МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
імені В.О. Сухомлинського

К.Т. КУЗЬМА, О.В. МЕЛЬНИК

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

навчальний посібник
для вищих закладів освіти

Затверджено рішенням вченої ради
Миколаївського національного університету
імені В.О. Сухомлинського

Миколаїв
2020

УДК 004.272+004.75
ББК 32.973
К89

Авторський колектив:

Кузьма К.Т., кандидат технічних наук, доцент кафедри інформаційних технологій Миколаївського національного університету імені В.О. Сухомлинського;

Мельник О.В., кандидат технічних наук, старший викладач кафедри інформаційних технологій Миколаївського національного університету імені В.О. Сухомлинського.

*Рецензенти***:**

І.І. Коваленко, доктор технічних наук, професор, професор кафедри комп'ютерної інженерії Чорноморського національного університету імені Петра Могили;

І.В. Устенко, кандидат технічних наук, доцент, доцент кафедри програмного забезпечення автоматизованих систем Національного університету кораблебудування імені адмірала Макарова.

Затверджено рішенням вченої ради Миколаївського національного університету імені В.О. Сухомлинського як навчальний посібник для студентів вищих навчальних закладів освіти (протокол № 19 від 28.05.2019 р.)

Кузьма К.Т.

К89 Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти / К.Т. Кузьма, О.В. Мельник. – Миколаїв: ФОП Швець В.М., 2020. – 172 с.

Навчальний посібник розроблений у відповідності з освітньо-професійною програмою підготовки бакалавра галузі знань 12 «Інформаційні технології» спеціальності 123 «Комп'ютерна інженерія», містить лекційний матеріал з нормативної навчальної дисципліни «Системне та прикладне програмне забезпечення (Частина 4. Паралельні та розподілені обчислення)». Наведений у посібнику матеріал може бути використаний при проведенні аудиторних, індивідуальних і самостійних занять. Посібник буде корисним для студентів інших спеціальностей галузі знань 12 «Інформаційні технології».

УДК 004.272+004.75
ББК 32.973

ISBN 978-617-7421-40-4
© Кузьма К.Т., Мельник О.В., 2020

Зміст

Вступ 6
РОЗДІЛ І. ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ 7
1 ПОЯВА ТА РОЗВИТОК ПАРАЛЕЛЬНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ 7
1.1 Поняття паралельних обчислювальних систем 8
1.2 Види паралелізму в архітектурі ЕОМ 10
1.3 Режими виконання незалежних частин програми 13
1.4 Рівні паралелізму в архітектурі ЕОМ 14
1.5 Контрольні запитання 18
2 КЛАСИФІКАЦІЯ ТИПІВ АРХІТЕКТУРИ ПАРАЛЕЛЬНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ 18
2.1 Параметри архітектури ПОС 19
2.2 Класифікація Флінна 20
2.3 Мультипроцесори 21
2.4 Мультикомп’ютери 24
2.5 Контрольні запитання 25
3 ПАРАДИГМИ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ 26
3.1 Моделі паралельних обчислень 26
3.2 Парадигми паралельного програмування 28
3.3 Модель паралельних обчислень у вигляді графа «операції-операнди» 30
3.4 Контрольні запитання 32
4 МОДЕЛІ СКЛАДНОСТІ ТА ПРОДУКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ 32
4.1 Модель послідовних та паралельних обчислень 33
4.2 Прискорення обчислень на паралельній системі . Закон Амдала 34
4.3 Принцип Брента 36
4.4 Контрольні запитання 37

5 МЕТОДИ РОЗРОБКИ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ 37
5.1 Метод балансованого дерева 37
5.2 Метод «розподіляй та володарюй» 39
5.3 Контрольні запитання 40
РОЗДІЛ II. СИНХРОНІЗАЦІЯ ПРОЦЕСІВ ТА ПОТОКІВ 41
6 ВЗАЄМНЕ ВИКЛЮЧЕННЯ ПАРАЛЕЛЬНИХ ПРОЦЕСІВ ІЗ ЗАГАЛЬНОЮ ПАМ'ЯТТЮ 41
6.1 Концепція процесу 41
6.2 Поняття ресурсу 43
6.3 Організація програм як системи процесів 44
6.4 Взаємодія та взаємовиключення процесів 46
6.5 Алгоритм Деккера 52
6.6 Семафори Дейкстри 54
6.7 Контрольні запитання 55
7 ПРОЦЕСИ І ПОТОКИ В ОС WINDOWS 56
7.1 Створення потоків 57
7.2 Синхронізація 67
7.3 Робота з об'єктами синхронізації 69
7.4 М'ютекси 70
7.5 Критичні секції 73
7.6 Події 77
7.7 Семафори 78
7.8 Контрольні запитання 79
8 СУЧАСНІ ІНТЕРФЕЙСИ ДЛЯ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ 80
8.1 Parallel Virtual Machine 81
8.2 Message Passing Interface 82
8.3 OpenMP 92
8.4 Контрольні запитання 102
РОЗДІЛ III. РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ 103
9 Основні механізми реалізації розподілених систем 103
9.1 Зв'язок у РОС 106
9.2 Сучасні РОС 107
9.2.1 Однорангові (peer-to-peer) мережі 108

9.2.2 Сервіс-орієнтована архітектура 110
9.2.3 Агенти 111
9.2.4 Хмарні обчислення 112
9.3 Модель «Клієнт-Сервер» 113
9.4 Контрольні запитання 114
10 ПРОГРАМУВАННЯ ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ З ВИКОРИСТАННЯМ СОКЕТІВ 115
10.1 Етапи роботи з об'єктами Windows Sockets: 115
10.2 Приклад серверної частини комплексу розподілених обчислень, які взаємодіють через протокол TCP-IP 122
10.3 Приклад клієнтської частини комплексу розподілених обчислень, які взаємодіють через протокол TCP-IP 126
10.4 Контрольні запитання 127
11 ВИЗНАЧЕННЯ ХМАРНИХ ОБЧИСЛЕНЬ 128
11.1 Архітектура хмарних додатків 131
11.1.1 Інфраструктура як сервіс (IaaS) 132
11.1.2 Платформа як сервіс (PaaS) 133
11.1.3 Програмне забезпечення як сервіс (SaaS) 133
11.2 Компоненти хмарних додатків 134
11.3 Переваги та недоліки хмарних обчислень 139
11.4 Контрольні запитання 141
12 КЛАСТЕРНІ СИСТЕМИ 142
12.1 Архітектура Грід-систем 142
12.2 Стандарти Грід 145
12.3 Порівняння Грід та Хмарних обчислень 147
12.4 Контрольні запитання 149
СИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 150
Інформаційні ресурси мережі Інтернет 152
ДОДАТОК А Встановлення MPICH 153
ДОДАТОК Б Наталштування MPI -програми в Visual Studio 167

Вступ

Розробка сучасного програмного забезпечення характеризуються динамічним розвитком й використанням моделей паралельних обчислень, які у своїй концептуальній основі стають розповсюдженими й пронизують більшість аспектів архітектури та засобів програмування сучасних комп'ютерних систем. Мережеві технології та засоби Інтернет, операційні системи й прикладне програмне забезпечення в сучасних умовах так чи інакше базуються на концепціях паралельних та розподілених обчислень. Поява та швидке розповсюдження мов програмування нового покоління, таких як Java, C# визначили новий напрям в дослідженнях проблем програмування багатоплатформенних розподілених та паралельних застосувань. Тому для інженера або програміста важливо знати й використовувати основні поняття та засоби мультипроцесорної обробки й паралельного програмування для підвищення ефективності та надійності обчислень.

За роки становлення й розвитку комп'ютерної науки область паралельних та розподілених обчислень накопичила достатню кількість знань, щоб перетворитися в самостійну дисципліну.

Метою дисципліни «Паралельні та розподілені обчислення» є, з одного боку, придбання студентами знань про теорію та методи паралельних обчислень, які вже стали класичними, а з іншого - ознайомлення студентів з новими досягненнями в розвитку паралельних обчислювальних систем, а також придбання ними практичних навичок в розробці сучасного паралельного програмного забезпечення. У методичному плані дисципліна спирається на знання, придбані студентами в циклі фундаментальних дисциплін з теорії алгоритмів та мов програмування.

РОЗДІЛ І. ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

1 ПОЯВА ТА РОЗВИТОК ПАРАЛЕЛЬНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ

Історія розвитку комп'ютерної науки і індустрії налічує вже понад 50 років. Ідеї створення програмованих обчислювальних пристроїв, які діють за програмою, що зберігається в пам'яті, були відомі ще в середині XIX ст. завдяки роботам англійського дослідника Ч. Беббіджа. Але успішне втілення цих ідей стало можливим тільки в перших електронних обчислювальних машинах (ЕОМ), які з'явилися після другої світової війни.

Перший в Україні комп'ютер (який був також першим в континентальній Європі) був створений в Києві в 1951 р. Загальні принципи побудови електронних обчислювальних машин були сформульовані американським математиком Дж. фон Нейманом (у 1946 р.) й відтоді відомі як принципи послідовних обчислень.

Головні положення принципів фон Неймана включали:

- послідовну організацію пам'яті;
- процес послідовного виконання команд (операцій програм) в порядку їх розташування в пам'яті машини;
- послідовну роботу пристроїв (арифметико-логічний пристрій, регістри, пристрої оперативної і постійної пам'яті), робота яких контролюється єдиним пристроєм управління.

Але вже тоді розробникам обчислювальної техніки була зрозуміло, що наявність одного виконуючого пристрою та єдиного пристрою управління не є принциповим обмеженням. Більше того, оскільки ЕОМ складалася з пристроїв різної швидкості, це призводило до значних втрат потенційної продуктивності обчислювальної машини. Проте тільки в 60-х роках, коли значно виросли потреби у високопродуктивних й високонадійних обчисленнях, з одного боку, і дозріла економічна доцільність побудови

мультипроцесорних ЕОМ - з іншого, створення комп'ютерних систем паралельної дії стало актуальним.

1.1 Поняття паралельних обчислювальних систем

Паралельна обчислювальна система (ПОС) - це мультипроцесорна система паралельної дії, яка забезпечує одночасне (паралельне) виконання операцій однією або декількома програмами.

У загальному плані під паралельними обчисленнями розуміються процеси обробки даних, в яких одночасно можуть виконуватися декілька операцій комп'ютерної системи.

Основною метою створення паралельних обчислювальних систем є отримання високих показників продуктивності (потужності) обчислень.

Такими показниками продуктивності є:

– швидкодія, яка вимірюється, як правило, у кількості операцій за секунду (оп/с) й визначає обчислювальну потужність паралельної системи; найуживанішою одиницею виміру швидкодії є 1 Мфлопс = 1 000 000 операцій з плаваючою точкою за секунду;

– пропускна спроможність - кількість інформації (транзакцій, запитів, команд), що обробляється мультипроцесорною системою за одиницю часу; може вимірюватися пропускною спроможністю каналів зв'язку мультипроцесорної системи в Мбайт/с;

– загальний об'єм оперативної пам'яті паралельної системи (Мбайт).

Методи паралельної обробки інформації виникали і удосконалювалися в ході загального розвитку архітектури обчислювальних машин. У другому поколінні ЕОМ разом з операційними системами і системами програмування виникли методи мультипрограмування, тобто виконання декількох програм на одному і тому ж устаткуванні. Проте комп'ютери залишалися практично однопроцесорними. Машини третього покоління далі розвинули методи мультипрограмування. У архітектурі ЕОМ з'явилися канали - спеціалізовані

процесори, які працювали паралельно з центральними процесорами і виконували виключно операції введення/виводу, а також виникло поняття мультикомп'ютерних систем. Проте це ще не було в повному розумінні мультипроцесорних систем. Ідея паралелізму обчислень утілювалася поступово [1].

З початку розвитку комп'ютерів і приблизно до 2000-2005 років провідні виробники мікропроцесорів досягали збільшення продуктивності останніх не на стільки за рахунок вдосконалення архітектурних рішень, як за рахунок збільшення тактової частоти процесорів. Але така тенденція на сьогоднішній день не є актуальною, оскільки подальше збільшення частоти при даному рівні технологій приводить до високих затрат. Тому сучасна стратегія розвитку процесорів персональних ЕОМ пов'язана із збільшенням процесорів або обчислювальних ядер.

Виникнення терміну «ядро» пов'язане з модифікацією архітектури процесора, що, у свою чергу, багато в чому визначається застосуванням технології виробництва мікропроцесорів з вищим рівнем інтеграції, що дає можливість реалізувати на кристалі нові технології обробки інформації.

Перший двоядерний процесор Power4 для серверів був анонсований фірмою IBM у 1999 році, а у 2001 році було розпочато їх продаж. У 2002 році AMD і Intel оголошують про перспективи створення своїх двоядерних процесорів.

Ідея багатоядерного процесора виглядає на перший погляд абсолютно тривіальною: просто «упаковуємо» два-три (або більш) процесори в один корпус - і комп'ютер отримує можливість виконувати декілька програмних потоків одночасно. Таке рішення є більш економічно вигідним, аніж застосування багатопроцесорних систем, проте часто є менш ефективним.

З метою систематизації всіх обчислювальних систем, доцільно їх класифікувати за певними ознаками.

1.2 Види паралелізму в архітектурі ЕОМ

Головними ідеями при розпаралелюванні операцій в обчислювальних системах є ідея конвеєризації обчислень потоку даних та ідея паралелізму - тобто незалежного (й одночасного) виконання операцій на різних пристроях обчислювальної машини. Використання цих двох ідей в чистому вигляді або в їх поєднанні привело до таких видів паралелізму в архітектурі ЕОМ :

– конвеєрна обробка;
– функціональна обробка;
– векторно-матрична обробка;
– мультипроцесорна обробка.

Конвеєрний спосіб розпаралелювання операцій передбачає наявність декількох етапів (стадій) обробки, через які послідовно проходить кожен елемент потоку даних. Якщо час обробки елементу даних на кожному етапі обробки приблизно однаковий, то паралелізм обчислень досягається за рахунок одночасної обробки різних елементів вхідного потоку даних на різних стадіях конвеєра.

Конвеєризація (чи конвеєрна обробка) в загальному випадку заснована на розподілі виконуваної функції на дрібніші частини, які називаються ступенями, й виділенні для кожної з них окремого блоку апаратури. Так обробку будь-якої машинної команди можна розділити на декілька етапів (ступенів), організувавши передачу даних від одного етапу до наступного. При цьому конвеєрну обробку можна використовувати для поєднання етапів виконання різних команд. Продуктивність при цьому зростає завдяки тому, що одночасно на різних ступенях конвеєра виконуються декілька команд. Конвеєрна обробка такого роду широко застосовується в усіх сучасних швидкодіючих процесорах.

Таким чином, чим довше конвеєр, тобто більша кількість його стадій, й чим довше вхідний потік даних - тим вища міра розпаралелювання операцій. Використання цього виду паралелізму в арифметичному й управляючому

пристроях ЕОМ стало хронологічно першим й найпоширенішим, особливо в науково-технічних розрахунках, які характеризуються високою регулярністю обчислень. Серед перших комерційно успішних прикладів конвеєрно-паралельних ЕОМ можна назвати американські машини фірми Control Data кінця 60-х й початку 70-х років CDC - 6600 та CYBER - 205. Зокрема, остання модель мала продуктивність 50 Мфлопс при довжині машинної операції в 64 розряди [2].

Функціональний паралелізм передбачає надання можливості одночасного виконання різних функцій обробки (наприклад, логічних і арифметичних) декільком незалежним функціональним пристроям процесора. Сучасним прикладом використання функціонального паралелізму є наявність співпроцесора в архітектурі персонального комп'ютера для виконання операцій з плаваючою точкою. Інші приклади можна знайти в [3].

Векторно-матрична обробка сполучає конвеєрний і функціональний паралелізм. Істотною ознакою при цьому є наявність множини ідентичних процесорних елементів з єдиною системою управління, які взаємодіють між собою через загальне поле пам'яті. Важливою віхою в реалізації цього виду паралелізму стала перша модель суперкомп'ютерної системи Cray (у 1976 р.), процесор якої містив 12 функціональних пристроїв, усі конвеєрні, а також 8 векторних регістрів, кожен з яких мав змогу містити 64-розрядні числа у форматі з плаваючою точкою і показував рекордну на свій час продуктивність 130 Мфлопс [2].

Мультипроцесорна обробка (чи справжній паралелізм), на відміну від відмічених вище видів паралелізму, реалізується множиною повнофункціональних процесорів, кожен з яких найчастіше є повноцінним комп'ютером, хіба що із спеціалізованими засобами зв'язку між процесорами і засобами введення/виведення. Справжній паралелізм, на відміну від конвеєрного, не вимагає організації вхідних даних в потік і характеризується незалежним та одночасним виконанням операцій над різними даними. Одним з перших прикладів розробки цього класу машин паралельної дії була 64-

процесорна система ILLIAC - IV (у 1972 р.), яка мала потужність понад 50 Мфлопс. Прикладом вітчизняної розробки є мультипроцесорна система ЕС - 1766 в Інституті кібернетики ім. В. М. Глушкова у 80-х роках, яка налічувала 64 різнорідні процесори, окрім хост-машини й інженерної машини-монітора, та об'єднувала конвеєрний і мультипроцесорний способи розпаралелювання [4].

Таким чином паралельна обробка даних, утілюючи ідею одночасного виконання декількох дій, має два різновиди: конвейєрність і власне паралельність. Обидва види паралельної обробки інтуїтивно зрозумілі, тому зробимо лише невеликі пояснення.

Паралельна обробка. Якщо деякий пристрій виконує одну операцію за одиницю часу, то тисячу операцій він виконає за тисячу одиниць. Якщо припустити, що є п'ять таких незалежних пристроїв, здатних працювати одночасно, то ту ж саму тисячу операцій система з п'яти пристроїв може виконати вже не за тисячу, а за двісті одиниць часу. Аналогічно система з N пристроїв ту ж роботу виконає за 1000/N одиниць часу.

Конвеєрна обробка. Ідея конвеєрної обробки полягає у виділенні окремих етапів виконання загальної операції, причому кожен етап, виконавши свою роботу, передавав би результат наступному, одночасно приймаючи нову порцію вхідних даних. Отримуємо очевидний виграш у швидкості обробки за рахунок поєднання раніше рознесених в часі операцій. Припустимо, що в операції можна виділити п'ять мікрооперацій, кожна з яких виконується за одну одиницю часу. Якщо є один послідовний пристрій, то 100 пар аргументів він обробить за 500 одиниць. Якщо кожну мікрооперацію виділити в окремий етап (ступінь) конвеєрного пристрою, то на п'ятій одиниці часу на різних стадіях обробки такого пристрою будуть знаходиться перші п'ять пар аргументів, а увесь набір із ста пар буде оброблений за 5+99=104 одиниці часу - прискорення в порівнянні з послідовним пристроєм майже в п'ять разів (число ступенів конвеєра).

1.3 Режими виконання незалежних частин програми

При розгляді проблеми організації паралельних обчислень слід розрізняти наступні можливі режими виконання незалежних частин програми:

1) Багатозадачний режим (режим розподілу часу), при якому для виконання декількох процесів використовується єдиний процесор; цей режим є псевдопаралельним, коли активним (виконуваним) може бути один єдиний процес, а усі інші процеси знаходяться в стані очікування своєї черги на використання процесора; використання режиму розподілу часу може підвищити ефективність організації обчислень (наприклад, якщо один з процесів не може виконуватися із-за очікування даних, що вводяться, процесор може бути задіяний для виконання іншого, готового до виконання процесу), крім того, в цьому режимі проявляються багато ефектів паралельних обчислень (необхідність взаємовиключення й синхронізації процесів та ін.) і, як результат, цей режим може бути використаний при початковій підготовці паралельних програм.

2) Паралельне виконання, коли в один і той же момент часу може виконуватися декілька команд обробки даних. Цей режим обчислень може бути забезпечений не лише за наявності декількох процесорів, але і реалізований за допомогою конвеєрних і векторних пристроїв обробки;

3) Розподілені обчислення; цей термін зазвичай використовують для паралельної обробки даних, при якій використовується декілька пристроїв обробки досить віддалених один від одного і в яких передача даних лініями зв'язку призводить до істотних тимчасових затримок; як результат, ефективна обробка даних при цьому способі організації обчислень можлива тільки для паралельних алгоритмів з низькою інтенсивністю потоків міжпроцесорних передач даних. Перераховані умови є характерними, наприклад, при організації обчислень в багатомашинних обчислювальних комплексах, що утворюються об'єднанням декількох

окремих ЕОМ за допомогою каналів зв'язку локальних або глобальних інформаційних мереж.

1.4 Рівні паралелізму в архітектурі ЕОМ

Рівень структуризації одиниць даних та обчислень, на якому відбувається розпаралелювання обробки в паралельних обчислювальних системах, визначає рівень паралелізму обчислень в цих системах. До відомих рівнів паралелізму належать:

- рівень завдань (процедур);
- рівень програм;
- рівень команд (арифметичних виразів);
- рівень розрядів.

Рівень завдань

На цьому рівні різні частини («процеси») однієї і тієї ж програми виконуються паралельно; кількість операцій обміну даними між процесами повинна бути мінімальною.

Основне застосування процедурного рівня розпаралелювання - загальне паралельне оброблення інформації, при якому застосовується поділ проблеми, що вирішується, на паралельні задачі - частини, які вирішуються багатьма процесорами з метою підвищення обчислювальної продуктивності (приклад - рис.1.1.).

Рівень програм

На цьому рівні одночасно (або щонайменше з часовим розподілом) виконуються комплексні програми (рис.1.2.). Комп'ютер, що виконує ці програми *не обов'язково повинен мати паралельну структуру,* достатньо, щоб на ньому була встановлена багатозадачна операційна система (наприклад, реалізована як система *розподілу часу).* В цій системі кожному користувачеві, відповідно до його пріоритету, планувальник *(Scheduler)* виділяє відрізок часу різної тривалості. Користувач одержує

ресурси центрального процесорного блоку тільки впродовж короткого часу, а потім стає в чергу на обслуговування.

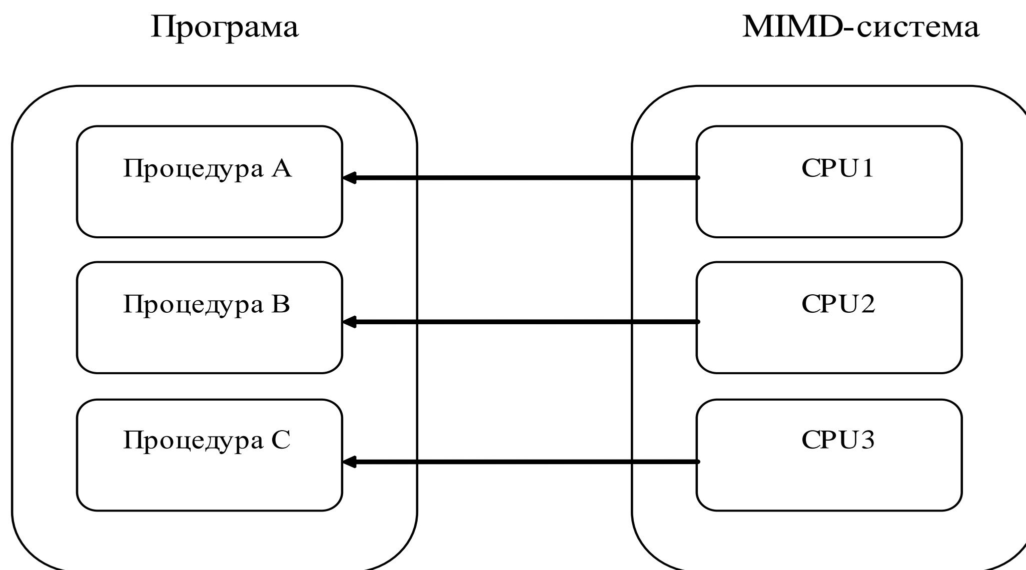


Рис.1.1. Паралельність на рівні процедур

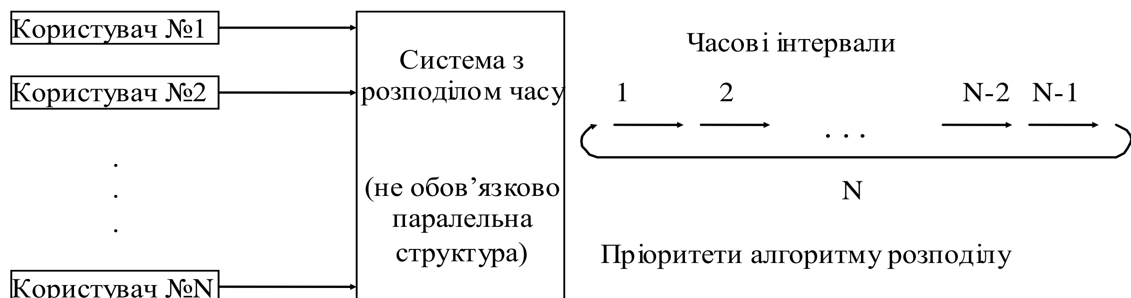


Рис.1.2. Паралельність на програмному рівні

У тому випадку, коли в системі недостатня кількість процесорів для всіх користувачів (або процесів), що, як правило, найбільш імовірно, в системі моделюється паралельне обслуговування користувачів за допомогою «квазіпаралельних» процесів.

Рівень команд (арифметичних виразів)

Арифметичні вирази виконуються паралельно покомпонентно, причому в суттєво простіших синхронних методах. Якщо, наприклад, йдеться про додавання матриць (рис.1.3.), то операція синхронно

розпаралелюється просто тому, що на кожному процесорові обчислюється один (чи кілька) елемент матриці.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 2 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 5 & 4 \end{pmatrix}$$

Неповні ALU без CPU

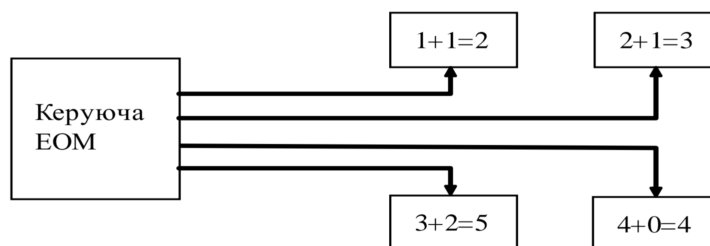


Рис.1.3. Паралельність на рівні операцій (формул)

При застосуванні $n*n$ процесорних елементів можна одержати суму двох матриць розмірністю $n*n$ за час виконання однієї операції додавання (за винятком часу, потрібного на виконання операцій читання та запису даних). Цьому рівню притаманні засоби векторизації так званої *паралельності даних*.

Рівень розрядів

На цьому рівні відбувається паралельне виконання бітових операцій в межах одного інформаційного слова (рис.1.4.). Паралельність на рівні бітів можна знайти в будь-якому працюючому мікропроцесорі. Наприклад, у 8-розрядному арифметико-логічному пристрої побітова обробка виконується паралельними апаратними засобами.

	01011101
AND	11011000
	01011000

Рис.1.4. Паралельність на рівні бітів

Певною мірою рівень паралелізму обчислень, який реалізується в паралельній обчислювальній системі, залежить від виду паралелізму, закладеного в архітектурі системи.

Методи паралельної обробки на мультипроцесорних системах грають провідну роль в першу чергу при створенні суперкомп'ютерних систем й додатків для таких системах. Суперкомп'ютери - це, як правило, дорогі обчислювальні системи паралельної дії, які забезпечують обчислення надвисокої продуктивності. Вони призначені для вирішення таких завдань, для яких існуючих засобів комп'ютерів масового споживання недостатньо або таке рішення неможливо виконати на цих засобах у відмічений термін. Прикладами таких завдань є задача із набору так званих викликів Grand Challenges, розроблена за завданням Національного наукового фонду (NSF) США на початку 90-х років, яка включала постановки задач й підходи їх рішення на суперкомп'ютерах в різних ділянках науково-технічних обчислень, зокрема прогнозування глобальних змін клімату, обчислення океанських течій, моделювання складних технічних систем тощо. Тривалий час на ринку суперкомп'ютерів домінували системи векторно-матричної архітектури, де найвидатніші досягнення мала компанія Cray. Проте в останнє десятиліття лідерство упевнено перейняли мультипроцесорні системи, які мали значну перевагу в нарощуванні кількості процесорів і засобах забезпечення високих показників продуктивності.

До найвидатніших досягнень суперкомп'ютерних систем останніх років належить розшифровка генома людини (що, як виявилось, налічує близько 30 тисяч різних генів, а не 120 тисяч, як вважалося раніше), а також перша у світі перемога шахової програми, яка виконувалася на мультипроцесорній системі Deep Blue фірми IBM над чемпіоном світу з шахів Г. Каспаровим.

Основними виробниками суперкомп'ютерних систем є провідні фірми США: Intel, IBM, SGI, Cray, HP, Sun, а також деякі фірми в Японії та Європі. Рейтинг найпотужніших у світі 500 суперкомп'ютерних систем визначається двічі на рік організацією TOP 500, яка оприлюднила результати оцінок в Інтернеті на сайті http://www.top500.org. Сучасний рівень потужності суперкомп'ютерів можна в цілому характеризувати

«трьома Т»: 1 Тфлопс швидкодії, 1 Тбайт/с пропускній спроможності і 1 Тбайт загального об'єму оперативної пам'яті.

Ось лише невеликий список областей людської діяльності, де використання суперкомп'ютерів дійсно потрібне:

- автомобілебудування;
- нафто- і газодобування;
- фармакологія;
- прогноз погоди і моделювання зміни клімату;
- сейсморозвідка;
- проектування електронних пристроїв;
- синтез нових матеріалів.

1.5 Контрольні запитання

1. Продуктивності ПОС: швидкодія, пропускна спроможність, загальний об’єм оперативної пам’яті.
2. Види паралелізму в архітектурі ЕОМ: конвеєрна, функціональна, векторно-матрична, мультипроцесорна обробка.
3. Історія появи паралелізму в архітектурі ЕОМ. Області застосування суперкомп‘ютерів.

Питання історії і розвитку обчислювальних машин паралельної дії детально висвітлені в книгах [1-3]. Огляд сучасних підходів до побудови паралельних обчислювальних систем наведено в [4]. На порталах [1-2] в мережі Інтернеті можна знайти також інші оглядові й навчальні матеріали з цих питань.

2 КЛАСИФІКАЦІЯ ТИПІВ АРХІТЕКТУРИ ПАРАЛЕЛЬНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ

Предметом вивчення в наступних лекціях є паралельні обчислювальні системи (ПОС) справжнього паралелізму (мультипроцесорні системи).

2.1 Параметри архітектури ПОС

Головними параметрами архітектури ПОС є:

– кількість і якість процесорів ПОС; системи з кількістю процесорів n > 100 прийнятий називати масовопаралельними. Такі ПОС, які складаються з однакових процесорів, називаються однорідними (гомогенними), а інші - різнорідними (гетерогенними);

– вид основної пам'яті - характеризує доступ до основної пам'яті з боку процесорів: загальна пам'ять є рівнодоступною для усіх процесорів ПОС й тому є глобальною для усієї ПОС; розподілена пам'ять розбита на окремі ділянки (в залежності від кількості процесорів), доступ до кожної з яких має тільки один процесор, тому така пам'ять є локальною;

– спосіб управління: синхронний (централізований), коли команди в усіх процесорах ПОС виконуються за єдиним сигналом від центрального блоку управління, й асинхронний (розподілений), коли виконання команди в процесорах не синхронізуються.

– система зв'язку між процесорами - характеризує топологію зв'язку процесорів ПОС, прикладами якої є: шинний зв'язок; статична топологія (лінійна, кільцева, зіркова, матрична, повнозв'язна, гіперкуб (2^n процесорів мають попарні зв'язки довжиною n)), динамічна топологія (із змінними зв'язками), комутатори.

Гіперкуб (hypercube) - ця топологія представляє окремий випадок структури решітки, коли за кожною розмірністю сітки є тільки два процесори; цей варіант організації мережі передачі даних широко поширений на практиці й характеризується наступним рядом відмітних ознак:

- два процесори мають з'єднання, якщо двійкові представлення їх номерів мають тільки одну позицію, що розрізняється;

- в n - мірному гіперкубі кожен процесор пов'язаний рівно з n сусідами;

- n - мірний гіперкуб може бути розділений на два ($n-1$)-мірних гіперкуби (всього можливо n різного такого розбиття).

2.2 Класифікація Флінна

Існує декілька класифікацій (таксономії) архітектури ПОС відповідно до відмічених вище параметрів. Найвідомішою серед них є запропонована американським дослідником М. Флінном ще в 60-х роках XX ст. таксономія архітектури, яка базується на характеристиці співвідношення потоків команд і даних. Під потоком команд спрощено розуміють послідовність команд однієї програми. Потік даних − це послідовність даних, яка оброблюється однією програмою.

За Флінном архітектура ПОС розділяються на чотири категорії:

1) SISD (Single Instruction Single Data) або ПКПД (поодинокий потік команд − поодинокий потік даних) - звичайна послідовна машина фон Неймана. До цього класу відносяться усі однопроцесорні системи;

2) MISD (Multiple Instruction Single Data) або МКПД (множинний потік команд - поодинокий потік даних) - декілька процесорів виконують різні потоки команд над одним потоком даних. Типовим представником цієї категорії є конвеєрна архітектура (наприклад, Cray [2-3]);

3) SIMD (Single Instruction Multiple Data) або ПКМД (поодинокий потік команд − множинний потік даних) - декілька процесорів одночасно (синхронно) виконують одну і ту ж команду, але над різними потоками даних. Типовими представниками є матричні комп'ютери, в яких усі процесорні елементи виконують одну і ту ж програму, що застосовується до своїх (різних для кожного елементу) локальних даних. (наприклад, ILLIAC - IV, ПС- 2000, Connection Machines CM - 2);

4) MIMD (Multiple Instruction Multiple Data) або МКМД (множинний потік команд − множинний потік даних) - кожен процесор виконує власну програму над власним потоком даних. Нині це найпоширеніший клас архітектури ПОС, який складає більшу частину із списку top500 (наприклад,

Intel Paragon Cray - T3D, CM. Такі системи зазвичай називають багатопроцесорними.

Слід зазначити, що хоча систематика Флінна широко використовується при конкретизації типів комп'ютерних систем, така класифікація призводить до того, що практично усі види паралельних систем (незважаючи на їх істотну різнорідність) відносяться до однієї групи MIMD. Як результат, багатьма дослідниками робилися неодноразові спроби деталізації систематики Флінна. Так, наприклад, для класу MIMD запропонована практично загальновизнана структурна схема, в якій подальший розподіл типів багатопроцесорних систем ґрунтується на використаних способах організації оперативної пам'яті в цих системах (рис. 2.1.).

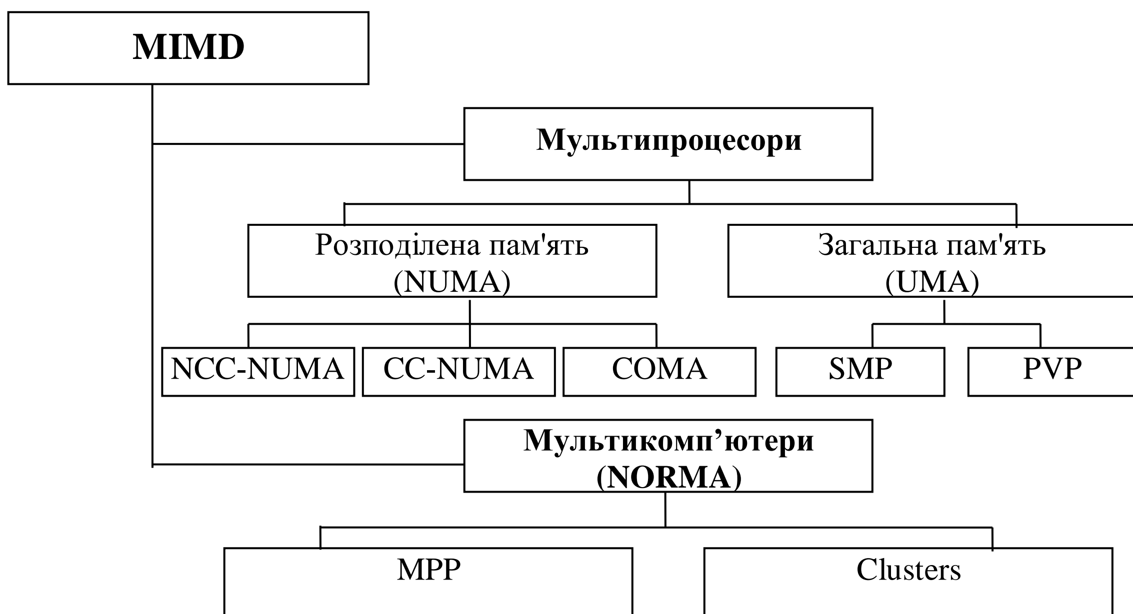
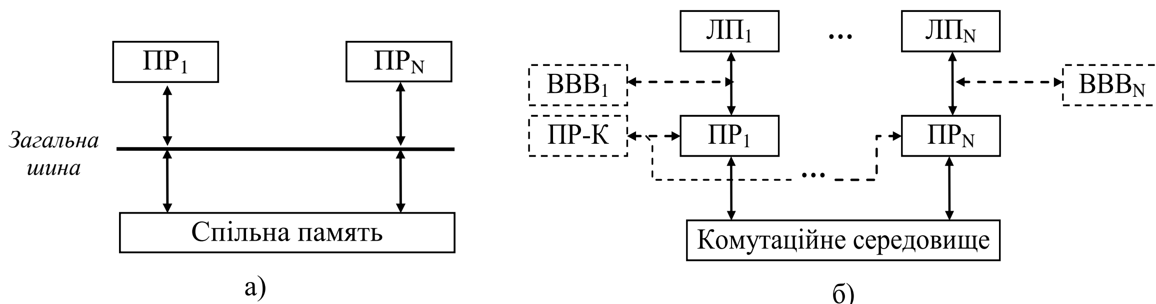


Рис. 2.1. Класифікація багатопроцесорних обчислювальних систем

Цей похід дозволяє розрізняти два важливі типи багатопроцесорних систем - multiprocessors (мультипроцесори або системи із загальною пам'яттю, що розділяється) і multicomputers (мультикомп'ютери або системи з розподіленою пам'яттю).

2.3 Мультипроцесори

Для подальшої систематики мультипроцесорів враховується спосіб побудови загальної пам'яті. Можливий підхід - використання єдиної (централізованої) загальної пам'яті (shared memory) - рис. 2.2(а). Такий підхід забезпечує однорідний доступ до пам'яті (uniform memory access or UMA) й служить основою для побудови векторних паралельних процесорів (parallel vector processor or PVP) та симетричних мультипроцесорів (symmetric multiprocessor or SMP). Серед прикладів першої групи суперкомп'ютер Cray T90, до другої групи відносяться IBM eServer, Sun StarFire, HP Superdome, SGI Origin та ін.



ЛП – локальна пам'ять, ПР – процесор, ПР-К – процесор комутаційний,

ВВВ – вузол введення-виведення.

Пунктирними лініями позначені необов’язкові вузли і лінії зв’язку.

Рис. 2.2. Архітектура багатопроцесорних систем:

а) системи з загальною пам’яттю; б) системи з розподіленою пам’яттю

Однією з основних проблем, які виникають при організації паралельних обчислень на такого типу системах, є доступ з різних процесорів до загальних даних і забезпечення однозначності (когерентності) вмісту різних кешів (cache coherence problem). Річ у тому, що за наявності загальних даних копії значень одних і тих же змінних можуть виявитися в кешах різних процесорів. Якщо в такій ситуації (за наявності копій загальних даних) один з процесорів виконає зміну

значення змінної, що розділяється, то значення копій в кешах інших процесорах виявляться не відповідними дійсності і їх використання приведе до некоректності обчислень. Забезпечення однозначності кешів зазвичай реалізується на апаратному рівні - для цього після зміни значення загальної змінної усі копії цієї змінної в кешах відзначаються як недійсні і наступний доступ до змінної приведе обов'язкового звернення до основної пам'яті. Слід зазначити, що необхідність забезпечення когерентності призводить до деякого зниження швидкості обчислень та ускладнює створення систем з досить великою кількістю процесорів.

Наявність загальних даних при виконанні паралельних обчислень призводить до необхідності синхронізації взаємодії одночасно виконуваних потоків команд. Завдання синхронізації належать до класичних проблем, їх розгляд при розробці паралельних програм є одним з основних питань паралельного програмування.

Загальний доступ до даних може бути забезпечений і при фізично розподіленій пам'яті (при цьому, природно, тривалість доступу вже не буде однакової для усіх елементів пам'яті) - рис. 2.2(б). Такий підхід називається як неоднорідний доступ до пам'яті (non - uniform memory access or NUMA). Серед систем з таким типом пам'яті виділяють:

− системи, в яких для представлення даних використовується тільки локальна кеш-пам'ять наявних процесорів (cache - only memory architecture or COMA); прикладами таких систем є, наприклад, KSR - 1 і DDM;

− системи, в яких забезпечується когерентність локальних кешів різних процесорів (cache - coherent NUMA or CC - NUMA); серед систем цього типу SGI Origin 2000, Sun HPC 10000, IBM/Sequent NUMA - Q 2000;

− системи, в яких забезпечується загальний доступ до локальної пам'яті різних процесорів без підтримки на апаратному рівні когерентності кеша (non - cache coherent NUMA or NCC - NUMA); до цього типу відноситься, наприклад, система Cray T3E.

Використання розподіленої загальної пам'яті (distributed shared memory or DSM) спрощує проблеми створення мультипроцесорів (відомі приклади систем з декількома тисячами процесорів), проте, проблеми ефективного використання розподіленої пам'яті (час доступу до локальної й віддаленої пам'яті може розрізнятися на декілька порядків), що виникають при цьому, призводять до істотного підвищення складності паралельного програмування.

2.4 Мультикомп'ютери

Мультикомп'ютери (багатопроцесорні системи з розподіленою пам'яттю) вже не забезпечують загальний доступ до усієї наявної в системах пам'яті (no - remote memory access or NORMA). При усій схожості подібної архітектури із системами з розподіленою загальною пам'яттю, мультикомп'ютери мають принципову відмінність - кожен процесор системи може використовувати тільки свою локальну пам'ять, тоді як для доступу до даних, що розташовуються на інших процесорах, необхідно явно виконати операції передачі повідомлень (message passing operations). Цей підхід використовується при побудові двох важливих типів багатопроцесорних обчислювальних систем (рис. 2.1.) - масивно-паралельних систем (massively parallel processor or MPP) та кластерів (clusters). Серед представників першого типу систем - IBM RS/6000 SP2, Intel PARAGON, ASCI Red, трансп'ютерні системи Parsytec та ін.; прикладами кластерів є, наприклад, системи AC3 Velocity і NCSA NT Supercluster.

Слід відмітити надзвичайно швидкий розвиток багатопроцесорних обчислювальних систем кластерного типу. Під кластером зазвичай розуміється безліч окремих комп'ютерів, об'єднаних в мережу, для яких за допомогою спеціальних апаратно-програмних засобів забезпечується можливість уніфікованого управління (single system image), надійного функціонування (availability) і ефективного використання (performance).

Кластери можуть бути утворені на базі вже існуючих окремих комп'ютерів, або ж сконструйовані з типових комп'ютерних елементів, що зазвичай не вимагає значних фінансових витрат. Застосування кластерів може також в деякій мірі понизити проблеми, пов'язані з розробкою паралельних алгоритмів і програм, оскільки підвищення обчислювальної потужності окремих процесорів дозволяє будувати кластери з порівняно невеликої кількості (декілька десятків) окремих комп'ютерів (lowly parallel processing). Це призводить до того, що для паралельного виконання в алгоритмах рішення обчислювальних завдань досить виділяти тільки великі незалежні частини розрахунків (coarse granularity), що, у свою чергу, знижує складність побудови паралельних методів обчислень і зменшує потоки даних, які передаються, між комп'ютерами кластера. Разом з цим слід зазначити, що організація взаємодії обчислювальних вузлів кластера за допомогою передачі повідомлень зазвичай призводить до значних тимчасових затримок, що накладає додаткові обмеження на тип паралельних алгоритмів і програм, що розробляються.

Окремі дослідники звертають особливу увагу на відмінність поняття кластера від мережі комп'ютерів (network of workstations or NOW). Для побудови локальної комп'ютерної мережі, як правило, застосовують простіші мережі передачі даних (близько 100 Мбіт/с). Комп'ютери мережі зазвичай більш розосереджені і можуть бути використані користувачами для виконання будь-якій додаткових робіт.

2.5 Контрольні запитання

1. Параметри архітектури ПОС: кількість та якість процесорів ПОС, вид основної пам‘яті, спосіб керування, система зв’язку між процесорами.
2. Поняття потоку команд та потоку даних.
3. Класифікація багатопроцесорних систем за Флінном (SISD, SIMD, MISD, MIMD системи).

4. Напрямки розвитку високопродуктивної обчислювальної техніки в даний час. (векторно-конвейєрні комп'ютери, масивно-паралельні комп'ютери, паралельні комп'ютери із загальною пам'яттю, кластери).

Детальніше питання параметрів архітектури обчислювальних машин паралельної дії розглядаються в книгах [2-3,5]. Приклади сучасної архітектури паралельних обчислювальних систем можна знайти в [4]. На порталі [1] у Інтернеті знаходяться також інші оглядові та навчальні матеріали.

3 ПАРАДИГМИ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

Паралельне програмування забезпечує спосіб організації програмного забезпечення, що складається з відносно незалежних частин. Воно також дозволяє використовувати багатопроцесорні системи. Існує три великі класи додатків – багатопотокові системи, розподілені системи й синхронні паралельні обчислення – та три відповідних їм типи паралельних програм.

3.1 Моделі паралельних обчислень

Процес - це послідовна програма, яка при виконанні має власний потік управління. Кожна паралельна програма містить декілька процесів, тому має декілька потоків. Проте термін багатопотоковий зазвичай визначає, що програма містить більше процесів (потоків), чим існує процесорів для їх виконання. Отже, процеси на процесорах виконуються по черзі.

Багатопоточна програмна система управляє безліччю незалежних процесів. Прикладами таких систем є:

– віконні системи на персональних комп'ютерах або робочих станціях;

– багатопроцесорні операційні системи і системи з розподілом часу;

– системи реального часу, що управляють електростанціями, космічними апаратами тощо.

Ці системи розроблені як багатопотокові програми, оскільки організувати код й структури даних у вигляді набору процесів набагато простіше, ніж реалізувати величезну послідовну програму. Крім того,

кожен процес може плануватися й виконуватися незалежно. Наприклад, коли користувач натискає кнопку миші персонального комп'ютера, посилається сигнал процесу, що управляє вікном, в якому в даний момент знаходиться курсор миші. Цей процес (потік) може виконуватися й відповідати на натискання миші. Застосування в інших вікнах можуть продовжувати при цьому своє виконання у фоновому режимі.

Другий широкий клас застосувань утворюють розподілені обчислення, в яких компоненти виконуються на ЕОМ, зв'язаних локальною або глобальною мережею. З цієї причини процеси взаємодіють, обмінюючись повідомленнями. Приклади систем розподілених обчислень:

- файлові сервери в мережі;
- системи баз даних для банків, замовлення авіаквитків тощо;
- Web-сервери мережі Internet;
- підприємницькі системи, що об'єднують філіали.

Такі системи створюються для розподілу обробки (як у файлових серверах), забезпечення доступу до віддалених даних (як в базах даних та в Web), інтеграції і управління даними, розподіленими за своєю суттю (як в промислових системах), або підвищення надійності (як у відмовостійких системах). Багато розподілених систем організовані як системи типу клієнт-сервер. Наприклад, файловий сервер надає файли даних для процесів, що виконуються на клієнтських машинах. Компоненти розподілених систем часто самі є багатопотоковими програмами.

Синхронні паралельні обчислення - третій широкий клас застосувань. Їх мета - швидко вирішити задане завдання або за той же час вирішити більш велике завдання. Приклади синхронних обчислень :

- наукові обчислення, які моделюють та імітують такі явища, як глобальний клімат, еволюція сонячної системи або результат дії нових ліків;
- графіка й обробка зображень, включаючи створення спецефектів в кіно;

– великі комбінаторні або оптимізаційні завдання, наприклад, планування авіаперельотів або економічне моделювання.

Програми рішення таких завдань вимагають великих обчислювальних потужностей. Для досягнення високої продуктивності вони виконуються на паралельних процесорах, причому звичайна кількість процесів (потоків) дорівнює числу процесорів.

У багатопотокових програмах процеси (потоки) взаємодіють, використовуючи змінні, що розділяються (глобальні змінні). У розподілених системах взаємодія процесів забезпечується за допомогою обміну повідомленнями або віддаленого виклику операцій. При виконанні синхронних паралельних обчислень процеси взаємодіють, використовуючи або змінні, що розділяються, або передачу повідомлень, залежно від апаратного забезпечення, на якому виконується програма.

3.2 Парадигми паралельного програмування

Існує велика кількість паралельних програмних застосувань, проте в них використовується лише невелике число моделей рішень, або парадигм. Сукупність методів та засобів для виконання певної діяльності, пов'язаних встановленою схемою їх використання називається парадигмою (прикладом) цієї діяльності.

Зокрема, існує п'ять основних парадигм паралельного програмування:

- ітеративний паралелізм;
- рекурсивний паралелізм;
- «виробники й споживачі» (конвеєри;
- «клієнти та сервери»;
- «взаємодіючі рівні».

З використанням однієї або декількох з цих парадигм і програмуються застосування.

Ітеративний паралелізм використовується, коли в програмі є декілька процесів (часто ідентичних), кожен з яких містить один або декілька

циклів. Таким чином, кожен процес є ітеративною програмою. Процеси програми працюють спільно над рішенням однієї задачі; вони взаємодіють і синхронізуються за допомогою змінних, що розділяються, або передачі повідомлень. Ітеративний паралелізм найчастіше зустрічається в наукових обчисленнях, що виконуються на декількох процесорах.

Рекурсивний паралелізм може використовуватися, коли в програмі є одна або декілька рекурсивних процедур та їх виклики незалежні, тобто кожен з них працює над своєю частиною загальних даних. Рекурсія часто застосовується в імперативних мовах програмування, особливо при реалізації алгоритмів типу «розділяй та володарюй» або «перебір з поверненням» (backtracking). Рекурсія є однією з фундаментальних парадигм в символічних, логічних, функціональних мовах програмування. Рекурсивний паралелізм використовується для вирішення таких комбінаторних проблем, як сортування, планування (завдання комівояжера), ігри (шахи та інші).

«Виробники й споживачі» - це взаємодіючі процеси. Вони часто організовуються в конвеєр, через який проходить інформація. Кожен процес конвеєра являється фільтром, який споживає вихідні дані свого попередника та формує вхідні дані для свого послідовника. Фільтри зустрічаються на рівні застосувань (оболонки) в операційних системах типу ОС Unix, усередині самих операційних систем, усередині прикладних програм, якщо один процес формує вихідні дані, які споживає (читає) інший процес.

«Клієнти та сервери» - найбільш поширена модель взаємодії в розподілених системах, від локальних мереж до World Wide Web. Клієнтський процес запитує сервіс та чекає відповіді. Сервер чекає запитів від клієнтів, а потім діє відповідно до цих запитів. Сервер може бути реалізований як поодинокий процес, який не може обробляти одночасно декілька клієнтських запитів, або (при необхідності паралельного обслуговування запитів) як багатопотокова програма. «Клієнти та сервери»

є паралельним програмним узагальненням процедур та їх викликів: сервер виконує роль процедури, а клієнти її викликають. Але якщо код клієнта і код сервера розміщені на різних машинах, звичайний виклик процедури використовувати не можна. Замість цього необхідно використовувати віддалений виклик процедури або рандеву.

«Взаємодіючі рівні» - остання парадигма взаємодії. Вона зустрічається в розподілених програмах, в яких декілька процесів для вирішення завдання виконують один й той же код та обмінюються повідомленнями. «Взаємодіючі рівні» використовуються для реалізації розподілених паралельних програм, особливо при ітеративному паралелізмі та децентралізованому прийнятті рішень в розподілених системах.

3.3 Модель паралельних обчислень у вигляді графа «операції-операнди»

Для опису існуючих інформаційних залежностей в алгоритмах рішення завдань може бути використана модель у вигляді графа «операції-операнди». При побудові моделі передбачатиметься, що час виконання будь-яких обчислювальних операцій є однаковим і дорівнює 1 (у тих або інших одиницях виміру). Крім того, приймається, що передача даних між обчислювальними пристроями виконується миттєво без яких-небудь витрат часу (що може бути справедливим, наприклад, за наявності загальної пам'яті, що розділяється, в паралельній обчислювальній системі).

Представимо безліч операцій, що виконуються в алгоритмі рішення обчислювальної задачі, та існуючі між операціями інформаційні залежності у вигляді ациклічного орієнтованого графа $G=(V,R)$, де $V=\{1,..,|V|\}$ - безліч вершин графа, що представляють операції алгоритму, а R - безліч дуг графа (при цьому дуга $r=(i, j)$ належить графові тільки тоді, коли операція j використовує результат виконання операції i).

Для прикладу на рис. 3.1 показаний граф алгоритму обчислення площі прямокутника, заданого координатами двох протилежних кутів. Для наведеного прикладу можуть бути використані різні схеми обчислень й побудовані відповідно різні обчислювальні моделі. Різні схеми обчислень мають різні можливості для розпаралелювання й тим самим при побудові моделі обчислень може бути поставлене завдання вибору найбільш відповідної для паралельного виконання обчислювальної схеми алгоритму.

У даній обчислювальній моделі алгоритму вершини без вхідних дуг можуть використовуватися для завдання операцій введення, а вершини без вихідних дуг - для операцій виведення. Позначимо через $\overline{V}$ безліч вершин графа без вершин введення, а через $d(G)$ діаметр (довжину максимального шляху) графа.

Операції алгоритму, між якими немає шляху у рамках вибраної схеми обчислень, можуть бути виконані паралельно (для обчислювальної схеми на рис. 3.1., наприклад, паралельно можуть бути реалізовані спочатку усі операції множення, а потім перші дві операції віднімання).

Нехай p є кількість процесорів, які використовуються для виконання алгоритму. Тоді для паралельного виконання обчислень необхідно задати множину (розклад) $H_p = \{(i, P_i, t_i) : i \in V\}$, у якому для кожної операції $i \in |V|$ вказується номер використаного для виконання операції процесора P_i та час початку виконання операції t_i. Для того, щоб розклад був реалізованим, потрібне виконання наступних вимог при задані множини P:

1) $\forall (i, j)\ j \in V : t_i = t_j \Rightarrow P_i \neq P_j$, тобто один і той же процесор не повинен призначатися різним операціям в один і той же момент часу,

2) $\forall (i, j)\ j \in R \Rightarrow t_j \geq t_i + 1$, тобто до визначеного моменту виконання операції усі необхідні дані вже мають бути обчислені.

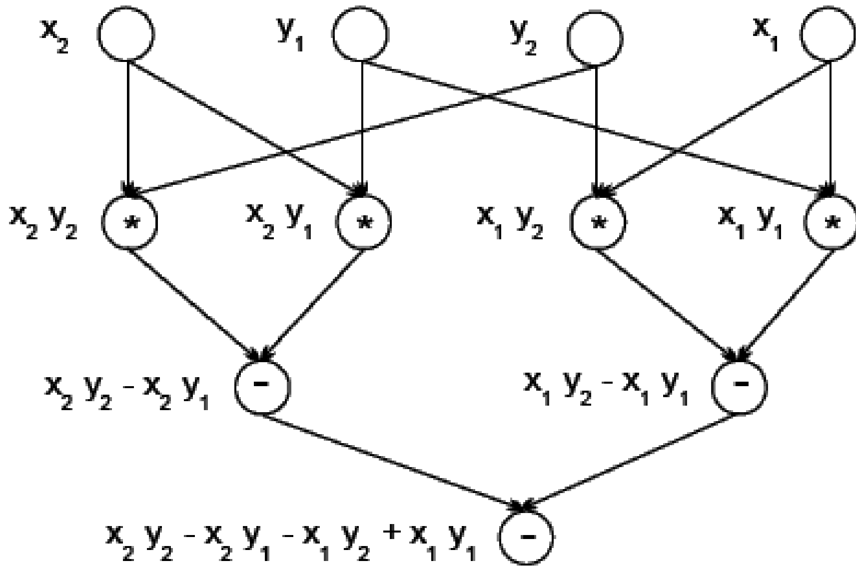


Рис. 3.1. Приклад обчислювальної моделі алгоритму у вигляді графа «операції-операнди»

3.4 Контрольні запитання

1. Парадигми паралельного програмування.
2. Характеристика моделі паралельних обчислень у вигляді графа «операції-операнди»

Детальніше огляд парадигм, алгоритмів паралельних обчислень та програмування розглядаються в книгах [5-9]. На порталі [1] у Інтернеті знаходяться також інші оглядові та навчальні матеріали.

4 МОДЕЛІ СКЛАДНОСТІ ТА ПРОДУКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Відомими одиницями виміру складності для послідовних алгоритмів є часова $T(n)$ складність, що вимірюється кількістю кроків (операцій), які

необхідно виконати, та просторова складність $S(n)$, що вимірюється кількістю пам'яті, яку необхідно мати для роботи алгоритму. Тут n - це розмір вхідних даних, які інтерпретуються відповідно до предметної області задачі. Наприклад, якщо вхідними даними є числа, то таким розміром може бути довжина двійкового запису чисел, для графів - це кількість вершин, для матриць - порядок матриць тощо.

Для паралельних алгоритмів часова та просторова складності також залишаються актуальними, але мають свої особливості. По-перше, вони стають залежними від кількості процесорів p, на яких розв'язується завдання: $Tp(n)$, $Sp(n)$. По-друге, одиницею виміру в $Tp(n)$ тепер є паралельна операція, тобто така, яка представляє одночасно виконувані обчислювальні кроки як одну операцію (а не сумарна кількість операторів, що виконуються в усіх процесорах разом). І, по-третє, сама просторова складність тепер визначається як кількість процесорів (як правило, максимальне), необхідних для вирішення завдання.

4.1 Модель послідовних та паралельних обчислень

Для оцінки складності послідовних обчислень добре відомою є модель RAM (random access memory) машини з пам'яттю довільного доступу [14].

Основними припущеннями цієї моделі є такі:

- модель складається з вхідної стрічки (тільки для читання), початкової стрічки (тільки для запису), процесора та пам'яті;
- пам'ять являє собою нескінченну множину регістрів, здатних зберігати числа довільної розрядності;
- операції RAM-машини складають функціонально повний набір (завантаження регістрів, запис результату, бінарні числові операції тощо).

RAM-машина є, таким чином, універсальною моделлю машини Тюрінга, яка зберігає поліноміальні оцінки часової та просторової складності з точністю до числового коефіцієнта.

Модель паралельних обчислень PRAM є поширенням моделі RAM за рахунок:

1) Збільшення кількості процесорів до $N > 1$, що є параметром моделі й визначається як функція розміру вхідних даних.

2) Надання нескінченній множині регістрів властивості загального доступу з боку процесорів.

3) Розширення множини команд командами читання і запису загальної пам'яті.

Така модель, зазвичай, є абстракцією паралельних обчислювальних машин, в якій не враховані важливі риси реальних систем. Так, в моделі не існує спеціальних засобів синхронізації, але передбачається певна стратегія для вирішення конфліктів при одночасному зверненні до загальної пам'яті декількох процесорів, наприклад, CREW PRAM (Concurent Read, Exclusive Write). Крім того, в моделі PRAM не враховуються втрати часу на обміни даними.

4.2 Прискорення обчислень на паралельній системі . Закон Амдала

Модель PRAM використовується не лише для архітектури із загальною пам'яттю. Ідеально, алгоритм T_1 в RAM повинен мати складність $T_n = T_1/n$ в моделі PRAM, де n - кількість процесорів.

Для оцінки реального прискорення обчислень на паралельній системі використовується коефіцієнт прискорення:

$$S_n = \frac{T_1}{T_n}.$$

Його значення залежить від кількості процесорів. Крім того, на практиці є ще ряд чинників, які можуть значно знижувати S_n, а саме:

– загальні ресурси (пам'ять, канали зв'язку), які можуть стати «вузькими» місцями паралельної системи й понизити її продуктивність;

– паралельні процеси можуть втрачати час, чекаючи виконання умов синхронізації;

– послідовна частина обчислень, яка не може мати паралельної реалізації, також є перешкодою для набуття високих значень коефіцієнта прискорення.

Нехай $T_1 = T_s + T_p$, де T_s - час виконання послідовної частини, а T_p - паралельної (тобто такої, яка допускає розпаралелювання) частини обчислень. Тоді:

$$S_n = \frac{T_1}{T_n} \le \frac{T_s + T_p}{T_s + T_p / n}$$

Введемо величину $x = \frac{T_s}{T_s + T_p}$, яка не залежить від n. Тоді:

$$S_n \le \frac{1}{x + (1 - x)/n}.$$

У такій формі залежність прискорення від співвідношення послідовної та паралельної частини обчислень відома як закон Амдала. Якщо прийняти $x = 1/k$, можна отримати важливий наслідок з цього запису $S_n \le k$. Тобто прискорення завжди обмежене величиною зворотної частини послідовних обчислень. Наприклад, якщо $k = 0,1$, то $S_n \le 10$. Таким чином, $k = 1/x$- максимальна кількість процесорів, на якій раціонально реалізовувати паралельні обчислень для цього завдання.

Закон Амдала показує, що залежність прискорення S_n від кількості процесорів є функцією що не зростає, а що найчастіше убуває.

Для характеристики якісного використання процесорів для паралельних обчислень використовується коефіцієнт ефективності:

$$E_n \le \frac{T_1}{nT_n} = \frac{S_n}{n}$$

З цього визначення слідує, що $0 \le E_n \le 1$, й таким чином коефіцієнт ефективності характеризує прискорення, яке задає один процесор.

Практично важливими є випадки, коли прискорення та ефективність паралельних обчислень є лінійними функціями кількості процесорів. У цих випадках паралельні алгоритми, які вирішують завдання, та самі завдання називаються масштабованими (scalable). Явища суперлінійних прискорень, які іноді можуть попадатися в паралельних обчисленнях, є аномальними й пояснюються:

1) Додатковими апаратними можливостями мультипроцесорної системи.
2) Наявністю ефективного «паралельного пошуку».

4.3 Принцип Брента

Окрім критерію часу $T_n(N)$, процесорів n, прискорення $S_n(N)$ та ефективності $E_n(N)$, використовується також інтегральний показник продуктивності паралельних алгоритмів, - ціна $C_n(N) = T_n(N) \cdot n$. Тобто для послідовних алгоритмів ціна співпадає з його тимчасовою складністю, тому ціну можна сприймати як узагальнення часової доладності послідовних алгоритмів. З цим поняттям пов'язано поняття оптимального паралельного алгоритму, а саме такого, для якого $C_n(N) = O(T_1(N))$.

Оцінка часової складності T_n, яка розглядалася до цього, пов'язана з кількістю процесорів. Але часто необхідно знайти найкращу оцінку часової складності та паралельного алгоритму, тобто таку, яка обчислюється при необмеженій кількості процесорів і від неї вже не залежить. Така величина називається часом необмеженого розпаралелювання і позначається T_∞. Зв'язок між кількістю процесорів і часом паралельного виконання встановлюється загальним принципом Брента, в наступній теоремі.

Теорема. Якщо А - паралельний алгоритм, який вимагає T_∞ кроків та оцінюється в C операцій, то А можна виконати на p процесорах за

$O(C/p+T_{\infty})$ кроків, $p<[n/2]$ та операції бінарні. Теорема Брента характеризує уповільнення обчислень паралельних алгоритмів при використанні ними меншої кількості процесорів.

4.4 Контрольні запитання

1. Моделі послідовних та паралельних обчислень.
2. Прискорення обчислень на паралельній системі. Закон Амдала. Наслідки закону.
3. Оцінка ефективності паралельних алгоритмів

Фундаментальні проблеми складності алгоритмів представлено в [7]. Аналіз продуктивності паралельних систем і їх показники розкриті в книгах [2,3].

5 МЕТОДИ РОЗРОБКИ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

Організація паралельних обчислень визначається, передусім, методом декомпозиції вирішуваної задачі на підзадачі, що у свою чергу визначає вибір структурних даних, засобів паралельного програмування тощо. Нижче розглядаються декілька основних способів декомпозиції, які є найуживанішими при проектуванні паралельних алгоритмів.

5.1 Метод балансованого дерева

При цій стратегії вершинам (корінь) дерева відповідають завдання, проміжні вершини - підзадачам, а термінальні вершини - вхідним даним. Основними структурними даними для таких завдань є вектори, масиви або списки. Якщо дерево підзадач збалансоване, тобто кількість вершин в піддеревах, підлеглих цій вершині, приблизно однакове, тоді можна очікувати приблизно на однаковий час виконання паралельних гілок алгоритму і, отже, на мінімальні непродуктивні витрати часу. Саме ця властивість дає можливість ефективно застосовувати розпаралелювання обчислень.

Для прикладу розглянемо завдання обчислення суми $n = 2k$ чисел, які є елементами масиву:

$$S = \sum_{i=1}^{n} A[\,i\,]\,.$$

Нижче на рисунку представлено «логарифмічну» схему процесів попарного підсумовування та передачі даних між ними. На початковому етапі маємо $n/2$ процесів $P_1,\ldots,P_{n/2}$ (рис. 5.1.).

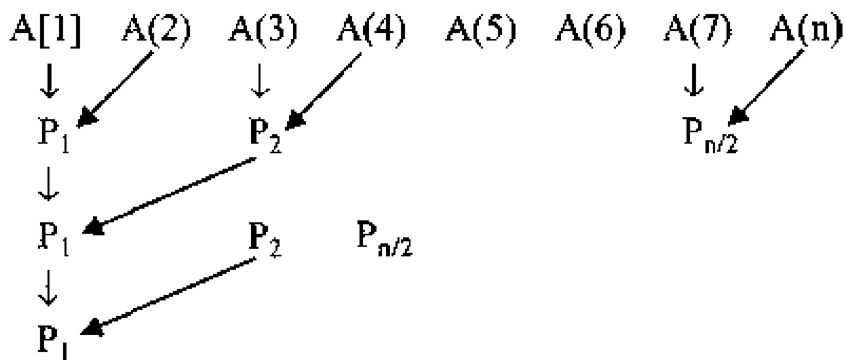


Рис. 5.1. «Схема каскадного підсумовування»

Як видно з рисунка P_i читає $A[2i-1]$ та $A[2i]$ та посилає суму цієї пари чисел до $A[i]$.

На першому етапі обчислюються значення правих частин усіх привласнень, а на другому одночасно відбувається привласнення значень, обчислених на першому етапі, відповідним змінним, визначеними лівими частинами операторів привласнення. Саме двоетапність мультиприсвоєння, а також синхронізованність обчислень в моделі PRAM гарантують інформаційну незалежність ітерацій паралельного циклічного оператора.

Проведемо тепер невелике дослідження продуктивності цього алгоритму. Часова складність алгоритму визначається висотою дерева процесів і складає величину $T_{n/2} = log_2\, n + 1$ та має порядок $O(log_2\, n).$ Програма максимально використовує $n/2$ процесорів, тому прискорення складає $S_{n/2} = (n+1)/(log_2\, n+1)$. При цьому ефективність паралельних обчислень

складає $E_{n/2} = 2(n+1)/n(log_2 n+1)$. Звідси ціна паралельних обчислень визначається як:

$$C_{n/2} = \sum_{i=1}^{log_2 n} \left(\frac{n}{2^i} \right) + 1,$$

яка за порядком $O(n)$. співпадає з величиною часової складності послідовного алгоритму. Отже, цей алгоритм є оптимальним.

До такого типу паралельних алгоритмів належать й інші алгоритми, які вимагають бінарних операцій над числами - елементами масиву, наприклад, знаходження найменшого або найбільшого елементу масиву. Практично, коли $p < n/2$, за принципом Брента маємо $T_p = O(C/p + T_\infty) = O(n/p + log\, 2n)$

Хоча розглянуте вище завдання обчислення суми елементів числового масиву виглядає як така, що відповідає парадигмі паралелізму за даними, метод балансованого дерева тісно не пов'язаний з цією схемою обчислень та може бути реалізований в контексті будь-якої парадигми паралельного програмування.

5.2 Метод «розподіляй та володарюй»

У загальному вигляді стратегія «розподіляй і володарюй» полягає в:

1) Діленні первинного завдання на декілька підзадач приблизно однаковій складності.

2) Рекурсивно-паралельному рішенні підзадач, виділених на попередньому кроці.

3)Використанні результатів підзадач як аргументів для вирішення первинного завдання.

Простим прикладом такого підходу може бути завдання множення двох числових матриць $A[q \cdot r]$ *та* $B[r \cdot s]$. Як відомо, елементи матриці добутку визначаються формулою:

$$C_{ij} = \sum_{k=1}^{r} A_{ik} \cdot B_{kj},$$

де обчислення елементів результуючої матриці не залежать один від одного.

Тому в даному випадку первинне завдання просто розкладається на $q * s$. паралельних підзадач.

Метод «розподіляй та володарюй» в загальному випадку найбільше відповідає схемі обчислень паралелізму за управлінням. Проте в простіших випадках цей метод може бути реалізований і у рамках інших парадигм паралельного програмування. Наприклад, синхронний характер паралелізму обчислень розглянутого вище завдання множення матриць найбільше відповідає парадигмі паралелізму за даними, проте очевидно може бути виконаний за схемою конвеєрного паралелізму та паралелізму за управлінням.

5.3 Контрольні запитання

1. Характеристика методу балансового дерева.
2. Загальна характеристика методу «розподіляй і володарюй».

Фундаментальні проблеми методів розробки алгоритмів, зокрема для моделі RAM, викладені в [16]. Питання методів проектування високопродуктивних паралельних систем розкрито в книгах [2-4, 9]. Довідковий матеріал з цієї теми знаходиться на порталі [1].

РОЗДІЛ ІІ. СИНХРОНІЗАЦІЯ ПРОЦЕСІВ ТА ПОТОКІВ

6 ВЗАЄМНЕ ВИКЛЮЧЕННЯ ПАРАЛЕЛЬНИХ ПРОЦЕСІВ ІЗ ЗАГАЛЬНОЮ ПАМ'ЯТТЮ

Основою для побудови моделей функціонування програм, що реалізовують паралельні методи вирішення завдань, являється поняття процесу як конструктивної одиниці побудови паралельної програми. Опис програми у вигляді набору процесів, що виконуються паралельно на різних процесорах або на одному процесорі в режимі розподілу часу, дозволяє сконцентруватися на розгляді проблем організації взаємодії процесів, визначити моменти й способи забезпечення синхронізації та взаємовиключення процесів, вивчити умови виникнення або довести відсутність безвихідної ситуації в ході виконання програм (ситуацій, в яких усі або частина процесів не можуть бути продовжені при будь-яких варіантах продовження обчислень).

Для мультипроцесорних систем із загальною пам'яттю взаємодію процесів підрозділяють на дві категорії:

– *взаємне виключенні процесів* - коли вони «змагаються» за загальний ресурс та можуть володіти їм або мати доступ до нього тільки окремо;

– *синхронізація процесів,* коли вони «співпрацюють» при рішенні однієї задачі й повинні погоджувати, синхронізувати свої дії при операціях над загальними об'єктами.

6.1 Концепція процесу

Поняття процесу є одним із базових в теорії й практиці паралельного програмування. Трактування цього поняття є досить широким, але в цілому більшість визначень зводяться до розуміння процесу як «деякої послідовності команд програми, що претендує, нарівні з іншими процесами, на використання процесора для свого виконання».

Конкретизація поняття процесу залежить від цілей дослідження паралельних програм. Для аналізу проблем організації взаємодії процесів процес можна розглядати як послідовність команд:

$$p_n = (i_1, i_2, ... i_n) .$$

(для простоти викладу матеріалу припускаємо, що процес описується єдиною командною послідовністю). Динаміка розвитку процесу визначається моментами часу початку виконання команд:

$$t(p_n) = t_p = (\tau_1, \tau_2, .., \tau_1),$$

де $\tau_j, 1 \le j \le n$, є час початку виконання команди τ_j. Послідовність τ_p представляє тимчасову траєкторію розвитку процесу. Припускаючи, що команди процесу виконуються послідовно, в ході своєї реалізації не можуть бути припинені (тобто є неподільними) та мають однакову тривалість виконання, яка дорівнює 1 (у тих або інших часових одиницях), отримаємо, що моменти часу траєкторії процесу повинні задовольняти співвідношенням:

$$\forall, 1 \le i < n \Rightarrow \tau_{i+1} \ge \tau_i + 1 .$$

Рівняння $\tau_{i+1} = \tau_i + 1$ досягається, якщо для виконання процесу виділений процесор і після завершення чергової команди процесу відразу ж починається виконання наступної команди. В цьому випадку говорять, що процес є активним і знаходиться в стані виконання. Співвідношення $\tau_{i+1} > \tau_i + 1$ означає, що після виконання чергової команди процес припинений та чекає можливості для свого продовження. Це призупинення може бути викликане необхідністю розподілу використання єдиного процесора між одночасно виконуваними процесами. В цьому випадку припинений процес знаходиться в стані очікування моменту надання процесора для свого виконання. Крім того, призупинення процесу може бути викликане й тимчасовою неготовністю процесу до подальшого виконання (наприклад, процес може бути продовжений тільки після

завершення операції введення-виведення). У подібних ситуаціях говорять, що процес є блокованим і знаходиться в стані блокування.

В ході свого виконання стан процесу може багаторазово змінюватися; можливі варіанти зміни станів показані на діаграмі переходів (рис. 6.1.).

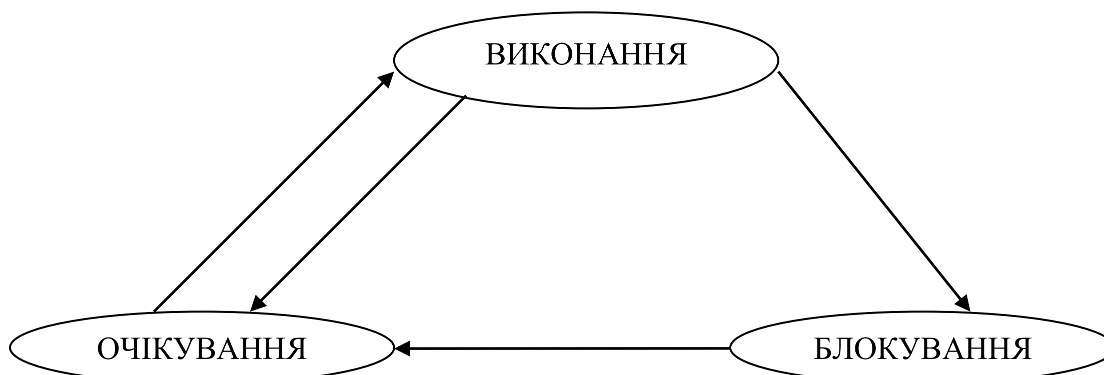


Рис. 6.1. Діаграма переходів процесу із стану в стан

6.2 Поняття ресурсу

Поняття ресурсу зазвичай використовується для позначення будь-яких об'єктів обчислювальної системи, які можуть бути використані процесом для свого виконання. Як ресурс може розглядатися процесор, пам'ять, програми, дані тощо. За характером використання можуть розрізнятися наступні категорії ресурсів:

− ресурси, що використовуються монопольно (неперерозподільчі), характеризуються тим, що виділяються процесам у момент їх виникнення та вивільняються тільки у момент завершення процесів;

− повторно розподільчі ресурси відрізняються можливістю динамічного запиту, виділення й вивільнення в ході виконання процесів (таким ресурсом є, наприклад, оперативна пам'ять);

− ресурси, що розділяються, особливість яких полягає в тому, що вони постійно залишаються в загальному використанні та виділяються процесам для використання в режимі розподілу часу (як, наприклад, процесор, файли, що розділяються тощо);

− багаторазово використовувані (реентерабельні) ресурси характеризуються можливістю одночасного використання декількома

процесами (що може бути забезпечене, наприклад, при незмінності ресурсу під час його використанні; як приклади таких ресурсів можуть розглядатися реентерабельні програми, файли, що використовуються тільки для читання і так далі).

Слід зазначити, що тип ресурсу визначається не лише його конкретними характеристиками, але й залежить від способу використання. Так, наприклад, оперативна пам'ять може розглядатися як повторно розподільчій, так й ресурс, що розділяється; використання програм може бути організоване у вигляді ресурсу будь-якого розглянутого типу.

6.3 Організація програм як системи процесів

Поняття процесу може бути використане як основний конструктивний елемент для побудови паралельних програм у вигляді сукупності взаємодіючих процесів. Така агрегація програми дозволяє отримати компактніші (що піддаються аналізу) обчислювальні схеми методів, що реалізовуються, приховати при виборі способів розпаралелювання несуттєві деталі програмної реалізації, забезпечує концентрацію зусиль на рішення основних проблем паралельного функціонування програм.

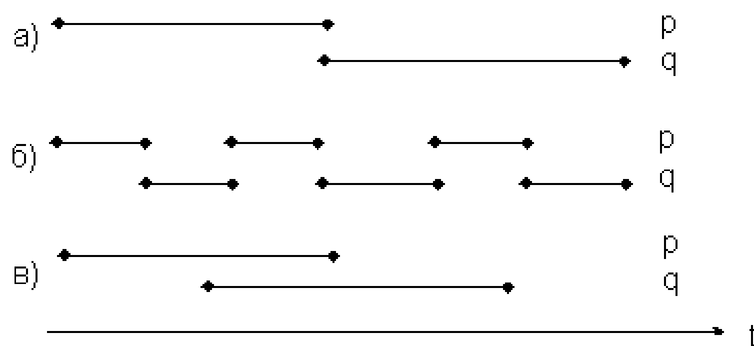


Рис. 6.2. Варіанти взаєморозташування траєкторій одночасно виконуваних процесів (відрізки ліній - фрагменти командних послідовностей процесів)

Існування декількох одночасно виконуваних процесів призводить до появи додаткових співвідношень, які повинні виконуватися для величин тимчасових траєкторій процесів. Можливі типові варіанти таких

співвідношень на прикладі двох процесів p та q полягають в наступному (рис. 6.2.):

- виконання процесів здійснюється строго послідовно, тобто процес q починає своє виконання тільки після повного завершення процесу p (однопрограмний режим роботи ЕОМ - рис. 6.2., а);
- виконання процесів може здійснюватися одночасно, але в кожен момент часу можуть виконуватися команди тільки одного або іншого процесу (режим розподілу часу або багатопрограмний режим роботи ЕОМ - рис. 6.2., б);
- паралельне виконання процесів, коли одночасно можуть виконуватися команди декількох процесів (цей режим виконання процесів можливий тільки за наявності в обчислювальній системі декількох процесорів - рис. 6.2., в).

Приведені варіанти взаєморозташування траєкторій процесів визначаються не вимогами необхідних функціональних взаємодій процесів, а є лише наслідком технічної реалізації одночасної роботи декількох процесів. З іншого боку, можливість чергування за часом командних послідовностей різних процесів слід враховувати при реалізації процесів.

Виділені особливості одночасного виконання декількох процесів можуть бути сформульовані у вигляді ряду принципових положень, які повинні враховуватися при розробці паралельних програм:

- моменти виконання командних послідовностей різних процесів можуть чергуватися за часом;
- між моментами виконання команд різних процесів можуть виконуватися різні тимчасові співвідношення; характер цих співвідношень залежить від кількості й швидкодії процесорів та завантаження обчислювальної системи і тому не може бути визначений заздалегідь;
- тимчасові співвідношення між моментами виконання команд можуть розрізнятися при різних запусках програм на виконання, тобто

одній і тій же програмі при одних і тих же початкових даних можуть відповідати різні командні послідовності внаслідок різних варіантів чергування моментів роботи різних процесів;

- доказ достовірності отримуваних результатів повинен проводитися для будь-яких можливих тимчасових співвідношень для елементів тимчасових траєкторій процесів;
- для виключення залежності результатів виконання програми від порядку чергування команд різних процесів потрібний аналіз ситуацій взаємовпливу процесів та розробка методів для їх виключення.

Перераховані моменти свідчать про істотне підвищення складності паралельного програмування в порівнянні з розробкою «традиційних» послідовних програм.

6.4 Взаємодія та взаємовиключення процесів

Однією з причин залежності результатів виконання програм від порядку чергування команд може бути розподіл одних й тих же даних між одночасно виконуваними процесами (наприклад, як це здійснюється у вище розглянутому прикладі). Ця ситуація може розглядатися як прояв загальної проблеми використання ресурсів (загальних даних, файлів, пристроїв тощо), що розділяються. Для організації розподілу ресурсів між декількома процесами необхідно мати можливість:

- визначення доступності затребуваних ресурсів (ресурс вільний і може бути виділений для використання, ресурс вже зайнятий одним з процесів програми та не може використовуватися додатково яким-небудь іншим процесом);
- виділення вільного ресурсу одному з процесів, що запитали ресурс для використання;
- призупинення (блокування) процесів, що видали запити на ресурси, зайняті іншими процесами.

Головною вимогою до механізмів розподілу ресурсів є гарантоване забезпечення використання кожного ресурсу, що розділяється, тільки одним процесом від моменту виділення ресурсу цьому процесу до моменту звільнення ресурсу. Ця вимога називається взаємовиключенням процесів; командні послідовності процесів, в яких процес використовує ресурс, називається *критичною секцією процесу*. З використанням останнього поняття, умова взаємовиключення процесів може бути сформульована як вимога знаходження в критичних секціях процесу ресурсу, що розділяється.

Розглянемо декілька варіантів програмного рішення проблеми взаємовиключення (для запису програм використовується мова програмування C++). У кожному з варіантів пропонуватиметься деякий частковий спосіб взаємовиключення процесів з метою демонстрації усіх можливих ситуацій при використанні загальних ресурсів, що розділяються. Послідовне удосконалення механізму взаємовиключення при розгляді варіантів приведе до викладу алгоритму Деккера, що забезпечує взаємовиключення для двох паралельних процесів. Обговорення способів взаємовиключення завершується розглядом концепції семафорів Дейкстри, які можуть бути використані для загального вирішення проблеми взаємовиключення будь-якої кількості взаємодіючих процесів.

Варіант 1

```
intProcessNum = 1; // номер процесу для доступу до ресурсу
Process_1() {
   while (1){
//   повторювати, поки право доступу до ресурсу у процесу 2
   while ( ProcessNum == 2 );
   < Використання загального ресурсу >
   // передача права доступу до ресурсу процесу 2
    ProcessNum = 2;
    }
```

```
 }
Process_2()
  {
    while (1){
 //   // повторювати, поки право доступу до ресурсу у процесу 1
    while ( ProcessNum == 1 );
    < Використання загального ресурсу >
   // передача права доступу до ресурсу процесу 1
    ProcessNum = 1;
    }
 }
```

Реалізований в програмі спосіб гарантує взаємовиключення, проте такому рішенню властиві два істотні недоліки:

- ресурс використовується процесами строго послідовно (по черзі) і, як результат, при різному темпі розвитку процесів загальна швидкість виконання програми визначатиметься найбільш повільним процесом;
- при завершенні роботи якого-небудь процесу інший процес не зможе скористатися ресурсом та може виявитися в постійно заблокованому стані.

Вирішення проблеми взаємовиключення відоме в літературі як спосіб жорсткої синхронізації.

Варіант 2

У цьому варіанті для відходу від жорсткої синхронізації використовуються дві керуючі змінні, які фіксують використання процесами ресурсу, що розділяється.

```
int ResourceProc1 = 0; // = 1 - ресурс зайнятий  процесом 1
int ResourceProc2 = 0; // = 1 - ресурс зайнятий  процесом 2
Process_1() {
  while (1){
// повторювати, поки ресурс використовується процесом 2
```

```
            while ( ResourceProc2 == 1 );
            ResourceProc1 = 1;
            < Використання загального ресурсу >
            ResourceProc1 = 0;
        }
    }
    Process_2()
        {       while (1){
// повторювати, поки ресурс використовується процесом 1
            while ( ResourceProc1 == 1 );
            ResourceProc2 = 1;
            < Використання загального ресурсу >
            ResourceProc2 = 0;      }
    }
```

Запропонований варіант розподілу ресурсів усуває недоліки жорсткої синхронізації, проте при цьому втрачається гарантія того, що при взаємовиключенні обидва процеси можуть виявитися одночасно у своїх критичних секціях (це може статися, наприклад, при перемиканні між процесами у момент завершення перевірки зайнятості ресурсу). Ця проблема виникає внаслідок відмінності моментів перевірки та фіксації зайнятості ресурсу.

Слід зазначити, що в окремих випадках взаємовиключення процесів в цьому прикладі може виконуватися й коректно - усе визначається конкретними моментами перемикання процесів. Звідси слідує два важливі висновки:

- успішність одноразового виконання не може бути доказом правильності функціонування паралельної програми навіть при незмінних параметрах вирішуваної задачі;
- для виявлення помилкових ситуацій потрібна перевірка різних тимчасових траєкторій виконання паралельних процесів.

Варіант 3

Можливий варіант відновлення взаємовиключення може полягати в установці значень змінних управління перед циклом перевірки зайнятості ресурсу.

```
int ResourceProc1 = 0; // = 1 - ресурс зайнятий процесом 1
int ResourceProc2 = 0; // = 1 - ресурс зайнятий процесом 2
Process_1() {
   while (1){
      // встановити, що процес 1 намагається зайняти ресурс
      ResourceProc1 = 1;
     // повторювати, поки ресурс зайнятий процесом 2
      while ( ResourceProc2 == 1 );
      < Використання загального ресурсу >
      ResourceProc1 = 0;
   }
}
Process_2()
   {
   while (1){
// встановити, що процес 2 намагається зайняти ресурс
      ResourceProc2 = 1;
  // повторювати, поки ресурс використовується процесом 1
      while ( ResourceProc1 == 1 );
      < Використання загального ресурсу >
      ResourceProc2 = 0;
   }
}
```

Представлений варіант відновлює взаємовиключення проте при цьому виникає нова проблема - обидва процеси можуть виявитися заблокованими внаслідок нескінченного повторення циклів очікування

звільнення ресурсів (відбувається при одночасній установці змінних управління в стан «зайнято»). Ця проблема відома під назвою ситуації безвиході («дедлок»). Виключення безвиході є одним з найбільш важливих завдань в теорії і практиці паралельних обчислень.

Варіант 4

Пропонований підхід для усунення безвиході полягає в організації тимчасового зняття значення зайнятості змінних процесів в циклі очікування ресурсу.

```
int ResourceProc1 = 0; // =1 - ресурс зайнятий процесом 1
int ResourceProc2 = 0; // =1 - ресурс зайнятий процесом 2
Process_1() {
  while (1){
    ResourceProc1 = 1; // процес 1 намагається зайняти ресурс
// повторювати, поки ресурс зайнятий процесом 2
    while ( ResourceProc2 == 1 ){
      ResourceProc1 = 0; // зняття зайнятості ресурсу
      < тимчасова затримка >
      ResourceProc1 = 1;
    }
    < Використання загального ресурсу >
    ResourceProc1 = 0;     }   }
Process_2()
  {
  while (1){
    ResourceProc2 = 1; // процес 2 намагається зайняти ресурс
// повторювати, поки ресурс використовується процесом 1
    while ( ResourceProc1 == 1 ){
      ResourceProc2 = 0; // зняття зайнятості ресурсу
      < тимчасова затримка >
```

```
            ResourceProc2 = 1;
        }
        < Використання загального ресурсу >
        ResourceProc2 = 0;
    }
}
```

Тривалість тимчасової затримки в циклах очікування повинна визначатися за допомогою деякого випадкового датчика. За таких умов реалізований алгоритм забезпечує взаємовиключення та виключає виникнення безвиході, але знову таки не позбавлений вагомого недоліку. Проблема полягає в тому, що потенціальне вирішення питання про виділення може відкладатися до нескінченності (при синхронному виконанні процесів). Ця ситуація відома під найменуванням нескінченне відкладення (starvation).

6.5 Алгоритм Деккера

У алгоритмі Деккера пропонується об'єднання пропозицій варіантів 1 та 4 для вирішення проблеми взаємовиключення.

```
int ProcessNum=1; // номер процесу для доступу  до ресурсу
int ResourceProc1 = 0; // = 1 - ресурс зайнятий  процесом 1
int ResourceProc2 = 0; // = 1 - ресурс зайнятий  процесом 2
Process_1() {
    while (1){
        ResourceProc1 = 1; // процес 1 намагається зайняти ресурс
 /* цикл очікування доступу до ресурсу */
        while ( ResourceProc2 == 1 ){
            if ( ProcessNum == 2 ){
                ResourceProc1 = 0;
    // повторювати, поки ресурс зайнятий процесом 2
                while ( ProcessNum == 2 );
```

```
            ResourceProc1 = 1;          }     }
        < Використання загального ресурсу >
        ProcessNum  = 2;
        ResourceProc1 = 0;
    }   }
Process_2()
    {
    while (1){
        ResourceProc2 = 1; // процес 2 намагається зайняти ресурс
 /* цикл очікування доступу до ресурсу */
        while ( ResourceProc1 == 1 ){
          if ( ProcessNum == 1 ){
            ResourceProc2 = 0;
// повторювати, поки ресурс використовується процесом 1
            while ( ProcessNum == 1 );
            ResourceProc2 = 1;
          }
        }
        < Використання загального ресурсу >
        ProcessNum = 1;
        ResourceProc2 = 0;       }     }
```

Алгоритм Деккера гарантує коректне рішення проблеми взаємовиключення для двох процесів. Керуючі змінні ResourceProc1, ResourceProc1 забезпечують взаємовиключення, змінна ProcessNum унеможливлює нескінченне відкладення. Якщо обидва процеси намагаються отримати доступ до ресурсу, то процес, номер якого вказаний в ProcessNum, продовжує перевірку можливості доступу до ресурсу (зовнішній цикл очікування ресурсу). Інший же процес в цьому випадку знімає свій запит на ресурс, чекає своєї черги доступу до ресурсу (внутрішній цикл очікування) та поновлює свій запит на ресурс.

Алгоритм Деккера може бути узагальнений на випадок довільної кількості процесів, проте, таке узагальнення призводить до помітного ускладнення виконуваних дій. Крім того, програмне рішення проблеми взаємовиключення процесів призводить до нераціонального використання процесорного часу ЕОМ (процесу, який очікує звільнення ресурсу, постійно виділяється процесорний час для перевірки можливості продовження - активне очікування (busy wait)).

6.6 Семафори Дейкстри

Під семафором *S* зазвичай розуміється змінна особливого типу, значення якої може опитуватися та змінюватися тільки за допомогою спеціальних операцій *P(S)* й *V(S)*, що реалізовуються відповідно до наступних алгоритмів:

операція P(S)

```
якщо S>0
  то S = S - 1
  інакше
  < чекати S >
```

операція V(S)

```
якщо      < один або декілька процесів чекають S >
  то < зняти очікування у одного із очікуючих процесів >
  інакше
  S = S + 1
```

Принциповим в розумінні семафорів є те, що операції *P(S)* й *V(S)* передбачаються неподільними, що гарантує взаємовиключення при використання загальних семафорів (забезпечення неподільності операції обслуговування семафорів зазвичай реалізуються засобами операційної системи).

Розрізняють два основні типи семафорів. Двійкові семафори набувають тільки значень *0* та *1*, область значень загальних семафорів -

додатні цілі значення. У момент створення семафори ініціалізуються деяким цілим значенням.

Семафори широко використовуються для синхронізації та взаємовиключення процесів. Так, наприклад, проблема взаємовиключення за допомогою семафорів може мати наступне просте рішення.

```
Semaphore
    Mutex=1; // семафор взаємовиключення  процесів
Process_1() {
    while (1){
  // перевірити семафор та чекати, якщо ресурс зайнятий
        P(Mutex);
        < Використання загального ресурсу >
     // звільнити один з очікуючих ресурсу процесів
 // збільшити семафор, якщо немає очікуючих процесів
        V(Mutex);  }    }
Process_2() {
    while (1){
 // перевірити семафор та чекати, якщо ресурс зайнятий
        P(Mutex);
        < Використання загального ресурсу >
 // звільнити один з очікуючих ресурсу процесів
  // збільшити семафор, якщо немає очікуючих процесів
        V(Mutex);       }   }
```

Наведений приклад розглядає взаємовиключення тільки двох процесів, але, як можна помітити, абсолютно аналогічно може бути організоване взаємовиключення   довільної кількості процесів.

6.7 Контрольні запитання

1. Поняття процесу, його стани (виконання, очікування, блокування).
2. Визначення ресурсу, категорії ресурсів.

3. Поняття взаємодії та взаємовиключення процесів.
4. Алгоритм Деккера.
5. Семафори Дейкстри.

Питання організації роботи процесів та потоків висвітлені в книгах [1,12,13]. Огляд механізмів синхронізації наведено в [10]. На порталі [1] в мережі Інтернеті можна знайти також інші оглядові й навчальні матеріали з цих питань.

7 ПРОЦЕСИ І ПОТОКИ В ОС WINDOWS

Процесом (process) називається екземпляр програми, завантаженої в пам'ять. Цей екземпляр може створювати потоки (thread), які є послідовністю інструкцій на виконання. Важливо розуміти, що виконуються не процеси, а саме потоки. Причому будь-який процес має хоча б один потік. Цей потік називається головним (основним) потоком додатку.

Оскільки практично завжди потоків значно більше, чим фізичних процесорів для їх виконання, то потоки насправді виконуються не одночасно, а по черзі. (розподіл процесорного часу відбувається саме між потоками.) Але перемикання між ними відбувається так часто, що здається ніби вони виконуються паралельно.

Залежно від ситуації потоки можуть знаходитися в трьох станах. По-перше, потік може виконуватися, коли йому виділений процесорний час, тобто він може знаходитися в стані активності. По-друге, він може бути неактивним та чекати виділення процесора, тобто бути в стані готовності. І є ще третій, теж дуже важливий стан - стан блокування. Коли потік заблокований, йому взагалі не виділяється час. Зазвичай блокування ставиться на час очікування якої-небудь події. При виникненні цієї події потік автоматично переводиться із стану блокування в стан готовності. Наприклад, якщо один потік виконує обчислення, а інший повинен чекати результатів, щоб зберегти їх на диск. Другий потік може заблокувати себе до тих пір, поки перший не встановить сигнальну подію, що читання закінчене.

У системі виділяються два види потоків - інтерактивні, такі, що реалізують свій цикл обробки повідомлень (наприклад, головний потік програми), та робочі, що є простою функцією. У другому випадку потік завершується під час завершення виконання цієї функції.

Заслуговує уваги також спосіб організації черговості потоків. Можна було б, звичайно, обробляти усі потоки за чергою але такий спосіб не завжди найефективніший. Набагато ефективнішим виявилося ранжувати усі потоки за пріоритетами. Пріоритет потоку позначається числом від 0 до 31 та визначається виходячи із пріоритету процесу, що «породив» потік й відносного пріоритету самого потоку. Таким чином, досягається найбільша гнучкість, кожен потік в ідеалі отримує стільки часу, скільки йому необхідно.

Іноді пріоритет потоку може змінюватися динамічно. Так інтерактивні потоки, що мають зазвичай клас пріоритету normal, система обробляє інакше та дещо підвищує фактичний пріоритет таких потоків, коли процес, який їх породив, знаходиться на передньому плані (foreground). Це зроблено для того, щоб застосування, з якому в даний момент працює користувач, швидше реагувало на його дії.

7.1 Створення потоків

Для створення потоку на мові C++ використовується функція CreateThread().

Синтаксис функції CreateThread():

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES
lpThreadAttributes,
 // дескриптор захисту
DWORD dwStackSize, //початковий розмір стеку
LPTHREAD_START_ROUTINE lpStartAddress, // функція потоку
LPVOID lpParameter, // параметр потоку
DWORD dwCreationFlags, // опції створення
```

LPDWORD lpThreadId); //ідентифікатор потоку

Перший та четвертий параметри можуть бути рівними NULL, другий та п'ятий дорівнює 0. Представляють інтерес третій та шостий параметри (lpStartAddress і lpThreadId).

lpStartAddress - це адреса функції, код якої виконуватиметься у новому потоці. lpThreadId - адреса ідентифікатора нового потоку.

Функція, код якої виконуватиметься у новому потоці повинна мати стандартизоване визначення: тип значення, яке функція повертає, unsigned long (чи DWORD), а аргумент всього один - посилання на тип void (LPVOID). Також ця функція повинна мати модифікатор WINAPI.

Головними аргументами цієї функції є:

- покажчик на функцію потоку (функцію, в якій виконуватиметься потік, або по-іншому - яка виконуватиме дії потоку);

- єдиний аргумент типу void* (чи lpvoid), значення якого буде передано у функцію потоку. За допомогою цього покажчика можна передавати будь-які дані в новий з місця його створення. Наприклад, можна виділити частину пам'яті, записати в нього дані, необхідні новому потоку, та передати покажчик на цей розділ пам'яті у функцію createthread.

В результаті виконання функції createthread буде створений новий потік, функція якого почне виконуватися відразу ж (чи буде призупинена до виклику resumethread). Функція createthread повертає спеціальне значення типу handle - хэндл потоку, який може бути використаний для призупинення, знищення потоку, синхронізації. При створенні кожному потоку також призначається унікальний id.

При кожному виклику CreateThread створюється об’єкт ядро – компактна структура, яка використовується ОС для управління потоком. У випадку коли об’єкт ядра потік не вдалось створити, то функція повертає NULL. Роботою потоків керує ОС, зокрема планувальник потоків. Змінити алгоритм роботи планувальника неможливо.

Приклад виклику CreateThread ():

```
#include <windows.h>
#include <conio.h>
DWORD WINAPI ThreadFunc( LPVOID lpParam )
{   char szMsg[80];
    wsprintf( szMsg, "Parameter = %d.", *(DWORD*)lpParam );
    MessageBox( NULL, szMsg, "ThreadFunc", MB_OK );
    return 0;
}
 void main( void )
{    DWORD dwThreadId, dwThrdParam = 1;
    HANDLE hThread;
    char szMsg[80];
    hThread = CreateThread(
        NULL,          // безпека за замовчуванням
        0,             // стек за замовчуванням
        ThreadFunc,    // функція потоку
        &dwThrdParam,        // параметр функції потоку
        0,             // потік виконувати негайно
        &dwThreadId);        // індентифікатор потоку
   // Перевірка створення потоку
   if (hThread == NULL)
   {   wsprintf( szMsg, "CreateThread failed." );
      MessageBox( NULL, szMsg, "main", MB_OK );   }
   else    {
      _getch();
      CloseHandle( hThread );   }
```

Функція потоку може виконувати будь-які завдання.

Після виконання потоку, його стек знищується, а лічильник користувачів об’єкта зменшується на одиницю.

Основні характеристики функції потоку:

−Ім'я функції потоку (за винятком головного) може бути довільним. Якщо різні потоки використовують різні за змістом функції, то їх імена повинні бути унікальними.

−На відміну від функції головного потоку, для функцій користувацьких потоків передається тільки один параметр. Тому тут немає проблем ANSI/UNICODE.

−Функції потоку повинні завжди повертати значення, яке буде трактуватись значенням, яке повертає потік.

−Функції потоків повинні працювати із власними локальними змінними і мінімально використовувати глобальні змінні.

−До завершення виконання функції потоку треба уникати знищення потоку.

Потік завершується одним із чотирьох способів:

1. Функція потоку повертає управління (рекомендований спосіб) .
2. Потік самознищується викликом void ExitThread(DWORD *dwExitCode*); (небажаний спосіб).
3. Потік знищується (або самознищується) викликом BOOL TerminateThread (HANDLE *hThread*, DWORD *dwExitCode*) (небажаний спосіб).
4. Завершується потік, який володіє процесом.

- *При першому способі*:
  - Знищуються усі об'єкти C++ відповідними деструкторами.
  - Коректно знищується стек потоку.
  - Код завершення потоку коректно встановлюється
  - Лічильник користувачів потоку зменшується на 1.
- *При другому способі* коректно знищуються лише ресурси виділені ОС. Але ресурси C++ не знищуються. Код завершення при цьому задається параметром функції ExitThread - *dwExitCode*.

- *При третьому способі* можна знищити будь-який потік, на який вказує *hThread*, із кодом завершення *dwExitCode*. Ця функція є асинхронна, а тому на результати її дії треба зачекати через WaitForSingleObject. Після виклику TerminateThread (на відміну від двох попередніх випадків) не знищується стек потоку. Окрім того при завершенні потоку система повідомляє про це усі DLL, які підключені до процесу, але при виклику TerminateThread – цього не робиться.

При завершенні потоку:

- Дескриптори усіх USER-об'єкти, які створені потоком, звільняються. Власником потік виступає лише по відношенню до двох об'єктів: вікон (windows) та пасток (hooks). Саме вони і знищуються. Усі решта об'єктів належать процесу лише після звільнення посилань на них.
- Код завершення процесу міняється із значення STILL_ACTIVE на код, переданий в ExitThread або TerminateThread.
- Об'єкт ядра переходить у вільний (незайнятий, signaled) стан.
- Якщо даний потік є останнім активним потоком в процесі, то завершується і сам процес.
- Лічильник об'єкта ядра зменшується на 1. Сам об'єкт буде знищений як тільки лічильник стане рівний 0.

Код завершення потоку можна дізнатись через функцію BOOL GetExitCodeThread (HANDLE *hThread*, LPDWORD *lpExitCode*);

Внутрішню структуру потоку зображено на рисунку 7.1.

В об'єкті ядро - потік є змінна – лічильник простоювань, яка при значенні відмінному від нуля призупиняє роботу потоку. При створенні кожен потік має лічильник простоювання рівний 1. Якщо у функцію CreateProcess не передано прапорець CREATE_SUSPENDED, то лічильник обнуляється і потік включається до планованих. В противному випадку він вважається призупиненим.

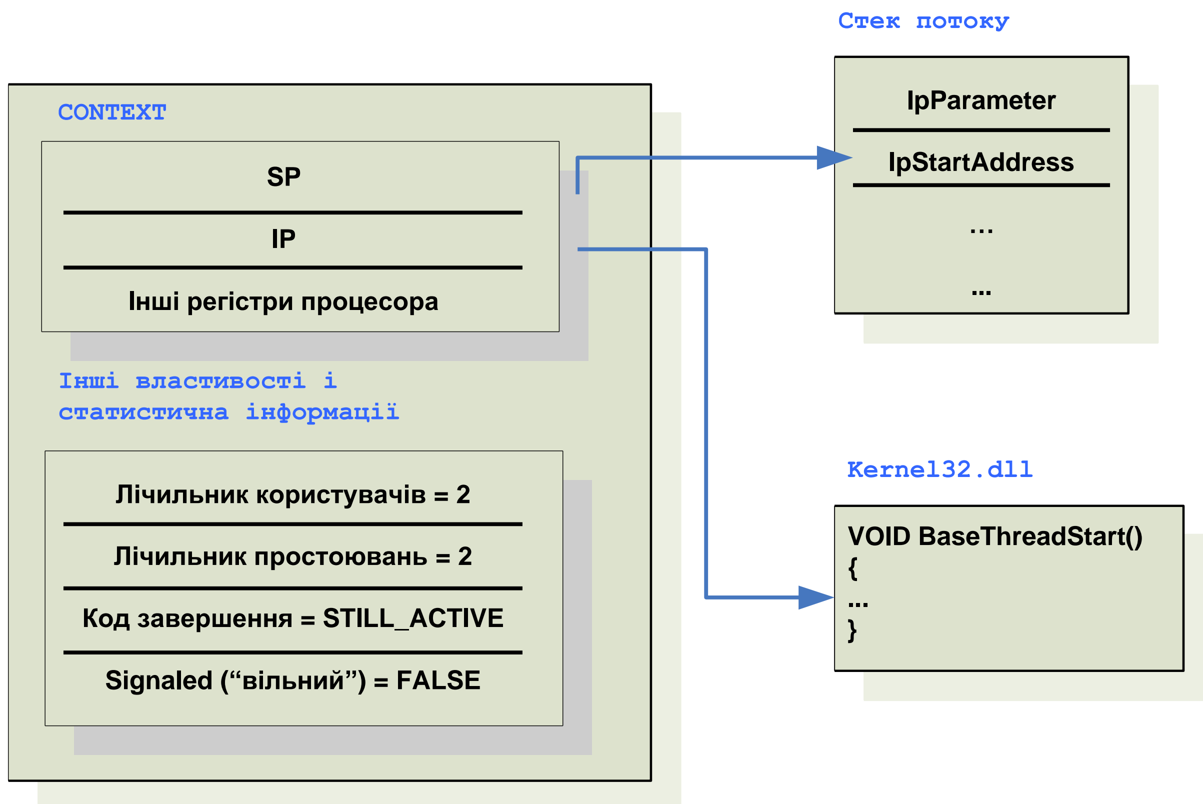


Рис. 7.1. Внутрішня структура потоку

Для відновлення виконання потоку треба використати функцію DWORD ResumeThread (HANDLE hThread), яка за параметром hThread дає можливість відновити виконання будь-якого потоку.

Якщо потік призупинений три рази, то відновлюватись він повинен також три рази.

Призупиняється виконання потоку функцією DWORD SuspendThread(HANDLE hThread), яка за параметром hThread дає можливість призупинити виконання будь-якого потоку.

Функція SuspendThread працює в асинхронному режимі. У зв’язку з цим операція призупинення є небезпечною, оскільки невідомо у який момент потік буде призупинений.

Обидві функції повертають попереднє значення лічильника простоювань.

Потік можна призупиняти не більше MAXIMUM_SUSPEND_COUNT.

Потік себе може призупинити, але відновлюватись він повинен з іншого потоку чи процесу.

Потік можливо призупинити на деякий проміжок часу, який задається в мілісекундах void Sleep(DWORD dwMilliseconds).

Деякі важливі моменти пов'язані з функцією Sleep:

1) Викликаючи Sleep, потік добровільно відмовляється від залишку процесорного часу.

2) Визначаючи час засипання, треба пам'ятати, що реально час відновлення потоку буде більшим.

3) Якщо у Sleep передати INFINITE можна взагалі заборонити планування потоку.

4) Якщо у Sleep передати нуль, то система здійснить підключення до процесора іншого потоку. Якщо інших планованих потоків з таким пріоритетом в системі немає, то ядро відновить виконання поточного.

5) Підключається черговий потік до процесора функцією BOOL SwitchToThread(void), яка потребує процесорний час. Якщо такого потоку немає, то управління повертається активному потоку.

6) Функція SwitchToThread дозволяє перехопити процесор у потоку, який має нижчий пріоритет.

7) Значення FALSE повертається у випадку, коли у системі немає жодного потоку, готового до виконання.

Визначити поточний час (у мілісекундах) від моменту запуску потоку дає змогу функція DWORD GetTickCount(void). При цьому у цей час закладається і час очікування потоком ресурсів процесора.

Детальніше часові періоди роботи потоку дає змогу визначити функція BOOL GetThreadTimes(HANDLE *hThread*, LPFILETIME *lpCreationTime*, LPFILETIME *lpExitTime*, LPFILETIME *lpKernelTime*, LPFILETIME *lpUserTime*);

- *lpCreationTime* – час створення потоку;

- *lpExitTime* – час завершення потоку (значення невизначене, якщо потік ще виконується);
- *lpKernelTime* - час, затрачений на виконання коду ОС;
- *lpUserTime* - час, затрачений на виконання користувацького коду;

Приклад: Необхідно підрахувати суму усіх елементів масиву із 10 000 елементів, де значення i-го елементу задається відповідною формулою:

$$a_i = \sin i + i^2,\ S = \sum_{i=1}^{10000} a_i,\ S\text{ - ?}$$

Стандартне рішення (непаралельне) :

```
for(int i=1; i<10001; i++)
{
a[i]=sin(i)+i*i;
sum=sum+a[i];
}
```

Дана задача розпаралелюється на два потоки.

Код для «паралельної» програми на мові C/C++ буде наступним:

```
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <windows.h>
const int N=10000000;
double S2=0;
long start,stop;
unsigned long WINAPI parallel(void* param)
{
 int i;
 double ai;
 for(i=start;i<=stop;i++)
 {
```

```
        ai=sin(i)+i*i;
        S2=S2+ai;
 }
 return 1; }
int main(int argc, char* argv[])
{
long double ai,S1=0,S;
time_t time1,time2;
HANDLE hThr;
unsigned long ThrId;
time(&time1);
start=0;
stop=(int)N/2;
if(!(hThr=CreateThread(NULL,0,parallel,NULL,0,&ThrId)))
{
 printf("\nCannot create thread!");      return 0;
}
S1=0;
printf("Computing...");
for(int i=(int)N/2+1;i<=N;i++)
{
 ai=sin(i)+i*i;
 S1=S1+ai;
}
WaitForSingleObject(hThr,INFINITE);
time(&time2);
S=S1+S2;
printf("\nS= %lf, time= %d s\n",S,time2-time1);
return 0; }
```

Усі потоки народжуються із пріоритетом normal. Список визначених констант пріоритету потоку наведено в таблиці 7.1.

Таблиця 7.1.-Константи, що визначають пріоритет потоку

Константа	Призначення
THREAD_PRIORITY_ABOVE_NORMAL	Встановлює пріоритет на один пункт вище нормального
THREAD_PRIORITY_BELOW_NORMAL	Встановлює пріоритет на один пункт нижче нормального
THREAD_PRIORITY_HIGHEST	Встановлює пріоритет на два пункти вище нормального
THREAD_PRIORITY_IDLE	Встановлює базовий пріоритет процесу рівним 1. Для процесу realtime_priority_class встановлює пріоритет рівним 16
THREAD_PRIORITY_LOWEST	Встановлює пріоритет на два пункти нижче нормального
THREAD_PRIORITY_NORMAL	Встановлює нормальний пріоритет
THREAD_PRIORITY_TIME_CRITICAL	Встановлює базовий пріоритет процесу рівним 15. Для realtime_priority_class встановлює пріоритет рівним 30.

Для того, щоб визначити відносний рівень пріоритету треба використати функцію

int GetThreadPriority(HANDLE hThread).

Зміна пріоритету здійснюється функцією:

BOOL SetThreadPriority (HANDLE hThread, int nPriority);

Для того, щоб його змінити, треба його породжувати призупиненим, змінити пріоритет, а далі запустити потік, наприклад:

```
DWORD dwThreadId;
HANDLE hThread = CreateThread(NULL, 0,ThreadFunc,NULL,
CREATE_SUSPENDED, &dwThreadId);
SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST);
ResumeThread(hThread);
CloseHandle(hThread);
```

Рівень пріоритету, який утворюється комбінацією відносного рівня пріоритету та пріоритету процесу, називається базовим рівнем пріоритету потоку. Для потоків із базовим рівнем від 1 до 15 (область динамічного пріоритету) система може автоматично змінювати пріоритет на вищий за умови наявності вільного процесорного часу. Такий процес називається динамічною зміною пріоритету. Пріоритет не може піднятись вище значення 15.

Аналогічно при появі потоків з вищим пріоритетом, у потоках із динамічно-зміненим пріоритетом, останній буде понижуватись але тільки до базового рівня.

Для встановлення чи відміни режиму динамічного управління пріоритетом визначено функції:

BOOL SetProcessPriorityBoost(HANDLE hProcess, BOOL DisablePriorityBoost);

BOOL SetThreadPriorityBoost(HANDLE hThread, BOOL DisablePriorityBoost);

Перша з них через другий параметр встановлює чи відміняє режим для усіх потоків процесу hProcess, а друга – тільки до потоку hThread.

7.2 Синхронізація

Отже, в Windows виконуються не процеси, а потоки. При створенні процесу автоматично створюється його основний потік. Цей потік в процесі виконання може створювати нові потоки, які, у свою чергу, теж можуть створювати потоки і так далі. Процесорний час розподіляється саме між потоками, і виходить, що кожен потік працює незалежно.

Усі потоки, що належать одному процесу, розділяють деякі загальні ресурси - такі, як адресний простір оперативної пам'яті або відкриті файли. Ці ресурси належать усьому процесу, а значить, і кожному його потоку. Отже, кожен потік може працювати з цими ресурсами без яких-небудь обмежень. Але чи так це насправді? Згадаємо, що у Windows реалізована

витісняюча багатозадачність - це означає, що у будь-який момент система може перервати виконання одного потоку й передати управління іншому. (Раніше використовувався спосіб організації, який називався кооперативною багатозадачністю. Система чекала, поки потік сам не зволить передати їй управління. Саме тому у разі «глухого» зависання одного застосування доводилося перезавантажувати комп'ютер. Так була організована, наприклад, Windows 3.1). Що станеться, якщо один потік ще не закінчив працювати з яким-небудь загальним ресурсом, а система перемкнулася на інший потік, що використовує той же ресурс? Відбудеться конфлікт й результат роботи цих потоків буде надзвичайно сильно відрізнятися від задуманого. Такі конфлікти можуть виникнути й між потоками, що належать різним процесам. Завжди, коли два або більше потоки використовують який-небудь загальний ресурс, виникає ця проблема.

Саме тому потрібний механізм, що дозволяє потокам погоджувати свою роботу із загальними ресурсами. Цей механізм отримав назву механізму синхронізації потоків (thread synchronization).

Це набір об'єктів операційної системи, які створюються і управляються програмно, є загальними для усіх потоків в системі (деякі - для потоків, що належать одному процесу) та використовуються для координування доступу до ресурсів. Ресурсами може виступати усе, що може бути загальним для двох й більше потоків - файл на диску, порт, запис в базі даних, об'єкт gdi, і навіть глобальна змінна програми (яка може бути доступна з потоків, що належать одному процесу).

Об'єктів синхронізації існує декілька, найважливіші з них - це взаємовиключення (mutex), критична секція (critical section), подія (event) та семафор (semaphore). Кожен з цих об'єктів реалізує свій спосіб синхронізації. Також, як об'єкти синхронізації можуть використовуватися самі процеси й потоки (коли один потік чекає завершення іншого потоку

або процесу), а також файли, комунікаційні пристрої, консольне введення й повідомлення про зміну .

У чому сенс об'єктів синхронізації? Кожен з них може знаходитися в так званому сигнальному стані. Для кожного типу об'єктів цей стан має різний сенс. Потоки можуть перевіряти поточний стан об'єкту й/або чекати зміни цього стану і таким чином погоджувати свої дії. Гарантується, що коли потік працює з об'єктами синхронізації (створює їх, змінює стан) система не перерве його виконання, поки він не завершить цю дію. Таким чином, усі кінцеві операції з об'єктами синхронізації є атомарними (неподільними), такими що виконуються за один такт.

Важливо розуміти, що ніякого реального зв'язку між об'єктами синхронізації та ресурсами немає. Вони не зможуть запобігти небажаному доступу до ресурсу, вони лише підказують потокам, коли можна працювати з ресурсом, а коли треба почекати. Можна провести аналогію зі світлофорами - вони показують, коли можна їхати, але ж в принципі водій може й не звернути уваги на червоне світло.

7.3 Робота з об'єктами синхронізації

Щоб створити той або інший об'єкт синхронізації, проводиться виклик спеціальної функції WINAPI типу Create (напр. CreateMutex()). Цей виклик повертає дескриптор об'єкту (handle), який може використовуватися усіма потоками, що належать цьому процесу. Є можливість отримати доступ до об'єкту синхронізації з іншого процесу - або успадкувавши дескриптор цього об'єкту, або скориставшись викликом функції відкриття об'єкту (Open...). Після цього виклику процес отримає дескриптор, який надалі можна використовувати для роботи з об'єктом. Об'єкту, якщо тільки він не призначений для використання усередині одного процесу, обов'язково привласнюється ім'я. Імена усіх об'єктів мають бути різні (навіть якщо вони різного типу). Не можна, наприклад, створити подію та семафор з одним й тим же ім'ям.

За наявним дескриптором об'єкту можна визначити його поточний стан. Це робиться з допомогою функцій очікування. Найчастіше використовується функція WaitForSingleObject(). Ця функція приймає два параметри, перший з яких - дескриптор об'єкту, другий, - час очікування в мсек. Функція повертає wait_object_0, якщо об'єкт знаходиться в сигнальному стані, wait_timeout - якщо витік час очікування, wait_abandoned, якщо об'єкт-взаємовиключення не був звільнений до того, як потік, що володіє ним, завершився. Якщо час очікування дорівнює нулю, функція повертає результат негайно, інакше вона чекає протягом вказаного проміжку часу. У разі, якщо стан об'єкту стане сигнальним до закінчення цього часу, функція поверне wait_object_0, інакше функція поверне wait_timeout.

Якщо в якості часу вказана символічна константа infinite, то функція чекатиме необмежено довго, поки стан об'єкту не стане сигнальним. Якщо необхідно дізнаватися про стан відразу декількох об'єктів, слід скористатися функцією WaitForMultipleObjects.

Щоб закінчити роботу з об'єктом та звільнити дескриптор викликається функція closehandle.

Дуже важливий той факт, що звернення до функції очікування блокує поточний потік, тобто доки потік знаходиться в стані очікування, йому не виділяється процесорного часу.

7.4 М’ютекси

Об'єкти взаємовиключення (м’ютекси, mutex - від mutual exclusion) дозволяють координувати взаємне виключення доступу до ресурсу, що розділяється. Сигнальний стан об'єкту (тобто стан «встановлений») відповідає моменту часу, коли об'єкт не належить жодному потоку і його можна «захопити». І навпаки, стан «скинутий» (не сигнальне) відповідає моменту, коли який-небудь потік вже володіє цим об'єктом. Доступ до об'єкту дозволяється, коли потік, що володіє об'єктом, звільнить його.

Для того, щоб оголосити, взаємовиключення таким, що належить поточному потоку, потрібно викликати одну із очікуючих функцій. Потік, якому належить об'єкт, може його «захоплювати» повторно скільки завгодно раз (це не приведе до самоблокування), але стільки ж раз він повинен буде його звільняти за допомогою функції releasemutex.

М'ютекс (взаємовиключення, mutex) - це об'єкт синхронізації, який встановлюється в особливий сигнальний стан, коли не зайнятий яким-небудь потоком. Тільки один потік володіє цим об'єктом у будь-який момент часу, звідси і назва таких об'єктів - одночасний доступ до загального ресурсу виключається. Наприклад, щоб виключити запис двох потоків в загальну ділянку пам'яті в один і той же час, кожен потік чекає, коли звільниться м'ютекс, стає його власником і тільки потім пише що-небудь в цю ділянку пам'яті. Після усіх необхідних дій м'ютекс звільняється, надаючи іншим потокам доступ до загального ресурсу.

Два (чи більш) процеси можуть створити м'ютекс з одним і тим же ім'ям, викликавши метод CreateMutex. Перший процес дійсно створює м'ютекс, а наступні процеси отримують хендл вже існуючого об'єкту. Це дає можливість декільком процесам отримати хендл одного і того ж м'ютекса, звільняючи програміста від необхідності піклуватися про той, хто насправді створює м'ютекс. Якщо використовується такий підхід, бажано встановити прапор bInitialOwner у FALSE, інакше виникнуть певні труднощі при визначенні дійсного власника м'ютекса.

Декілька процесів можуть отримати хендл одного і того ж м'ютекса, що робить можливою взаємодію між процесами. Mutex дозволяє проводити синхронізацію не лише між потоками(thread), але і процесами (process), тобто між застосуваннями. Цей об'єкт синхронізації реєструє доступ до ресурсів і може знаходитися в двох станах:

– встановлений
– скинутий

Mutex - встановлений в той момент коли ресурс вільний. Якщо до об'єкту є доступ, то говорять, що скинутий. Для роботи з Mutex є ряд функцій. Спочатку об'єкт треба створити CreateMutex(), для доступу OpenMutex(), а для звільнення ресурсу ReleaseMutex(). Для доступу до об'єкту Mutex використовується функція WaitForSingleObject(). Тонкість тут наступна. Кожна програма створює об'єкт Mutex за ім'ям. Тобто Mutex це іменований об'єкт. А якщо такий об'єкт синхронізації вже створила інша програма, то за викликом CreateMutex() ми отримаємо покажчик на об'єкт, який вже створила перша програма. Тобто у обох програм буде один і той же об'єкт. Ось це і дозволяє проводити синхронізацію.

Приклад м'ютексу:

Код WriteData:

```
// WriteData.cpp: Defines the entry point for the console application.
#include "windows.h"
#include "fstream.h"
#include "iostream.h"
void main()
{
HANDLE hShared = CreateMutex(NULL, TRUE, "WriteData");
ofstream ofs("d:\\write.txt");
ofs << "TestDataWrite" << endl;
ofs.close();
int i;
cout << "Press Key and Enter for access to file " << endl;
cin >> i;
ReleaseMutex(hShared);
CloseHandle(hShared);
}
```

Код ReadData:

```
// // ReadData.cpp: Defines the entry point for the console application.
#include "windows.h"
#include "fstream.h"
#include "iostream.h"

void main()
{
HANDLE hShared = OpenMutex(MUTEX_ALL_ACCESS, FALSE,
"WriteData");
cout << "Wait !!!! Write File Data Proccess " << endl;
WaitForSingleObject(hShared, INFINITE);
ifstream ifs("d:\\write.txt");
char buffer[100];
ifs >> buffer;
cout << "Read File Data  -  " << buffer << endl;
CloseHandle(hShared);
}
```

Основа програми - функція CreateMutex, синтаксис якої:

```
HANDLE CreateMutex(
   LPSECURITY_ATTRIBUTES, // атрибути
   BOOL bInitialOwner,    // ініціалізація
   LPCTSTR lpName         // ім'я об’єкта
);
```

7.5 Критичні секції

Критичні секції забезпечують синхронізацію подібно м’ютексам за винятком того, що об'єкти, які представляють критичні секції, доступні в межах одного процесу. Події, м’ютекси та семафори також можна використовувати в «однопроцесорному» застосуванні, проте критичні

секції забезпечують швидший та ефективніший механізм синхронізації. Подібно м’ютексам об'єкт, що представляє критичну секцію, може використовуватися тільки одним потоком в даний момент часу, що робить їх корисними при розмежуванні доступу до загальних ресурсів. Важко припустити що-небудь про порядок, в якому потоки будуть отримувати доступ до ресурсу, можна сказати лише, що система буде «справедливою» до усіх потоків.

Критична секція (Critical Section) це ділянка коду, в якій потік (thread) дістає доступ до ресурсу (наприклад змінної), який доступний з інших потоків.

Об'єкт критична секція забезпечує синхронізацію. Цим об'єктом може володіти тільки один потік, що і забезпечує синхронізацію. Для роботи з критичними секціями є ряд функцій API і тип даних CRITICAL_SECTION. Для використання критичної секції треба створити змінну цього типу, і проініціалізувати її перед використанням за допомогою функції InitializeCriticalSection(). Для того, щоб увійти до секції треба викликати функцію EnterCriticalSection(), а після завершення роботи LeaveCriticalSection(). Що буде, якщо потік звернеться до секції, в якій зараз інший потік? Той потік, який звернеться буде заблокований доки критична секція не буде звільнена. Саму критичну секцію можна видалити функцією DeleteCriticalSection(). Для того, щоб обійти блокування потоку при зверненні до зайнятої секції є функція TryEnterCriticalSection(), яка дозволяє перевірити критичну секцію на зайнятість.

Тут є один маленький і тонкий момент, який відноситься до синхронізації як такої. Цей момент в тому, що фізично дані не захищені. Ніхто не завадить потоку звернутися до даних, що розділяються. Синхронізація це підказка, як повинен поводитися об'єкт при доступі до даних.

Приклад критичної секції:

```
#include "windows.h"
#include "iostream.h"
#include "process.h"
#define MAX_ARRAY 5
CRITICAL_SECTION critsect;
int array[MAX_ARRAY];
void EmptyArray(void *);
void PrintArray(void *);
void FullArray(void *);
void main()
{		InitializeCriticalSection(&critsect);
			if (_beginthread(EmptyArray, 1024, NULL)==- 1) cout
<< "Error begin thread " << endl;
			if (_beginthread(PrintArray, 1024, NULL)==- 1) cout
<< "Error begin thread " << endl;
			if (_beginthread(FullArray, 1024, NULL)==- 1)  cout
<< "Error begin thread " << endl;
			if (_beginthread(PrintArray, 1024, NULL)==- 1) cout
<< "Error begin thread " << endl;
			if (_beginthread(EmptyArray, 1024, NULL)==- 1) cout
<< "Error begin thread " << endl;
			if (_beginthread(PrintArray, 1024, NULL)==- 1) cout
<< "Error begin thread " << endl;
			Sleep(10000);
}
void EmptyArray(void *)
{			cout << "EmptyArray" << endl;
	EnterCriticalSection(&critsect);
	for (int x=0;x<(MAX_ARRAY+1); x++) array[x]=0;
```

```
        Sleep(1000);
        LeaveCriticalSection(&critsect);
        _endthread();
    }
    void PrintArray(void *)
    {           cout << "PrintArray" << endl;
        EnterCriticalSection(&critsect);
        for (int x=0;x<(MAX_ARRAY+1); x++) cout << array[x] << " ";
        cout << endl;
        Sleep(1000);
        LeaveCriticalSection(&critsect);
        _endthread();
    }
    void FullArray(void *)
    {           cout << "FullArray" <<
     endl;
        EnterCriticalSection(&critsect);
        for (int x=0;x<(MAX_ARRAY+1); x++) array[x]=x;
        Sleep(1000);
        LeaveCriticalSection(&critsect);
        _endthread();
    }
```

Результат роботи наступний:

EmptyArray
PrintArray
FullArray
PrintArray
EmptyArray
PrintArray
0 0 0 0 0 0
0 1 2 3 4 5
0 0 0 0 0 0

Потоки запускаються, починається доступ до критичної секції. Чекають своєї черги і виконують необхідні дії.

Функції для роботи із критичними секціями представлені в таблиці 7.2.

Таблиця 7.2. - Функції для роботи із критичними секціями

Найменування	Опис
1	2
InitializeCriticalSection	Ініціалізує змінну типу CRITICAL_SECTION.
InitializeCriticalSectionAndSpinCount	Ініціалізує змінні типу CRITICAL_SECTION і лічильник її очікування.
EnterCriticalSection	Входить у критичну секцію й блокує інші потоки, що намагаються ввійти в критичну секцію.
LeaveCriticalSection	Залишає критичну секцію, дозволяючи іншим потокам увійти до неї.
TryEnterCriticalSection	Намагається ввійти в критичну секцію, повертаючи помилку у випадку якщо критична секція вже виконується будь-яким потоком.
SetCriticalSectionSpinCount	Установлює значення лічильника очікування критичної секції.
DeleteCriticalSection	Знищує змінні CRITICAL_SECTION.

7.6 Події

Об'єкти-події використовуються для повідомлення очікуючих потоків про настання якої-небудь події. Розрізняють два види подій - з ручним і автоматичним скиданням. Ручне скидання здійснюється функцією resetevent. Події з ручним скиданням використовуються для повідомлення відразу декількох потоків. При використанні події з автоскиданням

повідомлення отримає і продовжить своє виконання тільки один очікуючий потік, інші чекатимуть далі.

Функція createevent створює об'єкт-подію, setevent - встановлює подію в сигнальний стан, resetevent - скидає подію. Функція pulseevent встановлює подію, а після відновлення очікуючих потоків (усіх при ручному скиданні і тільки одного при автоматичному), скидає його. Якщо очікуючих потоків немає, pulseevent просто скидає подію.

7.7 Семафори

Об'єкт-семафор - це фактично об'єкт-взаємовиключення з лічильником. Цей об'єкт дозволяє «захопити» себе певній кількості потоків. Після цього «захоплення» буде неможливе, поки один з потоків, що раніше «захопили» семафор, не звільнить його. Семафори застосовуються для обмеження кількості потоків, що одночасно працюють з ресурсом. Об'єкту при ініціалізації передається максимальне число потоків, після кожного «захоплення» лічильник семафора зменшується. Сигнальному стану відповідає значення лічильника більше нуля. Коли лічильник дорівнює нулю, семафор вважається не встановленим (скинутим).

Створюється семафор функцією CreateSemaphore() :

```
HANDLE CreateSemaphore
(
   LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
// атрибут доступу
   LONG lInitialCount,
 // початковий стан лічильника
   LONG lMaximumCount,
 // максимальна кількість звернень
   LPCTSTR lpName
 // ім'я об'єкту);
```

При успішному виконанні функція поверне ідентифікатор семафора, інакше NULL. Після того, як необхідність в роботі з об'єктом відпала треба викликати функцію ReleaseSemaphore(), щоб звільнити лічильник.

```
BOOL ReleaseSemaphore
(
   HANDLE hSemaphore,       // хендл семафора
   LONG lReleaseCount,      // на скільки змінювати лічильник
   LPLONG lpPreviousCount   // попереднє значення
);
```

При успішному виконанні повертає ненульове значення. Для знищення семафора треба викликати CloseHandle().

7.8 Контрольні запитання

1. Поняття процесу та потоку.
2. Функція створення потоку CreateThread().
3. Механізми синхронізації за допомогою функцій бібліотеки Win32. Функції очікування SignalObjectAndWait, WaitForSingleObject.
4. Види пріоритетів потоку.
5. Структура механізму синхронізації. Поняття критичної секції. Синтаксис функцій EnterCriticalSection(), InitializeCriticalSection, LeaveCriticalSection ().
6. Об’єкти синхронізації семафори. Основні завдання і методи. Функції CreateSemaphore (), ReleaseSemaphore(),CloseHandle(). Параметри функцій.
7. М‘ютекси як об’єкт синхронізації між процесами. Функції CreateMutex() та ReleaseMutex().
8. Повідомлення як об’єкт синхронізації. Функції CreateEvent(), SetEvent(), ResetEvent(). Приклади застосування.

Питання синхронізації роботи процесів та потоків висвітлені в книгах [1,10-14]. Роботу з об’єктами синхронізації детально наведено в [1]. На порталах [1-3] в мережі Інтернеті можна знайти також інші оглядові й навчальні матеріали з цих питань.

8 СУЧАСНІ ІНТЕРФЕЙСИ ДЛЯ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

Засоби паралельного програмування застосовуються для різної архітектури мультипроцесорних систем та різних процесорних платформ з використанням різних мов програмування. При такій великій різноманітності з метою уніфікації й можливості перенесення паралельного програмного забезпечення йдуть двома основними шляхами:

1) Розширення існуючих мов програмування і створення нових компіляторів для них (наприклад, від C до C++).

2) Створення бібліотек функцій та програмних оболонок, які дають можливість, не змінюючи компілятора з базової мови програмування, забезпечувати стандартний інтерфейс програмування.

Другий підхід має перевагу, оскільки робить можливим використання існуючого програмного забезпечення. Проте обидва підходи не виключають, а доповнюють один одного. Прикладом першого з них являється поява нової мови програмування Java [17], яка базується на сучасних концепціях об'єктного орієнтування, багатоплатоформленій реалізації й підтримки паралелізму. Прикладами другого підходу є сучасні інтерфейси паралельного програмування, які є надбудовами над стандартними компіляторами та бібліотеки функцій, за допомогою яких паралельні алгоритми можуть програмуватися єдиним способом на багатьох мультипроцесорних системах. Матеріал цієї лекції присвячений останньому з названих підходів.

Існує декілька відомих бібліотек інтерфейсних функцій для паралельного програмування. Найвідомішими з них є PVM (Parallel Virtual Machine) та MPI (Message Passing Interface).

8.1 Parallel Virtual Machine

PVM - розробка кінця 80-х - початку 90-х рр., місце розробки - США. Головною метою було забезпечення можливості паралельного програмування на різнорідних (гетерогенних) мережах комп'ютерів, сполучених протоколом TCP/IP. PVM - це пакет програм, написаних мовою програмування C, що дає можливість розглядати гетерогенну мережу Unix-машин як єдиний паралельний комп'ютер з розподіленою пам'яттю (віртуальну машину). До складу віртуальної машини можуть входити як послідовні, так і паралельні комп'ютери.

Основною одиницею роботи в PVM є завдання (task), яке, як правило, відповідає поняттю процесу в Unix. Завдання можуть обмінюватися повідомленнями та утворювати таким чином застосування (application), яке може бути написане мовою C.

PVM складається з двох частин: клієнтської та серверної. Серверна частина є множиною образів (демонів) програм, розподілених між комп'ютерами віртуальної машини. Клієнтська частина утворена бібліотечними функціями PVM, які знаходяться у розпорядженні користувача та призначені для підтримки обміну повідомленнями, координації її завдань й зміни конфігурації віртуальної машини. Щоб скористатися можливостями PVM, програма застосування має бути пов'язана з бібліотекою PVM.

На одному і тому ж устаткуванні мережі може існувати декілька віртуальних машин, які одночасно здатні виконувати декілька застосувань.

Для виконання застосування користувач спочатку породжує демон на своєму комп'ютері, після чого застосування може стартувати з будь-якого комп'ютера віртуальної машини. У разі виходу з ладу одного з комп'ютерів PVM автоматично реєструє цю подію і виводить цей комп'ютер із складу віртуальної машини. Застосування може передбачити заміну машини користувача резервною.

Модель взаємодії завдань у PVM не накладає обмежень ні на кількість, ні на об'єм повідомлень, якими обмінюються завдання. Обмін повідомленнями може відбутися між будь-якими завданнями у віртуальному комп'ютері. Передбачені такі режими обміну:

- блокована передача - виконується відразу після отримання буфера для передачі;
- асинхронна блокована передача - не залежить від виклику відповідного прийому повідомлення;
- неблокований прийом - завершується відразу з альтернативним результатом: або прийняте повідомлення, або ознака того, що повідомлення ще не поступило.

Окрім цих парних взаємодій можливі також групові операції: «група – одному» (multicast), «один – групі» (broadcast).

Модель обміну повідомленнями PVM передбачає також збереження порядку проходження повідомлень. Тобто якщо завдання 1 передало повідомлення А і В завданню 2 в порядку їх переліку, то в тому ж порядку вони будуть прийняті завданням 2.

Отже, особливостями PVM є:

- підтримка гетерогенних мереж;
- динамічне управління ресурсами віртуальної машини.

Ці особливості є перевагами PVM, проте вони тягнуть і головний недолік PVM - порівняно невисоку ефективність паралельних обчислень через великі накладні витрати на підтримку своїх переваг.

8.2 Message Passing Interface

Однією із найпоширеніших сучасних технологій організації паралельних обчислень є MPI. MPI (Message Passing Interface) — інтерфейс обміну повідомленнями (інформацією) між одночасно працюючими обчислювальними процесами. Він широко використовується для створення

паралельних програм для обчислювальних систем з розподіленою пам'яттю (кластерів).

Так само, як і в PVM, в MPI реалізується підхід розширення відомих мов за рахунок бібліотеки функцій, які забезпечують засоби взаємодії паралельних процесів. Проте метою MPI є забезпечення високої продуктивності і можливості перенесення застосувань на різні мультипроцесорні платформи. На теперішній час MPI є машинно-незалежним стандартом для паралельного програмування, який включає понад 100 бібліотек функцій [16].

На відміну від PVM, яку можна розглядати як розподілену операційну систему для гетерогенних мереж, MPI є тільки інтерфейсом, розрахованим на однорідні мультипроцесорні системи.

Однією із найпоширеніших сучасних технологій організації паралельних обчислень є MPI. MPI (Message Passing Interface) — інтерфейс обміну повідомленнями (інформацією) між одночасно працюючими обчислювальними процесами. Він широко використовується для створення паралельних програм для обчислювальних систем з розподіленою пам'яттю (кластерів).

MPICH — найвідоміша реалізація MPI, створена в Арагонській національній лабораторії (США). Існують версії цієї бібліотеки для усіх популярних операційних систем. До того ж, вона безкоштовна. Перераховані чинники роблять MPICH ідеальним варіантом для того, щоб почати практичне освоєння MPI.

MPICH2 - це не версія програмного забезпечення, а номер того стандарту MPI, який реалізований у бібліотеці. MPICH2 відповідає стандарту MPI 2.0, звідси і назва. Мета створення MPICH2 наступна:

1) Надати реалізацію MPI, яка ефективно підтримує різні обчислювальні і комунікаційні платформи, включаючи загальнодоступні кластери (настільні системи, системи із загальною пам'яттю, багатоядерна архітектура), високошвидкісні мережі (Ethernet 10 ГБіт/с, InfiniBand, Myrinet, Quadrics) та ексклюзивні обчислювальні системи (Blue Gene, Cray,

SiCortex).

2) Зробити можливими передові дослідження технології MPI за допомогою легко розширюваної модульної структури для створення похідних реалізацій.

В якості кластера можна використовувати комп’ютерну мережу навчального класу. За відсутності комп’ютерної мережі можна запускати усі обчислювальні процеси і на одному комп’ютері. Якщо комп’ютер одноядерний, то, природно, приросту швидкодії ви не отримаєте, - тільки уповільнення. Якщо багатоядерний - то можна завантажити всі обчислювальні ядра.

В якості середовища розробки доцільно використовувати Visual Studio 2008 або 2010 (можливе використання безкоштовної версії Express).

MPICH для Windows складається з наступних компонентів:

1) Менеджер процесів smpd.exe, який є системною службою (сервісне застосування). Менеджер процесів веде список обчислювальних вузлів системи, і запускає на цих вузлах MPI-програми, надаючи їм необхідну інформацію для роботи і обміну повідомленнями.

2) Заголовні файли (.h) і бібліотеки компіляції (.lib), необхідні для розробки MPI -програм.

3) Бібліотеки виконання (.dll), необхідні для роботи MPI -програм.

4) Додаткові утиліти (.exe), необхідні для налаштування MPICH і запуску MPI -програм.

Усі компоненти, окрім бібліотек виконання, встановлюються по замовчуванню в теку C:\Program Files\MPICH2; dll-бібліотеки встановлюються в C:\Windows\System32.

Менеджер процесів є основним компонентом, який має бути встановлений і налагоджений на усіх комп’ютерах мережі (бібліотеки часу виконання можна, в крайньому випадку, копіювати разом з MPI програмою). Інші файли потрібно для розробки MPI-програм і налаштування деякого «головного» комп’ютера, з якого

здійснюватиметься їх запуск.

Менеджер працює у фоновому режимі і чекає запитів до нього з мережі з боку «головного» менеджера процесів (за умовчанням використовується мережевий порт 8676). Щоб захистити себе від хакерів і вірусів, менеджер вимагає пароль при зверненні до нього. Коли один менеджер процесів звертається до іншого менеджера процесів, він передає йому свій пароль. Звідси витікає, що треба вказувати один і той же пароль при установці MPICH на комп’ютери мережі.

Запуск MPI-програми робиться таким чином (рис. 8.1.) :

2) Користувач за допомогою програми Mpirun (чи Mpiexec, при використанні MPICH2 під Windows) вказує ім’я виконуваного файлу MPI - програми і необхідне число процесів. Крім того, можна вказати ім’я користувача і пароль: процеси MPI-програми запускатимуться від імені цього користувача.

2) Mpirun передає відомості про запуск локальному менеджерові процесів, у якого є список доступних обчислювальних вузлів.

3) Менеджер процесів звертається до обчислювальних вузлів за списком, передаючи запущеним на них менеджерам процесів вказівки по запуску MPI - програми.

Менеджери процесів запускають на обчислювальних вузлах декілька копій MPI-програми (можливо, по декілька копій на кожному вузлі), передаючи програмам необхідну інформацію для зв’язку один з одним.

Дуже важливим моментом є те, що перед запуском MPI -програма не копіюється автоматично на обчислювальні вузли кластера. Замість цього менеджер процесів передає вузлам шлях до виконуваного файлу програми точно в тому вигляді, в якому користувач вказав цей шлях програмі Mpirnn. Це означає, що якщо ви, наприклад, запускаєте програму C :.exe, то усі менеджери процесів на обчислювальних вузлах намагатимуться запустити файл C :.exe. Якщо хоч би на одному з вузлів такого файлу не виявиться, станеться помилка запуску MPI -програми.

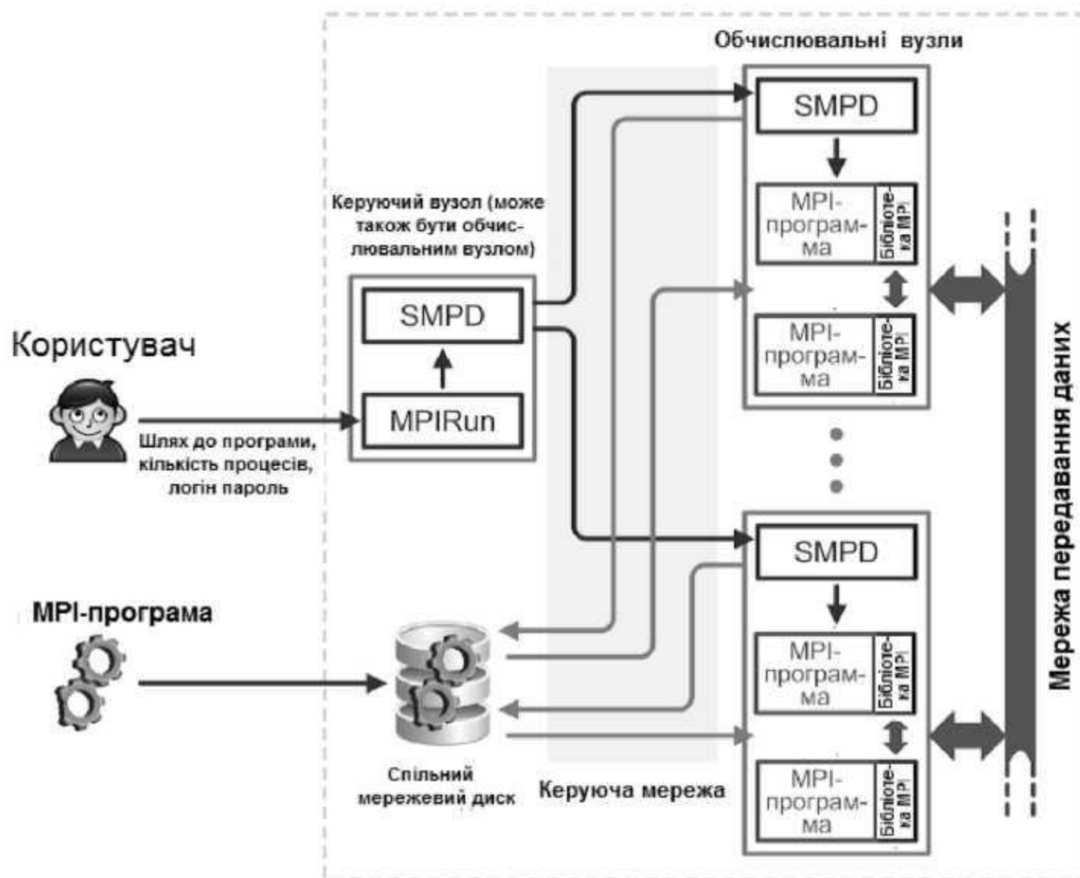


Рис. 8.1. Схема роботи MPICH на кластері

Щоб кожного разу не копіювати вручну програму і усі необхідні для її роботи файли на обчислювальні вузли кластера, зазвичай використовують загальний мережевий ресурс. В цьому випадку користувач копіює програму і додаткові файли на мережевий ресурс, видимий усіма вузлами кластера, і вказує шлях до файлу програми на цьому ресурсі. Додатковою зручністю такого підходу є те, що за наявності можливості запису на загальний мережевий ресурс запущені копії програми можуть записувати туди результати своєї роботи.

Робота MPI програми відбувається таким чином:

1) Програма запускається та ініціалізує бібліотеку часу виконання MPICH шляхом виклику функції MPI_Init.

2) Бібліотека отримує від менеджера процесів інформацію про кількість і місце розташування інших процесів програми, і встановлює з ними зв’язок.

3) Після цього запущені копії програми можуть обмінюватися один з одним інформацією за допомогою бібліотеки MPICH. З точки зору операційної системи бібліотека є частиною програми (працює в тому ж процесі), тому можна вважати, що запущені копії MPI-програми обмінюються даними безпосередньо один з одним, як будь-які інші застосування, передавальні дані по мережі.

4) Консольне уведення-виведення усіх процесів MPI-програми перенаправляється на консоль, на якій запущена Mpirun. Перенаправленням введення-виведення займаються менеджери процесів, оскільки саме вони запустили копії MPI-програми, і тому можуть отримати доступ до потоків введення-виведення програм.

5) Перед завершенням усі процеси викликають функцію MPI_Finalize, яка коректно завершує передачу і прийом усіх повідомлень, і відключає MPICH.

Усі описані вище принципи діють, навіть якщо MPI-програма запускається на одному комп’ютері.

В додатку А наведено опис процесу встановлення системи MPICH.

MPI-програма складається з процесів, представлених унікальними номерами $0..N-1$, не має загальної пам'яті та розрахована на архітектуру MIMD паралельної системи. У більшості реалізацій MPI кількість процесів є величиною постійною, що визначається при ініціалізації програми. Стандартними мовами програмування для MPI являються C та FORTRAN.

В даному прикладі використовується декілька головних функцій MPI:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{ int size, rank, i;
MPI_Init(&argc,&argv); //Ініціалізація бібліотеки
MPI_Comm_size(MPI_COMM_WORLD,&size); /* Кількість процесів в
додатку */
```

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* Власний номер
процесу */
if(rank==0)
printf("Amount of tasks=%d\n",size);
printf("My number in MPI_COMM_WORLD=%d\n",rank);
/* Точка синхронізації, після неї процес 0 друкує
* аргументи командного рядка. В командному рядку
* можуть бути параметри, що додаються
* завантажувачем MPIRUN.
*/
MPI_Barrier(MPI_COMM_WORLD);
if(rank==0)
for(puts("CommandLine for task 0:"),i=0;
i<argc; i++)
printf("%d: \"%s\"\n",i,argv[i]);
MPI_Finalize();//Всі процеси завершують виконання
return 0;
}
```

Ця програма демонструє використання функцій MPI для ініціалізації та завершення роботи паралельної програми, а також наводить звичайний приклад використання інформаційних функцій MPI_Comm_size() та MPI_Comm_rank(). Окрім використаних в даному прикладі чотирьох функцій до шестірки найуживаніших належать також функції передачі MPI_SEND і прийому MPI_RECV повідомлення.

Важливим поняттям MPI є поняття комунікатора - тобто середовища, в якому група процесорів може здійснювати обмін повідомленнями. Стандартним комунікатором є MPI_Comm_World, який визначає універсальне середовище для обміну - тобто групу процесорів, пронумерованих послідовно від *0* до *N - 1*.

Так само, як і PVM, MPI має декілька режимів передачі/прийому повідомлень.

Номер процесу - ціле додатне число, що є унікальним атрибутом кожного процесу.

Атрибути повідомлення - номер процесу-відправника, номер процесу-одержувача і 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 і додаткові поля.

Ідентифікатор повідомлення (msgtag) - атрибут повідомлення, що є цілим додатнім числом, що лежить в діапазоні від 0 до 32767.

Процеси об'єднуються в групи, можуть бути вкладені групи. Усередині групи всі процеси перенумеровані. З кожною групою асоційований свій комунікатор. Тому при здійсненні пересилки необхідно вказати ідентифікатор групи, усередині якої проводиться ця пересилка. Всі процеси містяться в групі з визначеним ідентифікатором MPI_COMM_WORLD.

При описі процедур MPI користуватимемося 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 додатку. Реальна ініціалізація для кожного додатку виконується не більше одного разу, а якщо MPI вже ініціалізував, то ніякі дії не виконуються і відбувається негайне повернення з підпрограми. Всі MPI-процедури, що залишилися, можуть бути викликані тільки після виклику MPI_Init.

Повертає: у разі успішного виконання – MPI_SUCCESS, інакше – код помилки. (Те ж саме повертає і вся решта функцій) int MPI_Finalize(void).

MPI_Finalize – завершення паралельної частини додатку. Всі подальші звернення 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 arge, char** argv)
{
MPI_Init(&argc &argv);
MPI_Finalize();
}
```

int MPI_Comm_size(MPI_Comm comm , int* size)

Визначення загального числа паралельних процесів в групі. comm – ідентифікатор групи.

OUT size – розмір групи (OUT означає, шо цей параметр є вихідним результатом функції).

Int MPI_Comm_rank(MPI_Comm comm , int* rank)

Визначення номера процесу в групі comm. Значення, що повертається за адресою &rank, лежить в 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 в секундах (дійсне число), що пройшов з деякого моменту у минулому. Гарантується, що цей момент не буде змінений за час існування процесу.

Для реалізації процедур попарного обміну повідомленнями використовуються функції MPI_Send (для передавання) і MPI_Recv (для приймання), які мають наступний формат:

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

де:

- buf –адреса буфера пам’яті, де розміщується повідомлення;

- count – кількість елементів у повідомленні;
- type - тип елементів у повідомленні;
- dest - номер процесу отримувача;
- tag – тег повідомлення, використовується для ідентифікації повідомлення. Це число від 0 до 255;
- comm – група процесів, в рамках якої відбувається передавання.

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

де:

- buf –адреса буфера пам’яті, куди прийматиметься повідомлення;
- count – кількість елементів у повідомленні;
- type – тип елементів у повідомленні;
- dest –номер процесу, від якого отримується повідомлення;
- tag – тег повідомлення, використовується для ідентифікації повідомлення. Значення тегу повинно співпадати при відправці і прийманні;
- comm – група процесів, в рамках якої відбувається передавання.

При заданні номера процесу і тегу при прийманні можна використовувати константи відповідно MPI_ANY_SOURCE та MPI_ANY_TAG, що означають прийняти повідомлення з від довільного процесу та з будь-яким тегом.

Приклад використання функцій:

MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);

MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,

MPI_ANY_TAG, MPI_COMM_WORLD, &Status);

Налаштування проекту Visual Studio для виконання MPI-програми наведено в додатку Б.

8.3 OpenMP

OpenMP задуманий як стандарт для програмування на масштабованих SMP-системах в моделі загальної пам'яті (shared memory model). У стандарт OpenMP входять специфікації набору директив компілятора, процедур і змінних середовища. Розробкою стандарту займається організація OpenMP ARB (Architecture Board), до якої увійшли представники найбільших компаній – розробників SMP-архітектур і програмного забезпечення..

До появи OpenMP не було відповідного стандарту для ефективного програмування на SMP-системах.

POSIX-інтерфейс для організації ниток (Pthreads) підтримується широко (практично на всіх UNIX-системах), однак з багатьох причин не підходить для практичного паралельного програмування: немає підтримки Fortrana, занадто низький рівень, немає підтримки паралелізму за даними, механізм ниток спочатку розроблявся не для цілей організації паралелізму.

OpenMP можна розглядати як високорівневу надбудову над Pthreads (чи аналогічними бібліотеками). Багато постачальників SMP-архітектур (Sun, HP, SGI) у своїх компіляторах підтримують спеціальні директиви для розпаралелювання циклів. Однак ці набори директив, як правило дуже обмежені, несумісні між собою; в результаті чого розробникам доводиться розпаралелювати застосування окремо для кожної платформи. OpenMP є багато в чому узагальненням і розширенням згаданих наборів директив. OpenMP надає розробнику наступні переваги:

1) За рахунок ідеї «інкрементального розпаралелювання» OpenMP ідеально підходить для розробників, що бажають швидко розпаралелити свої обчислювальні програми з великими паралельними циклами. Розробник не створює нову паралельну програму, а просто послідовно додає в текст послідовної програми OpenMP-директиви.

2) При цьому OpenMP – досить гнучкий механізм, що надає розробнику більші можливості контролю над поведінкою паралельного застосування.

3) Передбачається, що OpenMP-програма на багатопроцесорній платформі може бути використана в якості послідовної програми. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs).

4) Однією з переваг OpenMP його розробники вважають підтримку так званих «orphan» (відірваних) директив, тобто директиви синхронізації і розподілу роботи можуть не входити безпосередньо в лексичний контекст паралельної області.

На даний момент технологія OpenMP підтримується більшістю компіляторів мови C/C++. Дещо гірше справа йде з інструментами тестування паралельних OpenMP програм. Інструменти аналізу, перевірки та оптимізації паралельних програм хоча й існують давно, до недавнього часу були мало задіяні при розробці прикладного програмного забезпечення. Тому вони часто є менш зручними, ніж інші інструментальні засоби розробки. Найбільш повно процес розробки паралельних OpenMP програм підтриманий у пакеті Intel Parallel Studio. Є інструмент попереднього аналізу коду, для виявлення ділянок коду, які потенційно можна ефективно розпаралелити. Є добре оптимізований компілятор з підтримкою OpenMP. Додатково можна виділити інструмент VivaMP, що входить до складу PVS-Studio. Це статичний аналізатор коду, спеціалізований на виявленні помилок у OpenMP програмах на етапі їх написання.

Робота OpenMP-застосування починається з єдиного потоку – основного. У застосуванні можуть міститися паралельні регіони, входячи в які, основний потік створює групи потоків (що включають основний потік). Наприкінці паралельного регіону групи потоків зупиняються, а виконання основного потоку триває. У паралельний регіон можуть бути вкладені інші

паралельні регіони, в яких кожен потік первісного регіону стає основним для своєї групи потоків. Вкладені регіони можуть у свою чергу включати регіони більш глибокого рівня вкладеності. Паралельну обробку в OpenMP ілюструє рис. 8.2.

Паралельні області

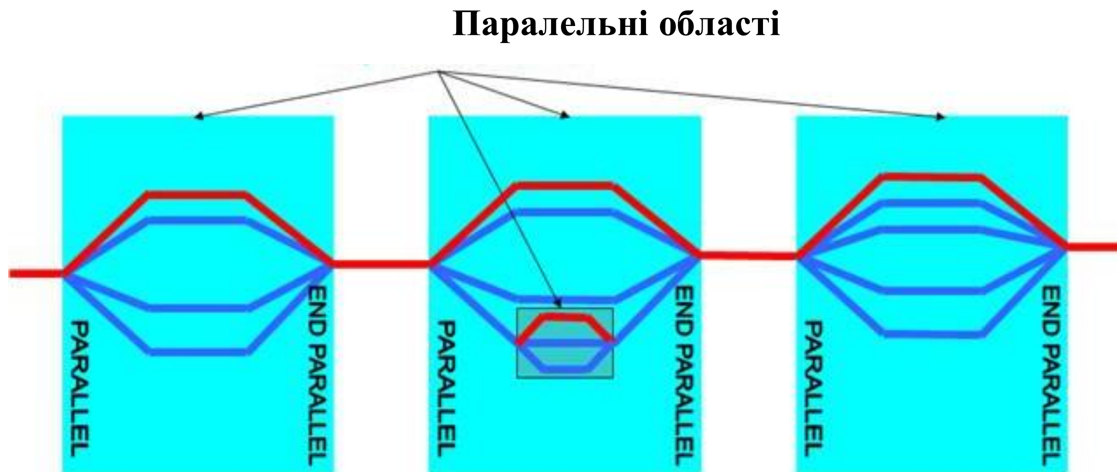


Рис. 8.2. Модель виконання Open-MP програми

При вході в паралельну секцію виконується операція fork, що породжує сімейство ниток. Кожна нитка має свій унікальний числовий ідентифікатор (Головної нитки відповідає 0). При розпаралелюванні циклів всі паралельні нитки виконують один код. У загальному випадку нитки можуть виконувати різні фрагменти коду. При виході з паралельної секції виконується операція join. Завершується виконання всіх ниток, крім головної.

OpenMP складають наступні компоненти:

1) Директиви компілятора - використовуються для створення потоків, розподілу роботи між потоками і їх синхронізації. Директиви включаються у вихідний текст програми. Підпрограми бібліотеки - використовуються для встановлення та визначення атрибутів потоків. Виклики цих підпрограм включаються у вихідний текст програми.

2) Змінні оточення - використовуються для управління поведінкою паралельної програми. Змінні оточення задаються для середовища виконання паралельної програми відповідними командами (наприклад,

командами оболонки в операційних системах UNIX). Використання директив компілятора і підпрограм бібліотеки підпорядковується правилам, які розрізняються для різних мов програмування. Сукупність таких правил називається прив'язкою до мови.

У програмах на мові С, імена функцій і змінних оточення OMP починається з omp, omp_ або OMP_. В OpenMP-програмі використовується заголовний файл omp. h

Формат директиви:

```
# pragma omp директива [оператор_1 [, оператор_2, …]]
```

Найважливіша і поширена директива - parallel. Вона створює паралельний регіон для наступного за нею структурованого блоку, наприклад:

```
# pragma omp parallel [розділ [[,] розділ ]...] структурований блок
```

Ця директива повідомляє компілятору, що структурований блок коду повинен бути виконаний паралельно, в декількох потоках. Кожен потік буде виконувати один і той же потік команд, але не один і той же набір команд - все залежить від операторів, керуючих логікою програми, таких як if-else. В якості прикладу розглянемо класичну програму «Hello World»:

```
#pragma omp parallel
{
printf("Hello World\n");
}
```

У двопроцесорній системі , звичайно ж, розраховували б отримати наступне:

Hello World

Hello World

Тим не менш, результат міг би бути й таким:

HellHell oo WorWlodrld

Другий варіант можливий через те, що два виконуваних паралельно потоки можуть спробувати вивести рядок одночасно. Коли два або більше

потоки одночасно намагаються прочитати або змінити загальний ресурс (у нашому випадку їм є вікно консолі), виникає ймовірність гонок (race condition). Це не детерміновані помилки в коді програми, знайти які вкрай важко. За запобігання гонок відповідає програміст як правило, для цього використовують блокування або зводять до мінімуму звернення до загальних ресурсів.

Розглянемо приклад, який визначає середні значення двох сусідніх елементів масиву і записує результати в інший масив. У цьому прикладі використовується OpenMP-конструкція # pragma omp for, яка відноситься до директив поділу роботи (work-sharing directive). Такі директиви застосовуються не для паралельного виконання коду, а для логічного розподілу групи потоків, щоб реалізувати вказані конструкції керуючої логіки.

Директива # pragma omp for повідомляє, що при виконанні циклу for в паралельному регіоні ітерації циклу повинні бути розподілені між потоками групи:

```
#pragma omp parallel
{
#pragma omp for
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
}
```

Якщо б цей код виконувався на чотирьох процесорному комп'ютері, а у змінній size було б значення 100, то виконання ітерацій 1-25 могло б бути доручено першому процесору, 26-50 - другому, 51-75 - третьому, а 76-99 - четвертим. Це характерно для політики планування, так званої статичної. Слід зазначити, що наприкінці паралельного регіону виконується бар'єрна синхронізація (barrier synchronization). Інакше кажучи, досягнувши кінця регіону, всі потоки блокуються до тих пір, поки останній потік не завершить свою роботу. Якщо з тільки що наведеного прикладу виключити

директиву # pragma omp for, кожен потік виконає повний цикл for, проробивши багато зайвої роботи:

```
#pragma omp parallel
{
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
}
```

Так як цикли є найпоширенішими конструкціями, де виконання коду можна розпаралелити, OpenMP підтримує скорочений спосіб запису комбінації директив # pragma omp parallel і # pragma omp for:

```
#pragma omp parallel for
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
```

В цьому циклі немає залежностей, тобто одна ітерація циклу не залежить від результатів виконання інших ітерацій. А ось у двох наступних циклах є два види залежності:

```
for(int i = 1; i <= n; ++i) // цикл 1
a[i] = a[i-1] + b[i];
for(int i = 0; i < n; ++i) // цикл 2
x[i] = x[i+1] + b[i];
```

Розпаралелити цикл 1 проблематично тому, що для виконання ітерації і потрібно знати результат ітерації і-1, тобто ітерація і залежить від ітерації і-1. Розпаралелити цикл 2 теж проблематично, але з іншої причини. У цьому циклі можемо вирахувати значення x [i] до x [i-1], однак, зробивши так, ми більше не зможете обчислити значення x [i-1].

Спостерігається залежність ітерації і-1 від ітерації і. При розпаралелюванні циклів потрібно переконатися в тому, що ітерації циклу не мають залежностей. Якщо цикл не містить залежностей, компілятор може виконувати цикл в будь-якому порядку, навіть паралельно. Дотримання цієї важливої вимоги компілятор не перевіряє. Якщо ми

задамо компілятору розпаралелити цикл, у якому є залежності, компілятор підкориться, що призведе до помилки. Крім того, OpenMP накладає обмеження на цикли for, які можуть бути включені в блок # pragma omp for або # pragma omp parallel for block. Цикли for повинні відповідати формату:

for ([цілочисельний тип] i = інваріант циклу; i {<,>,=,<=,>=} інваріант циклу; i {+,-}= інваріант циклу)

Ці вимоги введені для того, щоб OpenMP міг при вході в цикл визначити число ітерацій.

Приклад програми на мові C з використанням OpenMP:

```
#include "omp. h"
#include <stdio. h>
double f (double x)
{ return 4.0/ (1 + x * x); }
main () { const long N = 100000;
long i;
double h, sum, x;
sum = 0; 21
h = 1.0/N;
#pragma omp parallel shared (h)
{ #pragma omp for private (x) reduction (+: sum)
for (i = 0; i < N; i++) {
x = h * (i + 0.5);
sum = sum + f (x); } }
printf ("PI = %f\n", sum / N); }
```

OpenMP включає і функції, призначені для синхронізації коду. У OpenMP два типи блокувань: прості і вкладені (nestable); блокування обох типів можуть знаходитися в одному з трьох станів - неініціалізованому, заблокованому і розблокованому. Прості блокування (omp_lock_t) не можуть бути встановлені більше одного разу, навіть тим самим потоком. Вкладені блокування (omp_nest_lock_t) ідентичні простим з тим винятком,

що, коли потік намагається встановити вже приналежну йому вкладене блокування, він не блокується. Крім того, OpenMP веде облік посилань на вкладені блокування і стежить за тим, скільки разів вони були встановлені. OpenMP надає підпрограми, що виконують операції над цими блокуваннями. Кожна така функція має два варіанти: для простих і для вкладених блокувань. Можна виконати над блокуванням п'ять дій: ініціалізувати її, встановити (захопити), звільнити, перевірити і знищити. Всі ці операції дуже схожі на Win32-функції для роботи з критичними секціями, і це не випадковість: насправді технологія OpenMP реалізована як оболонка цих функцій. Відповідність між функціями OpenMP і Win32 ілюструє табл. 8.1.

Для синхронізації коду можна використовувати і підпрограми виконуючого середовища, і директиви синхронізації. Перевага директив в тому, що вони прекрасно структуровані. Це робить їх більш зрозумілими і полегшує пошук місць входу в синхронізовані регіони і виходу з них. Перевага підпрограм виконуючого середовища - гнучкість. Наприклад, можливість передати блокування в іншу функцію і встановити / звільнити її в цій функції. При використанні директив це неможливо. Як правило, якщо потрібна гнучкість, що забезпечується лише підпрограмами виконуючого середовища, краще використовувати директиви синхронізації.

Таблиця 8.1. - Відповідність між функціями OpenMP і Win32

Просте блокування OpenMP	Вкладене блокування OpenMP	Win32-функція
omp_lock_t	omp_nest_lock_t	CRITICAL_SECTION
omp_init_lock	omp_init_nest_lock	InitializeCriticalSection
omp_destroy_lock	omp_destroy_nest_lock	DeleteCriticalSection
omp_set_lock	omp_set_nest_lock	EnterCriticalSection
omp_unset_lock	omp_unset_nest_lock	LeaveCriticalSection
omp_test_lock	omp_test_nest_lock	TryEnterCriticalSection

В прикладі показано код двох паралельно виконуваних циклів, на початку яких виконуючому середовищі невідома кількість їх ітерацій. У першому прикладі виконується перебір елементів STL-контейнера std:: vector, а в другому - стандартного зв’язаного списку.

Приклад. Виконання заздалегідь невідомого числа ітерацій

```
# pragma omp parallel {
/ / Паралельна обробка вектора STL
std:: vector <int>:: iterator iter;
for (iter = xVect.begin ();
iter! = xVect.end (); + + iter)
{ # pragma omp single nowait { process1 (* iter); }
}
/ / Паралельна обробка стандартного пов'язаного списку
 for (LList * listWalk = listHead; listWalk! = NULL; listWalk = listWalk-
> next) { # pragma omp single nowait { process2 (listWalk); } } }
```

У прикладі з вектором STL кожен потік з групи потоків виконує цикл for і має власний примірник ітератора, але при кожній ітерації лише один потік входить в блок single (така семантика директиви single). Всі дії, що гарантують одноразове виконання блоку single при кожній ітерації, бере на себе виконуюче середовище OpenMP. Такий спосіб виконання циклу пов'язаний зі значними витратами, тому він корисний, тільки якщо у функції process1 виконується багато роботи. У прикладі зі зв'язаним списком реалізована та ж логіка. Варто відзначити, що в прикладі з вектором STL ми можемо до входу в цикл визначити число його ітерацій за значенням std:: vector.size, що дозволяє привести цикл до канонічної форми для OpenMP:

```
# pragma omp parallel for for (int i = 0; i <xVect.size (); + + i) process (xVect [i]);
```

Це суттєво зменшує витрати в період виконання, і саме такий підхід найкраще застосовувати для обробки масивів, векторів і будь-яких інших

контейнерів, елементи яких можна перебрати в циклі for, відповідно канонічній формі для OpenMP.

За умовчанням в OpenMP для планування паралельного виконання циклів for застосовується алгоритм, так званий статичним плануванням (static scheduling). Це означає, що всі потоки з групи виконують однакове число ітерацій циклу. Якщо n - число ітерацій циклу, а T - число потоків у групі, кожний потік виконає n / T ітерацій (якщо n не ділиться на T без залишку, нічого страшного). Однак OpenMP підтримує й інші механізми планування, оптимальні в різних ситуаціях: динамічне планування (dynamic scheduling), планування в період виконання (runtime scheduling) і кероване планування (guided scheduling). Щоб задати один з цих механізмів планування, використовують розділ schedule в директиві # pragma omp for або # pragma omp parallel for. Формат цього розділу виглядає так:

schedule(алгоритм планування[, число ітерацій])

Приклади цих директив:

pragma omp parallel for schedule (dynamic, 15) for (int i = 0; i <100; + + i) ...
pragma omp parallel # pragma omp for schedule (guided)

При динамічному плануванні кожен потік виконує вказане число ітерацій. Якщо це число не задано, за замовчуванням воно дорівнює 1. Після того як потік завершить виконання заданих ітерацій, він переходить до наступного набору ітерацій. Так триває, доки не будуть пройдені всі ітерації. Останній набір ітерацій може бути менше, ніж спочатку заданий.

Завершивши виконання призначених ітерацій, потік запитує виконання іншого набору ітерацій, число яких визначається по щойно наведеній формулі. Таким чином, число ітерацій, що призначаються кожному потоку, з часом зменшується. Останній набір ітерацій може бути менше, ніж значення, обчислене за формулою.

OpenMP - проста, але потужна технологія розпаралелювання програм. Вона дозволяє реалізувати паралельне виконання як циклів, так і

функціональних блоків коду. Вона легко інтегрується в існуючі програми і включається / вимикається одним параметром компілятора. OpenMP дозволяє більш повно використовувати обчислювальну потужність багатоядерних процесорів.

Поради коли використовувати технологію OpenMP:

1) Цільова платформа є багатопроцесорною або багатоядерною. Якщо програма повністю використовує ресурси одного ядра або процесора, то, зробивши її багатопотоковою за допомогою OpenMP, то це майже напевно підвищить швидкодію.

2) Програма повинна бути кросплатформенною. OpenMP - багатоплатформенний та широко підтримуваний API. А так як він реалізований на основі директив pragma, програму можна скомпілювати навіть за допомогою компілятора, який не підтримує стандарт OpenMP.

3) Виконання циклів потрібно розпаралелити. Весь свій потенціал OpenMP демонструє при організації паралельного виконання циклів. Якщо в програмі є тривалі цикли без залежностей, OpenMP - ідеальне рішення.

4) Перед випуском програми потрібно підвищити її швидкодію. Оскільки технологія OpenMP не вимагає переробки архітектури програми, вона чудово підходить для внесення в код невеликих змін, що дозволяють підвищити швидкодію.

8.4 Контрольні запитання

1. Система MPI. Загальна характеристика. Підтримка моделі взаємодії паралельних обчислювачів за допомогою передачі повідомлень. Основні програмні примітиви системи MPI. Приклад використання.
2. Загальна характеристика стандарту PVP.
3. Характеристика OpenMP.Основні компоненти OpenMP.
4. Модель виконання OpenMP-програми.

Матеріал щодо сучасних інтерфейсах паралельного програмування розміщений на порталах [1-3]. Детальніше розгляд питань про основні концепції, механізм і функції MPI можна знайти в книгах [14-16]. Програмування з використанням технології OpenMP розглядається в [17].

РОЗДІЛ ІІІ. РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

9 ОСНОВНІ МЕХАНІЗМИ РЕАЛІЗАЦІЇ РОЗПОДІЛЕНИХ СИСТЕМ

Розподілена обчислювальна система (РОС) – це набір з'єднаних каналами зв'язку незалежних комп'ютерів, які з точки зору користувача деякого програмного забезпечення виглядають єдиним цілим [18].

У цьому визначенні фіксуються два істотні моменти: автономність вузлів РОС і уявлення системи користувачем, як єдиної структури. При цьому, основною сполучною ланкою розподілених обчислювальних систем є програмне забезпечення.

Розподілена обчислювальна система являє собою програмно-апаратний комплекс, орієнтований на вирішення певних завдань. З одного боку, кожен обчислювальний вузол є автономним елементом. З іншого боку, програмна складова РОС повинна забезпечувати користувачам видимість роботи з єдиною обчислювальною системою.

У зв'язку з цим виділяють наступні важливі характеристики РОС:

- можливість роботи з різними типами пристроїв: з різними постачальниками пристроїв; з різними операційними системами; з різними апаратними платформами.
- можливість простого розширення та масштабування;
- перманентна (постійна) доступність ресурсів (навіть якщо деякі елементи РОС деякий час можуть перебувати поза доступу);
- приховування особливостей комунікації від користувачів.

Обчислювальні середовища, що складаються з безлічі обчислювальних систем на базі різних програмно-апаратних платформ, називаються гетерогенними.

Для забезпечення роботи гетерогенного обладнання РОС у вигляді єдиного цілого, стек програмного забезпечення (ПЗ) зазвичай розбивають на два шари.

На верхньому шарі розташовуються розподілені додатки, що відповідають для вирішення певних прикладних задач засобами РОС. Їх функціональні можливості базуються на нижньому шарі - проміжному програмному забезпеченні (ППЗ). ППЗ взаємодіє з системним ПЗ і мережевим рівнем для забезпечення прозорості роботи додатків в РОС (рис. 9.1.).

Для того щоб РОС могла бути представлена користувачеві як єдина система, застосовують такі типи прозорості в РОС:

- прозорий доступ до ресурсів - від користувачів повинна бути прихована різниця в поданні даних і в способах доступу до ресурсів РОС;
- прозоре розташування ресурсів - місце фізичного розташування необхідного ресурсу повинно бути несуттєве для користувача;
- реплікація - приховування від користувача того, що в реальності існує більше однієї копії використовуваних ресурсів;
- паралельний доступ - можливість спільного (одночасного) використання одного і того ж ресурсу різними користувачами незалежно один від одного. При цьому факт спільного використання ресурсу повинен залишатися прихованим від користувача;
- прозорість відмов - відмова (відключення) будь-яких ресурсів РОС не повинна впливати на роботу користувача та його додатків.

Термінологія РОС:

1. *Ресурсом* називається будь-яка програмна або апаратна сутність, представлена або використовувана в розподіленій мережі. Наприклад, комп'ютер, пристрій зберігання, файл, комунікаційний канал, сервіс тощо.

2. *Вузол* – будь-який апаратний пристрій у розподіленій обчислювальній системі.

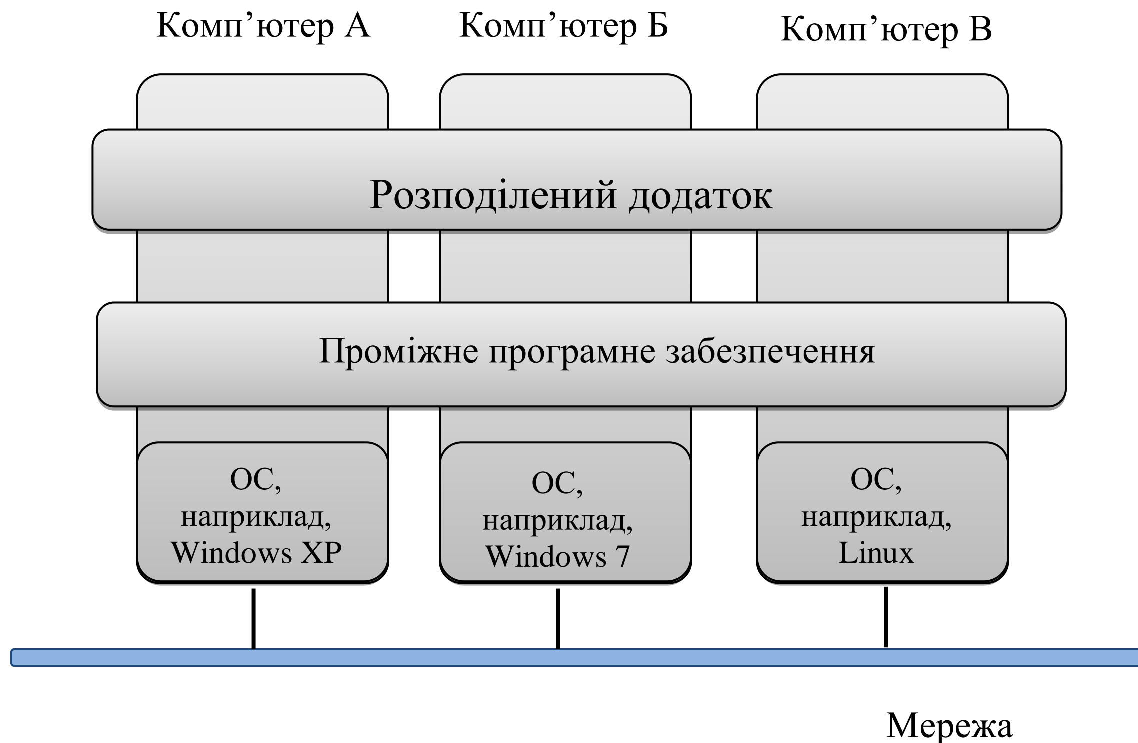


Рис. 9.1. Компоненти програмного забезпечення РОС

3. *Сервер* - це постачальник інформації у РОС (наприклад, веб-сервер).

4. *Клієнт* - це споживач інформації в РВС (наприклад, веб-браузер).

5. *Пір* - це вузол, який поєднує в собі як клієнтську, так і серверну частину (тобто і постачальник, і споживач інформації одночасно).

6. *Сервіс* - це мережева сутність, що надає певні функціональні можливості (наприклад, веб-сервер може надавати сервіс передачі файлів протоколом HTTP). В рамках одного вузла можуть надаватися кілька різних сервісів.

На рисунку 9.2. наведена схема, що встановлює взаємини між даними термінами. Зі схеми видно, що кожен комп'ютер або пристрій являє собою сутність у розподіленій обчислювальній системі у вигляді вузла. При цьому на кожному вузлі може розташовуватися кілька клієнтів, серверів, сервісів або пірів. Важливо зауважити, що будь-який вузол, сервер, пір або сервіс (*але не клієнт*!) є ресурсами розподіленої обчислювальної системи. Сервіс отримує запит на надання певних даних (майже як аргументи, що передаються при виклику локальної функції) і повертає відповідь. Таким

чином, сервіс можна визначити як певну заміну виклику функції на локальному комп'ютері. Існує безліч технологій, що забезпечують створення і супровід сервісів у розподілених обчислювальних системах: технологія XML веб-сервісів, сервіси REST та інші.

Рис. 9.2. Рівні моделі OSI

9.1 Зв'язок у РОС

Поняття «розподілена обчислювальна система» має на увазі, що компоненти такої системи розподілені, тобто віддалені один від одного. Очевидно, що функціонування подібних систем неможливе без ефективного зв'язку між її компонентами. Завдання організації обміну між розподіленими (територіально, адміністративно, тощо) компонентами давно та в значній мірі успішно вирішуються в обчислювальних мережах, і, природно, що РОС використовують напрацьований досвід. Взаємодія в обчислювальних мережах базується на протоколах. *Протокол* - це набір

правил та угод, що описують процедуру взаємодії між компонентами системи (в тому числі і обчислювальній).

Якщо система підтримує певний протокол, вона, з великою часткою ймовірності, виявиться здатною взаємодіяти з іншою системою, яка так само підтримує даний протокол. В області обчислювальних комунікацій вже тривалий час існує загальноприйнята система протоколів - мережева модель *OSI* (англ. *Open Systems Interconnection basic reference model* - базова еталонна модель взаємодії відкритих систем). Ця модель являє собою стек протоколів різного рівня, які дозволяють описати практично всі аспекти взаємодії компонентів РОС.

9.2 Сучасні РОС

На сьогоднішній день, РОС відходять від традиційних понять високопродуктивних розподілених обчислень в сторону розвитку віртуального співробітництва та віртуальних організацій. Віртуальна організація - це ряд людей та/або організацій, об'єднаних спільними правилами колективного доступу до певних обчислювальних ресурсів. Методи надання доступу до обчислювальних ресурсів стають сервісно-орієнтованими, що дозволяє гнучко використовувати одні і ті ж обчислювальні ресурси різними споживачами. Значно розширилися області автоматизованого управління ресурсами. Людина не в силах вручну вирішити задачу розподілу обчислень в системах такого масштабу і гетерогенності. Таким чином, необхідно використання автоматизованих систем управління завданнями, які беруть на себе завдання управління, які надаються системою. Також, зі зростанням масштабу обчислювальних мереж, необхідні автоматизовані засоби обробки помилок і відновлення обчислювального процесу.

9.2.1 Однорангові (peer-to-peer) мережі

У 1999 році, в Північно-східному Університеті (Массачусетс, США) першокурсник Шон Феннінг написав систему обміну MP3 файлами між користувачами. Цей проект отримав назву Napster. Він став першим проектом, котрий поклав початок технології однорангових (Peer-to-peer (P2P) - англ. «Рівний-до-рівного») розподілених обчислювальних мереж. Через 2 роки Napster був закритий спільними зусиллями власників авторських прав на музичні твори, що розповсюджувалися через цю мережу. Але за прикладом Napster розвинувся цілий клас P2P систем нового, децентралізованого типу, які закрити було значно складніше.

У 2000 році Джастін Франкел (20-ти річний хакер з США, в 1997 році випустив безкоштовний MP3 плеєр WinAmp) написав Gnutella - P2P протокол передачі файлів. На відміну від Napster, який використовував центральний сервер для встановлення зв'язку між пірами, Gnutella покладалася виключно на системи кінцевих користувачів для організації мережі. Таким чином, на відміну від Napster, мережу Gnutella виявилося неможливо «закрити», відключивши центральний сервер. Мільйони людей до сих пір користуються цією системою.

При роботі в рамках парадигми P2P, комп'ютери обмінюються ресурсами безпосередньо один з одним, без використання центрального сервера.

Підхід P2P забезпечує вирішення проблем, що виникли в результаті експоненціального зростання Інтернет та Веб. Застосування P2P дозволило безлічі користувачів, які раніше були простими споживачами інформації, взяти участь в наданні контенту. У момент своєї появи, P2P було швидше модним словом, ніж продуманою концепцією. В результаті потужного просування за допомогою засобів масової інформації, технологія P2P поширилася в академічних та промислових колах.

В рамках концепції P2P всі комп'ютери, які входять в мережу, взаємодіють один з одним безпосередньо, без використання централізованих серверів. Основні переваги однорангових обчислювальних систем:

- спрощується підтримка масштабованості при значному зростанні кількості вузлів в обчислювальної мережі;
- підвищується відмовостійкість мережі, тому що збій будь-якого обчислювального вузла не може привести до зупинки функціонування мережі цілком.

Проте, існує ряд перешкод при побудові P2P мереж:

1) При роботі з P2P додатками, обчислювальний вузол бере на себе функції, як клієнта, так і сервера. Це призводить до збільшення вимог до продуктивності кожного комп'ютера, який включений до такої мережі.

2) Низька ступінь захищеності машин, що беруть участь в P2P мережі пояснюється тим, що вони надають відкритий доступ до своїх ресурсів (таким як такти процесора, певні папки на жорсткому диску тощо).

Таким чином, при відсутності засобів захисту, комп'ютери, включені в P2P схильні до ризику злому або зараження з боку недобросовісних учасників.

3) При побудові P2P мережі доводиться долати можливу гетерогенності апаратного й програмного забезпечення її потенційних учасників. Це питання може бути вирішене шляхом застосування таких технологій як XML або Java.

4) Основна проблема P2P мереж - це пошук доступних ресурсів, без використання централізованої точки управління. Кожному вузлу доводиться проводити пошук серед сотень і тисяч ресурсів усередині мережі, що є дуже трудомістким і ресурсомістким завданням.

Незважаючи на всі труднощі, відбувається бурхливий розвиток та використання P2P мереж. Серед найбільш відомих та значущих прикладів, варто відзначити такі проекти як BitTorrent, Napster, Skype,

Незважаючи на те, що базова філософська концепція грід та P2P різниться, обидві технології намагаються вирішити одну і ту ж проблему - створення віртуального шару над існуючою інфраструктурою Інтернет для забезпечення спільної роботи й використання спільних ресурсів. Проте, в реалізації, підходи сильно відрізняються. Грід швидше орієнтована на об'єднання віртуальних організацій для забезпечення спільної роботи їх учасників, а P2P пов'язує окремих користувачів, які знаходяться за кінцевими вузлами мережі Інтернет (тобто за NAT, брандмауерами і ін.).

9.2.2 Сервіс-орієнтована архітектура

На початку 2000-х років бізнес-співтовариство зайнялося розробкою наступного покоління специфікацій, покликаних вирішити проблеми ранніх стандартів розподілених об'єктних технологій за допомогою веб-сервісів та *сервіс-орієнтованої архітектури (Service-Oriented Architecture - SOA).*

Стандарти веб-сервісів були розроблені за ініціативою організацій, що займаються наданням віддаленого доступу до певних обчислювальних ресурсів, і закріплені консорціумом W3C. До основних стандартів розробки та функціонування веб-сервісів можна віднести:

- SOAP - заснований на XML протокол взаємодії веб-сервісів;
- WSDL (Web Services Description Language - Мова опису веб-сервісів) - це методологія опису ресурсів, що надаються веб-сервісом;
- UDDI (Universal Description Discovery and Integration - Універсальний метод пошуку та інтеграції) - метод опису, пошуку, взаємодії та використання веб-сервісів.

На сьогоднішній день, сервіс-орієнтований підхід є стандартом «де-факто» при розробці розподілених обчислювальних систем.

9.2.3 Агенти

Незважаючи на всі переваги технологій веб-сервісів, вони не пропонують нових методологій та рішень побудови широкомасштабних обчислювальних мереж. Для пошуку рішень в цьому напрямку, необхідно розглянути агентно-орієнтовану парадигму побудови РОС.

Обчислювальні мережі на основі так званих агентів - це принципово інший підхід до організації РОС. Програмний агент - це автономний процес, здатний реагувати на середовище виконання та викликати зміни в середовищі виконання, можливо, в кооперації з користувачами або іншими агентами. Розглянемо основні принципи роботи агентних мереж [20]:

− автономність - агенти функціонують автономно, без можливості стороннього втручання в їх внутрішній стан;

− соціальна поведінка - агенти взаємодіють один з одним за допомогою певної мови;

− активність - агенти взаємодіють з навколишнім середовищем, отримуючи певні сигнали та відповідаючи на них;

− про-активність - агенти діють цілеспрямовано.

Агентні мережі принципово пристосовані для функціонування в динамічно-змінному навколишньому середовищу. В цьому випадку, автономність агентів дозволяє організувати динамічне підлаштування обчислювального алгоритму під умови обчислювального середовища. Таким чином, РОС можна уявити як набір взаємодіючих компонентів, а інформація, якою вони обмінюються, розбивається на певні категорії:

− інформація про компоненти та їх функціональні можливості, в рамках певної області;

− інформація про взаємодії між компонентами;

− узагальнена інформація про робочий процес та більш конкретна інформація з тієї чи іншої задачі.

Для забезпечення функціонування такої системи, необхідна стандартизація методів взаємодії між компонентами. Для вирішення цього завдання розробляються та стандартизуються мови взаємодії агентів (*Agent Communication Languages, ACLs*). Одним з найбільш відомих, є архітектура взаємодії *FIPA* (*Foundation for Intelligent Physical Agents*, базис інтелектуальних фізичних агентів). Ця архітектура стандартизує методи взаємодії агентів та агентних систем.

9.2.4 Хмарні обчислення

Хмара - це парадигма великомасштабних розподілених обчислень, заснована на ефекті масштабу, в рамках якої пул абстрактних, віртуалізованих, динамічно-масштабованих обчислювальних ресурсів, ресурсів зберігання, платформ та сервісів надаються за запитом зовнішнім користувачам через Інтернет.

Не дивлячись на те, що метафора «хмара» вже давно використовується фахівцями в області мережевих технологій для зображення на мережевих діаграмах складної обчислювальної інфраструктури (або ж Інтернету як такого), що приховує свою внутрішню організацію за певним інтерфейсом, термін «Хмарні обчислення» з'явився на світло зовсім недавно. Згідно з результатами аналізу пошукової системи *Google*, термін «Хмарні обчислення» («*Cloud Computing*») почав набирати важливості в кінці *2007* - початку *2008* року, поступово витісняючи популярне в той час словосполучення «Грід-обчислення» («*Grid Computing*»). Судячи із заголовків новин того часу, однією з перших компаній, що дали світові цей термін, стала компанія *IBM*, яка розгорнула на початку 2008 року проект «*Blue Cloud*» і стала спонсором Європейського проекту «*Joint Research Initiative for Cloud Computing*».

На сьогоднішній день вже можна говорити про те, що хмарні обчислення міцно увійшли в повсякденне життя кожного користувача Інтернету. Однак до сих пір немає єдиної думки про те, що таке «Хмарні

Обчислення» і яким чином вони співвідносяться з парадигмою «Грід-обчислень».

9.3 Модель «Клієнт-Сервер»

Згідно парадигмі клієнт-серверної архітектури кілька клієнтів та кілька серверів спільно із проміжним програмним забезпеченням та середовищем взаємодії утворюють єдину систему, що забезпечує розподілені обчислення, аналіз та подання даних. Використання клієнт-серверного підходу дозволило користувачеві персонального комп'ютера отримати доступ до різних ресурсів віддалених серверів, таких як бази даних, файли, принтери, процесорний час тощо.

У базовій моделі клієнт-сервер всі процеси в розподілених системах діляться на дві групи, які можуть перекриватися. Процеси, що реалізують деякий сервіс, наприклад, сервіс файлової системи або бази даних, називаються серверами. Процеси, що запитують сервіси у серверів шляхом посилки запиту та подальшого очікування відповіді від сервера, називаються клієнтами.

Якщо базова мережа так само надійна, як локальні мережі, взаємодія між клієнтом та сервером може бути реалізована за допомогою простого протоколу, що не потребує встановлення з'єднання (наприклад, протокол UDP). У цьому випадку клієнт, запитуючи сервіс, наділяє свій запит в форму повідомлення із зазначенням в ньому сервісу, яким він бажає скористатися, та необхідних для цього вихідних даних. Потім повідомлення надсилається серверу. Останній, в свою чергу, постійно очікує вхідного повідомлення, отримавши його, обробляє, упаковує результат обробки у повідомлення-відповідь та відправляє його клієнту.

Використання, яке не потребує з'єднання протоколу дає суттєвий виграш у ефективності. До тих пір поки повідомлення не почнуть пропадати або пошкоджуватися, можна цілком успішно застосовувати протокол типу запит-відповідь. На жаль, створити протокол, стійкий до

випадкових збоїв зв'язку, - нетривіальне завдання. Все, що можна зробити - це дати клієнту можливість повторно надіслати запит, на який не було отримано відповіді. Проблема, однак, полягає в тому, що клієнт не може визначити, чи дійсно первинне повідомлення із запитом було втрачено або помилка сталася під час передачі відповіді. Якщо загубилася відповідь, повторна посилка запиту може привести до повторного виконання операції.

У якості альтернативи в багатьох системах клієнт-сервер використовується надійний протокол з установкою з'єднання (наприклад, протокол TCP). Хоча це рішення у зв'язку з його відносно низькою продуктивністю не дуже добре підходить для локальних мереж, воно чудово працює в глобальних системах, для яких ненадійність є «вродженою» властивістю з’єднань. Так, практично всі прикладні протоколи Інтернету засновані на надійних з'єднаннях на основі стека TCP / IP. У цьому випадку щоразу, коли клієнт запитує сервіс, до посилки запиту серверу він повинен встановити з ним з'єднання. Сервер зазвичай використовує для посилки листа у відповідь те ж саме з'єднання, після чого воно розривається. Проблема полягає в тому, що установка та розрив з'єднання в сенсі затраченого часу та ресурсів відносно вартісні, особливо якщо повідомлення із запитом і відповіддю невеликі.

9.4 Контрольні запитання

1. Розподілені обчислення. Компоненти програмного забезпечення РОС.
2. Зв'язок в РОС. Модель OSI.
3. Характеристика сучасних РОС.
4. Поняття клієнт-серверної архітектури. Види клієнт-серверних архітектур.

Розгляд питань про основні концепції, механізм і функції розподілених обчислювальних систем можна знайти в книгах [18-20].

10 ПРОГРАМУВАННЯ ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ З ВИКОРИСТАННЯМ СОКЕТІВ

Сокети - це програмний інтерфейс, призначений для передачі даних між додатками. З англійської мови слово «socket» можна перекласти як «розетка», тому образно сокети можна уявити собі як дві розетки, до яких включено кабель для передачі даних через мережу.

За своєю суттю програмний інтерфейс сокетів являє собою набір функцій, які дозволяють програмісту вирішувати завдання, пов'язані з передачею інформації між ЕОМ мережею. Якщо локальна або глобальна мережа побудована на основі протоколу TCP / IP, то на прикладному рівні взаємодію між вузлами мережі можна реалізовувати за допомогою технології сокетів.

Спочатку інтерфейс сокетів розроблявся в рамках роботи над операційною системою Unix (стандарт Berkeley Sockets). Пізніше, фірма Microsoft допрацювала вказаний інтерфейс (наприклад, додала в нього деякі необхідні для ОС Windows функції), що призвело до появи інтерфейсу Windows Sockets або WinSock. Функції, що складають сучасну версію інтерфейсу WinSock, зібрані в динамічно підключеній бібліотеці WS2_32.DLL.

10.1 Етапи роботи з об'єктами Windows Sockets:

1. *Ініціалізація додатків*, тобто отримання доступу до інтерфейсу Windows Sockets. Для цієї мети служить функція WSAStartup, визначається таким чином:

```
int WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData);
```

wVersionRequested - двобайтний параметр, який вказує номер версії інтерфейсу WinSock, який потрібен для програми. Молодший байт параметра повинен містити старший номер версії, старший байт - молодший номер версії. В сучасних операційних системах сімейства

Windows використовується інтерфейс Windows Sockets версії 2.0 і вище. Параметр **lpWSAData** - це покажчик на структуру типу *WSADATA*, в яку будуть записані відомості про конкретну реалізацію інтерфейсу Windows Sockets. Функція повертає нульове значення, якщо її виконання завершилося успішно. В іншому випадку буде повернуто код помилки.

2. *Створення та ініціалізація сокета* - об'єкта, через який відбуватиметься обмін даними. Для цих цілей використовується функція socket:

SOCKET socket (int af, int type, int protocol);

af - цілочисельний параметр, що визначає формат мережевих адрес тих вузлів, з якими передбачається здійснювати взаємодію. Для роботи з адресами в форматі, прийнятому для Internet, цей параметр повинен прийняти значення *AF_INET*.

type - цілочисельний параметр, що визначає тип сокета. Для передачі даних через організований канал зв'язку значення цього параметра має дорівнювати *SOCK_STREAM.* Якщо сокет створюється для обміну інформацією без організації каналів зв'язку (швидший, але менш надійний спосіб), то значення параметра встановлюється рівним *SOCK_ DGRAM*).

protocol - цілочисельний параметр, що визначає протокол для роботи сокета. У більшості випадків його значення можна встановить рівним 0.

Функція socket в разі успішного виконання повертає дескриптор сокета. У разі виникнення помилки її результатом буде значення *INVALID_SOCKET* (або -1).

Перед тим, як починати роботу з сокетом, його необхідно ініціалізувати. Параметри сокета задаються з використанням спеціальних структур даних **sockaddr** або **sockaddr_in.** Структуру **sockaddr_in** потрібно використовувати в тих випадках, коли обрано формат мережевих адрес, прийнятий для мереж Internet (параметр **af** при створенні сокета

дорівнює *AF_INET*). Щоб взаємодія з іншими додатками мережею стала можливою, необхідно ініціалізувати наступні елементи зазначеної структури:

- *мережева адреса вузла*, до якого плануємо звертатися використовуючи сокет. Адреса задається із встановленням її типу і власне значенням. Тип адреси вказується в поле **sin_family** - його значення повинно дорівнюватися константі *AF_INET*. Щоб задати значення адреси, використовується поле **sin_addr**, яке, в свою чергу, є структурою спеціального формату. Для коректного визначення мережевої адреси в поле **s_addr** зазначеної структури достатньо записати результат роботи функції **inet_addr (const char * cp),** яка перетворює рядок з мережевою адресою, записаною у вигляді чотирьох десяткових чисел (наприклад, "217.71.128.65"), між якими ставиться крапка, в деяке ціле число. Слід пам'ятати, що в деяких ситуаціях, не можна вказати для сокета конкретне значення мережевої адреси. Наприклад, якщо додаток, який розробляється є *сервером*, тобто надає віддаленим додаткам (*клієнтам*) будь-які інформаційні послуги, він може з'єднуватися і обмінюватися інформацією з багатьма комп'ютерами в мережі, у яких можуть бути різні мережеві адреси. Для таких випадків передбачена константа *INADDR_ANY*, значення якої присвоюється полю **sin_addr.s_addr** і дозволяє сокету встановлювати з'єднання з будь-якої мережевої адреси.

- *номер порту*, який використовується сокетом при передачі даних. На кожному комп'ютері може бути одночасно запущено велику кількість додатків, які здійснюють обмін даних мережею. Щоб визначити, для якого додатка призначені ті чи інші пропозиції, що надійшли мережею даних, в протоколі TCP введено поняття *порту*. Мережевий порт являє собою число від 1 до 65535 і фактично є ідентифікатором встановленої між двома додатками взаємодії. Розробник може на власне бажання вибирати номер порту для своїх додатків (у кожної пари взаємодіючих мережею сокетів номер порту повинен бути один і той же). Однак перші 1024

номери портів вважаються зарезервованими для спеціального використання і не рекомендуються для довільного використання. Номер порту зберігається в поле **sin_port** в спеціальному *універсальному мережевому форматі*. Для перетворення 16-розрядного номера порту в універсальний мережевий формат в інтерфейсі WinSock передбачена функція **htons (u_short pt)**, результат виконання якої можна привласнювати у вказане вище поле структури **sockaddr_in**.

Таким чином, для коректного створення та ініціалізації сокета необхідно виконати наступні дії:

- створити сокет за допомогою функції **socket ();**
- створити змінну для структури даних **sockaddr_in**;
- ініціалізувати поля створеної змінної: **sin_family, sin_addr, sin_port**;
- ініціалізувати створений сокет шляхом його «прив'язки» до конкретної мережевої адреси.

Тут необхідно зробити деякі пояснення щодо *технології клієнт-сервер*, яка нерозривно пов'язана з використанням сокетів. Суть цієї технології полягає в тому, що будь-яка взаємодія між двома додатками розглядається як надання певної інформаційної послуги однієї програми (*сервера*) у відповідь на *запит* другого додатка (*клієнта*). Характер цієї послуги може бути найрізноманітнішим - видача файлів, надання даних, передача повідомлення тощо. У деяких випадках ролі додатків жорстко зафіксовані, тобто при їх взаємодії один завжди виступає як сервер, а інший - як клієнт. В інших ситуаціях обидва взаємодіючих додатки можуть виступати за відношенням один до одного і в якості сервера, і в якості клієнта.

Інтерфейс Windows Socket орієнтований на реалізацію технології клієнт-сервер при взаємодії між додатками. З цієї причини порядок ініціалізації сокета різниться в залежності від того, чи відкривається сокет на стороні сервера або на стороні клієнта. На стороні сервера етап

ініціалізації може відбуватися окремо від етапів встановлення каналу зв'язку та з'єднання з клієнтом, оскільки сервер повинен бути готовий до установки з'єднання, але точного моменту, в який клієнт звернеться до нього з запитом, він не знає. Клієнтські сокети не потрібно форматувати окремо від установки з'єднання, так як клієнт сам ініціює обмін інформацією з сервером (надсилає запит і чекає відповідь від сервера) і, отже, знає точний момент початку мережевої взаємодії.

Ініціалізація сокетів для серверних додатків здійснюється за допомогою функції **bind ():**

int bind (SOCKET sock, const struct sockaddr FAR * addr, int addrlen);

sock - дескриптор сокета (отриманий в результаті виклику функції **socket**), **addr** - покажчик на підготовлену структуру **sockaddr, addrlen** - параметр, який вказує розмір цієї структури в байтах. У разі, коли підготовлювалась структура **sockaddr_in**, а не **sockaddr**, необхідно провести відповідне приведення типів для покажчика **addr**. При успішно виконаній ініціалізації, функція **bind ()** повертає нульове значення. В іншому випадку буде повернуто код помилки.

Ініціалізація сокетів клієнтських додатків, як уже було сказано, здійснюється в момент установки зв'язку.

3. *Створення каналу зв'язку*. Даний етап виконується тільки в тому випадку, якщо для взаємодії додатків мережею використовується протокол TCP. Крім того, цей етап виділяється окремо тільки для хостів, які відкриваються в додатках-серверах. Для створення каналу зв'язку використовується функція **listen ():**

int listen (SOCKET sock, int backlog);

sock - це дескриптор сокета, а цілочисельний параметр **backlog** визначає максимальний розмір черги для очікування з'єднання (тобто, скільки клієнтів одночасно можуть чекати установки з'єднання з сервером). Щоб задати для сокета чергу максимального розміру в якості

зазначеного параметра можна передати значення константи *SOMAXCONN*. У разі успішного виконання функція **listen ()** повертає 0.

4. *Встановлення з'єднання*. Реалізація цього етапу виконується різним чином для серверних та клієнтських сокетів. Додаток-сервер має підготувати сокет до установки з'єднання та перейти в режим очікування запитів від клієнта (одного або декількох). Для цього необхідно викликати функцію **accept ():**

SOCKET accept (SOCKET sock, struct sockaddr FAR * addr,
int FAR * addrlen);

sock - дескриптор сокета, параметр **addr** вказує на структуру, в яку буде записана інформація про мережеву адресу, до якої підключився клієнт, **addrlen** - покажчик на довжину цієї структури. Якщо з'єднання встановлено успішно, функція **accept ()** повертає ідентифікатор сокета, до якої підключився клієнт. Якщо в момент виклику сервером функції **accept ()** немає клієнтів, які очікують підключення, то результат її виконання буде залежати від режиму роботи сокета. При роботі в *блокуючому режимі* функція заблокує виконання серверної програми, але не завершиться до тих пір, поки з'єднання з клієнтом не буде встановлено. При роботі сокета в *неблокованому режимі* функція завершить виконання зі спеціальним кодом помилки (WSAEWOULDBLOCK). За замовчуванням, всі сокети Windows Sockets працюють в блокуючому режимі.

Клієнтські програми після створення сокета та ініціалізації структури **sockaddr_in** з інформацією про мережеву адресу та номер порту для серверного додатка повинні ініціювати установку з'єднання. Для цієї мети служить функція **connect ():**

int connect (SOCKET sock, const struct sockaddr * addr, int addrlen);

sock - дескриптор раніше створеного сокета, **addr** - покажчик на структуру з інформацією про мережеву адресу та номер порту сервера, **addrelen** - довжина цієї структури. При роботі сокета в блокуючому

режимі функція поверне 0 при успішній установці з'єднання. Щоб з'єднання могло бути встановлено до моменту виклику клієнтом функції **connect (),** серверний додаток має встигнути створити канал зв'язку (функція **listen** ()) і перевести свій сокет в режим очікування з'єднання (функція **accept** ()).

5. *Обмін даними*. Сокети, що відносяться до типу *SOCK_STREAM*, забезпечують додаткам повнодуплексну взаємодію. Передача та отримання даних при взаємодії з протоколу TCP здійснюється за допомогою функцій **send ()** і **recv ().**

Функція передачі даних **send ()** має чотири параметри - дескриптор сокета **sock**, через який виконується передача, адреса буфера **buf**, що містить передане повідомлення, розмір цього буфера **bufsize** та прапори **flags**:

int send (SOCKET sock, const char FAR * buf, int bufsize, int flags);

Набір прапорів дозволяє налаштувати деякі специфічні параметри передачі даних.

Параметри функції **recv (),** призначеної для прийому даних, подібні до установок функції **send ():**

int recv (SOCKET sock, char FAR * buf, int bufsize, int flags);

Функції **recv ()** і **send ()** повертають кількість, відповідно, прийнятих та переданих байт даних. Додаток, який приймає дані, повинен викликати функцію **recv ()** в циклі до тих пір, поки не будуть прийняті всі передані дані. При цьому на один виклик функції **send ()** може доводитися декілька викликів функції **recv ().** При роботі сокета в блокуючому режимі функція **recv ()** не буде завершена до тих пір, поки від виконуваної програми не надійдуть дані, або не виникне помилка (наприклад, з'єднання буде перервано). У разі виникнення помилки функції **recv ()** і **send ()** повертають значення *SOCKET_ERROR (-1).*

6. *Закриття сокета і деініціалізація*. Після завершення мережевої взаємодії між додатками слід закрити сокети і припинити використання в

додатку інтерфейсу Windows Socket. Закриття сокета здійснюється за допомогою функції **closesocket (SOCKET sock)**, єдиним аргументом якої є дескриптор сокета sock, який повинен бути закритий. Відключення доступу до інтерфейсу Windows Socket (відключення відповідної DLL-бібліотеки від процесу користувальницького додатка) виконується функцією **WSACleanup ()**, яка не має вхідних параметрів. Обидві зазначені функції повертають нуль, якщо відповідна операція була викликана успішно.

10.2 Приклад серверної частини комплексу розподілених обчислень, які взаємодіють через протокол TCP-IP

Програма використовує механізм сокетів ОС Windows, тому спочатку необхідно проініціалізувати систему сокетів функцією WSAStartup (а перед завершенням програми звільнити систему сокетів функцією WSACleanup).

Після цього можна використовувати функції для роботи з TCP-IP у стилі Берклі, стандартному для будь-якої ОС. Таким чином, створене ПЗ буде кросплатформеним (за винятком двох згаданих функцій WSAStartup та WSACleanup, виклик яких можна обмежити за допомогою директив умовної компіляції ifdef/ifndef).

Отже, після ініціалізації системи WinSocket, необхідно виконати стандартні функції для запуску з'єднання за протоколом TCP-IP: створення сокета, зв'язування сокета, запуск циклу очікування вхідних з'єднань.

Оскільки всі програми (і серверна і клієнтська) будуть запускатися на одній машині, то необхідно забезпечити їх взаємодію через зворотну петлю TCP-IP - адреса 127.0.0.1 (localhost). Це спеціальна адреса, створена для налагодження мережевих додатків та тестування роботи протоколу TCP-IP. Таким чином, і клієнт і сервер будуть запускатися на одній мереженій адресі, тільки використовувати будуть різні порти. У реальних системах клієнт «знає» адресу сервера, яка ніколи не змінюється.

Серверу необхідно встановити з'єднання з клієнтом та передати йому два числа: початок і кінець діапазону параметра для розрахунку суми. Число клієнтів в програмі зроблено змінним (задається адміністратором сервера при запуску), тому потрібно розробити алгоритм для поділу заданого діапазону на стільки приблизно рівних частин, скільки клієнтів задано.

Для успішної компонування потрібно підключити бібліотеку wsock32.lib для WinSock 1.1 і ws2_32.lib для WinSock 2.0.

```
#define SRV_PORT 1234
#define CLNT_PORT 1235
void Error(int ErrNo)
{
char mess[127]; switch(ErrNo)
{
case 0 :
stropy(mess,"\nCannot initialize UinSock system! Exiting..."); break;
 case 1:
strcpy(mess,"\nBad UinSock version! Exiting..."); break;
case 2:
strcpy(mess,"\nCannot create socket! Exiting..."); break;
case 3:
strcpy(mess,"\nCannot bind socket! Exiting..."); break;
case 4:
strcpy(mess,"\nCannot connect to the server! Exiting..."..."); break;
case 5:
strcpy(mess,"\nExiting..."); break;
case 6:
strcpy(mess,"\nCannot resolve host name! Exiting..."); break;
case 7:
```

```
strcpy(mess,"\nCannot receive data! Exiting..."); break;
case 8:
strcpy(mess,"\nCannot send data! Exiting..."); break;
}
printf("%s",mess);
WSACleanup();
ExitProcess(0); return;
}
int main(int argc, char* argv[])
{
int C= 0,N= 0,subrange,i,sockaddr_inlength,numb;
unsigned int s,s_new;
struct sockaddr_in sin,from_sin;
int min=0,max=0;
long double sum=0,result=0;
char local[] ="127.0.0.1";
WSADATA wsadata;
i£(WSAStartup(MAKEWORD(1,1),&wsadata)) //Version == 257
Error(O);
if(wsadata.wVersion!=MAKEWORD(1,1))
Error(1);
if((s=socket(AF_INET,SOCK_STREAM,0)) = = SOCKET_ERROR) //-1
Error(2);
memset((char*)&sin,0,sizeof(sin));
sin.sin_family=AF_INET;
sin.sin_addr.s_addr=inet_addr(local);
sin.sin_port=SRV_PORT;
if(bind(s,(struct sockaddr*)&sin,sizeof(sin))= =SOCKET_ERROR)
Error(3);
if(listen(s,3))
```

```
Error(4);
do {
printf("Enter maximum N value (1000-1000000000): ");
 scanf("%d",&N);
}while(!N);
do {
printf("How many clients you wanna use?"); scanf("%d",&C);
}         while(C<2||C>5);
sockaddr_inlength=sizeof(from_sin); sum=i=0;
if(N%C)subrange=N/C+1; else subrange=N/C;
while(i<C)
{
min=subrange*i;
max=subrange*(i+1)-1;
if(i==C-l) max=N%subrange;
printf("\nWaiting for incoming connections... IP:
%lx:%d",htonl(sin.sin_addr.S_un.S_addr),sin.sin_port);
do
s_new=accept(s,(struct sockaddr*)&from_sin,&sockaddr_inlength);
while(s_new< =0);
numb=send(s_new,(char*)&min,sizeof(int),0);
if(!numb)Error(6);
print f("\nSent %d bytes",numb);
numb=send(s_new,(char*)&max,sizeof(int),0);
if(!numb)Error(6);
printf("\nSent %d bytes",numb);
numb=recv(s_new,(char*)&result,sizeof(long double),0);
if(!numb)Error(7);
printf("\nReceived %d bytes\nResult is %lt", numb,result);
sum+=result; i++;
```

```
closesocket(s_new);}
closesocket(s);
printf("\nResult is %lf\n",sum);
Error(5);  return 0; }
```

10.3 Приклад клієнтської частини комплексу розподілених обчислень, які взаємодіють через протокол TCP-IP

Клієнтська частина в своїй роботі повинна створити сокет, зв'язати його, приєднатися до сервера, отримати два числа (початок і кінець діапазону), порахувати результат і послати його на сервер, закрити сокет.

Для створення з'єднання клієнту необхідно знати параметри відкритого сокета сервера. Для цього клієнтська частина створює ще один сокет srv_sin. Необхідно звернути увагу, щоб порти сервера збігалися в обох частинах програми: клієнтській та серверній.

```
int main(int argc, char* argv[])
{
int s, min=-l,max=-l,numb; struct sockaddr_in clnt_sin,srv_sin;
char local[]="127.0.0.1"; long double sum=0;
WSADATA wsadata;
if(WSAStartup(MAKEWORD(1,1),wsadata)) //Version == 257
 Error(0);
if(wsadata.wVersion!=MAKEW0RD(1,1))
Error(1);
if((s=socket(AF_INET,S0CK_STREAM,0)) == SOCKET_ERR0R) //-1
Error(2);
memset((char*)&clnt_sin,0,sizeof(clnt_sin));
clnt_sin.sin_family=AF_INET;
clnt_sin.sin_addr.s_addr=inet_addr(local);
clnt_sin.sin_port=htons(CLNT_P0RT);
```

```
if(bind(s,(struct sockaddr*)&clnt_sin,sizeof(clnt_sin))==
SOCKET_ERR0R)
Error(3);
memset((char*)&srv_sin,0,sizeof(srv_sin));
srv_sin.sin_f amily=AF_INET;
srv_sin.sin_addr.s_addr=inet_addr(local);
srv_sin.sin_port=SRV_PORT;
if(connect(s,(struct sockaddr*)&srv_sin,sizeof(struct sockaddr)) ==
SOCKET_ERR0R)
Error(4);
numb=recv(s,(char*)&min,sizeof(int),0);
if(!numb)Error(7);
printf("\nReceived %d bytes",numb);
numb=recv(s,(char*)&max,sizeof(int),0);
if(!numb)Error(7);
printf("\nReceived %d bytes",numb);
printf("\nGot range from %d to %d",min,max);
for(int i=min;i<=max;i++)
sum+=sin(i)+i*i;
numb=send(s,(char*)&sum,sizeof(long double),0);
if (!numb)Error(8);
printf("\nSent %d bytes",numb);
closesocket(s);
printf("\n\nThe data was successfully computed and sent!\n");
Error(5);
return 0;
}
```

10.4 Контрольні запитання

1. Поняття сокету. Функція локального керування.

2. Функція встановлення зв’язку.

3. Функції обміну даними. Функції закриття зв’язку.

4. Призначення та синтаксис структури WSAStartup.

5. Етапи створення з’єднання на стороні сервера. Створення сокету. Зв’язування сокету.

6. Етапи створення з’єднання на стороні клієнта.

7. Створення сокету. Зв’язування сокету. Запит на встановлення зв’язку.

Розгляд питань про основні концепції, механізм і функції розподілених обчислювальних систем можна знайти в книгах [18-20].

11 ВИЗНАЧЕННЯ ХМАРНИХ ОБЧИСЛЕНЬ

З одного боку, термін «Хмарні обчислення» немає стандартного визначення. З іншого боку, безліч різних корпорацій, вчених та аналітиків дають власні визначення цьому терміну.

Визначення хмарних обчислень викликало дебати і в науковому співтоваристві. На відміну від визначень, які можна знайти в комерційних виданнях, наукові визначення орієнтуються не тільки на те, що буде надано користувачеві, але і на архітектурні особливості запропонованої технології.

Наприклад, в лабораторії Берклі дають таке визначення хмарних обчислень:

«Хмарні обчислення - це не тільки додатки, які поставляються як послуг через Інтернет, а й апаратні засоби та програмні системи в центрах обробки даних, які забезпечують надання цих послуг. Послуги самі по собі вже давно називають «надання програмного забезпечення як послуги» (Software-as-a-Service або SaaS). Хмарою називається апаратне та програмне забезпечення центру обробки даних. Суспільна хмара надає ресурси широкому колу користувачів за принципом «оплата у міру використання» (pay-as-you-go - принцип надання послуг, при якому

користувач оплачує тільки ті ресурси, які були за фактом витрачені на вирішення поставленого завдання). Приватна хмара - це внутрішні центри обробки даних, в комерційній або іншій організації, які не доступні широкому колу користувачів. Таким чином, хмарні обчислення є сумою SaaS і «комунальних обчислень» (Utility Computing - модель обчислювальних систем, в якій надання даних та процесорних потужностей організовано за принципами комунальних послуг). Люди можуть бути користувачами або провайдерами SaaS, або користувачами або постачальниками комунальних обчислень».

Дане складне визначення виділяє іншу сторону хмарних обчислень: з точки зору провайдера, найважливішою складовою хмари є центр обробки даних (ЦОД). ЦОД містить обчислювальні ресурси та сховища інформації, які разом з програмним забезпеченням надаються користувачеві за принципом «оплата у міру використання».

Ян Фостер визначає хмарні обчислення як «парадигму великомасштабних розподілених обчислень, засновану на ефекті масштабу, в рамках якої пул абстрактних, віртуалізованних, динамічно-масштабованих обчислювальних ресурсів, ресурсів зберігання, платформ та сервісів пропонується за запитом зовнішнім користувачам через Інтернет».

Дане визначення додає два найважливіших аспекти у визначення хмарних обчислень: віртуалізацію та масштабованість. Хмарні обчислення абстрагуються від базової апаратної та програмної інфраструктури за допомогою віртуалізації. Віртуалізовані ресурси надаються за допомогою певних абстрактних інтерфейсів (програмних інтерфейсів API або сервісів). Така архітектура забезпечує масштабованість та гнучкість фізичного рівня хмари без наслідків для інтерфейсу кінцевого користувача.

Всі визначення ілюструють одну просту думку: феномен хмарних обчислень об'єднує кілька різних концепцій інформаційних технологій та представляє собою нову парадигму надання інформаційних ресурсів

(апаратних та програмних комплексів). З боку власника обчислювальних ресурсів хмарні обчислення орієнтовані на надання інформаційних ресурсів зовнішнім користувачам. З боку користувача, хмарні обчислення - це отримання інформаційних ресурсів у вигляді послуги у зовнішнього постачальника, оплата за яку проводиться в залежності від обсягу спожитих ресурсів відповідно до встановленого тарифу. Ключовими характеристиками хмарних обчислень є масштабованість та віртуалізація.

Масштабованість є можливістю динамічного налаштування інформаційних ресурсів до мінливого навантаження, наприклад до збільшення або зменшення кількості користувачів, зміни необхідної ємності сховищ даних або обчислювальної потужності. Віртуалізація, яка також розглядається як найважливіша технологія всіх хмарних систем, в основному використовується для забезпечення абстракції і інкапсуляції.

Можна виділити наступні основні риси хмарних обчислень (рис. 11.1.):

– хмарні обчислення представляють собою нову парадигму надання обчислювальних ресурсів;

– базові інфраструктурні ресурси (апаратні ресурси, системи зберігання даних, системне ПЗ) та додатки надаються у вигляді сервісів. Дані сервіси можуть надаватися незалежним постачальником для зовнішніх користувачів за принципом «оплата у міру використання»;

– основними особливостями хмарних обчислень є віртуалізація і динамічна масштабованість;

– хмарні сервіси можуть надаватися кінцевому користувачеві через веб-браузер або за допомогою певного програмного інтерфейсу.

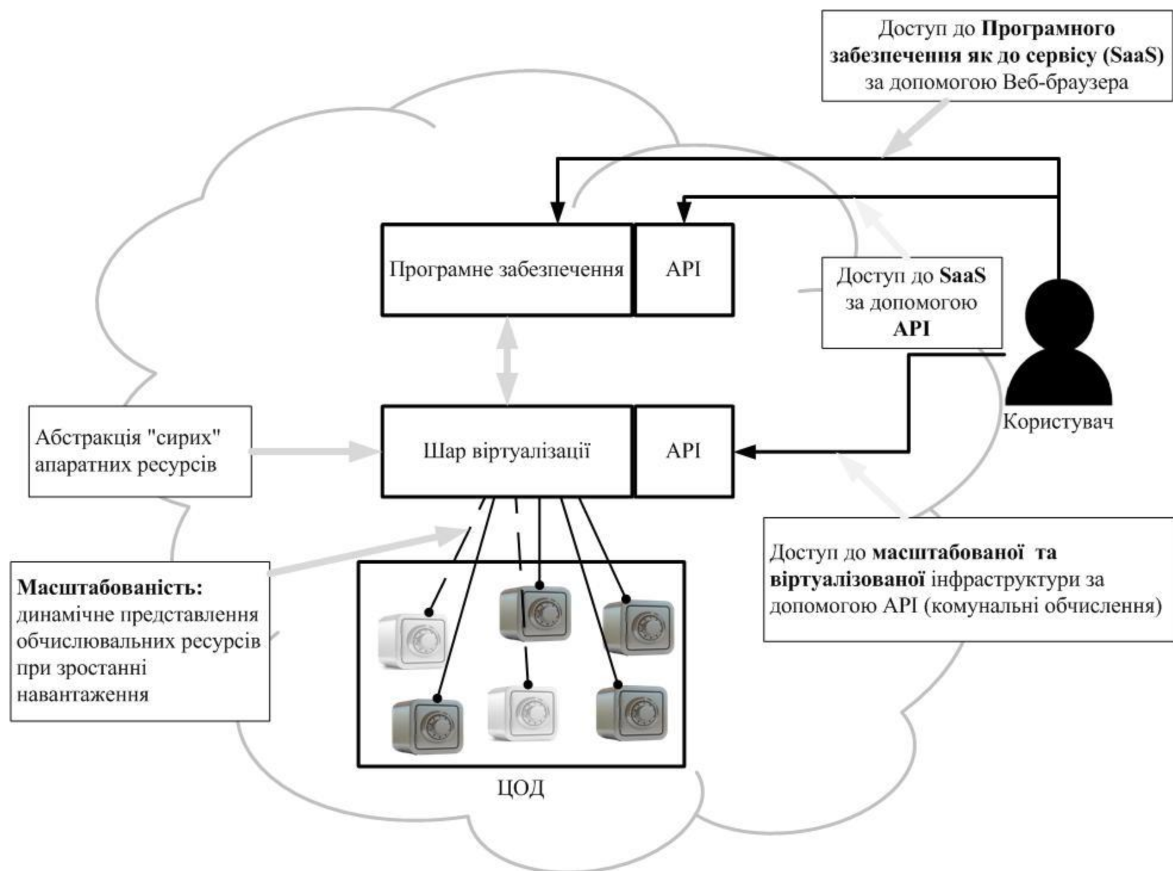


Рис. 11.1. Основні риси хмарних обчислень

11.1 Архітектура хмарних додатків

Всі можливі методи класифікації хмар можна звести до тришарової архітектури хмарних систем, що складається з наступних рівнів (рис. 11.2.):

- інфраструктура як сервіс (Infrastructure as a Service: IaaS);
- платформа як сервіс (Platform as a Service: PaaS);
- програмне забезпечення як сервіс (Software as a Service: SaaS).

Далі розглядається більш докладно, що собою представляє кожен із зазначених рівнів, і яким чином вони взаємодіють один з одним.

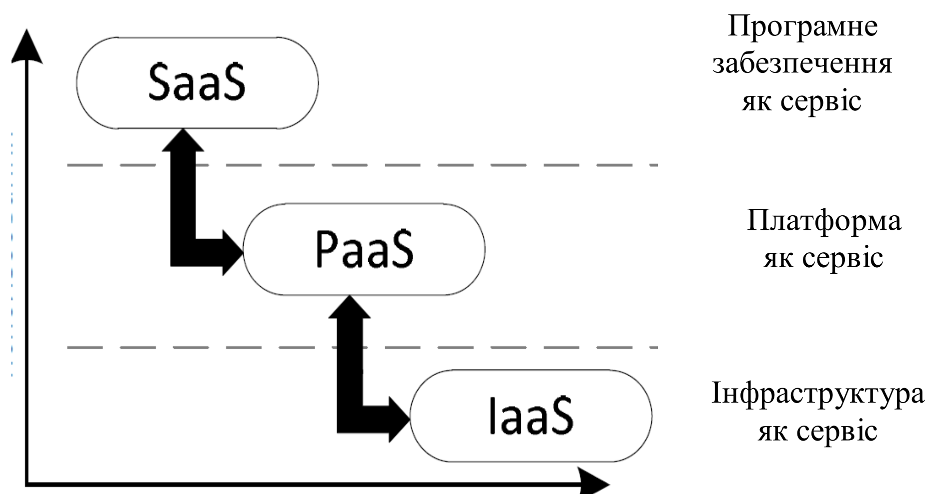


Рис. 11.2. Рівні хмарних систем

11.1.1 Інфраструктура як сервіс (IaaS)

IaaS пропонує інформаційні ресурси, такі як обчислювальні цикли або ресурси зберігання інформації, у вигляді сервісу. Яскравим прикладом такого підходу є хмара компанії Amazon - Amazon Web Services, що складається з Elastic Compute Cloud (EC2), що надає інформаційні ресурси у вигляді сервісів та Simple Storage Service (S3) для зберігання інформації.

Іншим прикладом такого підходу може бути сервіс Joynet, який забезпечує хостинг високо-масштабованих веб-сайтів та веб-додатків. Замість надання доступу до «сирих» обчислювальних пристроїв та систем зберігання, постачальники IaaS зазвичай надають віртуалізовану інфраструктуру у вигляді сервісу. Зазвичай «сирі» ресурси (процесорні цикли, мережеве обладнання, системи зберігання) розташовують на базовому рівні, над яким за допомогою віртуалізації надбудовують шари сервісів, які і надаються кінцевим користувачам у вигляді IaaS.

Треба сказати, що ще задовго до появи хмарних обчислень інфраструктура була доступна як сервіс. Такий підхід мав назву

«комунальні обчислення», і це словосполучення часто застосовується деякими авторами під час опису інфраструктурного рівня хмарних систем.

У порівнянні з ранніми спробами організації комунальних обчислень, підхід IaaS надає розробникам зрозумілий інтерфейс, до якого легко отримати доступ та використовувати у власних додатках. Даний інтерфейс повинен легко інтегруватися з інфраструктурою потенційних користувачів та розробників рішень SaaS. Таким чином, ресурси постачальників комунальних обчислень можуть бути ефективно використані тільки в тому випадку, якщо вони використовуються великим числом користувачів, а цього можна досягти шляхом організації гарного програмного інтерфейсу до своїх ресурсів.

11.1.2 Платформа як сервіс (PaaS)

Платформа - це шар абстракції між програмними додатками (SaaS) і віртуалізованою інфраструктурою (IaaS). Основною цільовою аудиторією PaaS є розробники додатків. Розробники можуть писати власні додатки на основі специфікацій певної платформи, не піклуючись про те, яким чином організувати взаємодію з нижчою інфраструктурою (IaaS). Розробники завантажують свій програмний код на платформу, яка забезпечує автоматичне масштабування додатків в залежності від навантаження. Яскравим прикладом реалізації підходу PaaS є платформа Google App Enginge, що забезпечує виконання призначених для користувача додатків на інфраструктурі Google. Шар PaaS грунтується на стандартизованому інтерфейсі, що надається шаром IaaS, який віртуалізує базові обчислювальні ресурси та надає стандартний інтерфейс для розробки додатків, що функціонують на шарі SaaS.

11.1.3 Програмне забезпечення як сервіс (SaaS)

SaaS - це програмне забезпечення, яке надається за принципом «оплата у міру використання» та управляється дистанційно одним або

декількома постачальниками. SaaS - це найбільш помітний шар хмарних обчислень, так як саме він представляє реальну цінність для кінцевого користувача і забезпечує рішення його завдань.

З точки зору користувача, основною перевагою SaaS є цінова перевага перед «класичним» ПЗ. Оплата SaaS здійснюється за моделлю «оплата у міру використання», що означає відсутність необхідності інвестицій у власну апаратну та програмну інфраструктуру.

Яскравим прикладом SaaS є комплекс Google Apps, що включає в себе такі системи як Google Mail та Google Docs.

Типовий користувач SaaS не може контролювати базову інфраструктуру, будь це програмна платформа, на якій SaaS засноване (PaaS) або ж безпосередньо апаратна інфраструктура (IaaS). Однак постачальник SaaS зобов'язаний пропрацювати взаємодію даних шарів, тому що вони необхідні для роботи системи. Наприклад, SaaS-додаток може бути розроблено на базі існуючої платформи та виконуватися на інфраструктурі, наданою сторонньою компанією. Робота з платформами і/або інфраструктурою як з сервісом є дуже привабливою з точки зору постачальників SaaS, так як це може зменшити в рази відрахування на використовувані ліцензії або витрати на інфраструктуру. Це також дозволяє їм зосередитися на тих областях, в яких вони по-справжньому компетентні. Як можна помітити, це дуже схоже на ті переваги, які мотивують кінцевих користувачів ПЗ переходити до використання рішень SaaS. За даними ринкових аналітиків, високий тиск ринку приводить компанії до необхідності скорочення витрат на IT, що призводить до високого попиту і зростання рішень SaaS, і, тим самим, зростання хмарних обчислень в цілому.

11.2 Компоненти хмарних додатків

На сьогоднішній день не існує єдиної компонентної архітектури хмарних додатків. Це викликано високою закритістю різних аспектів

реалізації найбільш поширених хмарних систем. Але, не дивлячись на це, можна виділити основні найбільш важливі компоненти, властиві практично всім існуючим хмарним платформ.

Хмару можна розбити на основні компоненти, що відображають наступні ключові особливості хмарних рішень (рис. 11.3).

1) Платформа: середовище та набір утиліт, що забезпечують розробку, інтеграцію та надання хмарних сервісів. Платформа є центральним компонентом моделі хмари. Платформа з одного боку, надає набір базових сервісів, доступних розробнику хмарного додатка, а з іншого накладає певні обмеження на методи розробки та подання додатка. При виборі платформи можна ґрунтуватися як на вже готові рішеннях (Google App Engine або Microsoft Azure), так і самостійно розробити масштабовану платформу на базі готової хмарної інфраструктури. При виборі базової платформи необхідно виходити з критеріїв вартості закінченого рішення, продуктивності та необхідної масштабованості. Також необхідно пам'ятати, що будь-яка вибрана платформа потребує використання певних мов програмування та програмних фреймворків для реалізації програми.

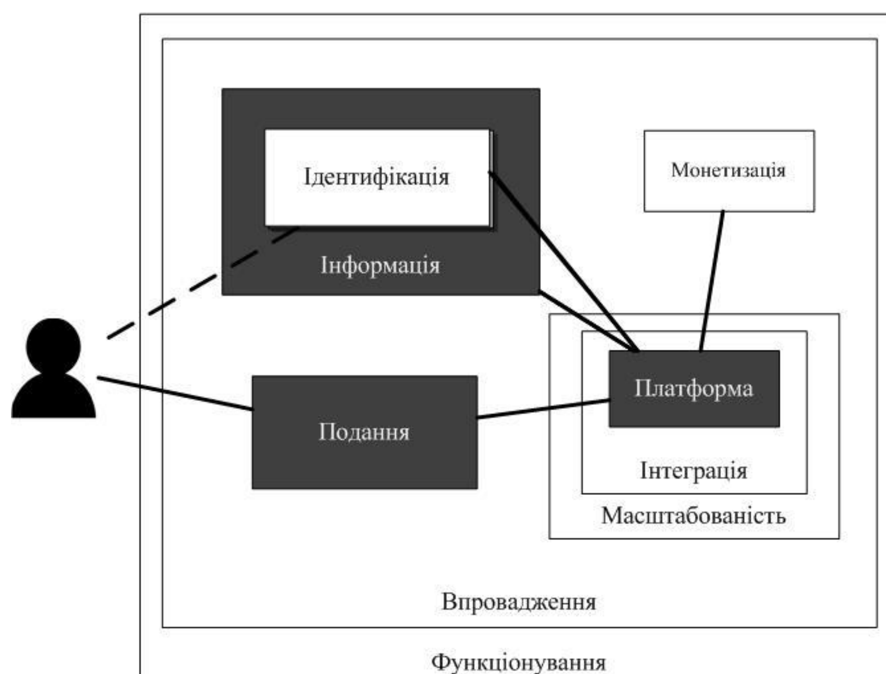


Рис. 11.3. Компоненти хмарних систем

2) Представлення: інтерфейс, через який користувач проводить взаємодію з хмарою. Цей компонент забезпечує отримання вхідних даних та надання інформації кінцевому користувачу. Найбільш типовим методом реалізації представлення є веб-додаток, що забезпечує взаємодію з користувачем за допомогою веб-браузера. Останнім часом все частіше використовуються всі можливості надання користувачу зручного інтерфейсу роботи незалежно від пристрою, з якого він виходить в Інтернет. Це призводить до розробки окремих користувацьких інтерфейсів для мобільних пристроїв (смартфонів, планшетів) які можуть представляти собою як окремі Інтернет-сторінки, так і повноцінні мобільні додатки, які взаємодіють з хмарою за допомогою API.

3) Інформація: джерела даних, що забезпечують розподілене зберігання структурованих або неструктурованих, статичних або динамічно-змінюваних даних. Призначена для користувача інформація в хмарних системах може досягати величезних обсягів. Наприклад, в системі Gmail на початок 2010 року було понад 170 мільйонів активних користувачів. Навіть якщо припустити, що в середньому використовується не більше 5% від доступних користувачеві 7 Гб, виходить що Google необхідно забезпечувати зберігання більш ніж 60 000 Тб даних поштового листування. І цей обсяг експоненціально збільшується з кожним місяцем. На таких обсягах даних класичні SQL бази даних вже не дають задовільних результатів по швидкості обробки. Більш того, хмарним платформам часто доводиться обробляти повнозв'язані структури даних (графи, дерева) що стає дуже важко при використанні SQL-підходу та реляційних баз даних. У зв'язку з цим, в останні кілька років стали активно розвиватися альтернативні «NoSQL» системи управління базами даних (стовпчикові СУБД, такі як Hadoop; документно-орієнтовані СУБД, такі як CouchDB і MongoDB) та альтернативні підходи до обробки надвеликих обсягів інформації. Найбільш відомою реалізацією такого підходу є

фреймворк MapReduce, представлений компанією Google. Він використовується для паралельних обчислень над дуже великими (кілька петабайт) наборами даних в комп'ютерних кластерах.

4) Ідентифікація: інформація про основних споживачів хмарних ресурсів, використовується для оптимізації та налаштування хмари під їх завдання. Більшості додатків необхідно вміти відрізняти користувачів один від одного для надання релевантної інформації (наприклад, поштовому клієнту необхідно забезпечити аутентифікацію та авторизацію користувачів, щоб представити кожному з них можливість читання їх особистої поштової кореспонденції). Така інформація має першорядну значимість для хмарної платформи, так як з нею зав'язані великі обсяги призначених для користувача даних, і при цьому необхідно забезпечити її максимальну доступність в рамках системи, тому що процедура авторизації повинна проходити максимально швидко. Також, необхідно забезпечити прозору авторизацію користувача в усіх сервісах однієї хмарної платформи, щоб не було потрібно щоразу вводити ім'я користувача і пароль заново. У зв'язку з розподіленою природою хмарних сервісів необхідно забезпечити високий рівень безпеки при роботі з призначеною для користувача інформацією.

5) Інтеграція: інфраструктура, яка спрощує обмін інформацією та виконання завдань в розподіленому обчислювальному середовищі. Високий ступінь декомпозиції сервісів дозволяє досягти максимальної ефективності та гнучкості виконання хмарних додатків, так як з'являється можливість завантаження відразу декількох обчислювальних машин при виконанні одного користувальницького завдання. У зв'язку з цим з'являється необхідність організації обміну інформацією та виконання завдань в розподілених обчислювальних середовищах. В рамках цього компонента необхідно забезпечити максимальну продуктивність та безпеку процесу обміну даними між сервісами. Далі, необхідно забезпечити сумісність форматів даних та розробити механізми синхронної

й асинхронної взаємодії з успадкованим ПЗ. На більш високому рівні необхідно забезпечити зв'язок програмних компонентів та переконатися у відсутності вузьких місць в програмній архітектурі системи.

6) Масштабованість: гнучкість методів надання ресурсів, що забезпечує підтримку виділення додаткових інформаційних ресурсів при зростанні навантаження на додаток. При цьому необхідно враховувати не тільки можливість короткочасного збільшення навантаження на додаток (наприклад, в результаті напливу відвідувачів після появи рекламної статті на одному з популярних Інтернет-ресурсів) але і планувати довгострокове збільшення продуктивності системи в результаті постійного приросту аудиторії. В обох випадках, необхідно забезпечити декомпозицію хмарного додатка на окремі модульні компоненти, які можуть бути розподілені на декілька обчислювальних пристроїв.

7) Монетизація: облік та білінг ресурсів, витрачених на виконання користувальницьких завдань. Це ключовий компонент безлічі комерційних додатків. Для організації якісного білінгу хмарних платформ необхідно організувати збір та надання повноцінної інформації про всілякі ресурси, що витрачаються на рішення завдань користувача. Також, необхідно забезпечити користувачеві можливість зручної та швидкої оплати витрачених ресурсів.

8) Впровадження: процес розробки нового хмарного додатка, який включає в себе розробку, тестування та впровадження в експлуатацію. На етапі розробки хмарної програми потрібен зовсім невеликий обсяг обчислювальних ресурсів, який значно збільшується при переході до етапу тестування та впровадження в експлуатацію. При цьому очевидно, що застосування готової хмарної інфраструктури дозволяє значно скоротити витрати на розробку та впровадження високомасштабованого додатку, так як оплата використаних інформаційних ресурсів проводиться на основі моделі комунальних обчислень та не вимагає значних інвестицій у власну

інфраструктуру. Це дозволяє мінімізувати початкові витрати та сконцентрувати фінансування на всебічному тестуванні програми.

Але, не дивлячись на всі перераховані переваги, необхідно оцінити всі можливі недоліки моделі хмарних обчислень, як то складності організації реплікації даних між сервісами, складність відкату на попередні версії при появі несподіваних помилок в процесі впровадження, необхідність обережного і всебічного тестування розроблених сервісів на сумісність даних та злагодженість роботи програми.

9) Функціонування: моніторинг та підтримка додатків, які перебувають в стадії експлуатації. Додаток, який запущено в експлуатацію, необхідно адмініструвати, що може виявитися надзвичайно складним завданням, якщо врахувати велике число окремих сервісів, що становлять хмарний додаток. У зв'язку з цим необхідно забезпечити інтеграцію процесів адміністрування та управління сервісами у вигляді єдиного «центру управління сервісами». Паралельно, в нього можна включити моніторинг навантаження додатку, панель управління призначеними для користувача завданнями тощо.

Всі перераховані вище компоненти хмарного додатка повинні бути заплановані з самого початку розробки для забезпечення високого рівня масштабованості та автоматизації.

11.3 Переваги та недоліки хмарних обчислень

Як раніше було описано, хмарні обчислення орієнтовані на надання інформаційних ресурсів на трьох рівнях: рівні інфраструктур (IaaS), рівні платформ (PaaS) та рівні програмного забезпечення (SaaS).

Надаючи інтерфейси до всіх трьох різних рівнів, хмари взаємодіють з трьома різними типами споживачів:

1) Кінцеві користувачі, які використовують SaaS-рішення через веб-браузер, або ж будь-які базові ресурси інфраструктурного шару які

надаються за допомогою шару SaaS (наприклад, хмарні ресурси зберігання за допомогою Dropbox.com).

2) Корпоративні споживачі, які можуть використовувати всі три шари: IaaS - для того, щоб розширити власну програмно-апаратну інфраструктуру або отримати додаткові обчислювальні ресурси на вимогу; PaaS - для того, щоб мати можливість запуску власних програм в хмарі; SaaS - для отримання можливостей тих додатків, які вже доступні в хмарі.

3) Розробники та незалежні постачальники програмного забезпечення, які розробляють додатки у вигляді хмарних SaaS-рішень. Зазвичай, ця категорія користувачів безпосередньо взаємодіє з шаром PaaS, і вже через нього, опосередковано, з шаром IaaS.

З точки зору користувача, основна перевага хмарних обчислень - це модель оплати ресурсів у міру їх використання. Не має потреби попередніх інвестицій в інфраструктуру: немає необхідності інвестицій в ліцензійне ПЗ, відсутня необхідність інвестиції в апаратну інфраструктуру та пов'язані з цим витрати на обслуговування і персонал. Таким чином, капітальні витрати перетворюються в поточні витрати.

Покупці хмарних сервісів використовують тільки той обсяг інформаційних ресурсів, який їм насправді потрібен, та оплачують тільки той обсяг інформаційних ресурсів, якими вони реально скористалися. У той же час, вони користуються такими перевагами хмарних обчислень як масштабованість та гнучкість. Хмарні обчислення дозволяють легко і швидко надати необхідні обчислювальні ресурси на вимогу.

Проте, існують деякі недоліки моделі хмарних обчислень, які випливають з однією очевидною особливості - хмари обслуговують відразу безліч різних клієнтів. Користувач хмарної платформи не знає, завдання яких користувачів будуть виконуватися на тому чи іншому сервері, що входить в хмарну інфраструктуру. Типова хмара є зовнішньою по відношенню до внутрішньої інфраструктури будь-якої компанії, тобто знаходиться поза зоною відповідальності адміністраторів та служб безпеки

компанії-споживача. У той час як для окремого споживача це може бути несуттєвим, великі компанії приділяють цьому питанню дуже велике значення.

Користувачеві доводиться покладатися на обіцянки постачальника хмарних рішень в питаннях надійності, продуктивності та якості обслуговування інфраструктури хмари. Використання хмар також пов'язане з високими ризиками безпеки та захисту конфіденційної інформації. Це пов'язано з двома факторами: 1) необхідність завантаження та отримання даних із хмари; 2) зберігання даних проводиться на базі віддалених сховищ, в зв'язку з чим власнику інформації доводиться покладатися на гарантії постачальника хмарних рішень, що несанкціонованого доступу до даних не відбудеться. Більш того, використання хмар вимагає певних інвестицій в інтеграцію власної інфраструктури та додатків у хмару.

У зв'язку з цим, необхідно завжди враховувати ризики, пов'язані з використанням хмар. У кожному окремому випадку необхідно уважно зважити всі потенційні вигоди та загрози при переході на хмарну платформу. Також, необхідно вирішити які дані та яка обробка може проводитися на базі «хмарного аутсорсингу», а які дані краще ніколи не виводити за рамки локальної мережі організації.

11.4 Контрольні запитання

1. Визначення та основні риси хмарних обчислень.
2. Архітектура хмарних додатків.
3. Характеристика рівнів хмарних систем.
4. Компоненти хмарних додатків.
5. Переваги та недоліки хмарних обчислень

Більш детальний розгляд питань про основні концепції, організацію хмарних обчислень можна знайти в [18,19].

12 КЛАСТЕРНІ СИСТЕМИ

Термін «грід» був введений Яном Фостером на початку 1998 року публікацією книги «Грід. Нова інфраструктура обчислень »[20]:

Грід - це система, яка координує розподілені ресурси за допомогою стандартних, відкритих, універсальних протоколів та інтерфейсів для забезпечення нетривіальної якості обслуговування (QoS - Quality of Service).

12.1 Архітектура Грід-систем

Основною ідеєю в концепції грід-обчислень, є централізоване віддалене надання ресурсів, необхідних для вирішення різного роду обчислювальних завдань. Можна запустити будь-яке завдання з будь-якого комп'ютера або мобільного пристроїв на обчислення, ресурси ж для цього обчислення повинні бути автоматично надані на віддалених високопродуктивних серверах, незалежно від типу нашої задачі.

З більш практичної точки зору, основне завдання, що лежить в основі концепції грід, це узгоджений розподіл ресурсів та вирішення задач в умовах динамічних, багатопрофільних віртуальних організацій. Розподіл ресурсів, в якому зацікавлені розробники грід, це не обмін файлами, а прямий доступ до комп'ютерів, програмного забезпечення, даних та інших ресурсів, які потрібні для спільного вирішення задач і стратегій управління ресурсами, що виникають в промисловості, науці та техніці. Віртуальної організацією (ВО) називають ряд окремих людей або установ, об'єднаних єдиними правилами колективного доступу до розподілених обчислювальних ресурсів [19]. Для організації роботи в рамках таких ВО виникає необхідність в наступному:

1) В гнучких механізмах розподілу ресурсів, починаючи від клієнт-серверних закінчуючи одноранговими.

2) В розвиненій системі контролю ресурсів, які використовуються, включаючи контроль над мікромодульними та іншими методами доступу та використання локальних й глобальних підходів.

3) В розподіленому доступі до різних ресурсів, починаючи від програм, файлів та даних закінчуючи комп'ютерами, сенсорами та мережами.

4) В різних моделях використання ресурсів (від одного користувача до багатокористувацьких, від високопродуктивних до мало витратних), що включають регулювання якості наданого обслуговування, планування, перерозподіл та ведення обліку ресурсів.

Виділяють такі рівні архітектури грід-систем:

1) Базовий рівень (Fabric) - містить різні ресурси, такі як комп'ютери, пристрої зберігання, мережі, сенсори та ін.

2) Зв'язуючий рівень (Connectivity) - визначає комунікаційні протоколи та протоколи аутентифікації.

3) Ресурсний рівень (Resource) - реалізує протоколи взаємодії з ресурсами РОС та їх управління.

4) Колективний рівень (Collective) - управління каталогами ресурсів, діагностика, моніторинг;

5) Прикладний рівень (Applications) - інструментарій для роботи з грід та користувальницькими програмами.

На базовому рівні визначаються служби, що забезпечують безпосередній доступ до ресурсів, використання яких розподілено за допомогою протоколів Грід.

1) Обчислювальні ресурси надають користувачеві Грід-системи (точніше кажучи, завданню користувача) процесорні потужності. Обчислювальними ресурсами можуть бути як кластери, так і окремі робочі станції. Попри всю різноманітність архітектур будь-яка обчислювальна система може розглядатися як потенційний обчислювальний ресурс Грід-системи.

2) Ресурси пам'яті представляють собою простір для зберігання даних. Для доступу до ресурсів пам'яті також використовується програмне забезпечення проміжного рівня, що реалізує уніфікований інтерфейс управління та передачі даних.

3) Інформаційні ресурси та каталоги є особливим видом ресурсів пам'яті. Вони служать для зберігання та надання метаданих й інформації про інші ресурси Грід-системи.

4) Мережний ресурс є ланкою, що зв'язує розподілені ресурси Грід-системи. Основною характеристикою мережного ресурсу є швидкість передачі даних.

Зв'язуючий рівень визначає комунікаційні протоколи та протоколи аутентифікації, забезпечуючи передачу даних між ресурсами базового рівня. Зв'язуючий рівень грід заснований на стеку протоколів TCP / IP:

1) Інтернет (IP, ICMP);

2) Транспортні протоколи (TCP, UDP);

3) Прикладні протоколи (DNS, OSRF).

Ресурсний рівень реалізує протоколи, що забезпечують виконання таких функцій:

- узгодження політик безпеки використання ресурсу;
- процедура ініціації ресурсу;
- моніторинг стану ресурсу;
- контроль над ресурсом;
- облік використання ресурсу.

Окремо виділяються два типи протоколів ресурсного рівня:

1) Інформаційні протоколи - використовуються для отримання інформації про структуру та стан ресурсу.

2) Протоколи управління - використовуються для узгодження доступу до ресурсів, що розділяються, визначаючи вимоги та допустимі дії по відношенню до ресурсу (наприклад, підтримка резервування, можливість створення процесів, доступ до даних).

Колективний рівень відповідає за глобальну інтеграцію різних наборів ресурсів та може включати в себе служби каталогів; служби спільного виділення, планування і розподілу ресурсів; служби моніторингу та діагностики ресурсів; служби реплікації даних.

На прикладному рівні розташовуються користувальницькі додатки, що виконують у середовищі ВО. Вони можуть використовувати ресурси, які розташовуються на будь-яких нижчих рівнях архітектури Грід.

12.2 Стандарти Грід

Ключовим моментом в розробці грід додатків є стандартизація, що дозволяє організувати пошук, використання, розміщення та моніторинг різних компонентів, які складають єдину віртуальну систему, навіть якщо вони надаються різними постачальниками послуг або управляються різними організаціями [19]. До початку 2001 року в різних проектах були представлені різні методи реалізації грід-обчислень. Але всі вони сходилися в одному: для гнучкого, прозорого і надійного надання доступу до обчислювальних ресурсів була запропонована сервіс-орієнтована модель.

У 2001 році в якості бази для створення стандарту архітектури грід-додатків була обрана технологія веб-сервісів. Даний вибір був обумовлений двома основними перевагами даної технології. По-перше, мова опису інтерфейсів веб-сервісів WSDL (Web Service Definition Language) підтримує стандартні механізми для визначення інтерфейсів окремо від їх реалізації, що в сукупності зі спеціальними механізмами зв'язування (транспортним протоколом і форматом кодування даних) забезпечує можливість динамічного пошуку та компонування сервісів в гетерогенних середовищах. По-друге, широко поширена адаптація механізмів веб-сервісів означає, що інфраструктура, побудована на базі веб-сервісів, може використовувати різні утиліти та інші існуючі сервіси [19]. Грід-сервіс підтримує такі стандартні інтерфейси.

1) Пошук. Грід-додаткам необхідні механізми для пошуку доступних сервісів та визначення їх характеристик.

2) Динамічне створення сервісів. Можливість динамічного створення та управління сервісами - це один з базових принципів OGSA, що вимагає наявності сервісів для створення нових сервісів.

3) Управління часом існування. Розподілена система повинна забезпечувати можливість знищення примірника грід-сервісу.

4) Повідомлення. Для забезпечення роботи грід-додатків набори грід-сервісів повинні мати можливість асинхронно повідомляти один одного про зміни в їх стані.

Перша реалізація моделі OGSA, розроблена в 2003 р, називалася OGSI (Open Grid Service Infrastructure). У зв'язку з тим, що існуючі тоді стандарти веб-сервісів (до яких належали WSDL, SOAP, UDDI) не могли забезпечити всіх вимог, що пред'являються розробниками до функціональних можливостей грід-сервісів, при створенні OGSI потрібно модифікувати та розширити відповідні стандарти. Це призвело до того, що спільне використання веб-сервісів та грід-сервісів в одному середовищі стало неможливим, через несумісність базових стандартів [19]. Подальші спільні зусилля спільноти грід та організацій з розробки стандартів веб-сервісів призвели до визначення стандартів, що відповідають вимогам грід. У запропонованому стандарті WSRF (Web Service Resource Framework) специфіковані універсальні механізми для визначення, перегляду та управління станом віддаленого ресурсу, що є критично-важливою з точки зору грід. На сьогоднішній день реалізація моделі OGSA за допомогою стандарту WSRF (та супутніх стандартів, таких як WS-Notification і WS-Addressing) є найбільш поширеною в середовищі грід.

В даний час, існують дві системи, що забезпечують інфраструктуру розробки грід-систем відповідно до стандартів OGSA, реалізованими за допомогою WSRF: Globus Toolkit і UNICORE.

12.3 Порівняння Грід та Хмарних обчислень

Опис концепції грід-обчислень в попередньому розділі і концепції хмарних обчислень, яка була представлена в цій, показують, що між ними багато спільного. Це спричинило за собою безліч дискусій, як в комерційній, так і в науковому середовищі. Основне питання цих дискусій полягав в наступному: чим хмарні та грід обчислення відрізняються один від одного? Чи дійсно термін «хмарні обчислення» - це просто новий маркетинговий прийом?

Деякі дослідники вважають, що основною відмінністю хмарних обчислень від грід є віртуалізація: «Хмарні обчислення, на відміну від грід, застосовують віртуалізацію для максимізації обчислювальної потужності. Віртуалізація, шляхом відокремлення логічного рівня від фізичного, вирішує безліч проблем, з якими стикаються грід-рішення ».

У той час як грід-системи забезпечують високе завантаження обчислювальних ресурсів за допомогою розподілу однієї складної задачі на кілька обчислювальних вузлів, хмарні обчислення йдуть шляхом виконання декількох завдань на одному сервері у вигляді віртуальних машин. Крім того, є особливості в основних варіантах використання грід та хмарних обчислень. Тоді як грід, в основному, використовується для вирішення завдань за певний (обмежений) проміжок часу, хмарні обчислення в основному орієнтовані на надання сервісів, які «довго» існують. Думки сходяться в одному - хмарні обчислення виросли з концепції грід. Фостер визначає взаємодію грід і хмарних обчислень в такий спосіб [20]:

«Ми вважаємо, що хмарні обчислення не просто перетинаються з концепцією грід. Насправді хмари виросли з грід-обчислень і ґрунтуються на концепції інфраструктури грід. Еволюція підходу полягає в тому, що замість надання «сирих» обчислювальних ресурсів та ресурсів зберігання забезпечується надання більш абстрактних ресурсів у вигляді сервісів ».

Таким чином, можна вважати що грід та хмарні обчислення доповнюють один одного. Інтерфейси та протоколи грід можуть забезпечити взаємодію між хмарними ресурсами або ж забезпечити об'єднання хмарних платформ. Також, більш високий рівень абстракції, який надається хмарними платформами, може допомогти користувачам грід-систем в організації прозорого та зручного надання ресурсів грід-платформ й залучити нові групи користувачів до використання таких ресурсів. З точки зору користувача, різниця між хмарними обчисленнями та грід обчисленнями буде полягати в наступному:

1) Хмарні платформи фокусуються на підході «все як сервіс». Грід обчислення фокусуються на проміжному програмному забезпеченні, яке надається у вигляді відкритих вихідних кодів або ж у вигляді готових пакетів. При цьому «комунальні» обчислення є всього лише однією з форм надання грід. У порівнянні з цим, хмарні обчислення фокусуються виключно на платному наданні інформаційних ресурсів кінцевому користувачеві. При цьому проміжне програмне забезпечення, яке дозволило б забезпечити розробку власної хмари, поки не дуже поширене.

2) Грід і хмарні обчислення фокусуються на різні типи обчислень. Спочатку, грід обчислення були орієнтовані на рішення наукових завдань за допомогою суперкомп'ютерних систем. В даний час грід застосовується для науково-дослідних завдань, вирішення яких вимагає об'єднання декількох суперкомп'ютерних платформ. З іншого боку, хмарні обчислення орієнтовані не на вирішення окремих завдань, а на перманентне надання певних сервісів кінцевим користувачам. Вони забезпечують динамічний розподіл фізичних ресурсів для задоволення змінного завантаження таких сервісів.

3) Різне взаємовідношення з постачальниками ресурсів. Грід обчислення ґрунтуються на понятті віртуальних організацій, що включають в себе кілька різних окремих організацій з чіткими правилами взаємодії між ними та чіткими політиками надання програмно-апаратних

ресурсів. Концепція хмарних обчислень забезпечує можливість будь-якій компанії використовувати хмарні сервіси для вирішення власних завдань, оплачуючи тільки ті ресурси, які необхідні.

4) Різні області застосування. Грід-платформи надають базу для розгортання обчислювальної інфраструктури. Хмарні обчислення надають інтегрований підхід на всіх рівнях надання інформаційних ресурсів: IaaS, PaaS, SaaS.

5) Розширення кількості призначених для користувача інтерфейсів. Грід обчислення орієнтовані на представлення різних обчислювальних ресурсів у гетерогенних обчислювальних середовищах для вирішення конкретних завдань. Таким чином, інтерфейси грід-орієнтовані на взаємодію обчислювальних інфраструктур на фізичному рівні за допомогою API, яким може скористатися тільки професійний програміст. Хмарні обчислення розробляються таким чином, щоб надавати інтерфейси кінцевим користувачам через веб-доступ або за допомогою API. На кожному рівні (IaaS, PaaS, SaaS) надається свій власний інтерфейс. Підвищення рівня абстракції дозволяє забезпечити застосування хмарних обчислень як на рівні окремих користувачів, так і на рівні корпоративних клієнтів.

У загальному, грід-обчислення забезпечують об'єднання гетерогенних обчислювальних ресурсів в єдину обчислювальну середу. Це те, з чого починаються і на чому ґрунтуються хмарні обчислення. Хмарні обчислення забезпечують більш високий рівень абстракції, надаючи обчислювальні ресурси кінцевим користувачам (будь то приватні клієнти або організації) у вигляді сервісів.

12.4 Контрольні запитання

1. Визначення кластерних систем.
2. Архітектура грід-систем.
3. Стандарти грід. Порівняння грід та хмарних обчислень.

Детальний огляд питань організації кластерних систем та обчислень відображено в [19,20].

СИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Эндрюс* Г. Р. Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс. - М.: Вильямс, 2003. – 512 с.

2. *Дорошенко А. Е.* Математические модели и методы организации высокопроизводительных параллельных вычислений *А. Е Дорошенко*. -К.: Наукова думка, 2000. – 177 с.

3. *Корнеев В. В.* Параллельные вычислительные системы / *В. В.* Корнеев. - М.: Нолидж, 1999. – 320 с.

4. *Гергель* В.П. Основы параллельных вычислений для многопроцессорных вычислительных систем / В.П. Гергель, Р.Г. Стронгин. Учебное пособие. – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. –184 с.

5. *Crichlow* J. M. An Introduction to Distributed and Parallel Computing / J. M. Crichlow. - Prentice Hall, 1997. – 209 p.

6. Элементы параллельного программирования / *В. А. Вольковский, В. Е. Котов, А. Г. Марчук, Н. Н. Миренков.* - М.: Радио й связь, 1983. – 240 с.

7. *Бурова* И.Г. Алгоритмы параллельных вычислений и программирование / И.Г. Бурова, Ю.К. Демьянович. Курс лекций. - СПб.: Изд-во С. -Пб. ун-та, 2007. – 206 с.

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

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

10. Synchronization of Parallel Programs / *Andre* J., Herman D., Verjus J.-P. Oxford: North Oxford Academic Publishing Company Limited, 1985. – 110 p.

11. Parallel Computing. Architectures, Algorithms and Applications / *Bischof C.*, Bücker M., Gibbon P., Joubert G.R., Lippert T., Mohr B., Peters F.

(eds.) OS Press, 2008. – 825 p.

12. Pllana Sabri. Programming multicore and many-core computing systems/ *Sabri Pllana*, Fatos Xhafa.Wiley, 2017. – 528 p.

13. *Рихтер Дж.* Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. Пер. с англ.: 4-е изд. — СПб.: Питер; М.: Русская Редакция, 2001. – 752 с.

14. *Качко Е.Г.* Параллельное программирование / Е.Г. Качко. Харьков: Форт, 2011. – 528 с.

15. *Корнеев В.Д.* Параллельное программирование в MPI / В.Д. Корнеев. 2-е изд., испр. - Новосибирск: Изд-во ИВМиМГ СО РАН, 2002. – 215 с.

16. *Лазарович І.М.* Паралельні обчислювальні середовища. Лабораторний практикум/ І. М. Лазарович. – Івано-Франківськ: Видавництво Прикарпатського національного університету імені Василя Стефаника, 2014. – 65 с.

17. *Антонов А.С.* Параллельное программирование с использованием технологии OpenMP / А.С. Антонов. Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.

18. *Таненбаум Э.* Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. – СПб.: Питер, 2003. – 877 с.

19. *Радченко Г.И.* Распределенные вычислительные системы / Г.И. Радченко. – Челябинск: Фотохудожник, 2012. – 184 с.

20. *Foster I.* The Grid. Blueprint for a new computing infrastructure / I. Foster, C. Kesselman. San Francisco: Morgan Kaufman, 1999. – 677 p.

Інформаційні ресурси мережі Інтернет

1. Інформаційно-аналітичні матеріали з паралельних обчислень (http://www.parallel.ru).
2. TOP500. Рейтинг 500 найпотужніших відомих суперкомп'ютерних систем (http://www.top500.org/).
3. Обчислювальний кластер Київського національного університету ім. Т. Г. Шевченка (http://cluster.univ.kiev.ua/).

ДОДАТОК А

Встановлення MPICH

1. Виконати інсталятор mpich2-L0.7-win32-ia32.msi , використовуючи права привілеями адміністратора. Для цього потрібно зайти в меню Пуск –> Програми –> Стандартні програму «Командний рядок», натисніть на неї правою кнопкою миші, і виберіть пункт «Запуск від імені адміністратора» (Рисунок 2). Підтвердити свої наміри і введіть пароль, якщо необхідно.

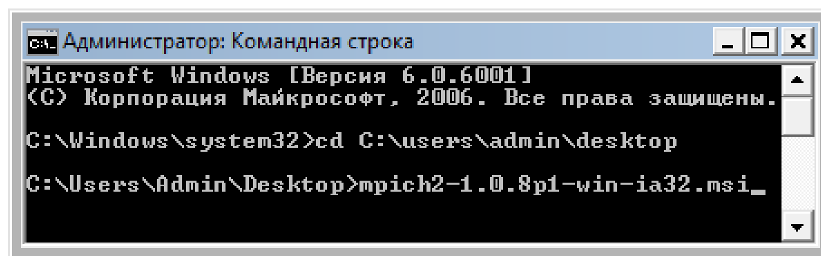
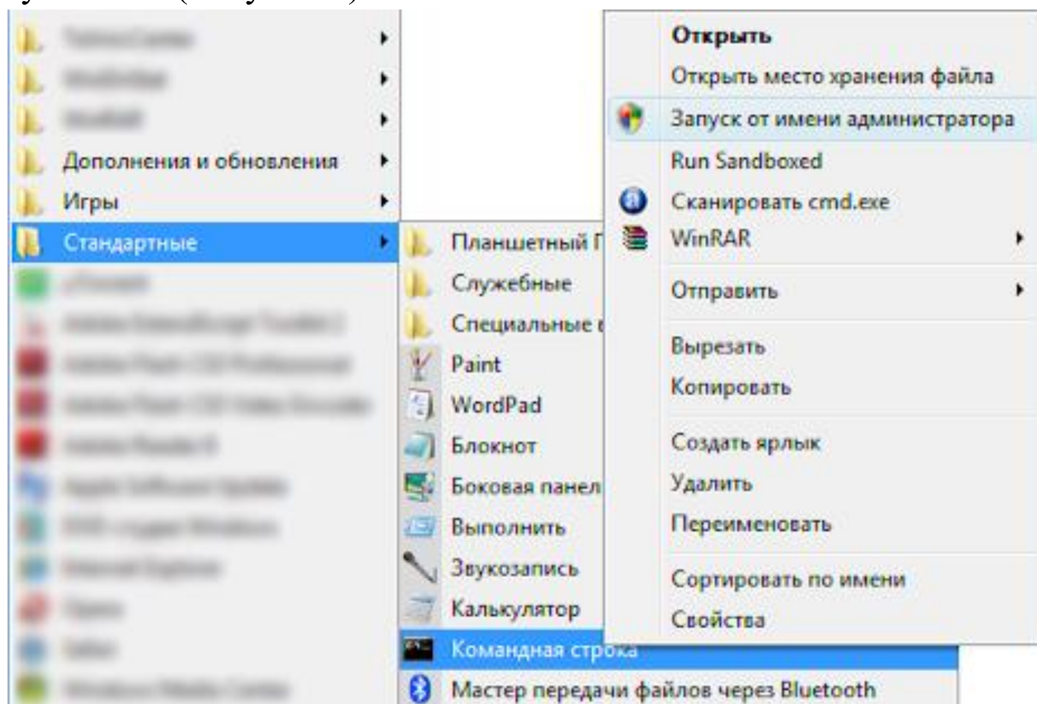


Рисунок 3. Запуск інсталятора з командного рядка

2. Ввести в командному рядку повний шлях до програми інсталяції і натиснути Enter (Рисунок 3).



3. Під час установки вам треба буде ввести пароль «mpi_pass» для

доступу до менеджера процесів SMPD. Необхідно ввести однаковий пароль на усіх комп'ютерах (Рисунок 4).

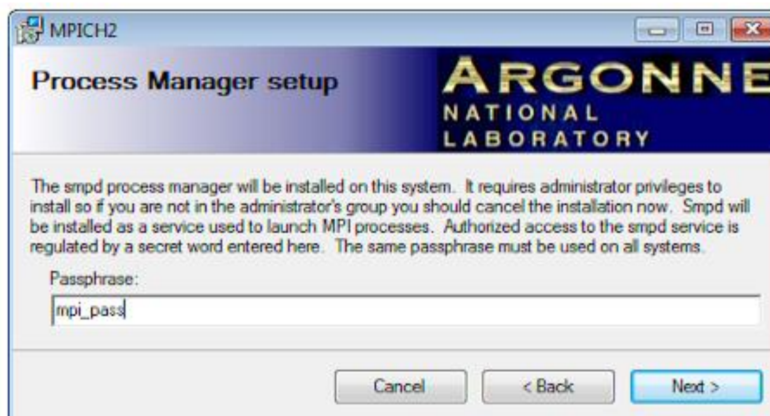


Рисунок 4. Вказівка пароля для доступу до менеджера процесів

4. У вікні вказівки шляху установки рекомендую залишити каталог за умовчанням. Крім того, поставите відмітку в пункті «Everyone» (Рисунок 5). Якщо Windows запитає, чи дозволити доступ в мережу програмі smpd.exe, то слід натиснути «Дозволити».

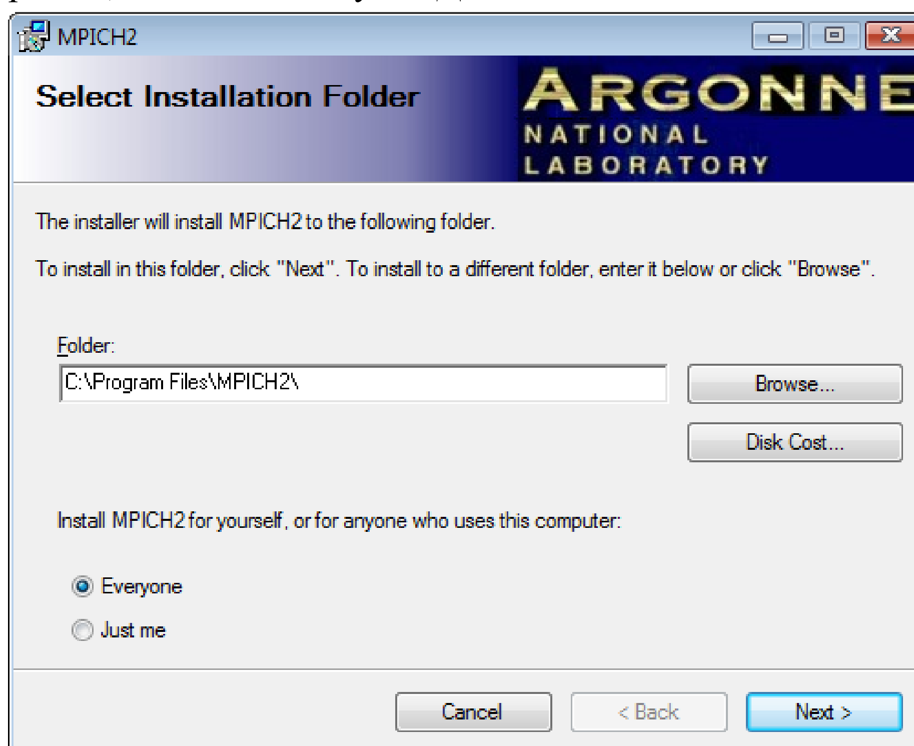


Рисунок 5. Вказівка шляху установки

Проте, перш ніж переходити до налаштування, обов'язково слід перевірити дві речі: чи запущена служба «MPICH 2 Process Manager», і чи

дозволений цій службі доступ в мережу. Для цього слід натиснути Пуск –> Налаштування –> Панель управління –> Адміністрування –> Служби. «MPICH2 Process Manager» повинен бути в списку служб (Рисунок 6). Ця служба повинна працювати. Якщо служба в списку відсутня, то запуск інсталятора не відбувся від імені адміністратора.

Microsoft .NET Framework NGEN v2.0.50727_X86	Microsoft .NET Framework NGEN		Вручную
Microsoft Office Diagnostics Service	Запуск центра диагностики Microsoft Office.		Вручную
MPICH2 Process Manager, Argonne National Lab	Process manager service for MPICH2 applications	Работает	Авто
NBService	Nero BackItUp Service is responsible to control a...		Вручную
NMIndexingService			Вручную

Рисунок 6. Служба «MPICH 2 Process Manager» в списку служб

Далі слід перевірити, чи дозволений доступ в мережу для MPICH. В меню Пуск –> Налаштування –> Панель управління –> Брандмауер Windows. слід натиснути «Дозвіл запуску програми через брандмауер Windows». В списку дозволених програм повинен бути «Process launcher for MPICH2 applications» і «Process manager service for MPICH2 applications» (Рисунок 7)

.Якщо якась з перерахованих програм відсутня в списку дозволених програм, то потрібно додати її вручну. Для цього слід натиснути кнопку «Додати програму.»., і додати C:\program files\mpich2\bin\mpiexec.exe, якщо відсутній «Process launcher for MPICH2 applications», і C:\program files\mpich2\bin\smpd.exe, якщо відсутній «Process manager service for MPICH2 applications».

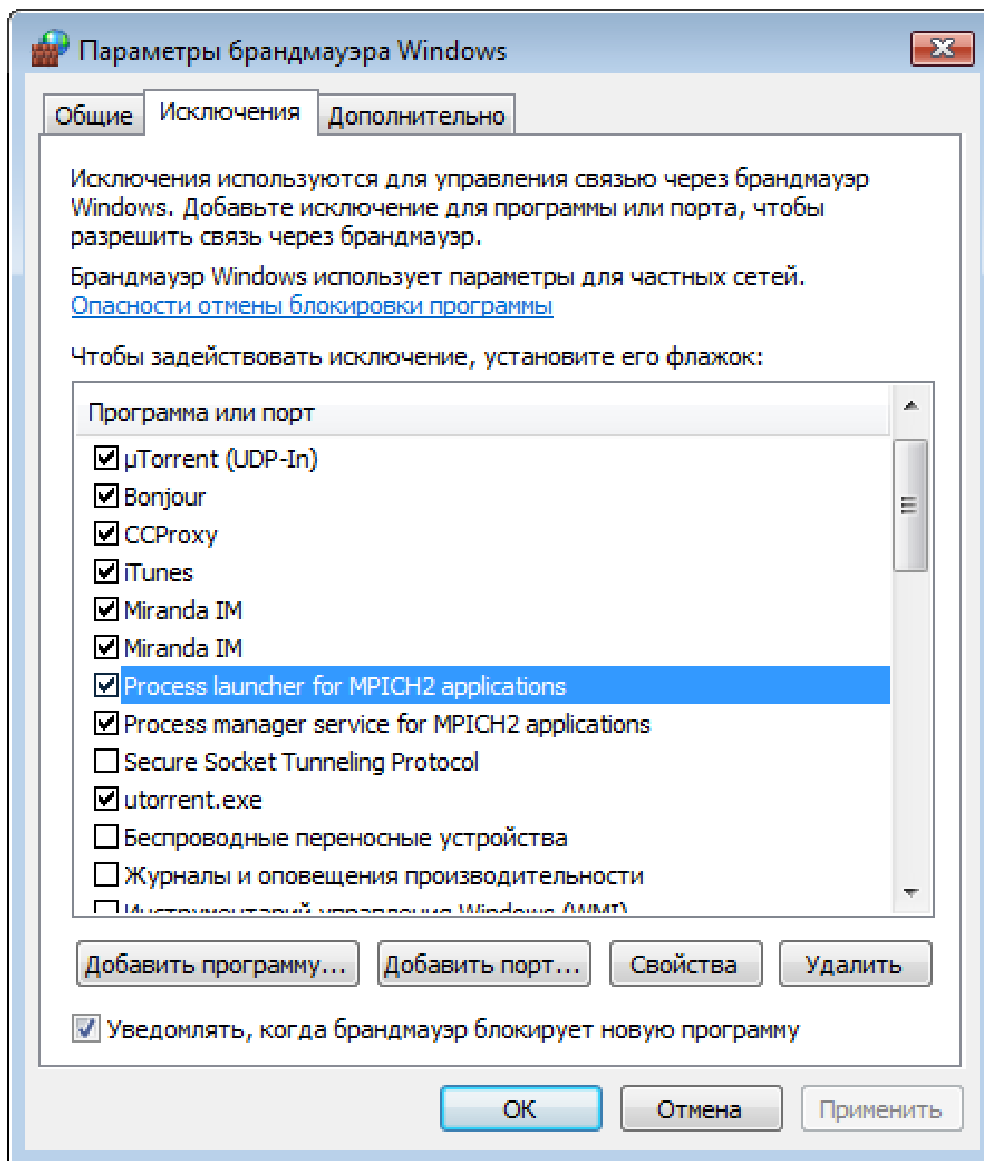


Рисунок 7. Програми MPICH в списку виключень брандмауера

Наступним етапом роботи є налаштування MPICH. Для цього потрібно виконати наступні пункти:

1) **Необхідно** створити на усіх комп'ютерах користувача з однаковим ім'ям і паролем**; від імені** цього користувача запускатимуться MPI - програми (якщо у вас один комп'ютер — цей крок можете пропустити). Найпростіше це зробити, встановивши однаковий пароль користувачам Адміністратор.

2) Щоб переконатися, що обліковий запис працює, і встановити на

нього пароль, потрібно зайти в систему з обліковим записом «Адміністратор», і встановити пароль з допомогою Пуск –> Налаштування –> Панель управління –> Облікові записи користувачів –> Зміна свого пароля.

3) Як вже було сказано раніше, будь-яку дію система MPICH виконує від вказаного імені користувача. Для того, щоб запитувати ім'я користувача і пароль, використовується програма Wmpiregister. Проблема в тому, що ім'я користувача і пароль запитується досить часто. Для того, щоб цього уникнути, Wmpiregister може зберігати ім'я користувача і пароль в реєстрі Windows. Необхідно запустити Wmpiregister на тому комп'ютері, з якого буде запускатися MPI-програма. Для цього слід натиснути Пуск –> Програми –> MPICH2 –> wmpiregister.exe. Вікно програми наведено на рисунку 8.

Зміст кнопок (справа-наліво) :

- «Cancel» — закрити програму без виконання будь-якої дії.
- «OK» — передати введені ім'я користувача і пароль програмі. Якщо Wmpiregister запущена як окреме застосування, то натиснення кнопки OK еквівалентне натисненню кнопки Cancel.
- «Remove» — натиснення цієї кнопки видаляє збережені раніше ім'я користувача і пароль з реєстру Windows.
- «Register» — зберігає ім'я користувача і пароль в реєстрі.

Введіть ім'я користувача і пароль у вікні програми і натисніть кнопку «Register». Повинен з'явитися напис «Password encrypted into the Registry» (Рисунок 8). Після цього вікно програми більше не з'являтиметься при роботі з MPICH. Якщо потрібно видалити ім'я користувача і пароль з реєстру, то потрібно знову запустити цю програму, і натиснути кнопку «Remove».

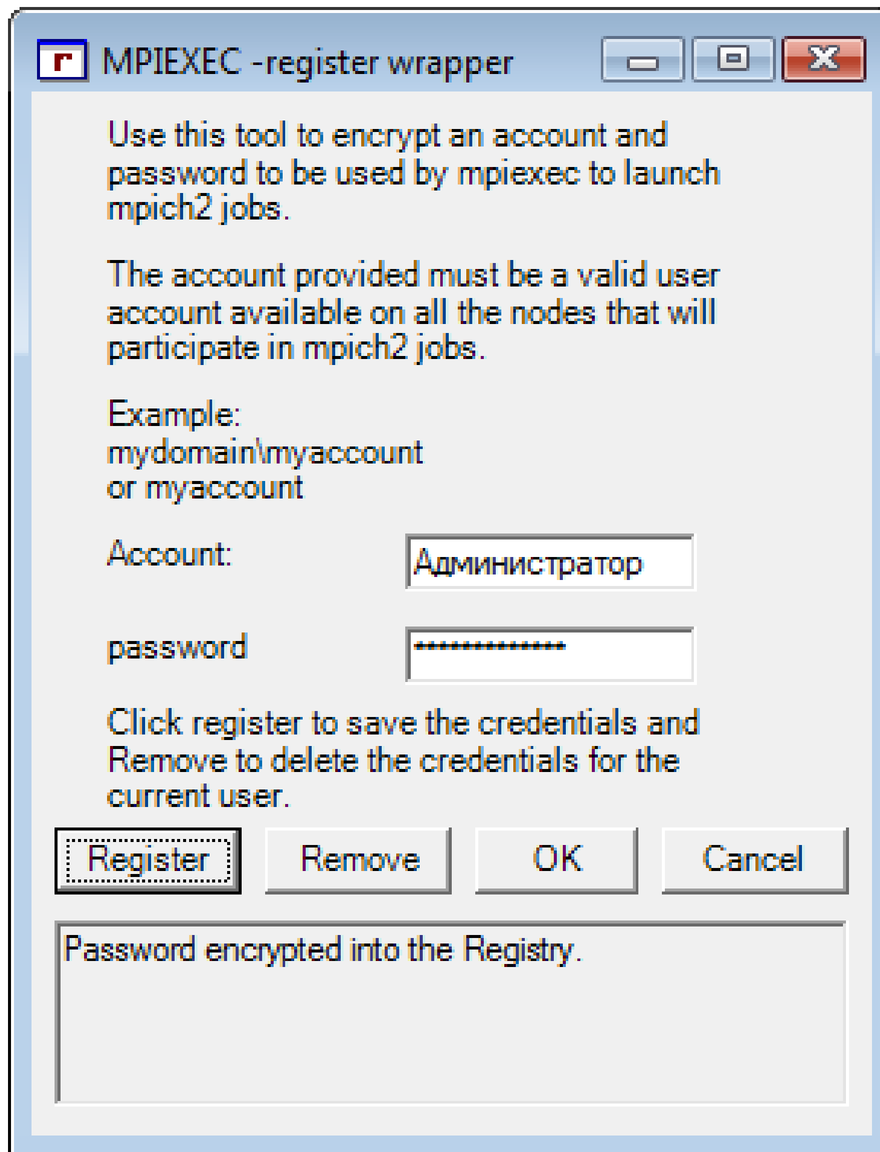


Рисунок 8. Програма Wmpiregister

4) Далі слід запустити на усіх комп'ютерах програму Wmpiconfig. Якщо усі попередні кроки зроблені правильно, то в полі «version» в лівій колонці таблиці повинна бути версія встановленого менеджера процесів (Рис.9). Якщо менеджер процесів не встановлений, або йому закритий доступ в мережу, то з'явиться повідомлення напис «MPICH 2 not installed or unable to query the host» в одному з полів лівого стовпця. В цьому випадку зверніться за допомогою до викладача.

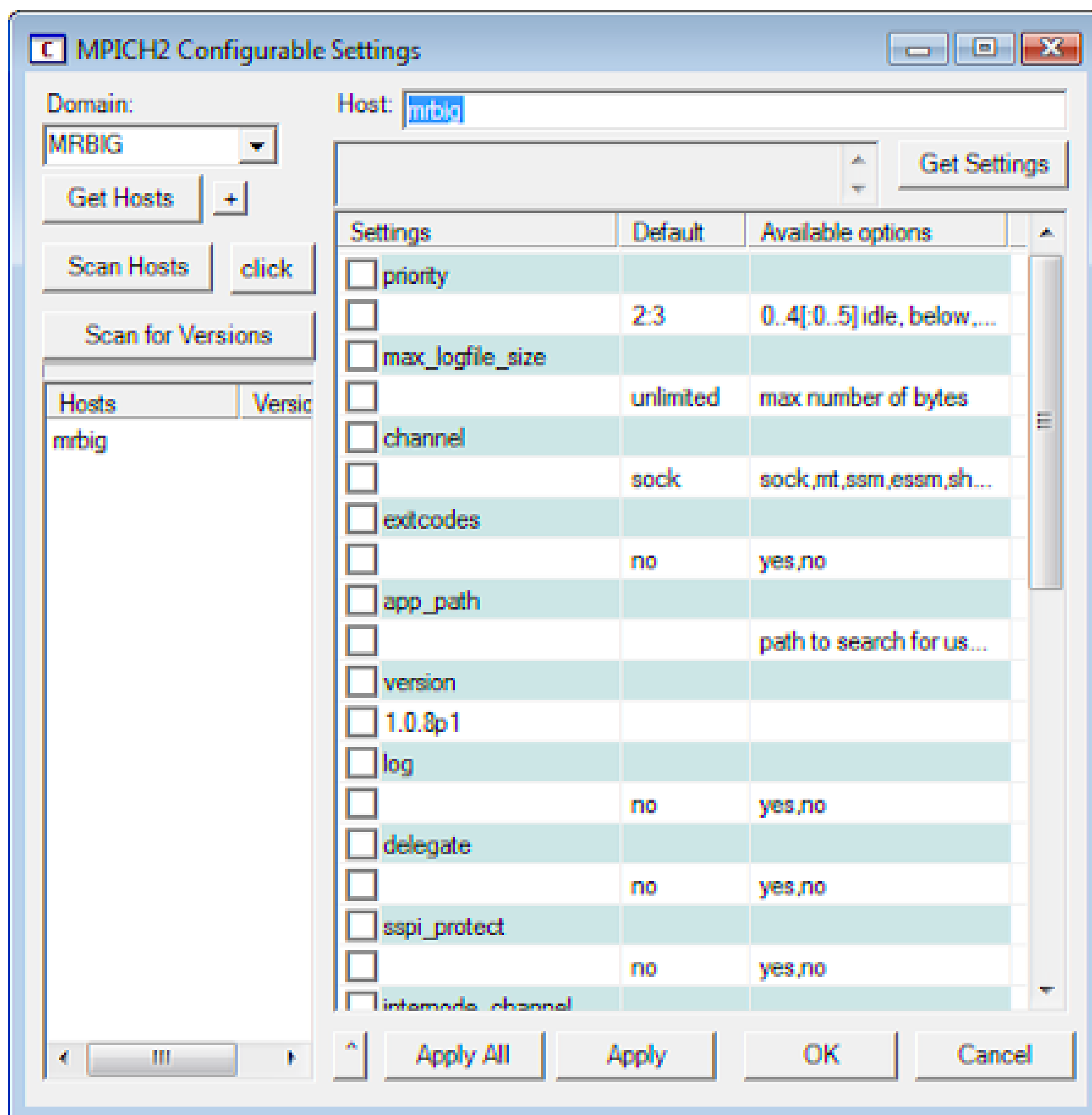


Рисунок 9. Програма Wmpiconfig

Програма Wmpiconfig призначена для налаштування менеджерів процесів на поточному комп'ютері і інших комп'ютерах мережі. Для цього вона під'єднується до менеджерів процесів на вибраних комп'ютерах, читає наявні у них налаштування, і повідомляє їм нові налаштування, якщо потрібно.

Елементи управління програми Wmpiconfig виконують наступні дії:

– Ліворуч-внизу є список комп'ютерів, з якими працює програма налаштування. Ім'я комп'ютера на білому фоні означає, що не було спроб зв'язатися з цим комп'ютером; зелений фон означає, що зв'язок

зроблений успішно; сірий фон означає, що при встановленні зв'язку виникла помилка.

- Видалити комп'ютер зі списку можна клавішею Del. Слід мати на увазі, що цей список призначений тільки для зручності налаштування, і не має ніякого відношення до списку комп'ютерів, на яких буде запущена MPI - програма.

- Кнопка «Get Hosts» отримує список комп'ютерів в заданому домені або робочій групі (задається у випадному списку «Domain»). Отриманий список замінює наявний список комп'ютерів або, якщо натиснута кнопочка «+», додає комп'ютери до поточного списку.

- Кнопка «Scan Hosts» отримує налаштування з усіх комп'ютерів списку; кнопка «Scan for Versions» отримує тільки номери версій.

- Кнопка «Get Settings» отримує поточні налаштування того комп'ютера, ім'я якого введене в поле введення «Host». При виборі комп'ютера в списку комп'ютерів його ім'я автоматично вводиться в поле «Host». Якщо натиснута кнопка «Click», то налаштування будуть отримані автоматично при виборі комп'ютера зі списку.

- Справа у вікні розташована таблиця налаштувань. Якщо ви хочете змінити які-небудь налаштування, то треба двічі клацнути на відповідному полі в першому стовпці таблиці. Порожнє поле означає, що використовується налаштування за умовчанням, вказана в другому стовпці. Налаштування, призначені до зміни, слід відмічати установкою опції ліворуч.

- Кнопка «Apply» застосовує виділені опцією налаштування до того комп'ютера, ім'я якого знаходиться в полі «Host». Кнопка «Apply All» застосовує налаштування до усіх комп'ютерів списку.

- Кнопка «Cancel» закриває програму. Наскільки я зрозумів,

дія кнопки «OK» нічим не відрізняється від дії кнопки «Cancel».

– Для успішного виконання потрібно, щоб імена комп'ютерів містили тільки латинські букви і цифри. Для того, щоб змінити ім'я, натисніть правою кнопкою миші на «Комп'ютер» –> Властивості –> Додаткові параметри системи –> Ім'я комп'ютера –> Змінити...

5) На тому комп'ютері, з якого планується запуск програм, треба вказати список доступних обчислювальних вузлів (якщо використовується тільки один комп'ютер — переходьте до пункту «запуск MPI — програм»). Цей список треба ввести (через пропуск) в поле hosts лівого стовпця таблиці (Рисунок 10), і натиснути кнопку «Apply».

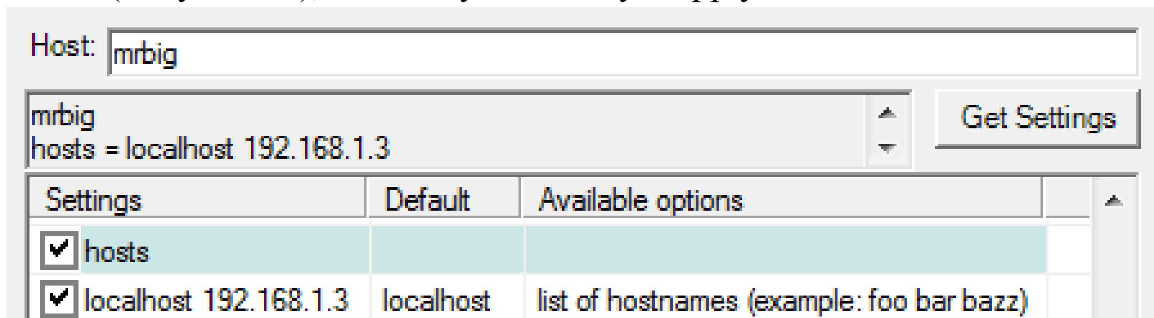


Рисунок 10. Вказуємо список доступних обчислювальних вузлів

6) Для зручного запуску MPI-програм слід створити на одному з комп'ютерів загальний мережевий ресурс. Якщо MPI-програми будуть запущені на одному комп'ютері, можна пропустити цей етап. Створіть папку (наприклад «Lab_Ivanov_MPI»), в яку ви викладатимете MPI - програми, натисніть на неї правою кнопкою миші, і виберіть «Властивості».

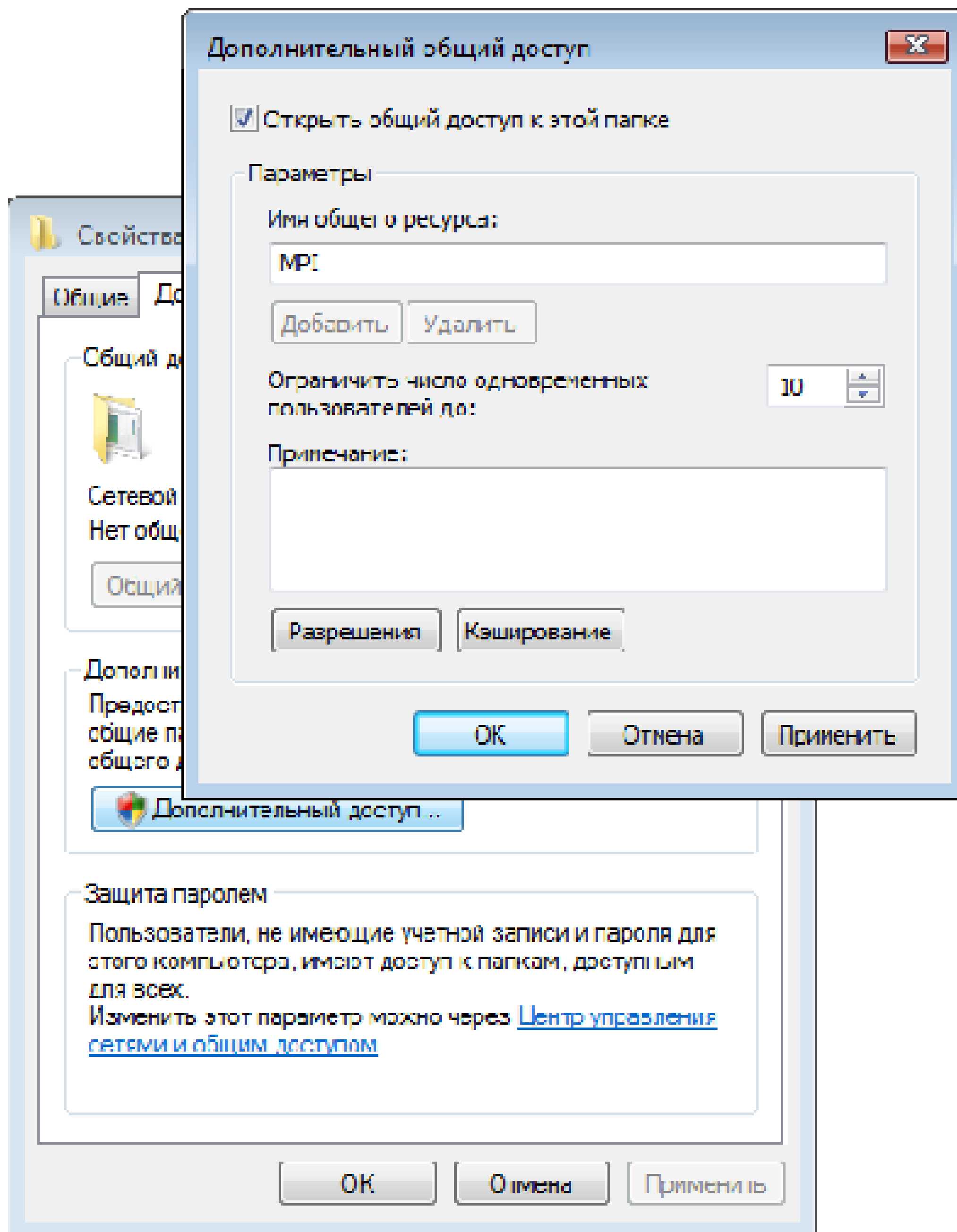


Рисунок 11. Вікно властивостей папки

7) У вікні, що з'явилося, виберіть вкладку «Доступ» (Рисунок 11), в якій натисніть кнопку «Додатковий доступ». У вікні, що з'явилося, «Додатковий загальний доступ» поставте опцію «Відкрити загальний доступ до цієї теки», і встановіть «число одночасних користувачів» таким, щоб воно перевищувало кількість комп'ютерів мережі, призначених для запуску MPI -програм.

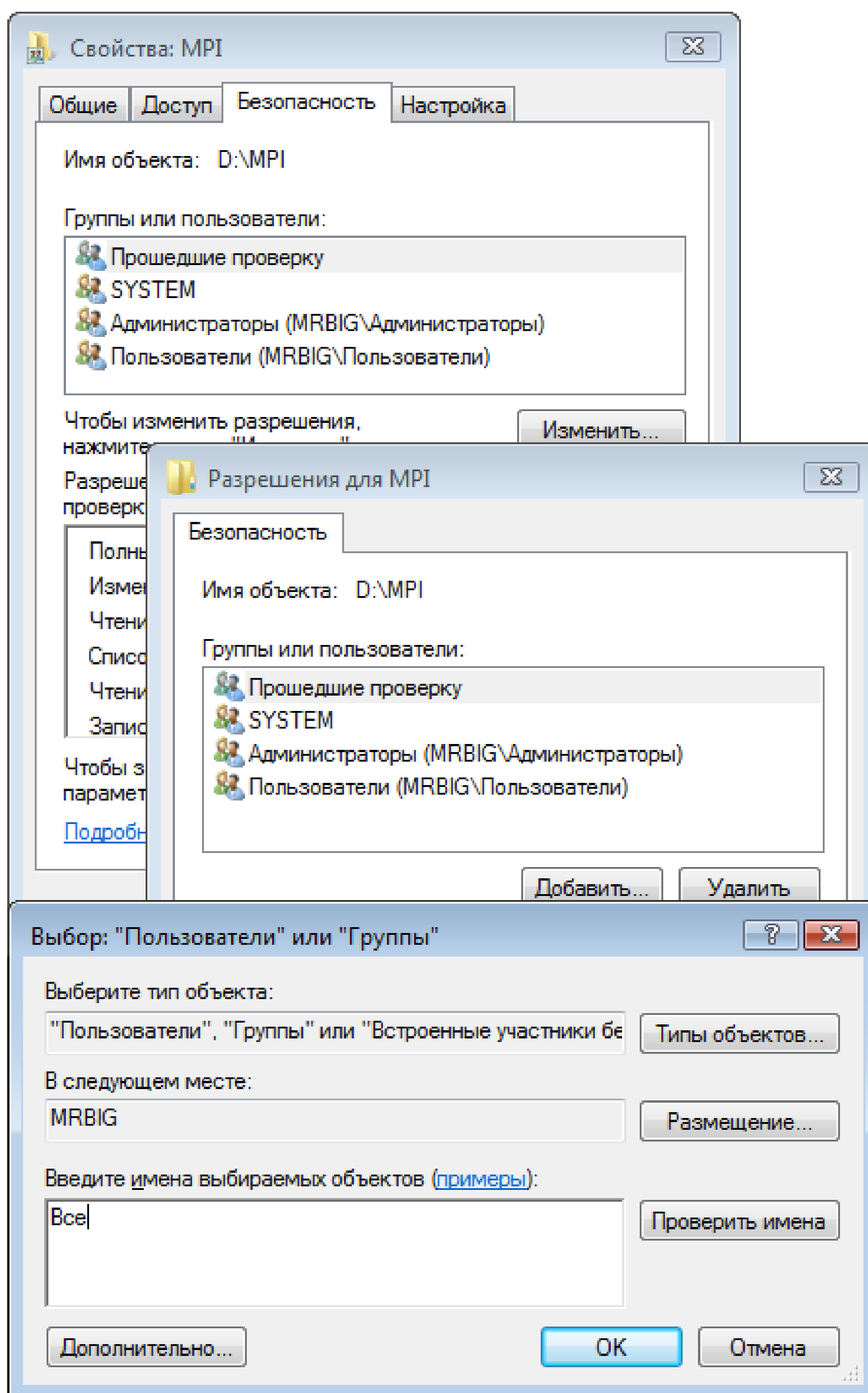


Рисунок 12. Додавання дозволів для доступу до теки

8. Натисніть «ОК» у вікні «Додатковий загальний доступ» і перейдіть у вкладку «Безпеку». Там натисніть кнопку «Змінити» (Рисунок 12), з'явиться вікно «Дозволу для [ім'я теки]». У цьому вікні натисніть кнопку «Додати.»., з'явиться вікно вибору об'єкту для додавання (користувача або групи). Введіть в поле введення «Усі» (чи «All», якщо у вас англійська версія Windows) .Натисніть кнопку «ОК» в двох вікнах. У вкладці «Безпека» у верхньому списку повинна додатися рядок «Все», при виборі якої в списку дозволів повинні стояти опції навпроти пунктів «Читання і виконання», «Список вмісту теки» і «Читання».

9. **Останнім** етапом є запуск MPI -програми. Для цього в комплект MPICH2 входить програма з графічним інтерфейсом Wmpiexec, яка є оболонкою навколо відповідної утиліти командного рядка Mpiexec. Вікно програми Wmpiexec показане на Рисунку 13 (зверніть увагу, що включений прапорець «more options»).

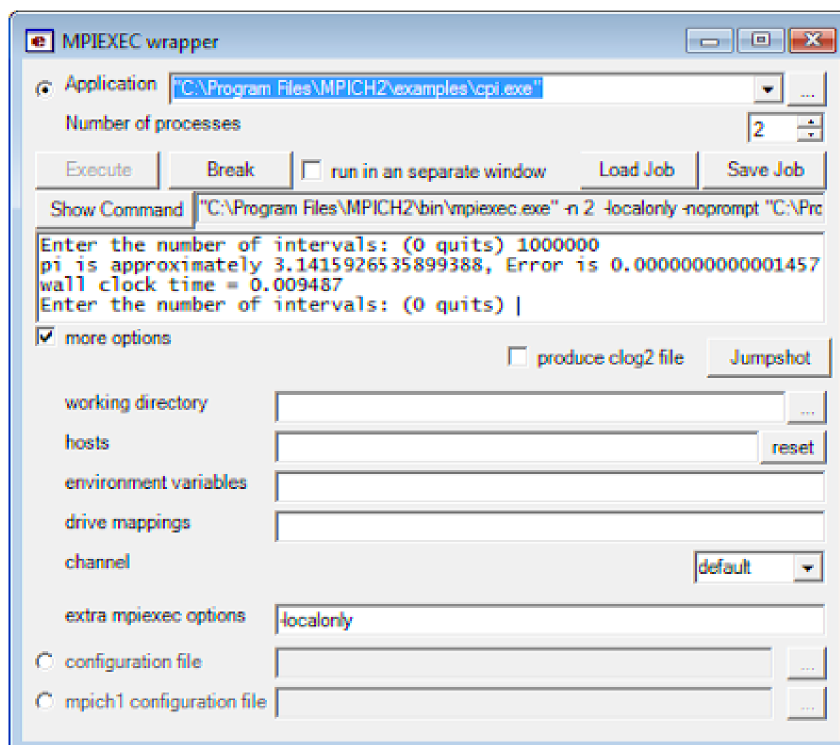


Рисунок 13. Програма Wmpiexec

Елементи управління вікна мають наступний зміст:

- Поле введення «Application» : сюди вводиться шлях до MPI - програми. Як вже було сказано раніше, шлях передається в незмінному вигляді на усі комп'ютери мережі, тому бажано, щоб програма розташовувалася в загальній мережевій папці.
- «Number of processes»: число процесів, що запускаються. По замовчуванню процеси розподіляються порівну між комп'ютерами мережі, проте цю поведінку можна змінити за допомогою конфігураційного файлу.
- Кнопка «Execute» запускає програму; кнопка «Break» примусово завершує усі запущені екземпляри.
- Прапорець «run in a separate window» перенаправляє виведення усіх екземплярів MPI -програми в окреме консольне вікно.
- Кнопка «Show Command» показує в полі справа командний рядок, який використовується для запуску MPI -програми (Wmpiexec — усього лише оболонка над Mpiexec). Командний рядок складається з усіх налаштувань, введених в інших полях вікна.
- Далі йде велике текстове поле, в яке потрапляє уведення-виведення усіх екземплярів MPI-програми, якщо не встановлений прапорець «run in a separate window».
- Прапорець «more options» показує додаткові параметри.
- «working directory»: сюди можна ввести робочий каталог програми. Знову ж таки, цей шлях має бути вірний на усіх обчислювальних вузлах. Якщо шлях не вказаний, то як робочий каталог використовуватиметься місце знаходження MPI -програми.
- «hosts»: тут можна вказати через пропуск список обчислювальних вузлів, використовуваних для запуску MPI -програми. Якщо це поле порожнє, то використовується список, що зберігається в налаштуваннях менеджера процесів поточного вузла (дивіться розділ

“Налаштування MPICH”).

– «environment variables»: в цьому полі можна вказати значення додаткових змінних оточення, що встановлюються на усіх вузлах на час запуску MPI - програми. Синтаксис наступний: імя1=значення1, імя2=значення2.

– «drive mappings»: тут можна вказати мережевий диск, що підключається на кожному обчислювальному вузлі на час роботи MPI - програми. Синтаксис: Z:\\winsrv\wdir.

– «channel»: дозволяє вибрати спосіб передання даних між екземплярами MPI -програми.

– «extra mpiexec options»: в це поле можна ввести додаткові ключі для командного рядка Mpiexec.

ДОДАТОК Б

Налаштування MPI -програми в Visual Studio

У Visual Studio 2010 налаштувань каталогів перенесені у *властивості проекту.* Це означає, що спочатку треба створити проект, а потім вже настроїти для нього каталоги у вікні Project –> Properties.

Передусім, треба налаштувати Visual Studio, щоб він знаходив заголовні файли і .lib-бібліотеки MPICH. Для цього запустіть Visual Studio і натисніть Tools–> Options, в дереві ліворуч виберіть Projects and Solutions – > VC++ Directories. Справа-вгорі виберіть Show directories for: Include files. Натисніть кнопку «Add» і додайте шлях до .h -файлів:

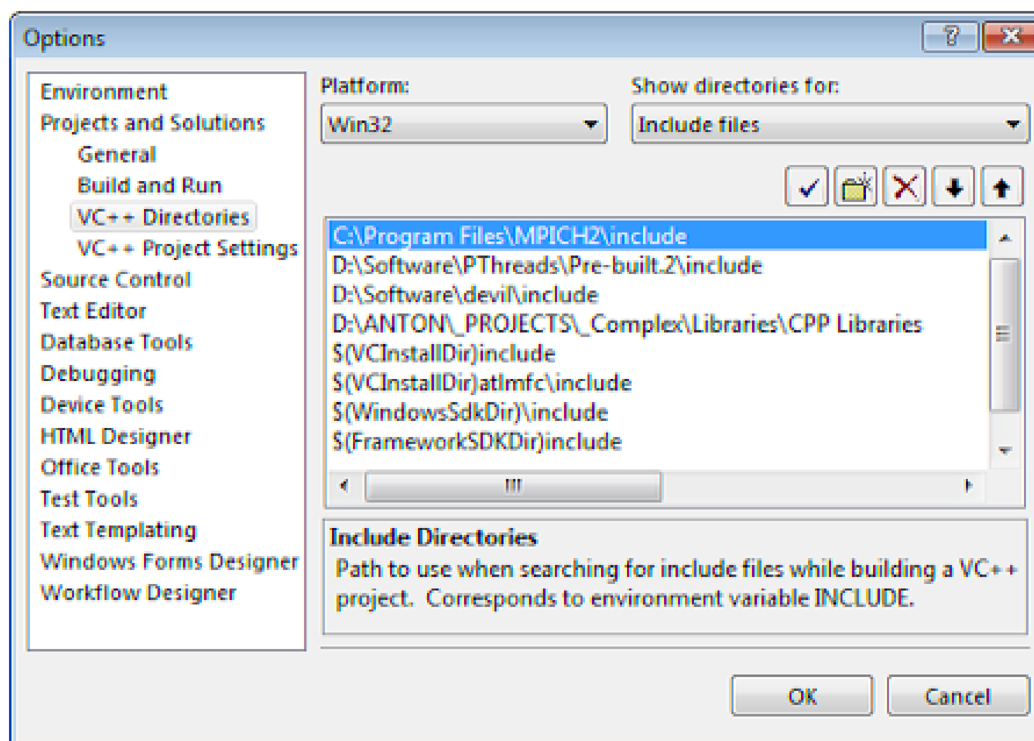


Рисунок 14. Налаштування шляху до заголовних файлів MPICH

Після цього слід виконати ту ж процедуру для бібліотек (Show directories for: Library files), рисунок 15.

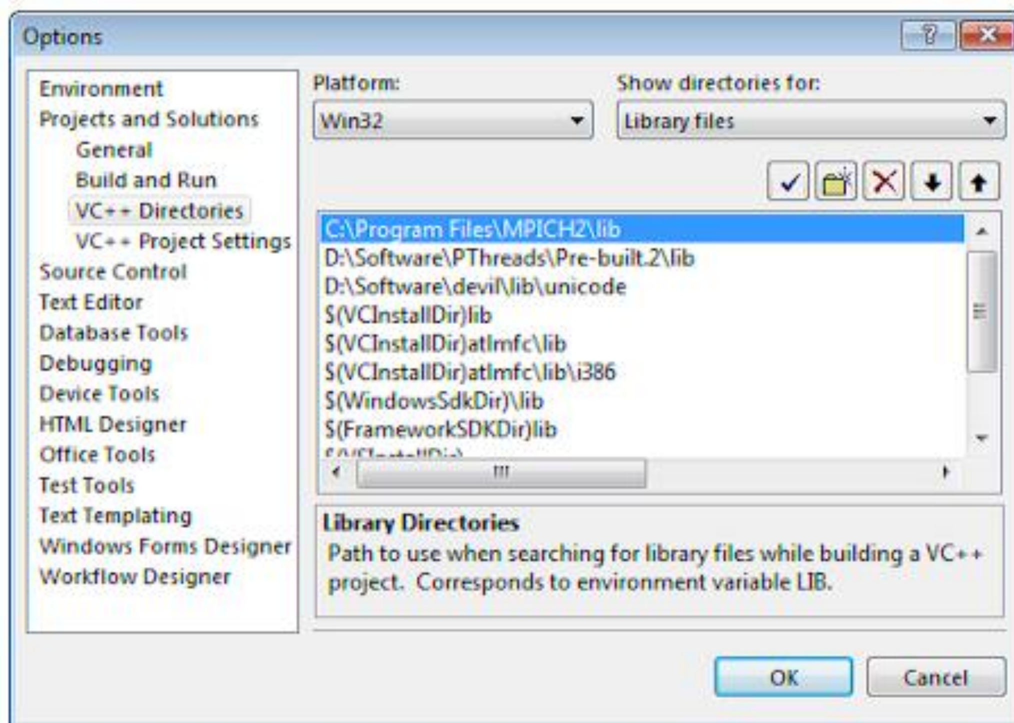


Рисунок 15. Налаштування шляху до бібліотечних файлів MPICH

Відкрийте вікно налаштувань проекту (Project–> Properties), виберіть Configuration: All Configurations, в дереві ліворуч виберіть Configuration справа.

Далі в полі Additional Dependencies, яке знаходиться в Properties –> Linker –> Input додайте mpi.lib. В список додаткових бібліотек для компонування окрім mpi.lib слід включати cxx.lib. Тому, якщо компонувальник (linker) видає помилку, спробуйте замість mpi.lib у вікні налаштувань, показаному на Рисунку 16, написати mpi.lib cxx.lib (через пропуск)

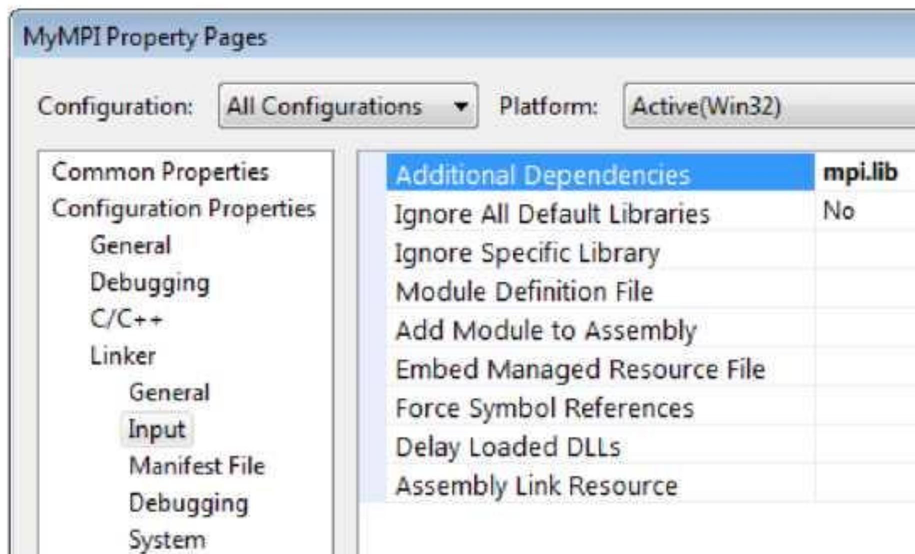


Рисунок 16. Додавання mpi.lib до програми

Тепер створіть консольний проект відповідно до рисунків 17,18.

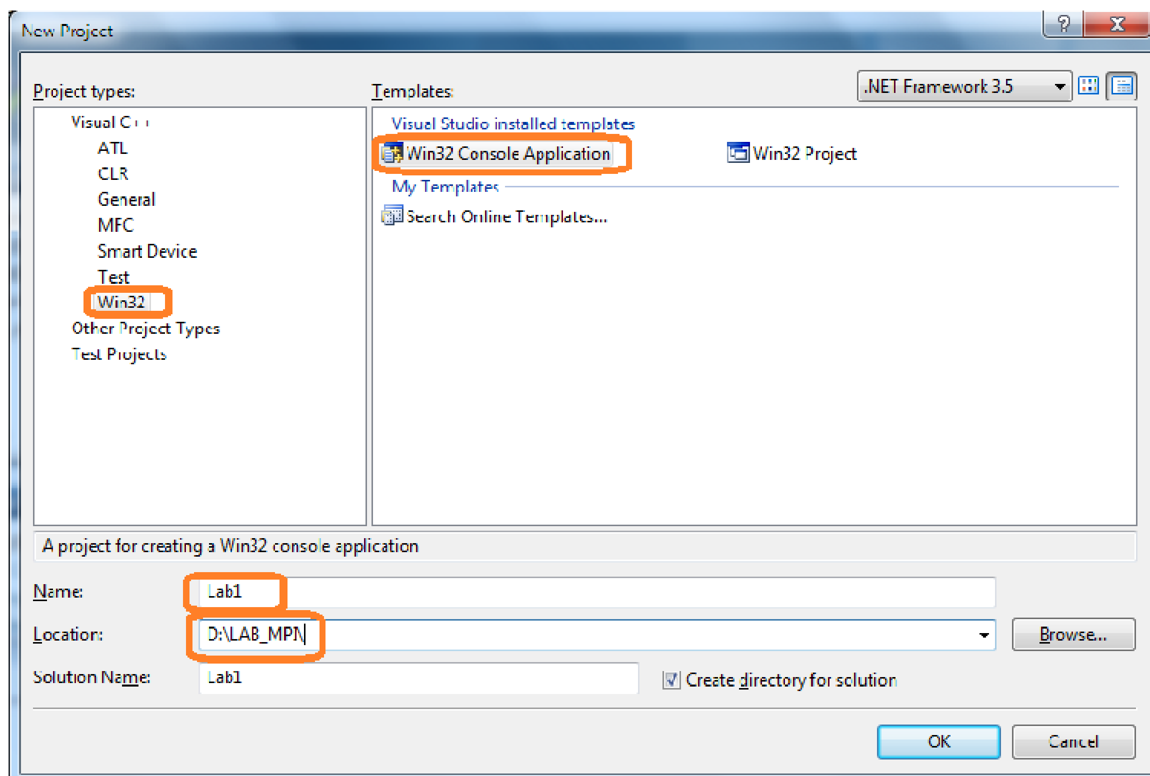


Рисунок 17. Створення нового проекту

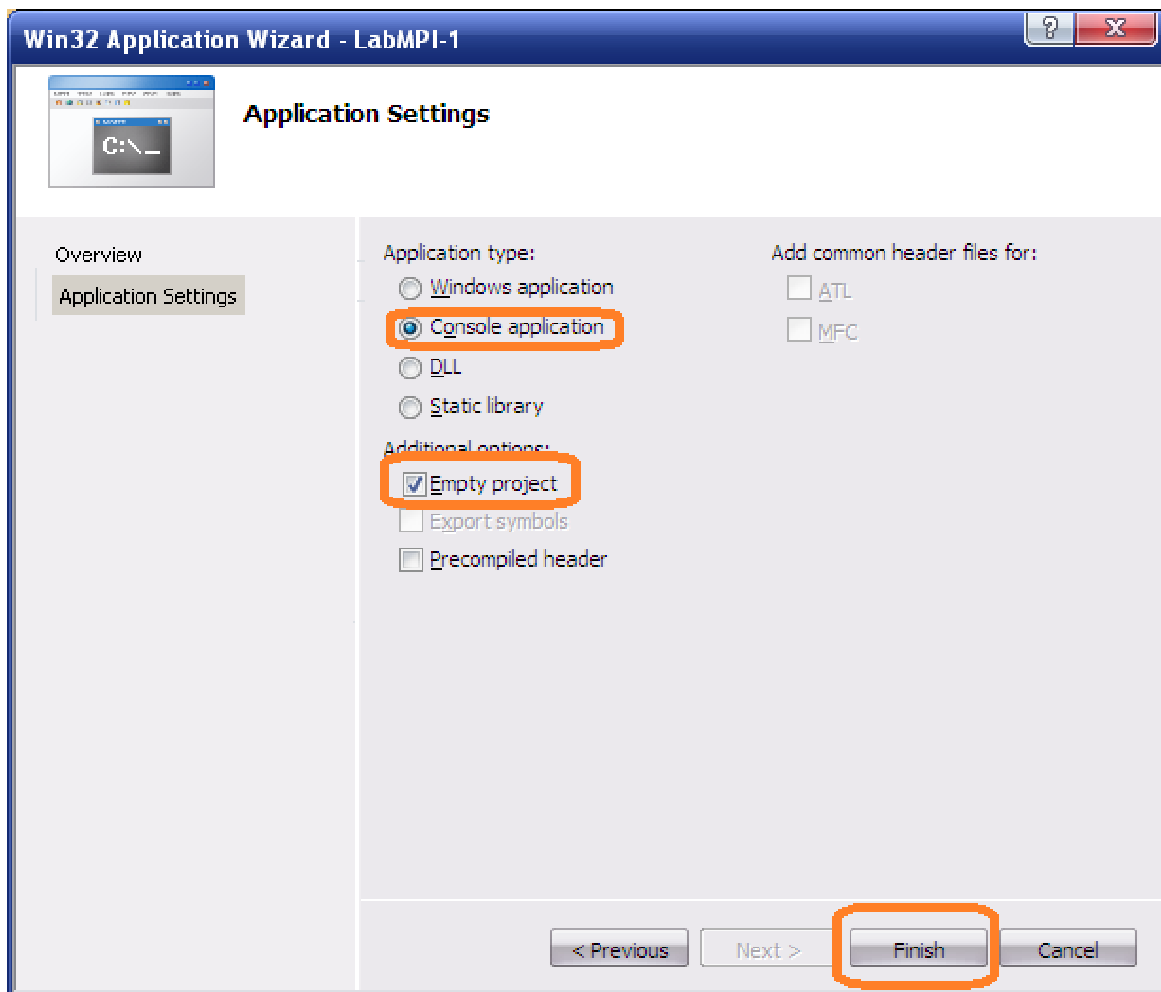


Рисунок 18. Задання властивостей створюваного проекту