засобів динамічного породження процесів і управління ними, згідно з стандартом MPI версії 2.0. Всі процеси програми послідовно перенумеровані від 0 до $p-1$, де p є загальна кількість процесів. Номер процесу іменується рангом процесу.

1.2. Операції передачі даних. Основу MPI складають операції передачі повідомлень. Серед передбачених у складі MPI функцій розрізняються парні (point-to-point) операції між двома процесами і колективні (collective) комунікаційні дії для одночасної взаємодії декількох процесів.

Для виконання парних операцій можуть використовуватися різні режими передачі (синхронний, блокуючий і ін.). Повний розгляд можливих режимів передачі розглядається нижче.

1.3. Поняття комунікаторів. Процеси паралельної програми об'єднуються в групи. Іншим важливим поняттям MPI, що описує набір процесів, є поняття комунікатора. Під комунікатором в MPI розуміється спеціально створений службовий об'єкт, який об'єднує групу процесів і ряд додаткових параметрів (контекст), використовуваних при виконанні операцій передачі даних.

Парні операції передачі даних можуть бути виконані між будь-якими процесами одного і того ж комунікатора, а в колективних операціях беруть участь всі процеси комунікатора. Як результат, вказання використовуваного комунікатора є обов'язковим для операцій передачі даних в MPI.

Логічна топологія ліній зв'язку між процесами має структуру повного графа (незалежно від наявності реальних фізичних каналів зв'язку між процесорами).

У MPI є можливість представлення множини процесів у вигляді ґраток довільної розмірності. Крім того, в MPI є засоби і для формування логічних (віртуальних) топологій будь-якого необхідного типу.

Під час обчислень можуть створюватися нові і видалятися існуючі групи процесів і комунікаторів. Один і той же процес може належати різним групам і комунікаторам. Всі наявні в паралельній програмі процеси входять до складу конструйованого за замовчуванням комунікатора з ідентифікатором MPI_COMM_WORLD.

У версії 2.0 стандарту з'явилася можливість створювати глобальні комунікатори (intercommunicator), об'єднуючи в одну структуру пару груп при необхідності виконання колективних операцій між процесами з різних груп.

1.4. Типи даних. При виконанні операцій передачі повідомлень для вказівки передаваних або отримуваних даних у функціях MPI необхідно вказувати тип даних, що пересилаються. MPI містить великий набір базових типів даних, багато в чому співпадаючих з типами даних в алгоритмічних мовах C і Fortran. Крім того, в MPI є можливість створення нових похідних типів даних для точнішого і коротшого опису вмісту повідомлень, що пересилаються.

2. Введення в розробку паралельних програм з використанням MPI

Мінімально необхідний набір функцій MPI, достатній для розробки порівняно простих паралельних програм.

2.1. Ініціалізація і завершення MPI-програм. Першою функцією MPI, що викликається, повинна бути функція

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

argc - вказівник на кількість параметрів командного рядка;

argv - параметри командного рядка.

яка використовується для ініціалізації середовища виконання MPI-програми. Параметрами функції є кількість аргументів в командному рядку і адреса вказівника на масив параметрів командного рядка.

Останньою функцією MPI, що викликається, обов'язково повинна бути функція:

int MPI_Finalize(void).

Структура паралельної програми з використанням MPI повинна мати такий вигляд:

```
#include "mpi.h"
int main(int argc, char *argv[])
{
    <програмний код без використання функцій MPI>
    MPI_Init(&argc, &argv);
    <програмний код з використанням функцій MPI>
    MPI_Finalize();
    <програмний код без використання функцій MPI>
    return 0;
}
```

Варто зазначити:

1. Файл *mpi.h* містить визначення іменованих констант, прототипів функцій і типів даних бібліотеки MPI.
2. Функції *MPI_Init* і *MPI_Finalize* є обов'язковими і повинні бути виконані (і лише один раз) кожним процесом паралельної програми.
3. Перед викликом *MPI_Init* може бути використана функція *MPI_Initialized* для визначення того, чи був раніше виконаний виклик *MPI_Init*, а після виклику *MPI_Finalize* - *MPI_Finalized* аналогічного призначення.

Розглянуті приклади функцій дають представлення синтаксису іменування функцій в MPI. Імені функції передують префікс MPI; далі одне або декілька слів назви; перше слово в імені функції починається із заголовного символу; слова розділяються знаком підкреслення. Назви функцій MPI, як правило, пояснюють призначення виконуваних функцією дій.

2.2. Визначення кількості і рангу процесів. Визначення кількості процесів у виконуваний паралельній програмі здійснюється за допомогою функції:

*int MPI_Comm_size(MPI_Comm comm, int *size), де*

comm - комунікатор, розмір якого визначається;

size - визначена кількість процесів в комунікаторі.

Для визначення рангу процесу використовується функція:

*int MPI_Comm_Rank(MPI_Comm comm, int *rank), де*

comm - комунікатор, в якому визначається ранг процесу;

rank - ранг процесу в комунікаторі.

Як правило, виклик функцій *Mpi_comm_size* і *Mpi_comm_rank* виконується відразу після *Mpi_init* для отримання загальної кількості процесів і рангу поточного процесу:

```
#include "mpi.h"
int main(int argc, char *argv[])
{
    int ProcNum, ProcRank;
    <програмний код без використання функцій MPI>
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    <програмний код з використанням функцій MPI>
    MPI_Finalize();
    <програмний код без використання функцій MPI>
    return 0;
}
```

Варто зазначити:

1. Комунікатор `MPI_COMM_WORLD`, як наголошувалося раніше, створюється за замовчуванням і представляє всі процеси виконуваної паралельної програми.
2. Ранг, що отримується за допомогою функції `MPI_Comm_rank`, є рангом процесу, що виконав виклик цієї функції, тобто змінна `ProcRank` прийме різні значення у різних процесів.

2.3. Передача повідомлень. Для передачі повідомлення процес-відправник повинен виконати функцію:

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

buf - адреса буфера пам'яті, в якому розташовуються дані повідомлення, що відправляється;

count - кількість елементів даних в повідомленні;

type - тип елементів даних повідомлення, що пересилається;

dest - ранг процесу, якому відправляється повідомлення;

tag - значення-тег, що використовується для ідентифікації повідомлення;

comm - комунікатор, в рамках якого виконується передача даних.

Для вказання типу даних, що пересилаються, в MPI є ряд базових типів, повний список яких наведений в таблиці

Базові типи даних MPI для алгоритмічної мови C

Тип даних MPI	Тип даних C
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	Double
MPI_FLOAT	Float
MPI_INT	Int
MPI_LONG	Long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	Short

MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Варто зазначити:

1. Повідомлення, що відправляється, визначається шляхом задання блоку пам'яті (буфера), в якому це повідомлення розташовується. Використовувана для цього тріада (*buf*, *count*, *type*) входить до складу параметрів практично всіх функцій передачі даних.

2. Процеси, між якими виконується передача даних обов'язково повинні належати комунікатору, що вказується у функції *MPI_Send*.

3. Параметр *tag* використовується тільки при необхідності розрізнення передаваних повідомлень, інакше, як значення параметра, може бути використане довільне додатнє ціле число.

Відразу після завершення виконання функції *MPI_Send* процес-відправник може почати повторно використовувати буфер пам'яті, в якому розташовувалося повідомлення, що відправлялося. Також необхідно пам'ятати, що у момент завершення функції *MPI_Send* стан самого повідомлення, що пересилається, може бути абсолютно різним: повідомлення може розташовуватися в процесі-відправнику, може знаходитися в стані передачі, може зберігатися в процесі-одержувачі або ж може бути прийнято процесом-одержувачем за допомогою функції *MPI_Recv*. Тим самим, завершення функції *MPI_Send* лише означає, що операція передачі почала виконуватися і пересилка повідомлення рано чи пізно буде виконана.

2.4. Прийом повідомлень. Для прийому повідомлення процес-одержувач повинен виконати функцію:

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

buf, *count*, *type* - буфер пам'яті для прийому повідомлення, призначення кожного окремого параметра відповідає опису в *MPI_Send*.

source - ранг процесу, від якого повинен бути виконаний прийом повідомлення.

tag - тег повідомлення, яке повинне бути прийняте для процесу.

comm - комунікатор, в рамках якого виконується передача даних.

status - вказівник на структуру даних з інформацією про результат виконання операції прийому даних.

Варто зазначити:

1. Буфер пам'яті повинен бути достатнім для прийому повідомлення. При браку обсягу пам'яті частина повідомлення буде втрачена і в коді завершення функції буде зафіксована помилка переповнення. З іншого боку, повідомлення, що приймається, може бути коротшим від розміру приймального буфера. У такому разі зміняться тільки області буфера, використані прийнятим повідомленням.

2. Типи елементів повідомлення, що передаються і приймаються, повинні співпадати.

3. При необхідності прийому повідомлення від будь-якого процесу-відправника для параметра *source* може бути вказане значення *MPI_ANY_SOURCE* (на відміну від функції передачі *MPI_Send*, яка посиляє повідомлення строго певного адресата).

4. При необхідності прийому повідомлення з будь-яким тегом для параметра *tag* може бути вказане значення *MPI_ANY_TAG* (знову-таки, при використанні функції *MPI_Send* повинне бути вказане конкретне значення тега).

5. На відміну від параметрів "процес-одержувач" і "тег", параметр "комунікатор" не має значення, що означає "будь-який комунікатор".
6. Параметр `status` дозволяє визначити ряд характеристик прийнятого повідомлення.
7. `Status.MPI_SOURCE` - ранг процесу - відправника прийнятого повідомлення.
8. `Status.MPI_TAG` - тег прийнятого повідомлення.

Значення змінної `status` дозволяє визначити кількість елементів даних в прийнятому повідомленні за допомогою функції:

*int MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count), де*

status - статус операції `MPI_Recv`;

type - тип прийнятих даних;

count - кількість елементів даних в повідомленні.

Виклик функції *Mpi_Recv* не зобов'язаний бути узгодженим з часом виклику відповідної функції передачі повідомлення *MPI_Send* - прийом повідомлення може бути ініційований до моменту, в момент чи після моменту початку відправки повідомлення.

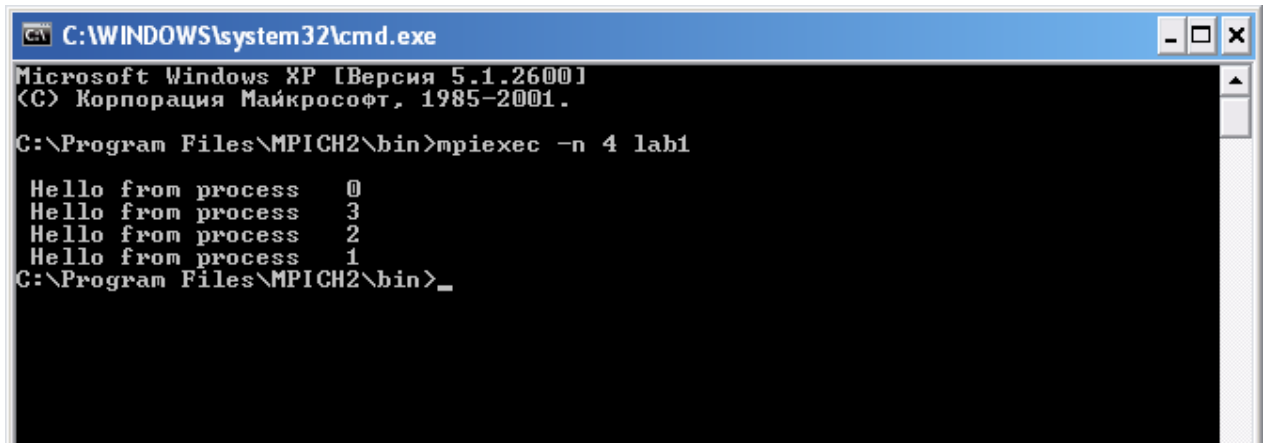
Після закінчення виконання функції `MPI_Recv` в заданому буфері пам'яті буде розташоване прийняте повідомлення. Принциповий момент полягає в тому, що функція `MPI_Recv` є блокуючою для процесу-одержувача, тобто його виконання припиняється до завершення роботи функції. Таким чином, якщо, по якихось причинах очікуване для прийому повідомлення буде відсутнє, виконання паралельної програми буде заблоковано.

ПРОСТИЙ ПРИКЛАД ПРОГРАМИ З ВИКОРИСТАННЯМ MPI

Обмін повідомленнями між процесами з використанням MPI для 4 процесів.

Текст програми:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
    int ProcNum, ProcRank, RecvRank;
    MPI_Status Status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    if ( ProcRank == 0 )
    {
        // Дії виконуються тільки процесом з рангом 0
        printf("\n Hello from process %3d", ProcRank);
        for (int i = 1; i < ProcNum; i++ )
        {
            MPI_Recv(&RecvRank,1, MPI_INT, MPI_ANY_SOURCE,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
            printf("\n Hello from process %3d", RecvRank);
        }
    }
    else // Повідомлення відправляється всіма процесами,
    // крім процесу з рангом 0
        MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```



The screenshot shows a Windows XP command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the output of the MPI program. The command executed is "C:\Program Files\MPICH2\bin>mpirun -n 4 lab1". The output shows four lines of "Hello from process" followed by the process ranks 0, 3, 2, and 1. The prompt is "C:\Program Files\MPICH2\bin>".

РОЗПАРАЛЕЛЕННЯ МНОЖЕННЯ МАТРИЦЬ

1. Принципи розпаралелювання

Для багатьох методів матричних обчислень характерним є повторення одних і тих же операцій для різних даних. Дана властивість свідчить про наявність паралелізму за даними при виконанні матричних обчислень, і, як результат, розпаралелювання матричних операцій зводиться, в більшості випадків, до розбиття оброблюваних матриць між процесорами використовуваної обчислювальної системи. Вибір способу поділу матриць приводить до визначення конкретного методу паралельних обчислень; існування різних схем розподілу даних породжує ряд паралельних алгоритмів матричних обчислень.

Найбільш загальні і широко використовувані способи поділу матриць полягають в розбитті даних на смуги (по вертикалі або горизонталі) або на прямокутні фрагменти (блоки).

1.1. Стрічкове розбиття матриці. При стрічковому (block-striped) розбитті кожному процесору виділяється певна підмножина рядків (rowwise або горизонтальне розбиття) або стовпців (columnwise або вертикальне розбиття) матриці. При такому підході для горизонтального розбиття по рядках, наприклад, матриця A представляється у вигляді (1)

$$A = (A_0, A_1, \dots, A_{p-1})^T, A_i = (a_{i0}, a_{i1}, \dots, a_{ik-1}), i_j = ik + j, 0 \leq j < k, k = m/p, \text{ де} \quad (1)$$

$$a_i = (a_{i1}, a_{i2}, \dots, a_{in}),$$

$0 \leq i < m$ і-й рядок матриці A (передбачається, що кількість рядків m кратна кількості процесорів p , тобто $m = kp$).

Інший можливий підхід до формування смуг полягає в застосуванні тієї чи іншої схеми чергування (циклічності) рядків або стовпців. Як правило, для чергування використовується кількість процесорів p - в цьому випадку при горизонтальному розбитті матриця A приймає вигляд

$$A = (A_0, A_1, \dots, A_{p-1})^T, A_i = (a_{i0}, a_{i1}, \dots, a_{ik-1}), i_j = i + jp, 0 \leq j < k, k = m/p, \quad (2)$$

1.2. Блокове розбиття матриці. При блоковому (chessboard block) розбитті матриця ділиться на прямокутні набори. Хай кількість процесорів складає $p = s \cdot q$, кількість рядків матриці є кратним s , а кількість стовпців - кратним q , тобто $m = k \cdot s$ і $n = l \cdot q$. Представимо початкову матрицю A у вигляді набору прямокутних блоків таким чином (3):

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots & A_{0q-1} \\ & \dots & & \\ A_{s-11} & A_{s-12} & \dots & A_{s-1q-1} \end{pmatrix}, \quad (3)$$

де A_{ij} - блок матриці, що складається з елементів:

$$A = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \dots & a_{i_0 j_{l-1}} \\ & \dots & & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & \dots & a_{i_{k-1} j_{l-1}} \end{pmatrix}, \quad \begin{matrix} i_\nu = ik + \nu, 0 \leq \nu < k, k = m/s, \\ j_u = jl + u, 0 \leq u \leq l, l = n/q. \end{matrix} \quad (4)$$

При такому підході доцільно, щоб обчислювальна система мала фізичну або, принаймні, логічну топологію процесорних ґраток з s рядків і q стовпців. При такому розбитті даних, сусідні в структурі ґраток процесори, обробляють суміжні блоки початкової матриці. Треба зазначити, що і для блокової схеми може бути застосоване циклічне чергування рядків і стовпців.

У лабораторній роботі розглядаються три паралельні алгоритми для множення квадратної матриці на вектор. Кожен підхід заснований на різному типі розбиття початкових даних (елементів матриці і вектора) між процесорами. Розбиття даних міняє схему взаємодії процесорів, тому кожен з представлених методів істотним чином відрізняється від решти.

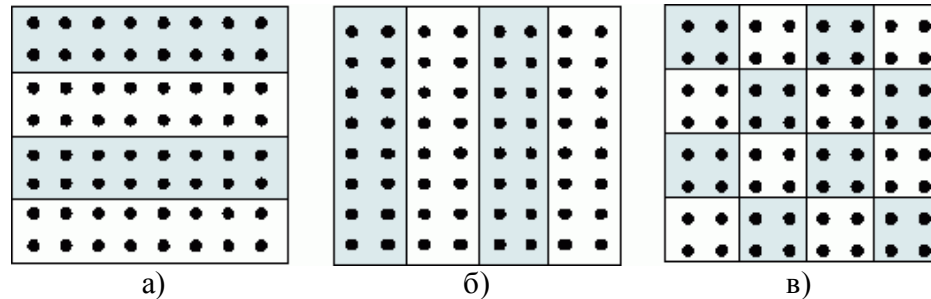


Рис. 1. Способи розбиття елементів матриці між процесорами обчислювальної системи

На рис. 1 схематично наведені способи розбиття матриць між процесорами: а) горизонтальне розбиття, б) вертикальне розбиття та в) блокове розбиття матриці.

2. Постановка задачі

В результаті перемноження матриці A розмірності $m \times n$ і вектора b , що складається з n елементів, отримується вектор розміру m , кожен i -й елемент якого є результат скалярного множення i -того рядка матриці A (позначимо цей рядок a_i) і вектора b :

$$c_i = (a_i, b) = \sum_{j=1}^n a_{ij} b_j; \quad 1 \leq i \leq m \quad (4)$$

Тим самим отримання результуючого вектора C припускає повторення m однотипних операцій по множенню рядків матриці A і вектора b . Кожна така операція включає множення елементів рядка матриці і вектора b (n операцій) і подальше підсумовування отриманих множень ($n - 1$ операцій). Загальна кількість необхідних скалярних операцій є величина

$$T_l = m \cdot (2n - 1). \quad (5)$$

2.1. Послідовний алгоритм. Послідовний алгоритм перемноження матриці на вектор може бути представлений таким чином.

```
// Послідовний алгоритм множення матриці на вектор
for (i = 0; i < m; i++) {
    c[i] = 0;
    for (j = 0; j < n; j++) {
        c[i] += A[i][j] * b[j]
    }
}
```

Векторно-матричне множення - це послідовність обчислення скалярних добутків. Оскільки кожне обчислення скалярного добутку векторів довжини n вимагає виконання n операцій множення і $(n-1)$ операцій додавання, його трудомісткість становить $O(n)$. Для виконання

векторно-матричного множення необхідно здійснити m операцій обчислення скалярного добутку, таким чином, алгоритм має трудомісткість порядку $O(mn)$.

2.2. Розділення даних. При виконанні паралельних алгоритмів перемноження матриці на вектор, окрім матриці A , необхідно розбити вектор b і вектор результату c . Елементи векторів можна продублювати, тобто скопіювати всі елементи вектора на всі процесори, складові багатопроцесорної обчислювальної системи, або розділити між процесорами. При блоковому розбитті вектора з n елементів кожен процесор обробляє безперервну послідовність із k елементів вектора (припускається, що розмірність вектора n без остачі ділиться на число процесорів, тобто $n = kp$).

Пояснимо, чому дублювання векторів b і c між процесорами є допустимим рішенням (далі для простоти викладу вважатимемо, що $m=n$). Вектори b і c складаються з n елементів, тобто містять стільки ж даних, скільки і один рядок або один стовпець матриці. Якщо процесор зберігає рядок або стовпець матриці і одиночні елементи векторів b і c , то загальне число елементів, що зберігаються, має трудомісткість порядку $O(n)$. Якщо процесор зберігає рядок (стовпець) матриці і всі елементи векторів b і c , то загальна кількість елементів, що зберігаються, також порядку $O(n)$. Таким чином, при дублюванні і при розбитті векторів вимоги до об'єму пам'яті з одного класу складності.

ПРИКЛАД ПРОГРАМИ МНОЖЕННЯ МАТРИЦІ НА ВЕКТОР З ВИКОРИСТАННЯМ MPI

Тип розбиття – стрічкове горизонтальне. Кількість процесорів – 7.

Розмір матриці	Тип розбиття	Кількість процесорів
120	Стрічкове (гор)	7

Розбиття матриці

При горизонтальному способі розбиття даних (розбиття матриці на горизонтальні смуги) вхідна матриця буде мати такий вигляд:

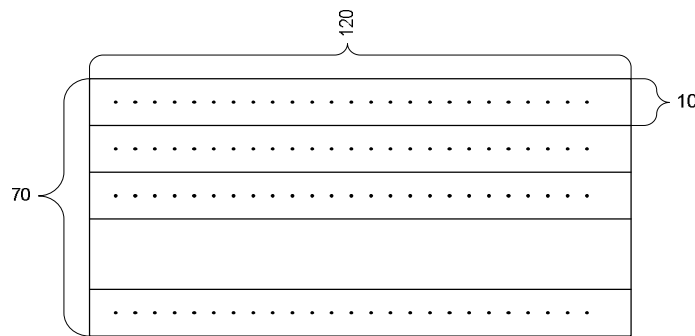


Рис. 2. Розбиття вхідної матриці на горизонтальні смуги для 7 процесорів.

Для кожного процесора визначено наступний розмір блоку для таких параметрів: матриця A розмірності $m \times n$, вектор b , що складається з n елементів та вектора результатів c розміру m . Вважається, що вектори b і c копіюються на кожний процесор.

Тоді: $m \times n / p + n + m = 120 \times 70 / 10 + 70 + 120 = 1390$ елементів;

Кількість операцій визначається на основі (5) та становить 2280 операцій для кожного процесора.

Як базова підзадача може бути вибрана операція скалярного множення одного рядка матриці на вектор.

Розробка схеми інформаційної взаємодії

Для виконання базової підзадачі скалярного добутоку процесор повинен містити відповідний рядок матриці A і копію вектора b . Після завершення обчислень кожна базова підзадача визначає один з елементів вектора результату c .

Для об'єднання результатів і отримання повного вектора c на кожному з процесорів обчислювальної системи необхідно виконати операцію узагальненого збору даних, в якій кожен процесор передає свій обчислений елемент вектора c решті всіх процесорів. Цей крок можна виконати, наприклад, з використанням функції *MPI_Allgather* з бібліотеки MPI.

У загальному вигляді схема інформаційної взаємодії підзадач в ході виконуваних обчислень наведена на рис. 3.

на основі методичних розробок Ваврука Є.Я.

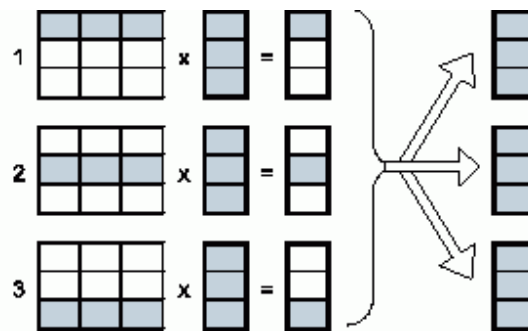


Рис. 3. Організація обчислень при виконанні паралельного алгоритму множення матриці на вектор, основане на розбитті матриці по рядках

Розбиття та масштабування підзадач по процесорах

В процесі множення матриці на вектор кількість обчислювальних операцій для отримання скалярного добутку однакова для всіх базових підзадач. Тому, у разі, коли кількість процесорів p менше від кількості базових підзадач, можна об'єднати базові підзадачі так, щоб кожен процесор виконував декілька таких підзадач, що відповідають безперервній послідовності (області) рядків матриці A . В цьому випадку після закінчення обчислень кожна базова підзадача визначає набір елементів результуючого вектора c .

Розробка програми з використанням MPI

Головна функція програми. Реалізує логіку роботи алгоритму, послідовно викликає необхідні підпрограми.

```
// Множення матриці на вектор - стрічкове горизонтальне розбиття
// (початковий і результуючий вектори дублюються між процесами)
void main(int argc, char* argv[])
{
    double* pMatrix; // Перший аргумент - початкова матриця
    double* pVector; // Другий аргумент - початковий вектор
    double* pResult; // Результат множення матриці на вектор
    int Size; // Розміри початкової матриці і вектора double* pProcRows;
    double* pProcResult;
    int RowNum;
    double Start, Finish, Duration;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    // Виділення пам'яті і ініціалізація початкових даних
    ProcessInitialization(pMatrix, pVector, pResult, pProcRows,
        pProcResult, Size, RowNum);

    // Розподіл початкових даних між процесами
    DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);

    // Паралельне виконання множення матриці на вектор
    ParallelResultCalculation(pProcRows, pVector, pProcResult,
        Size, RowNum);
}
```

```
// Збір результуючого вектора на всіх процесрах
ResultReplication(pProcResult, pResult, Size, RowNum);

// Завершення процесу обчислень
ProcessTermination(pMatrix, pVector, pResult, pProcRows,
                    pProcResult);

MPI_Finalize();
}
```

Функція ProcessInitialization. Ця функція задає розмір і елементи для матриці A і вектора b. Значення для матриці A і вектора b визначаються у функції Randomdatainitialization.

```
// Функція для виділення пам'яті і ініціалізації початкових даних
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, double* &pProcRows, double* &pProcResult,
    int &Size, int &RowNum)
{
    int RestRows; // Кількість рядків матриці, які ще
                  // не розподілені
    int i;

    if (ProcRank == 0) {
        do {
            printf("\nВведіть розмір матриці: ");
            scanf("%d", &Size);
            if (Size < ProcNum) {
                printf("Розмір матриці повинен перевищувати кількість
                    процесів! \n ");
            }
        }
        while (Size < ProcNum);
    }
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    RestRows = Size;
    for (i=0; i<ProcRank; i++)
        RestRows = RestRows-RestRows/(ProcNum-i);
    RowNum = RestRows/(ProcNum-ProcRank);

    pVector = new double [Size];
    pResult = new double [Size];
    pProcRows = new double [RowNum*Size];
    pProcResult = new double [RowNum];

    if (ProcRank == 0) {
        pMatrix = new double [Size*Size];
        RandomDataInitialization(pMatrix, pVector, Size);
    }
}
```

Функція DataDistribution. Здійснює розсилку вектора b і розподіл рядків початкової матриці A по процесрах обчислювальної системи. Слід зазначити, що коли кількість рядків матриці n не є кратною числу процесорів p, об'єм даних, що пересилаються, для процесів може опинитися різним і для передачі повідомлень необхідно використовувати функцію MPI_Scatterv бібліотеки MPI.

```
// Функція для розбиття початкових даних між процесами
void DataDistribution(double* pMatrix, double* pProcRows,
    double* pVector, int Size, int RowNum)
```

```
{
    int *pSendNum;          // Кількість елементів, що посилаються процесу
    int *pSendInd;          // Індекс першого елемента даних
                          // посиланого процесу

    int RestRows=Size;      // Кількість рядків матриці, які ще
                          // не розподілені

    MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Виділення пам'яті для зберігання тимчасових об'єктів
    pSendInd = new int [ProcNum];
    pSendNum = new int [ProcNum];

    // Визначення положення рядків матриці, призначених
    // кожному процесу

    RowNum = (Size/ProcNum);
    pSendNum[0] = RowNum*Size;
    pSendInd[0] = 0;
    for (int i=1; i<ProcNum; i++) {
        RestRows -= RowNum;
        RowNum = RestRows/(ProcNum-i);
        pSendNum[i] = RowNum*Size;
        pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
    }
    // Розсилка рядків матриці
    MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
        pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Звільнення пам'яті
    delete [] pSendNum;
    delete [] pSendInd;
}
```

3.3.4. Функція *ParallelResultCalculation*. Дана функція проводить множення на вектор тих рядків матриці, які розподілені на даний процес, і таким чином виходить блок результуючого вектора *c*.

```
// Функція для обчислення частини результуючого вектора
void ParallelResultCalculation(double* pProcRows, double* pVector,
    double* pProcResult, int Size, int RowNum) {
    int i, j;
    for (i=0; i<RowNum; i++) {
        pProcResult[i] = 0;
        for (j=0; j<Size; j++)
            pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
    }
}
```

3.3.5. Функція *ResultReplication*. Об'єднує блоки результуючого вектора *c*, отримані на різних процесах, і копіює вектор результату на всі процеси обчислювальної системи.

```
// Функція для збору результуючого вектора на всіх процесах
void ResultReplication(double* pProcResult, double* pResult,
    int Size, int RowNum)
{
```

```
int *pReceiveNum;    // Кількість елементів, що посиляються процесом
int *pReceiveInd;    // Індекс елементу даних в результуючому
                    // векторі

int RestRows=Size;   // Кількість рядків матриці, які ще не
                    // розподілені

int i;

// Виділення пам'яті для тимчасових об'єктів
pReceiveNum = new int [ProcNum];
pReceiveInd = new int [ProcNum];

// Визначення положення блоків результуючого вектора
pReceiveInd[0] = 0;
pReceiveNum[0] = Size/ProcNum;
for (i=1; i<ProcNum; i++) {
    RestRows -= pReceiveNum[i-1];
    pReceiveNum[i] = RestRows/(ProcNum-i);
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
}

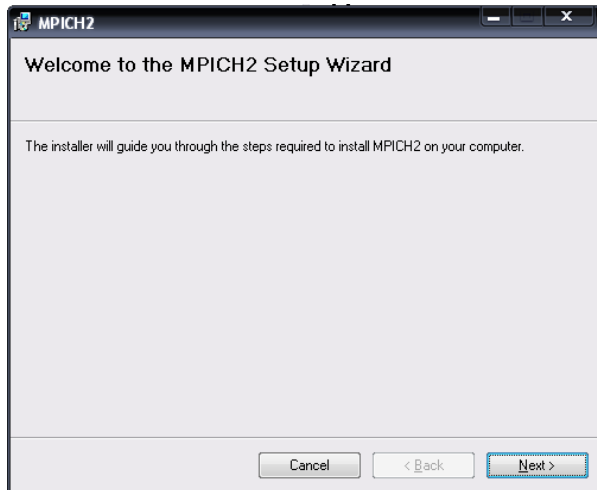
// Збір всього результуючого вектора на всіх процесах
MPI_Allgather(pProcResult, pReceiveNum[ProcRank],
    MPI_DOUBLE, pResult, pReceiveNum, pReceiveInd,
    MPI_DOUBLE, MPI_COMM_WORLD);

// Звільнення пам'яті
delete [] pReceiveNum;
delete [] pReceiveInd;
}
```

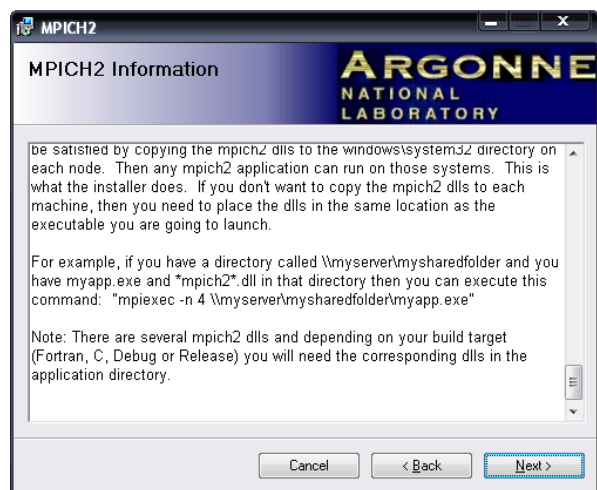
ІНСТАЛЯЦІЯ MPICH2 ДЛЯ ВИКОРИСТАННЯ MPI

3.1. Встановлення бібліотеки.

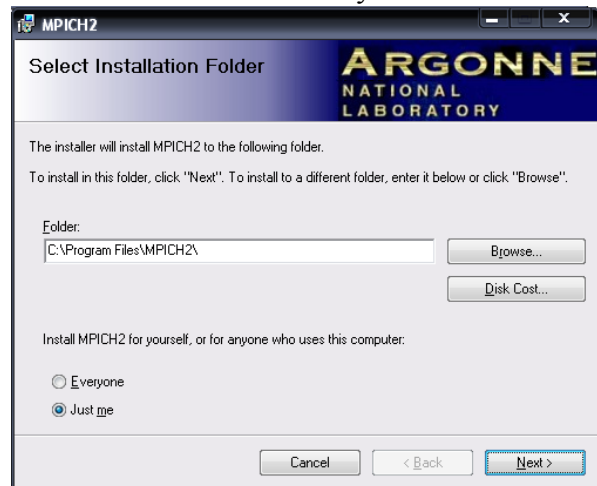
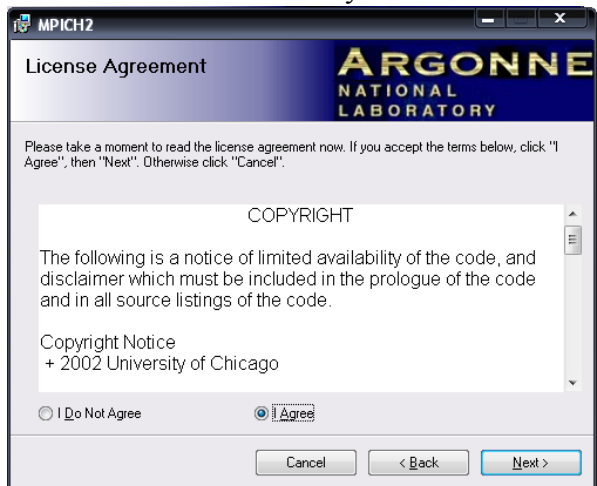
Запускаємо файл *mpich2-1.0.8-win-ia32.msi*



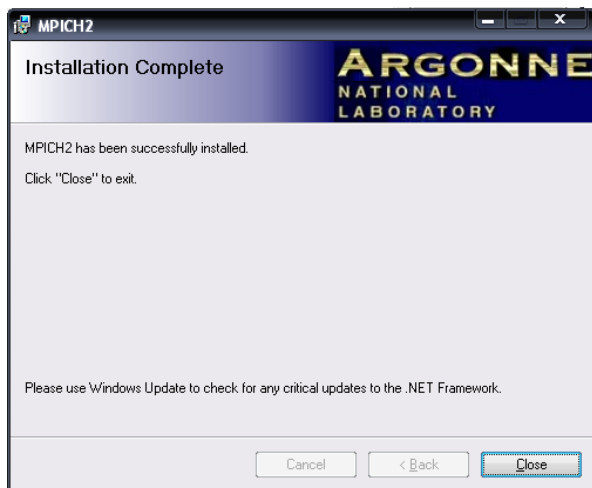
Натискаємо кнопку Next



Натискаємо кнопку Next

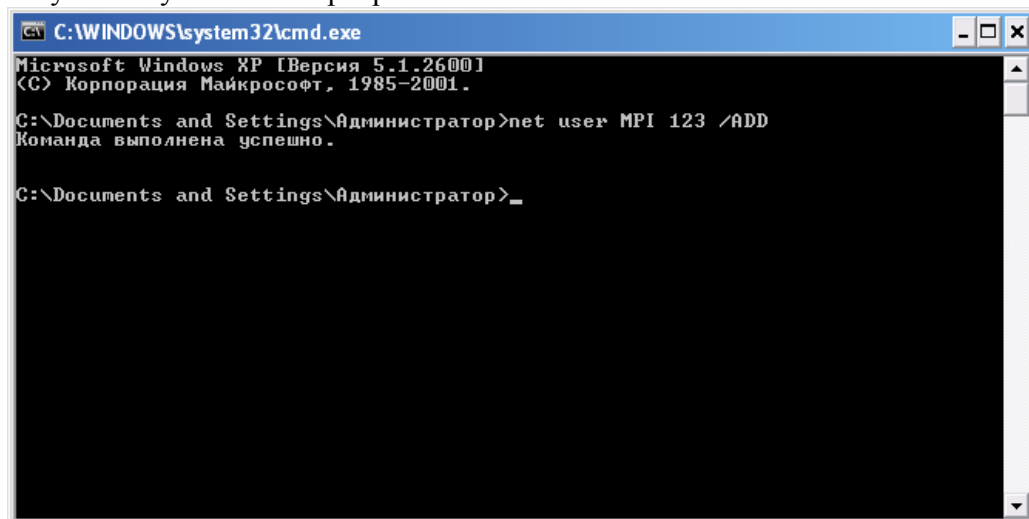


Погоджуємось з ліцензійними вимогами і натискаємо кнопку Next. Вибираємо директорію в якій буде розміщуватись MPI і натискаємо кнопку Next

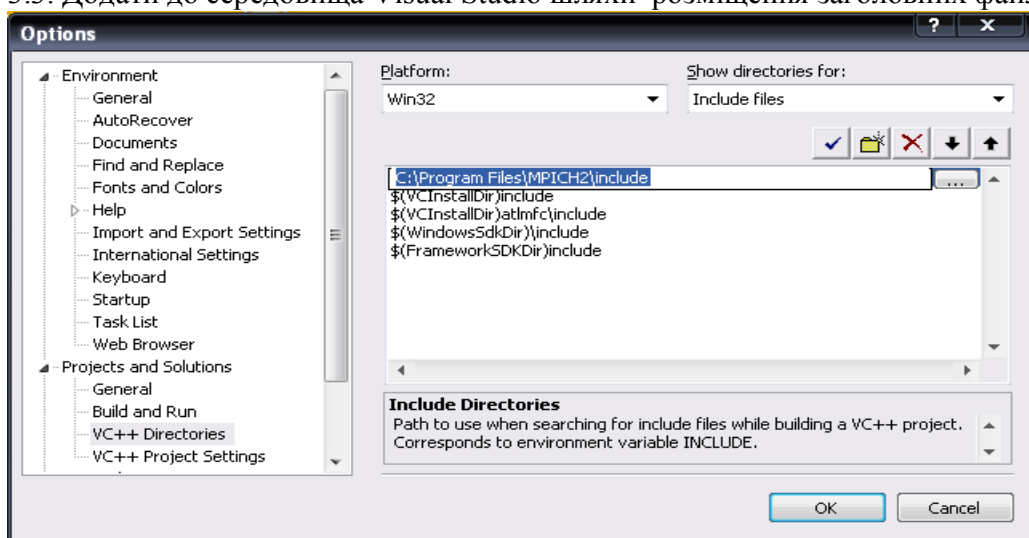


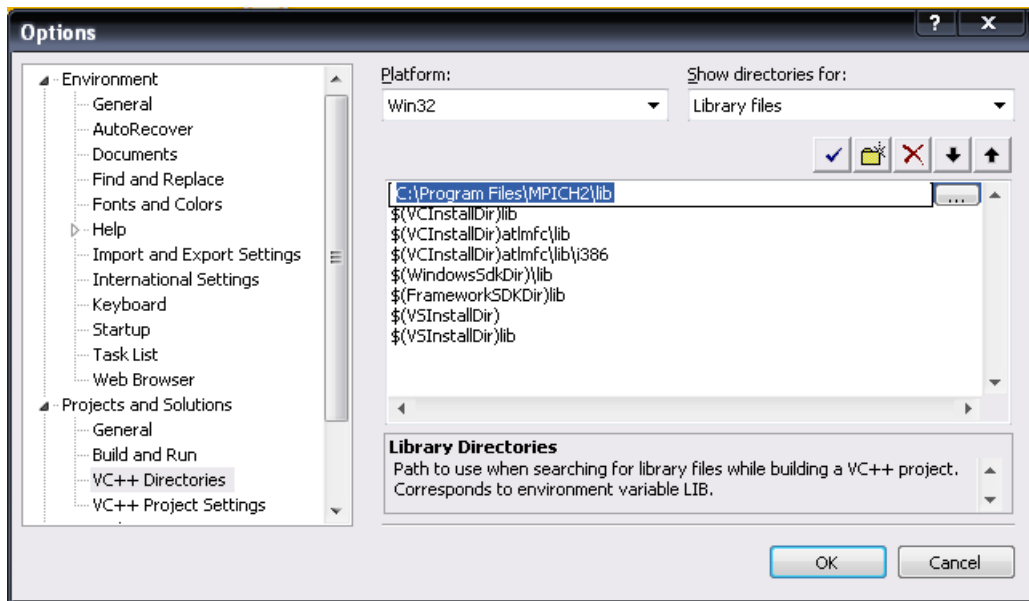
Чекаємо завершення встановлення та натискаємо кнопку Close

3.2. Командою `net user username password /add` прописати обліковий запис, під яким запускатимуться MPI- програми



3.3. Додати до середовища Visual Studio шляхи розміщення заголовних файлів та бібліотек





```
Administrator: Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd C:\MPICH2\bin

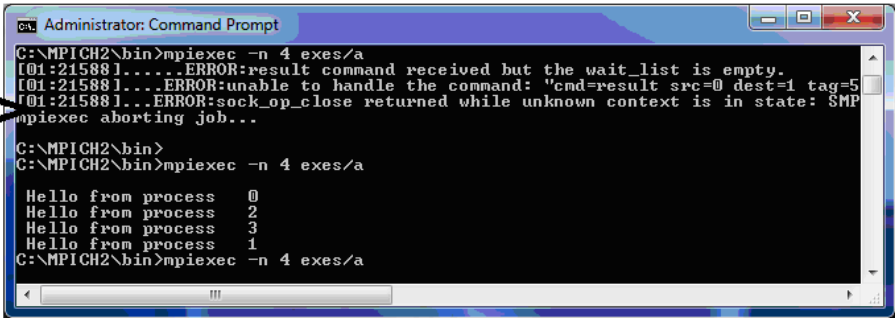
C:\MPICH2\bin>smpd
smpd options:
-port <port> or -p <port>
-phrase <passphrase>
-getphrase
-debug or -d
-noprompt
-restart [hostname]
-shutdown [hostname]
-console [hostname]
-status [hostname]
-anyport
-hosts
-sethosts
-set <option_name> <option_value>
-get <option_name>
-query [domain]
-help
unix only options:
-s
-r
-smpdfile <filename>
windows only options:
-install or -regserver
-remove or -unregserver or -uninstall
-start
-stop
-register_spn
-remove_spn
-traceon <logfilename> [<hostA> <hostB> ...]
-traceoff [<hostA> <hostB> ...]

bracketed [] items are optional

"smpd -d" will start the smpd in debug mode.
"smpd -s" will start the smpd in daemon mode for the current unix user.
"smpd -install" will install and start the smpd in Windows service mode.
This must be done by a user with administrator privileges and then all
users can launch processes with mpiexec.
Not yet implemented:
"smpd -r" will start the smpd in root daemon mode for unix.

C:\MPICH2\bin>mpiexec -validate
SUCCESS
```

Ctrl + C



```
C:\MPICH2\bin>mpiexec -n 4 exes/a
[01:215881].....ERROR:result command received but the wait_list is empty.
[01:215881].....ERROR:unable to handle the command: "cmd=result src=0 dest=1 tag=5
[01:215881].....ERROR:sock_op_close returned while unknown context is in state: SMP
mpiexec aborting job...

C:\MPICH2\bin>
C:\MPICH2\bin>mpiexec -n 4 exes/a

Hello from process 0
Hello from process 2
Hello from process 3
Hello from process 1

C:\MPICH2\bin>mpiexec -n 4 exes/a

Hello from process 0
Hello from process 2
Hello from process 3
Hello from process 1
```

Після виконаних вказаних кроків бібліотека MPI готова до використання.