Міністерство освіти і науки України
Національний технічний університет України "Київський політехнічний інститут"
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Курсова робота
з дисципліни "Паралельні та розподілені обчислення"

Керівник роботи:
Доц. Корочкін О. В.

Допущена до захисту

_____ «___» 2010 р.

Захищена

_____ «___» 2010 р.

Виконавець роботи:
ст. Грибенко Д. В.
ФІОТ гр. ІО-72

Київ — 2010

Технічне завдання

1. Огляд багатоядерних процесорів виробництва AMD.

2. Дана паралельна комп'ютерна система (рис. 1), що складається з P процесорів, двох пристроїв введення/виведення та спільної пам'яті. Для даної комп'ютерної системи розробити програмне забезпечення для обчислення виразу:

$$MA = \max(MO) \cdot (MR \cdot (MX \cdot MT)).$$

Вхідні та вихідні дані знаходяться на пристроях введення/виведення так, як показано на рисунку.

Мови для розробки програмного забезпечення: C++, Java.

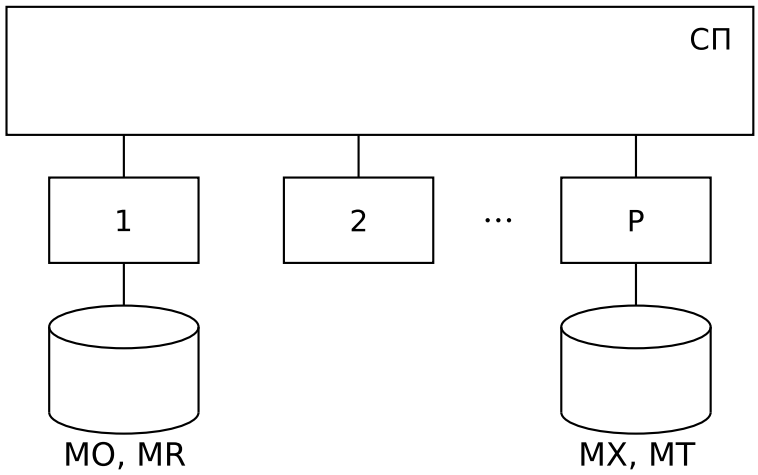

Рис. 1 — Паралельна обчислювальна система

Зміст

1 Огляд багатоядерних процесорів виробництва AMD

1.1 Багатоядерність як сучасна тенденція в розробці процесорів

Закон Мура прогнозує подвоєння кількості транзисторів на кристалі мікропроцесора кожні 2 роки. [1] Збільшення кількості транзисторів означає збільшення продуктивності. Але збільшення кількості транзисторів вимагає постійного вдосконалення технологічного процесу виробництва інтегральних схем. Останнім часом однією з проблем стало досягнення фундаментальних фізичних обмежень напівпровідникової технології. Зокрема, фізичні обмеження створюють проблеми тепловиділення та синхронізації даних. [2, 3] Ці проблеми в першу чергу пов'язані з максимальною швидкістю передачі сигналів через неідеальні лінії з ненульовою ємністю. Перезарядження цієї паразитної ємності і є основна причина тепловиділення в напівпровідникових інтегральних схемах, а час перезаряду — причина затримок передачі даних, більших, ніж в ідеальному випадку (коли єдиним обмеженням є швидкість світла). Мінімально можливий час передачі даних звичайно зумовлює максимально можливу тактову частоту мікропроцесора. Таким чином, можна зробити висновок, що необмежене зростання продуктивності за рахунок збільшення тактової частоти та інженерних рішень, продуктивність яких залежить від частоти, неможливе.

Інженери-розробники мікропроцесорів шукають різні шляхи обходу вищеназваних фізичних обмежень. Серед ранніх інженерних рішень варто виділити введення конвеєра та суперскалярних інструкцій. Але ці розробки самі по собі не можуть дати необмежений приріст продуктивності без одночасного збільшення тактової частоти мікропроцесора. Таким чином, вони можуть забезпечити збільшення продуктивності тільки в константну кількість разів, а провідним фактором все ж залишається тактова частота. З іншої сторони, ці розробки підходять не однаково гарно для різних програмних застосунків. Дійсно, якщо в програмному коді є багато переходів, напрямок яких важно передбачити, то переваги конвеєра зникають (мікропроцесор вимушений скидати конвеєр при виконанні неправильно передбаченого переходу). Якщо ж програмний код важко ефективно реалізувати з огляду на особливості суперскалярних архітектур, то і переваг суперскалярної архітектури застосунок отримати не зможе.

Сучасним підходом до вирішення згаданих проблем є створення багатоядерних мікропроцесорів. [4] Даний підхід передбачає розміщення декількох функціонально незалежних обчислювальних ядер разом з комунікаційним середовищем у спільному корпусі. [5] Окремі ядра є рівноцінними з точки зору програміста прикладних програм, що дозволяє проводити процес програмування так само, як і для класичних багатопроцесорних систем.

При цьому існує кілька варіантів реалізації багатоядерності: [5]

– Незалежні обчислювальні ядра з індивідуальною кеш-пам'яттю, розміщені на одному кристалі та під'єднані до системної шини.

– Незалежні обчислювальні ядра розміщені на різних кристалах.

– Обчислювальні ядра, що використовують спільні вузли, наприклад, кеш-пам'ять або системну шину.

Звичайно, такий підхід дозволяє забезпечити суб-лінійну масштабованість для застосунків, що розроблені з метою виконання на багатопроцесорних системах. Звідси випливає недолік багатопроцесорних систем: програмне забезпечення має бути розроблене з урахуванням їх особливостей. В той же час, оптимізації, які використовують паралелізм на рівні інструкцій (наприклад, вищезгаданий конвеєр), можна проводити повністю в автоматичному режимі, хоча і це вимагає наявності відповідного оптимізуючого компілятора. Наприклад, компілятор GCC може в автоматичному режимі оптимізувати код з використанням суперскалярних інструкцій, наявних у цільовому процесорі. [6, 7]

Порівняно зі звичайними багатопроцесорними системами, багатоядерні коштують в середньому менше. Це пов'язано з тим, що встановлення на системну плату додаткових процесорних роз'ємів та реалізація комунікаційного середовища суттєво підвищують її ціну.

В 1999 році IBM анонсувала перший в світі двоядерний процесор загального призначення — IBM Power4. IBM Power4 — 64-бітний процесор з набором команд PowerPC. [8] Так як Power4 призначався в першу чергу для високопродуктивних серверів, то IBM виробляла також багатопроцесорні модулі з кількома двоядерними процесорами Power4.

Важливо відмітити, що на сьогоднішній день багатоядерність використовується не тільки при розробці процесорів для настільних комп'ютерів та серверів [5], а також для управляючих мікроконтролерів. Одним з комерційно успішних багатоядерних процесорів, призначеного не для високопродуктивних обчислень, а в першу чергу для вирішення задач управління, є Parallax Propeller [9] — 8-ядерний 32-бітний RISC процесор.

1.2 Загальна характеристика багатоядерних процесорів AMD

До випуску багатоядерних процесорів AMD вже розробила декілька модельних рядів процесорів, призначених для мобільних та настільних комп'ютерів, а також для серверів. Перший двоядерний процесор був випущений в модельному ряді серверних процесорів Opteron в 2005 році. [10] Через деякий час ці розробки були перенесені і на модельний ряд для настільних комп'ютерів — Athlon 64 X2.

У багатоядерних процесорів AMD, призначених як для серверів, так і для настільних комп'ютерів, можна виділити наступні спільні риси:

– 64-бітна архітектура AMD64;

– підтримка виконання 32-бітних програм для архітектури x86 без емуляції;

– індивідуальна кеш-пам'ять другого рівня;

– спільна кеш-пам'ять третього рівня;

– використання архітектури Direct Connect на основі шини HyperTransport;

– інтегрований контролер пам'яті DDR або DDR2 або DDR3;

– в багатокристальних мікропроцесорах — NUMA архітектура з забезпеченням когерентності кеш-пам'яті (ccNUMA).

В наступних розділах ці особливості розглянуті детально.

1.3 Архітектура AMD64

Архітектура AMD64 — розроблена компанією AMD 64-бітна архітектура з забезпеченням сумісності з 32-бітною архітектурою x86 без емуляції. Так як при розробці одним із основних завдань біло саме забезпечення сумісності з x86, то був обраний не революційний, а еволюційний шлях розвитку. Фактично AMD64 стала набором розширень до x86, хоча деякі інструкції та режими x86 не доступні в 64-бітному режимі AMD64.

Першим процесором з архітектурою AMD64 став Opteron. [11]

Основні особливості архітектури AMD64, що вирізняють її від x86, перераховані нижче. [12, 13, 14]

– Розширення адресного простору до 64 біт. Всі вказівники мають довжину 64 біти. Реально існуючі реалізації AMD64 мають обмеження адресного простору до 52 біт.

– Розширення максимального можливого об'єму оперативної пам'яті до 2^{52} байт (по специфікації), а реально існуючі реалізації AMD64 мають обмеження 2^{48} байт.

– Розширення регістрів загального призначення до 64 біт. Для всіх команд в x86, що працювали з 32-бітними операндами, введені 64-бітні аналоги. Таким чином, є можливість виконувати логічні та арифметичні операції (включаючи множення), а також пересилки в пам'ять 8-байтних операндів. Робота зі стеком також ведеться блоками по 8 байт.

– Додаткові 64-бітні регістри r8, r9, . . . , r15. Таким чином, кількість регістрів загального призначення в AMD64 (16) порівняно з x86 (8) збільшилась в 2 рази. Це дозволяє зберігати більше операндів в регістрах, а не в стеку, тим самим зменшуючи кількість звернень до пам'яті. Також це дозволяє передавати аргументи підпрограмам в регістрах (в x86 після передачі аргументів в регістрах залишалось замало робочих регістрів для тимчасових даних підпрограми).

– Додаткові 8 регістрів SSE. Таким чином, в AMD64 всього 16 SSE регістрів.

– Новий режим адресації — адресація відносно вказівника команд. Це дозволяє більш ефективно реалізовувати код, що не залежить від адреси завантаження в пам'ять (такий код використовується в динамічно завантажуваних бібліотеках підпрограм щоб відображати одні і ті самі сторінки фізичної пам'яті по різним логічним адресам в адресні простори процесів і тим самим економити пам'ять).

– Команди SSE та SSE2 є обов'язковою частиною AMD64.

– Біт NX в атрибутах сторінок. Якщо біт NX встановлений, то сторінка не може містити програмного коду. Це ускладнює можливість виконання довільного коду при атаках на уразливості типу «переповнення буферу».

– Деякі режими та команди x86 не підтримуються в 64-бітному режимі. Але ці команди реалізовані в повному об'ємі в 32-бітному режимі сумісності, що дозволяє виконувати на процесорах AMD64 старі операційні системи та програми без змін.

1.4 Ієрархія кеш-пам'яті в багатоядерних процесорах AMD

Кеш — невелика за розміром пам'ять, що знаходиться на кристалі процесора і дозволяє скоротити простій процесора через доступ до пам'яті. В сучасних системах доступ до оперативної пам'яті може займати від 250 до 400 тактів. [15] Таким чином, кеш-пам'ять є обов'язковою частиною сучасних високопродуктивних процесорів.

Ідея кеш-пам'яті базується на принципах часової локальності та просторової локальності. [12]

– Відповідно до принципу **часової локальності**, дані з високою імовірністю використовуються в процесі обчислень декілька разів протягом невеликого проміжку часу.

– Відповідно до принципу **просторової локальності**, якщо в процесі обчислень використовуються деякі дані, то з високою імовірністю протягом невеликого проміжку часу знадобляться й дані, що знаходяться поряд в пам'яті.

Класифікація багаторівневих кешів проводиться за алгоритмами додавання та видалення даних з кешу. [5, 16]

– **Строго включаючими**, в цьому випадку вміст кешу більш низького рівня має знаходитись також і у всіх кешах більш високого рівня. Перевагою такого виду кешу є простота алгоритму видалення з кешу. Іншою перевагою є те, що кеші різних рівнів можуть мати різну довжину рядка (звичайно, чим вище рівень кешу, тим більше його об'єм і доцільно використовувати рядки більшого розміру). Недоліком такого підходу є зменшення корисного об'єму кешу високих рівнів. Цей фактор особливо важливий для

багатоядерних процесорів, так як багато ядер можуть мати локальні кеші низьких рівнів та спільні кеші високих рівнів. У цьому випадку строго включаючий кеш високого рівня включав би всі кеші всіх ядер, що значно знижувало б його корисний об'єм.

– **Невключаючими**, в цьому випадку деякий блок даних може знаходитись лише у кеші деякого одного рівня. Перевагою такого способу збереження даних в кеші є те, що корисний розмір кешу рівний його фактичному розміру. Недоліком є більш складні алгоритми роботи з кешем, порівняно зі строго включаючими кешами. Дійсно, витіснення рядка з кешу деякого рівня на рівень нижче може спричинити витіснення рядка і з нижчого рівня, і так по ланцюгу до найнижчого рівня. При попаданні в кеш низького рівня здійснюється обмін рядків між цим кешем та кешем першого рівня. Очевидно, обидва алгоритми є більш складними. Другим недоліком є те, що всі кеші повинні мати рядки однакової довжини (для того, щоб міняти місцями рядки в кешах різного рівня при попаданні).

– **В основному включаючими**, в цьому випадку дані з кешів низьких рівнів можуть і не знаходитись у кешах вищих рівнів, але з-за алгоритмів роботи з кешами, більшість даних з кешів низьких рівнів знаходиться у кешах вищих рівнів.

Строго включаючі та в основному включаючі кеші використовуються сучасними процесорами Intel [17]. В процесорах AMD використовуються невключаючі кеші. [12]

В багатоядерних процесорах кеш кожного рівня може бути спільним чи індивідуальним.

Спільний кеш в багатоядерному процесорі — такий кеш, що розміщений на одній інтегральній схемі з ядрами і може використовуватись рівноправно усіма ядрами. Недолік спільного кешу в тому, що необхідно розробити алгоритм, за яким буде можливий одночасний доступ кількох ядер до кешу.

Індивідуальний кеш в багатоядерному процесорі — такий кеш, що доступний лише одному ядру. Порівняно зі спільним кешем, недолік індивідуального кешу в меншому розмірі (при тій же самій кількості транзисторів).

Кеш першого рівня у процесорах Intel та AMD є індивідуальним. [17, 12] Кеш другого рівня в процесорах Intel є спільним, а в процесорах AMD — індивідуальним. Кеш третього рівня спільний в процесорах обох виробників.

Не зважаючи на той факт, що x86 процесори мають фон-неймановську архітектуру, кеш першого рівня в процесорах Intel та AMD розділений на кеш для даних та кеш для команд, тобто має гарвардську архітектуру. Це інженерне рішення було прийнято доволі давно і базувалось на евристичних спостереженнях, але на сьогоднішній день перевірено часом (наприклад, Intel вперше застосувала розділення кешу першого рівня в процесорі Pentium, випущеному в 1993 році, і з тих пір всі процесори Intel мають розділений кеш першого рівня). [17]

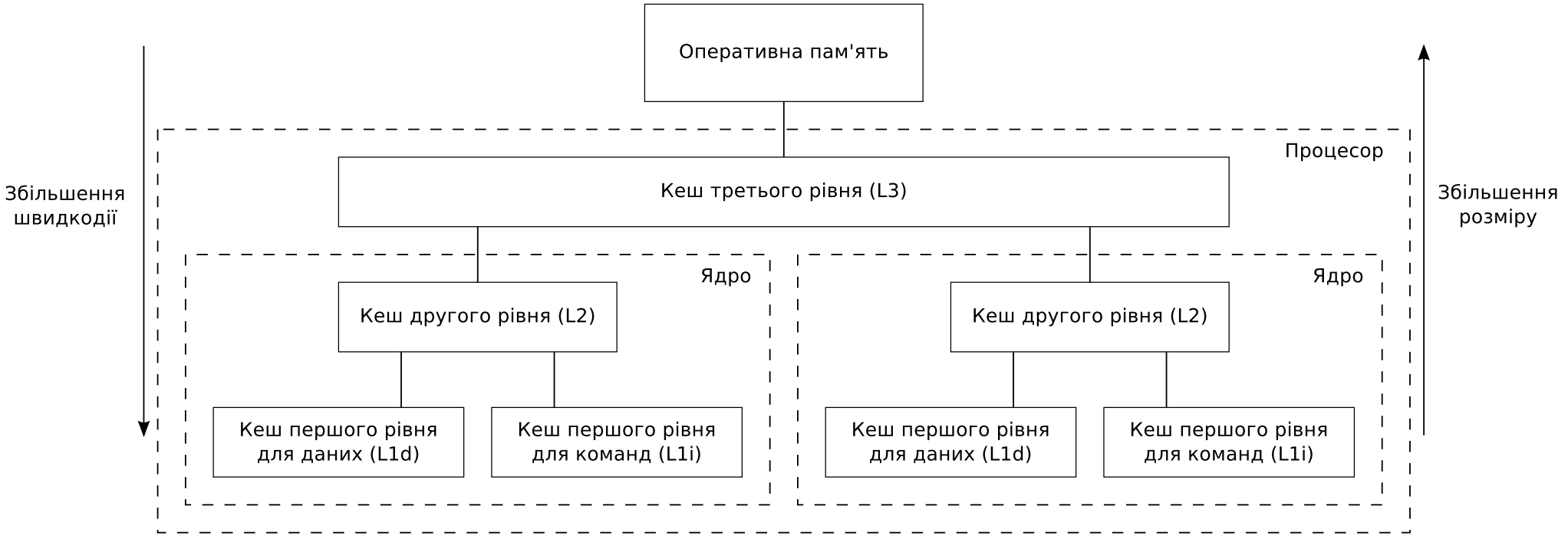

Рис. 1.1 – Ієрархія пам’яті в багатоядерних процесорах AMD

На рисунку 1.1 показана ієрархія пам'яті в сучасних багатоядерних процесорах AMD. Показані три рівня кешів, індивідуальні кеші першого та другого рівня для кожного ядра, та спільний кеш третього рівня для всіх ядер на інтегральній схемі.

1.5 Порівняльна характеристика мікроархітектур K8 та K10

На мікроархітектурі K10 базуються сучасні багатоядерні мікропроцесори AMD з модельних рядів Opteron, Athlon64 X2, Phenom, Phenom II. Розглянемо її характерні особливості та порівняємо з попередньою мікроархітектурою K8, яка була основою для процесорів модельних рядів Athlon64, Athlon64 X2, Opteron.

1.5.1 Вибірка команд

Процесор починає виконання коду завантажуючи команди з кешу першого рівня для команд. Після цього починається декодування команд. Команди x86 та AMD64 мають не фіксовану довжину, тому важко визначити їх границі не декодуючи потік команд послідовно. Для того, щоб прискорити декодування команд через паралельне декодування та виконання різних команд на різних етапах конвеєру, необхідно відмітити в потоці команд початки кожної команди. Процесори AMD з мікроархітектурою K8 та K10 забезпечують таке маркування, виконуючи його в процесі завантаження команд в кеш першого рівня. Маркування зберігається в спеціальних полях кешу, для цього відведено три додаткових біти на кожний байт команд. Таким чином, попереднє декодування з метою визначення границь команд дозволяє забезпечити стабільну інтенсивність декодування незалежно від форматів та довжин команд. [13, 18]

Процесор завантажує команди з пам'яті в кеш блоками (довжина блоку залежить від типу пам'яті, типова довжина — 64 біти). З кешу команди також завантажуються блоками в обчислювальні вузли процесору, а потім з блоку вибираються байти, що разом становлять необхідну в даний час команду. Процесор з мікроархітектурою K10 завантажує команди з кешу блоками довжиною 32 байти, а процесор з мікроархітектурою K8 (а також процесори Intel Core 2) — блоками по 16 байт. Розрахунки показують, що при середній довжині команди в 5 байт, завантаження 16 байт команд під час кожного такту дозволяє декодувати три команди кожний такт. Але, деякі команди x86 можуть мати довжину 16 байт, і в деяких доволі імовірних випадках три послідовні команди мають довжину більше 5 байт кожна. Це не дозволяє дати гарантію декодування трьох команд кожен такт. Тому в K10 вибірка команд з кешу відбувається 32-бітними блоками. [16, 18]

1.5.2 Передбачення переходів

Всі сучасні процесори виконують команди за принципом конвеєру, тобто, одночасно виконуються кілька команд, але кожна на своєму етапі обробки. Якщо в потоці команд зу-

стрічається команда умовного переходу, на першому ж етапі виконання необхідно обрати деякий напрям переходу для того, щоб вибрати наступну команду. Якщо обраний напрям був неправильним — конвеєр скидається починаючи з неправильно передбаченого переходу.

Продуктивність роботи конвеєра з командами умовних переходів повністю визначається обраним алгоритмом передбачення переходів. В мікроархітектурі K8 використовується дворівневий адаптивний алгоритм. Цей алгоритм приймає рішення базуючись не тільки на історії напрямів даного конкретного переходу, але й напрямів переходів серед 8 попередніх команд. Найбільшим недоліком мікроархітектури K8 є те, що її алгоритм не може передбачити переходи з адресою, заданою у вигляді операнда з непрямою адресацією. [18]

Команди переходу з непрямо заданою адресою здійснюють перехід за адресою, яку неможливо обчислити статичним аналізом коду. Такі команди звичайно використовуються компілятором при генерації кодів для операторів switch-case. Вони також використовуються для виклику віртуальних функцій в об'єктно-орієнтованих мовах програмування. Процесор з мікроархітектурою K8 завжди передбачав перехід за тією ж самою адресою, за якою був здійснений перехід в останній раз. Якщо адреса змінювалась, та перехід був передбачений невірно, конвеєр команд скидався. Якщо адреси переходів змінювались постійно та приймали навіть два різні значення, процесор передбачав перехід завжди невірно, що приводило о великих втрат швидкодії.

В мікроархітектурі K10 алгоритм передбачення переходів був покращений наступним чином: [18]

1. Реалізована підтримка переходів за непрямо заданою адресою. Для цього використовується таблиця з 512 елементами.

2. Регістр історії був розширений з 8 до 12 біт. Цей регістр зберігає напрями переходів для попередніх команд та дозволяє передбачати напрямок переходу базуючись на історії виконання попередніх команд переходу.

3. Стек адрес повернення збільшений з 12 до 24 слів. Цей стек використовується для швидкого визначення адреси повернення функції від час виконання команди ret на конвеєрі. Адреса повернення фактично є адресою переходу.

Вищеназвані вдосконалення мають збільшити швидкодію під час виконання коду на об'єктно-орієнтованих мовах програмування за рахунок зменшення простою через скидання конвеєру. Але об'єктивно оцінити приріст швидкодії неможливо через відсутність адекватної методики тестування.

1.5.3 Побічний оптимізатор стеку

Команди роботи в архітектурі x86 зі стеком створюють перепони для переставлення команд місцями та виконання кількох команд у конвеєрі. До команд, що працюють зі стеком

відносяться не тільки push та pop, а й call та ret. Ці команди зустрічаються доволі часто у потоці команд та використовуються для передачі параметрів функцій, виклику функцій, повернення з функцій, збереження значень регістрів. Проблема полягає в тому, що всі команди роботи зі стеком зчитують та записують регістр вказівника стеку esp, а тому вони є залежними одна від одної. Це не дозволяє аналізатору загального призначення знайти спосіб переставити ці команди місцями у випадках коли таке переставлення могло б покращити швидкодію.

В мікроархітектурі K10 для вирішення цієї проблеми введений спеціальний блок — побічний оптимізатор стеку.

Побічний оптимізатор стеку розкладає складні команди роботи зі стеком на більш прості мікрооперації, що виконуються процесором. Причому модифікація вказівника стеку esp не виконується до тих пір, доки програмний код не звертається до нього безпосередньо. Для того, щоб забезпечити коректну роботу зі стеком, побічний оптимізатор стеку обчислює зміщення між реальним значенням esp та його правильним значенням та коректує команди роботи за стеком на це зміщення. Таким чином досягається незалежність команд роботи зі стеком одна від одної, що приводить до збільшення швидкодії, та незмінність роботи процесора з точки зору прикладного програміста. [18]

1.5.4 Управління кеш-пам'яттю

Кеш-пам'ять має обмежений, причому доволі невеликий, розмір і тому має використовуватись якомога ефективніше. Алгоритми заповнення кеш-пам'яті базуються на вищезгаданих принципах часової та просторової локальності, але ці принципи є тільки евристичними спостереженнями і для деякого коду вони можуть не виконуватись і тим самим знижувати швидкодію через так зване забруднення кешу.

Забруднення кешу — заповнення кешу даними, які більше не знадобляться виконуваному коду в найближчий час.

Існують способи боротьби з забрудненням кешу за допомогою змін структур даних, але вони є прийнятними не у всіх випадках.

Крім того, не завжди евристичні алгоритми процесору можуть передбачити, які саме дані знадобляться в програмі в найближчий час. Якщо програмі відомі адреси цих даних, вона може «замовити» завантаження цих даних в кеш заздалегідь. Важливо розміщувати команди завантаження в кеш заздалегідь так, щоб підсистема пам'яті встигла обробити запит до того моменту, коли дані дійсно знадобляться програмі.

Команди завантаження в кеш необхідні як окремий клас команд та не можуть бути імітовані звичайними командами пересилки даних. По-перше, команди пересилки даних можуть заблокувати конвеєр команд до того часу, поки дані не будуть доставлені з пам'яті. По-друге, команди пересилки з пам'яті потребують використання регістра, яких в архітектурі AMD64 доволі небагато.

Для програмного контролю над вмістом кешу в процесорах AMD передбачені спеціальні команди: [12, 19]

1. CLFLUSH addr. Якщо блок даних з пам'яті за адресою addr знаходиться в кеші будь-якого рівня, то він видаляється з кешу. Якщо блок був змінений в кеші, то перед видаленням проводиться запис змінених даних в пам'ять.

2. PREFETCH addr. «Замовляє» завантаження даних з пам'яті за адресою addr в кеш першого рівня. Дані завантажуються в кеш з оптимізацією для читання, тобто, під час запису за адресою addr виникне затримка з-за необхідності передати сигнал іншим ядрам для підтримки когерентності кешів.

3. PREFETCHW addr. «Замовляє» завантаження даних з пам'яті за адресою addr в кеш першого рівня. Дані завантажуються в кеш з оптимізацією для запису, тобто, під час завантаження зразу посилається сигнал іншим ядрам процесору про те, що дане ядро має намір записувати в пам'ять по адресі addr та інші ядра мають видалити ці дані з власних індивідуальних кешів.

4. PREFETCH*рівень* addr. «Замовляє» завантаження даних з пам'яті за адресою addr в кеш заданого рівня.

1.6 Характеристика модельного ряду процесорів Opteron

Важливо розглянути процесори модельного ряду Opteron, так як всі новітні розробки AMD впроваджує в першу чергу в цих процесорах, орієнтованих на використання в високопродуктивних серверах. Так, Opteron став першим процесором з архітектурою AMD64, [11] перші двоядерні процесори також з'явились в модельному ряду Opteron. [10]

Як приклад сучасного процесору Opteron, в якому впроваджено всі нові розробки, розглянемо Opteron 6100. Модельний ряд процесорів AMD Opteron 6100 з кодовим ім'ям «Magny-Cours» представлений 29 березня 2010 року. Серед процесорів даного ряду є перший 12-ядерний x86 процесор. [20] Процесор базується на мікроархітектурі K10, описаній вище. Даний процесор має сумарно 12 Мб спільного кешу третього рівня, а також 512 Кб кешу другого рівня для кожного ядра. Різні моделі даного процесору працюють на частотах від 1,9 до 2,3 ГГц. В даному процесорі AMD вперше використала архітектуру Direct Connect версії 2.0.

На рисунку 1.2 представлена структурна схема AMD Opteron 6100. [20] В одному корпусі процесорів Opteron 6100 міститься дві інтегральні схеми, на кожній з яких реалізовано 4 або 6 ядер. Кожна інтегральна схема має 6 Мб кешу третього рівня. Кеш третього рівня є спільним для всіх ядер відповідної інтегральної схеми. Кожна інтегральна схема має контролер доступу до пам'яті з 2 каналами, а також 4 лінії HyperTransport. Максимальна пропускна здатність ліній HyperTransport — 25,6 Гб/с.

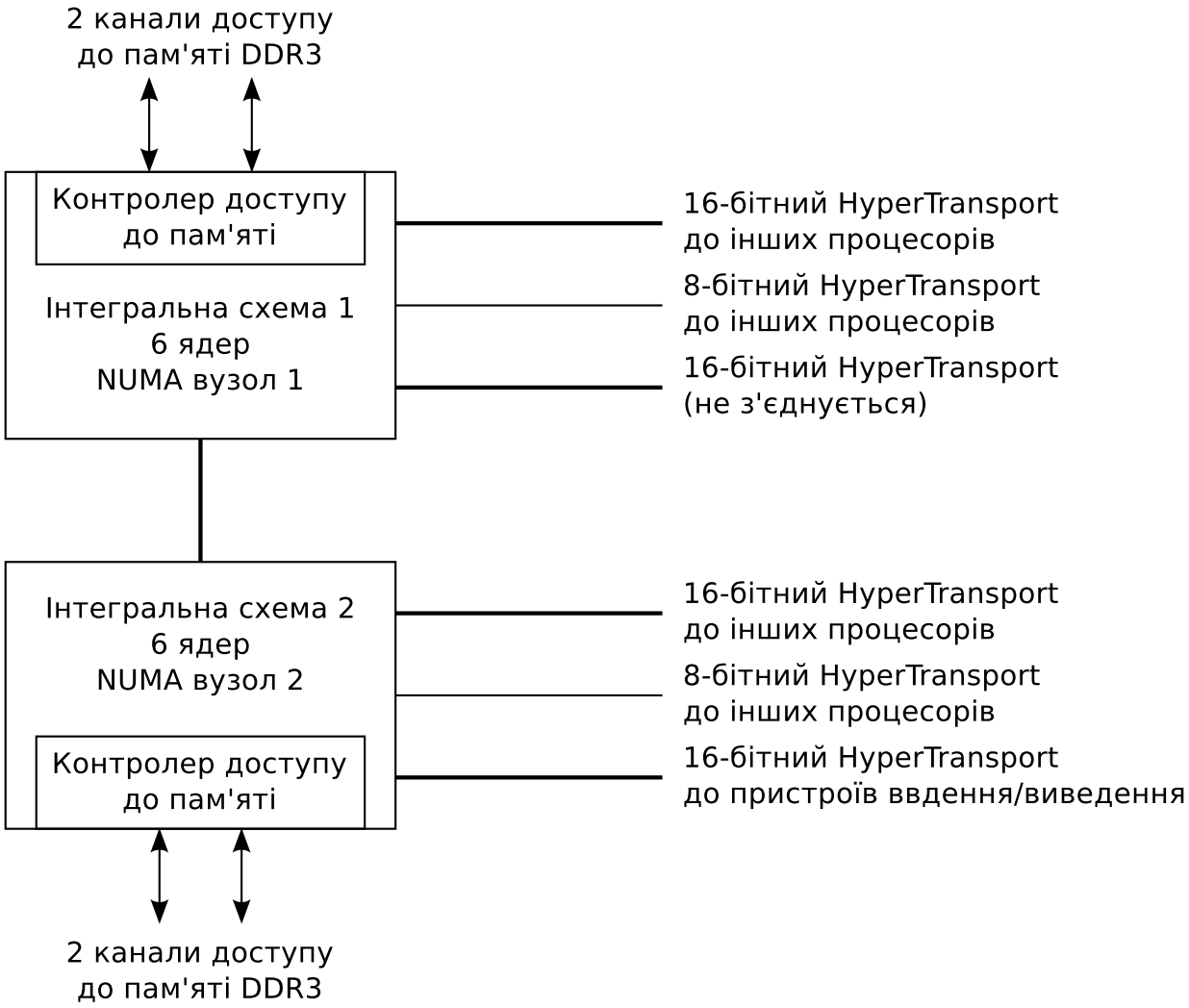

Рис. 1.2 – Структурна схема AMD Opteron 6100

З рисунку 1.2 видно, що ядра кожної інтегральної схеми мають безпосередній доступ лише до частини оперативної пам’яті. До решти пам’яті доступ здійснюється опосередковано через інші лінії передачі даних між ядрами. Таким чином, навіть однопроцесорні конфігурації Opteron 6100 є NUMA системами — системами з неоднорідним доступом до пам’яті.

Використавши NUMA архітектуру замість традиційної SMP, AMD збільшила продуктивність підсистеми пам’яті та уникла типової проблеми недостатньої пропускної здатності пам’яті в SMP системах та заклала засади масштабованості архітектури для майбутніх систем. [21]

NUMA вузли в Opteron 6100 мають неоднорідний доступ не лише до пам’яті, а й до периферійних пристроїв. Ланцюг з периферійних пристроїв з’єднується лише з одним NUMA вузлом, через який всі інші вузли отримують доступ до периферії.

На рисунку 1.3 показана топологія зв’язків між NUMA вузлами в двопроцесорній конфігурації. Це повнозв’язна топологія, тобто, кожен вузол має лінії зв’язку HyperTransport до кожного іншого вузла. Тим не менше, пропускна здатність цих зв’язків не рівноцінна: одні зв’язки є 16-бітними, інші — 8-бітними.

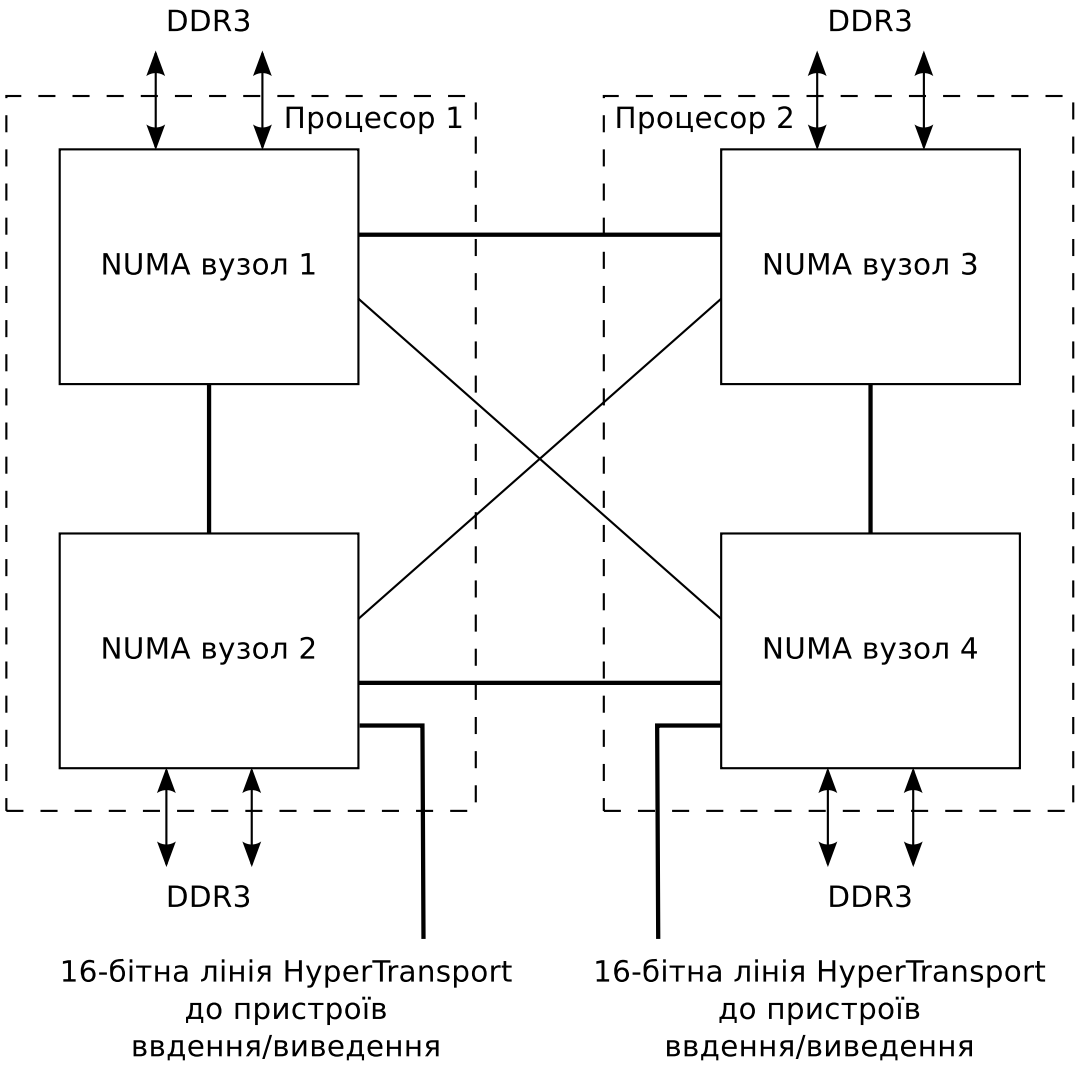

Рис. 1.3 – Структурна схема двопроцесорної конфігурації з AMD Opteron 6100

1.7 Особливості модельного ряду процесорів Phenom та Phenom II

Процесори Phenom базуються на мікроархітектурі K10, розглянутій вище. Процесори орієнтовані в першу чергу на застосування в настільних та мобільних комп'ютерах.

Серед процесорів є двох-, трьох-, чотирьох- та шестиядерні моделі. Вони мають маркування X2, X3, X4, X6 відповідно. [22] Трьохядерні процесори фактично є чотирьохядерними, але з одним відключеним ядром. Четверте ядро відключають у всієї партії на основі результатів тестування деяких процесорів з цієї партії. Тому існує ненульова імовірність того, що четверте ядро в Phenom II X3 є робочим. В Інтернеті легко знайти інструкції для ввімкнення четвертого ядра. [23]

Основні характеристики процесорів Phenom співпадають з характеристиками відповідних процесорів модельного ряду Opteron. В процесорах Phenom так само використовується архітектура DirectConnect, кеш третього рівня є спільним, контролер пам'яті реалізований в інтегральній схемі процесора.

В сучасних процесорах Phenom реалізовані технології, які орієнтовані на використання

в настільних комп'ютерах. Це такі технології як Turbo CORE, Cool'n'Quiet, CoolCore. [24] Розглянемо кожну з них більш детально.

Технологія Turbo CORE в шестиядерних процесорах Phenom II дозволяє підвищити тактову частоту трьох ядер у випадку, коли навантажено не більше трьох ядер. Частота та напруга трьох ненавантажених ядер при цьому знижується. Це дозволяє підвищити продуктивніть програмного забзепечення, яке не оптимізовано для роботи на будь-якій кількості процесорів. Важливим аспектом даної технології є уникнення перевищення максимальних рівнів тепловиділення. Для кожного процесору встановлений так званий теплопакет — максимальна теплова потужність, яку система охолождення відводити з процесору. Якщо просто підвищити тактову частоту ядер, що навантажені, то тепловиділення вийде за границі теплопакету. Таким чином зменшення тепловиділення ненавантажених ядер є необхідним. [25, 26]

Технологія Cool'n'Quiet призначена для зниження рівнів споживання електроенергії та зниження тапловиділення. Так як на сучасні процесори в основному встановлюються системи повітряного охолодження, то зниження тепловиділення приводить до зниження рівня шуму систем охолодження. Принцип роботи Cool'n'Quiet базується на моніторингу завантаження процесору в цілому та зниження напруги та тактової частоти під час малого завантаження. [24, 27]

Технологія CoolCore має ту ж саму мету, що й Cool'n'Quiet, але іншу технічну реалізацію. Принцип роботи CoolCore полягає у вимиканні блоків процесору, які не використовуються у даний момент. Наприклад, під час читання з пам'яті блок контролера пам'яті, що відповідає за запис, вимикається. [24]

1.8 Шина HyperTransport

HyperTransport — двонаправлена послідовно/паралельна високопродуктивна шина, розроблена консорціумом HyperTransport, в який входить компанія AMD. [28, 29]

На сьогоднішній день HyperTransport має широке застосування в тих комп'ютерних системах, де швидкодія є критичною. Ця шина застосовується не тільки в процесорах AMD Opteron та Athlon 64, а й в процесорах Apple G5 (IBM) та його чипсетах, в процесорах Transmeta, в різноманітних серверних пристроях виробництва Cisco та Broadcom, в чипсетах Nvidia та ALi/ULi.4. [21]

Основною проблемою, яка підштовхнула AMD до розробки HyperTransport було відставання пропускної спроможності шин передачі даних від тактової частоти мікропроцесорів. Тактова частота подвоювалась приблизно кожні 2 роки, а пропускна спроможність шин, що зв'язували мікропроцесори один з одним та з периферійними пристроями — лише кожні 3 роки. [30] За десятиліття розвитку обчислювальної техніки відставання стало створювати суттєву перепону на шляху створення високопродуктивних обчислювальних систем.

Крім того, при підвищенні пропускної здатності класичних багатопровідних шин інженери-

розробники друкованих плат зіткнулись з проблемою надмірного випромінювання електромагнітних хвиль.

Окрім вирішення основної проблеми, AMD, розробляючи HyperTransport, мала на меті зменшити кількість проводів, тим самим зменшивши електромагнітне випромінення, а також зменшивши необхідну потужність живлення. [28]

З точки зору сфери застосування, HyperTransport призначається для організації високошвидкісних каналів зв'язку типу точка-точка в межах системної плати. HyperTransport оптимізована для реалізації в кристалах мікропроцесорів та інших компонентів системної плати. Про це свідчать наступні особливості:

– наявність засобів управління живленням;

– ACPI-сумісність.

1.8.1 П'ятирівнева модель HyperTransport

Архітектура HyperTransport може бути представлена у вигляді наступних п'яти рівнів.

1. Фізичний рівень визначає фізичні та електричні вимоги до ліній передачі даних. На цьому рівні описується, як абстрактні сигнали з вищих рівнів безпосередньо передаються по лініям передачі даних. Описуються лінії даних, лінії управління, а також лінії передачі тактових сигналів.

2. Канальний рівень визначає послідовність дій при ініціалізації та налаштуванні, способи контролю коректності передачі даних (за допомогою підрахунку контрольних сум CRC), послідовність дій при відключенні та підключенні, формат інформаційних пакетів, що використовуються для управління, формат для пакетів даних.

3. На протокольному рівні визначаються команди, віртуальні канали для передачі команд, а також порядок їх передачі.

4. Рівень транзакцій використовує сервіси протокольного рівня для виконання дій, таких як запис та зчитування.

5. Сеансовий рівень описує правила, за яким пристрої налаштовують параметри живлення, параметри переривань, а також інших системних параметрів.

1.8.2 Фізичний рівень

В HyperTransport використовуються диференційні електричні сигнали з малою амплітудою (напруга низького рівня $-0{,}3$ В, високого $+1{,}7$ В). Використання диференційних сигналів дозволяє HyperTransport працювати у середовищах з високим рівнем електромагнітного випромінення (таких як системна плата). Використання малих амплітуд напруги дозволяє

використовувати високі тактові частоти передачі, так як час перезаряду паразитної ємності ліній зв'язку тим менше, чим менше амплітуда сигналу (але треба зауважити, що ця залежність не є прямо пропорційною). Максимальна довжина ліній зв'язку може складати 0,6 м, що більше, ніж достатньо для використання у системних платах. [28]

Ширина шини може обиратись інженером в залежності від потреб пристрою. За специфікацією, допустимими є 2, 4, 8, 16, 32-бітні шини. Аналогічно, тактова частота шини може обиратись в діапазоні від 400 до 1600 МГц. Збільшуючи кількість провідників можна збільшити ефективну швидкість передачі даних без зміни тактової частоти. Це дозволяє не змінювати технологічний процес виробництва інтегральної схеми, в яку інтегрують HyperTransport.

Треба відмітити, що допустимими є асиметричні з'єднання пристроїв. Тобто, кількість ліній на передачу даних не має бути рівною кількості ліній на прийом.

На рисунку 1.4 показано з'єднання двох пристроїв HyperTransport. Один пристрій може передавати дані по 4 лініям, а інший — по 16 лініям. Окрім ліній для безпосередньої передачі даних, в інтерфейсі присутня лінія CTL, логічний рівень сигналу на якій визначає, в конкретний момент часу передаються дані чи команда. Кожні 8 ліній передачі даних тактуються власним тактовим сигналом. Дані передаються за принципом DDR (англ. dual data rate — подвійна швидкість даних), тобто, передача даних відбувається як за переднім, так і за заднім фронтом тактового сигналу. [21, 28]

Інші лінії не зв'язані з передачею даних і призначені для керування самою шиною. Живлення $+2{,}5$ В призначене тільки для потреб самої шини (тобто, для тих електронних компонентів, що становлять інтерфейс пристрою); живлення ж самого пристрою по HyperTransport не передбачено. Лінія PWROK переводиться пристроєм у високий логічний стан коли пристрій виявив стабільну напругу живлення. Через деякий час, хост HyperTransport має видати низький рівень сигналу RESET#, за яким під'єднаний пристрій має розпочати процедуру ініціалізації. Лінія LDTSTOP# встановлюється в низький логічний рівень у випадку, коли пристрій знаходиться в режимі енергозбереження, а низький сигнал по лінії LDTREQ# дозволяє перевести пристрій з цього режиму в нормальний. [21]

В таблиці 1.1 показані можливі швидкості передачі даних при заданій частоті тактових сигналів та ширині шини.

З таблиці видно, що навіть в мінімальній конфігурації (2 біта на прийом, 2 біта на відправку, 21 фізичний провідник, 400 Мб/с) HyperTransport на 800 МГц має пропускну здатність вдвічі більшу, ніж PCI Express (1 біт на прийом, 1 біт на передачу, 36 фізичних провідників, 250 Мб/с). Пропускна здатність 32-бітних шин HyperTransport відповідає по швидкості двоканальній пам'яті DDR400.

Фактично, потенційні можливості HyperTransport перевищують потреби сучасних комп'ютерних систем. [29]

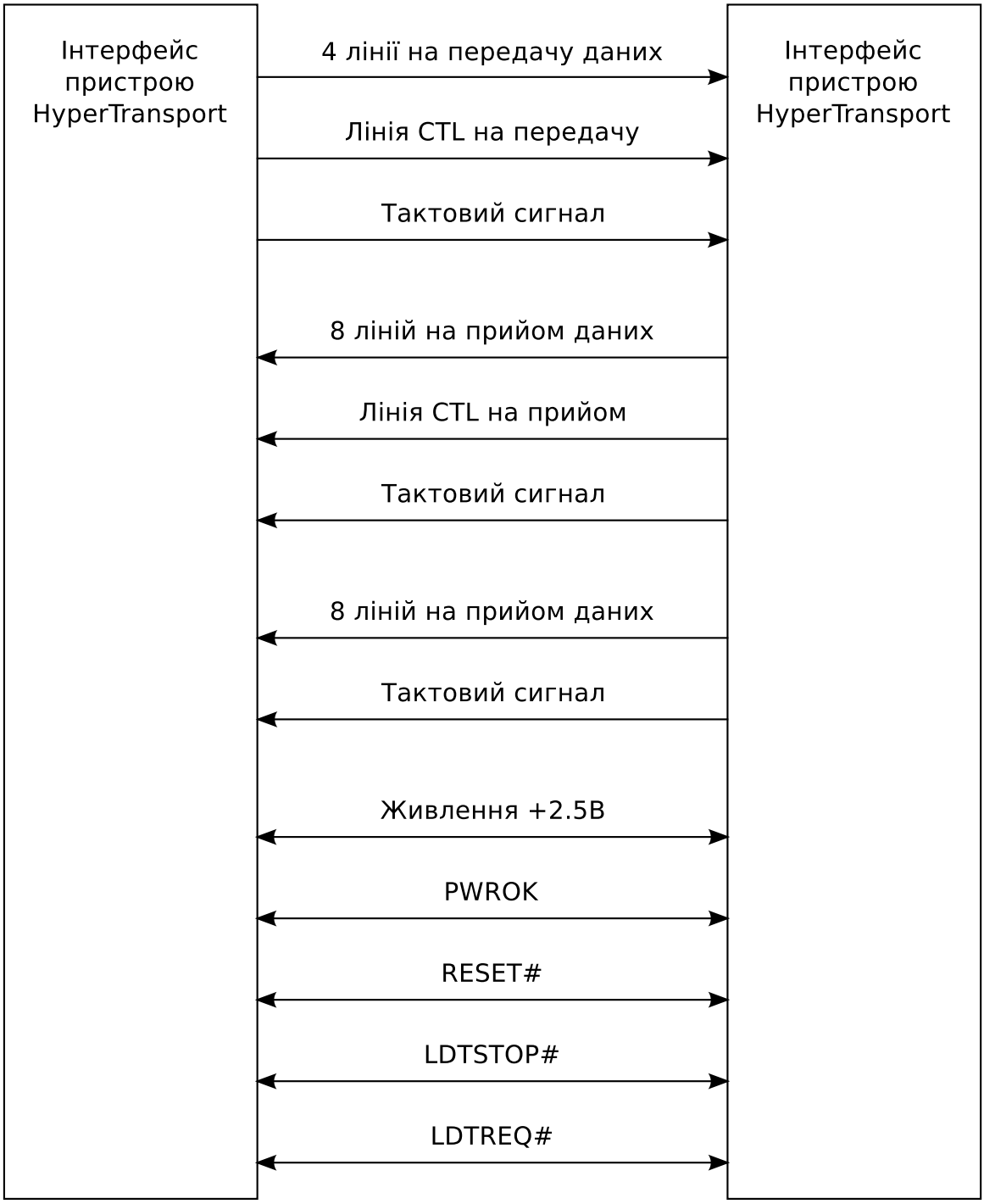

Рис. 1.4 – З'єднання інтерфейсів пристроїв HyperTransport

1.8.3 Канальний рівень

Передача даних в HyperTransport базується на моделі передачі пакетів. Це дозволяє абстрагуватись від фізичних особливостей каналу зв'язку (таких як кількість ліній) за рахунок використання багаторівневої моделі. Крім того, це дозволяє передавати дані різних видів по одним і тим же лініям зв'язку. В реальних комп'ютерних системах через HyperTransport передаються пакети з керувальною інформацією відносно самого HyperTransport, керувальні пакети ACPI, пакети з даними пристроїв.

Табл. 1.1 – Швидкість передачі даних по шині HyperTransport

Частота, МГц	Швидкість передачі, Мб/сек				
	2 біта	4 біта	8 біт	16 біт	32 біта
200	100	200	400	800	1600
400	200	400	800	1600	3200
600	300	600	1200	2400	4800
800	400	800	1600	3200	6400
1000	500	1000	2000	4000	8000
1200	600	1200	2400	4800	9600
1400	700	1400	2800	5600	11200

Пакети даних в HyperTransport мають довжину від 4 до 64 байт. Довжина пакету має бути кратна чотирьом. Керувальні пакети мають довжину 4 або 8 байт. Така мала довжина пакетів була обрана для виключення можливості монополізації каналу зв'язку одним пристроєм, а також забезпечення низьких затримок передачі даних.

Важливо відмітити відсутність можливості складної маршрутизації пакетів, а також підтримки QoS та передачі ізохронних даних в HyperTransport (на відміну від, наприклад, PCI Express). Це пояснюється високими вимогами до швидкодії, а також до затримок передачі даних.

В HyperTransport передбачений механізм, що дозволяє пристрою-тунелю передати свої дані під час транзитної передачі даних. В цьому випадку тунель перериває передачу транзитного пакету, передає свої дані, після чого передача транзитного пакету продовжується.

Контроль помилок HyperTransport виконується нетиповим способом для пакетної передачі даних. Контрольні коди обраховуються не для пакетів даних, а для вікон. Вікно — це частина потоку заданої довжини. Після передачі вікна передається контрольний код, незважаючи на те, де закінчується чи починається пакет даних. Це дозволяє легко обчислити, яка частина потоку даних буде використана на контрольні коди, а яка буде нести корисні дані. На контрольні коди використовується 0,78% пропускної здатності HyperTransport. [21]

Вищеназвані інженерні рішення дозволяють передавати дані по шині HyperTransport з середньою затримкою в 1-2 такти. [21]

1.8.4 Топології зв'язку

Множина можливих топологій в HyperTransport визначається допустимими типами пристроїв. Всього в специфікації описано три типи пристроїв.

1. Печера (англ. cave) — пристрій з однією лінією зв'язку в кінці ланцюга. Всі операції передачі даних для такого пристрою пов'язані тільки з даними, призначеними безпосередньо для цього пристрою.

2. Тунель (англ. tunnel) — пристрій з двома лініями зв'язку, який не є мостом. Такі пристрої окрім передачі власних даних, також займаються транзитною передачею даних для інших пристроїв.

3. Міст (англ. bridge) — пристрій з однією головною лінією зв'язку HyperTransport та кількома допоміжними лініями зв'язку (не-HyperTransport). Такі пристрої дозволяють під'єднувати до HyperTransport периферійні пристрої без ліній зв'язку HyperTransport.

Таким чином, HyperTransport дозволяє будувати ланцюги з пристроїв. Ланцюги можна об'єднувати в дерева за допомогою мостів.

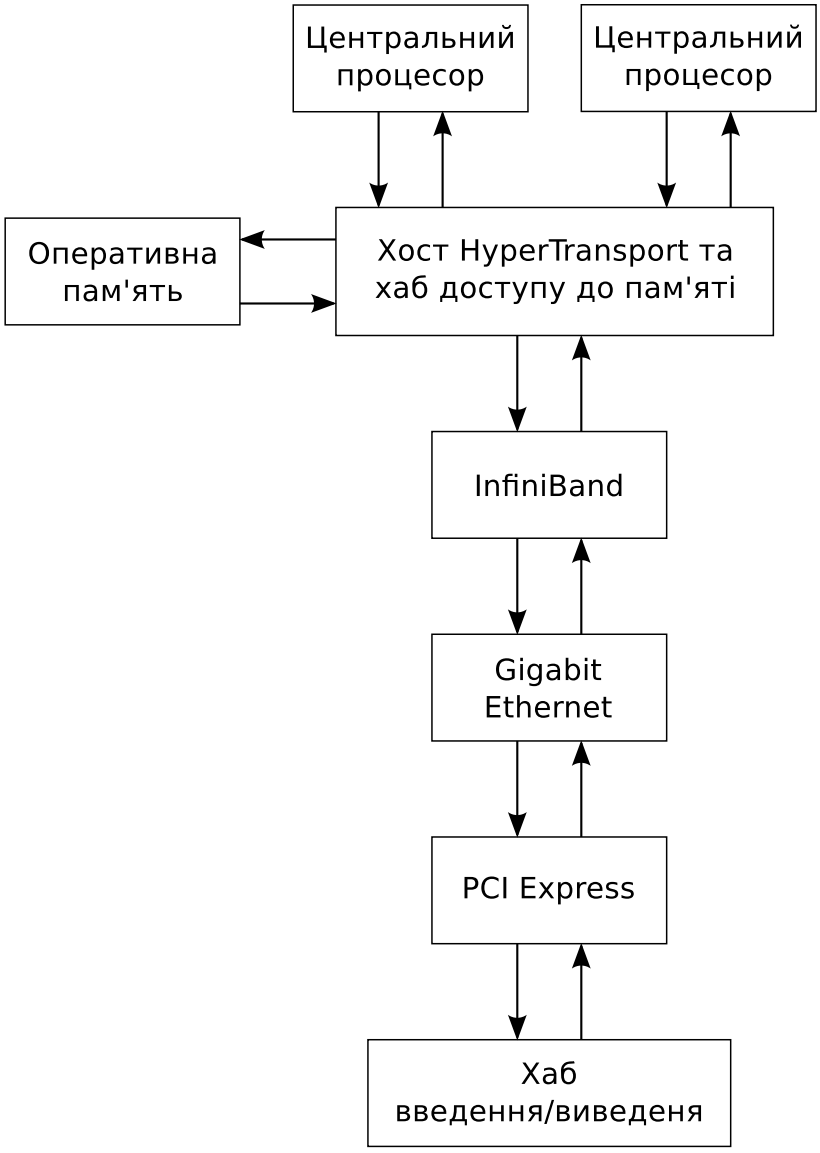

Рис. 1.5 – Приклад застосування HyperTransport для побудови комунікаційного середовища

На рис. 1.5 зображено приклад застосування HyperTransport для побудови комунікаційного середовища. [28] В даному випадку HyperTransport реалізовано в мікропроцесорних чипах та в контролерах периферійних пристроїв. Процесори під'єднані напряму до хабу, що

дозволяє їм обмінюватись інформацією з мінімальними затримками. Периферійні пристрої з'єднані в ланцюг. Ближче до початку ланцюга знаходяться найбільш швидкодійні пристрої.

Розглянемо цю схему з точки типів пристроїв, визначених специфікацією HyperTransport. Процесори становлять ланцюг з одного пристрою і є печерами. Таким чином, пристрій-печера не зайнятий транзитною передачею даних для будь-яких інших пристроїв. В кінці ланцюга периферійних пристроїв знаходиться хаб введення/виведення для найменш швидкодійних пристроїв — він також є печерою. Всі інші периферійні пристрої є тунелями, вони передають інформацію по ланцюгу для пристроїв, що знаходяться далі по ланцюгу. З іншого боку, всі периферійні пристрої є мостами.

1.9 Архітектура Direct Connect

Архітектура Direct Connect — розроблена компанією AMD архітектура введення/виведення, що застосовується у процесорах Athlon 64 X2, Opteron, та Phenom. Вона має наступні відмінності від класичної архітектури введення/виведення, що застосовувалась для процесорів x86:

1. Кожна інтегральна схема мікропроцесору має інтегрований контролер доступу до пам'яті, через який з'єднане з пам'яттю по одному або більшому числу каналів.

2. Мікропроцесор з'єднаний з підсистемою введення/виведення через реалізований на кристалі HyperTransport.

3. Ядра мікропроцесору з'єднуються з іншими ядрами (можливо, не з усіма) через шини HyperTransport. Це дозволяє забезпечити когерентність кешів в NUMA-системі.

На рисунку 1.3 показаний приклад системи з архітектурою Direct Connect.

1.10 Висновки

– Багатоядерність є сучасним інженерним підходом до вирішення проблем розробки мікропроцесорів, спричинених фундаментальними фізичними обмеженнями.

– Використання технології HyperTransport дозволяє побудувати високошвидкісне, масштабоване міжпроцесорне комунікаційне середовище з малими затримками передачі даних.

– Для забезпечення низьких затримок передачі даних в HyperTransport використовуються прийоми, нетипові для класичних систем передачі пакетів, як то: обчислення контрольних кодів не для пакетів, а для вікон, відсутність складної маршрутизації, простий алгоритм передачі даних високого пріоритету.

– Інженери-розробники системних плат можуть досягти необхідної пропускної здатності шини HyperTransport змінюючи тільки ширину шини. Це дозволяє не змінювати технологічний процес виробництва інтегральних схем для збільшення тактових частот.

– Шина HyperTransport може виступати в ролі системної шини, а також в ролі шини для обміну даними з периферійними пристроями; таке застосування HyperTransport дозволяє знизити складність та вартість комп'ютерної системи.

– Кеші процесорів AMD є невключаючими, тому їх корисний розмір рівний їх реальному розміру.

– Сучасні багатоядерні процесори AMD в однопроцесорній конфігурації мають NUMA архітектуру з двома вузлами. При під'єднанні більшого числа процесорів кількість NUMA вузлів збільшується.

– В мікроархітектурі K10 блок передбачення переходів було вдосконалено для роботи з переходами по адресі, заданою у вигляді операнда з непрямою адресацією. Це збільшило швидкодію операторів switch-case та викликів віртуальних функцій в об'єктно-орієнтованих мовах програмування.

– Побічний оптимізатор в стеку в мікроархітектурі K10 дозволяє зробити команди роботу зі стеком незалежними одна від одної, що приводить до збільшення швидкодії.

– В мікроархітектурі K10 передбачені спеціальні інструкції для попереднього завантаження даних в кеш, що дозволяє уникнути простоїв в роботі процесора через звернення до пам'яті, що не були передбачені евристичними алгоритмами.

2 Розробка програмного забезпечення для ПКС

2.1 Розробка програмного забезпечення на C++

Програмне забезпечення було розроблено з дотриманням рекомендованої методики [31].

2.1.1 Розробка паралельного математичного алгоритму

Паралельний математичний алгоритм відповідно до рекомендованої методики можна подати у вигляді наступних трьох етапів:

1. $\alpha_i = \max(MO_H), \quad (i = \overline{0, P-1})$
   $\alpha = \max(\alpha, \alpha_i); \quad (i = \overline{0, P-1})$
2. $MZ_H = MX_H \cdot MT$;
3. $MA_H = \alpha \cdot MR_H \cdot MZ$;

де:

- $H = \frac{N}{P} = \frac{N}{4}$;
- MZ_H — H рядків матриці MZ;
- MX_H — H рядків матриці MX.
- MR_H — H рядків матриці MR.
- MA_H — H рядків матриці MA.

Спільні ресурси: α, MT, MZ.

Треба зауважити, що другий етап обчислень не залежить по даним від першого. Тому можна не синхронізувати задачі по завершенню першого етапу.

Проведемо аналіз даної задачі з точки зору концепції необмеженого паралелізму (КНП). Для оцінки необхідного часу обчислень використаємо теорему Мунро-Петерсена, яка для комп'ютерної системи з необмеженим числом процесорів формулюється наступним чином: якщо виконується обчислення скалярної величини, яке потребує m бінарних операцій, то необхідний час обчислень t_p:

$$t_p \geq \lceil \log_2(m+1) \rceil.$$

Для обчислення $\alpha = \max(MO)$ необхідно виконати N^2 бінарних операцій порівняння. Тому час виконання буде:

$$t_{p1} \geq \lceil \log_2(N^2+1) \rceil.$$

Для обчислення одного елементу добутку матриць $MZ = MX \cdot MT$ необхідно виконати N множень та $N-1$ додавання. Тому час виконання буде:

$$t_{p2} \geq \lceil \log_2(2N) \rceil = 1 + \lceil \log_2 N \rceil.$$

Для обчислення одного елементу добутку $MA = \alpha \cdot MR \cdot MZ$ необхідно виконати $N+1$ множення та $N-1$ додавання. Тому час виконання буде:

$$t_{p3} \geq \lceil \log_2(2N+1) \rceil.$$

Так як перший і другий етап незалежні, то вони можуть виконуватись паралельно. Тому сумарний час їх виконання буде рівний максимальному з двох:

$$t_{p1,2} \geq \max(t_{p1}, t_{p2}) = t_{p1} = \lceil \log_2(N^2+1) \rceil.$$

Сумарний час виконання всіх трьох етапів обчислень буде виражатись наступною формулою:

$$t_p \geq t_{p1,2} + t_{p3} = \lceil \log_2(N^2+1) \rceil + \lceil \log_2(2N+1) \rceil.$$

2.1.2 Розробка алгоритмів задач

Так як розроблюване програмне забезпечення має бути масштабованим, тобто має працювати на системі з будь-якою кількістю процесорів, то зручним варіантом реалізації є написання єдиного алгоритму для всіх задач.

Задачі T(0) — T(P-1):

Крок алгоритму	ТС, КД
1. Якщо tid = 0, ввести MO, MR.	
2. Якщо tid = 0, сигнал задачам $1 \ldots P-1$ про завершення вводу1.	$S_{j,1}, \quad j = \overline{1, P-1}$
3. Якщо tid = $P-1$, ввести MX, MT.	
4. Якщо tid = $P-1$, сигнал задачам $0 \ldots P-2$ про завершення вводу2.	$S_{j,2}, \quad j = \overline{0, P-2}$
5. Якщо tid != 0, чекати сигналу про завершення вводу1 від задачі 0.	$W_{0,1}$
6. Якщо tid != $P-1$, чекати сигналу про завершення вводу2 від задачі $P-1$.	$W_{P-1,2}$
7. Обчислення $\alpha_i = \max(MO_H)$.	
8. Обчислення $\alpha = \max(\alpha, \alpha_i)$.	КД
9. Копія $MT_i = MT$.	КД
10. Обчислення $MZ_H = MX_H \cdot MT_i$.	
11. Сигнал всім задачам про завершення обчислень1.	$S_{j,3}, \quad j = \overline{0, P-1}$
12. Чекати сигналів про завершення обчислень1 від всіх задач.	$W_{j,3}, \quad j = \overline{0, P-1}$
13. Копія $MZ_i = MZ$.	КД
14. Копія $\alpha_i = \alpha$.	КД

Крок алгоритму	ТС, КД
15. Обчислення $MA_H = \alpha \cdot MR_H \cdot MZ_i$.	
16. Якщо tid != 0, сигнал задачі 0 про завершення обчислень3.	$S_{0,4}$
17. Якщо tid = 0, чекати сигналів про завершення обчислень3 від задач $1 \ldots P-1$.	$W_{j,4}, \quad j = \overline{1, P-1}$
18. Якщо tid = 0, вивести MA.	

2.1.3 Розробка структурної схеми взаємодії задач

На основі алгоритму для всіх задач, розробленого в попередньому розділі, була розроблена структурна схема взаємодії задач (рис. 2.1–2.3). Структурна схема розділена на три рисунки для більш наглядного сприйняття. Структурна схема дозволяє контролювати зв'язок точок синхронізації (сигналів S та очікувань W). Це дозволяє перевірити, що алгоритм розроблено вірно і при його реалізації будуть відсутні взаємні блокування через відсутність сигналів S та не прийняті сигнали через відсутність очікувань W. Крім того, схема взаємодії дозволяє зв'язати точки синхронізації з семафорами та подіями, що будуть використовуватись у програмі.

На структурній схемі показані 4 задачі: T(0), T(i), T(j), T(P-1). Необхідно показати саме ці 4 задачі, щоб продемонструвати всі види взаємодій, що виникають від час виконання алгоритму. Задачі T(0) та T(P-1) вводять дані, тому з ними взаємодіють всі інші (синхронізація по вводу). Задачі T(i), T(j) даних не вводять, але виконують обчислення і синхронізуються одна з одною та з усіма іншими.

На структурній схемі також показані засоби для вирішення задач синхронізації та взаємного виключення, а саме:

- подія ev_input_1 — для синхронізації з завершенням вводу в T(0);
- подія ev_input_2 — для синхронізації з завершенням вводу в T(P-1);
- критична секція cs_alpha — для вирішення задачі взаємного виключення під час зчитування та запису спільного ресурсу α;
- критична секція cs_MT — для вирішення задачі взаємного виключення під час зчитування спільного ресурсу MT;
- критична секція cs_MZ — для вирішення задачі взаємного виключення під час зчитування спільного ресурсу MZ;
- масив подій ev_comp2_end[N] — для синхронізації всіх задач з завершенням обчислень2 в усіх задачах;
- семафор sem_comp3_end — для синхронізації задачі T(0) з завершенням обчислень3 в задачах T(1)... T(P-1).

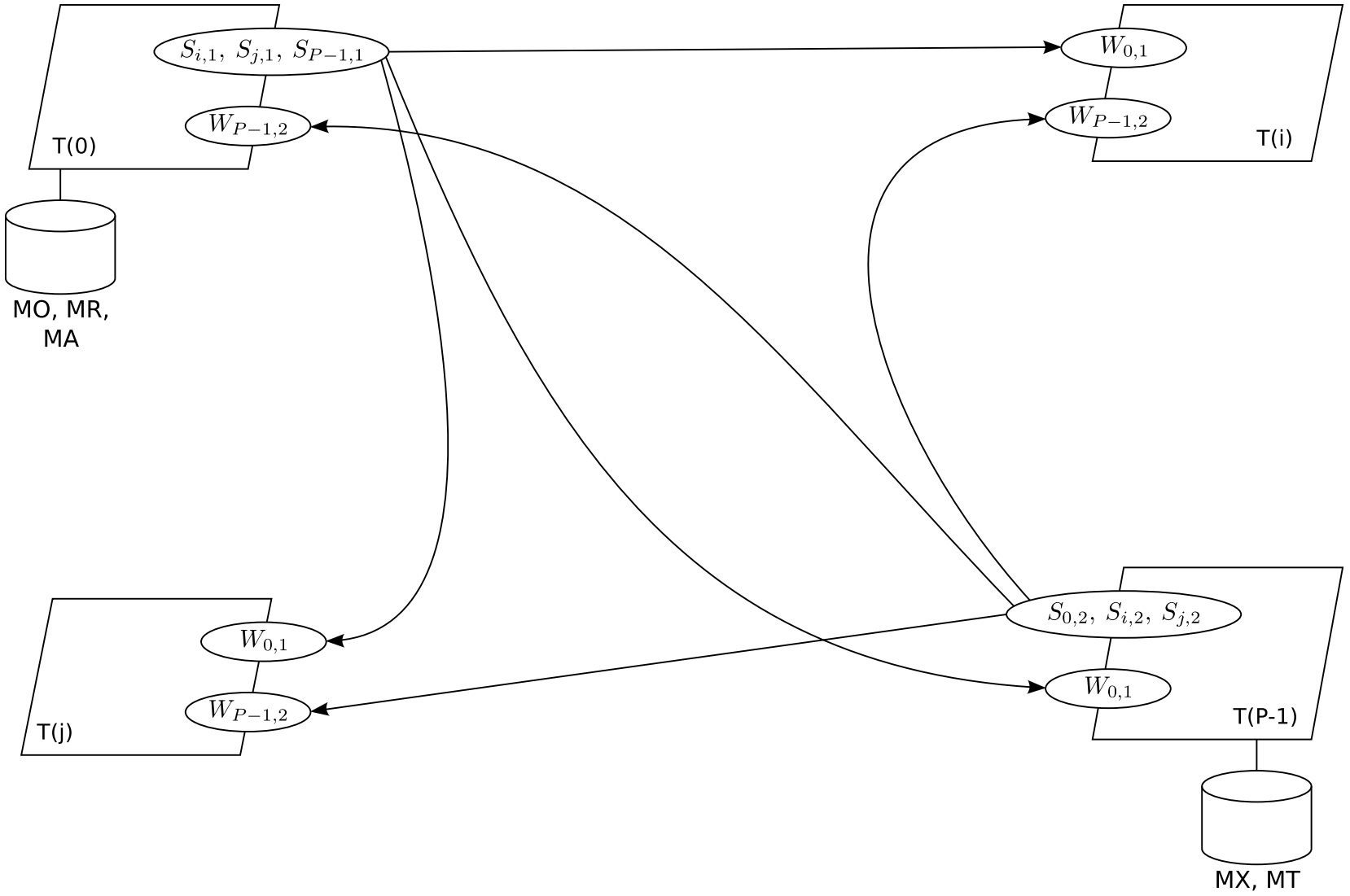

Рис. 2.1 – Структурна схема взаємодії задач під час кроків алгоритму 1–6

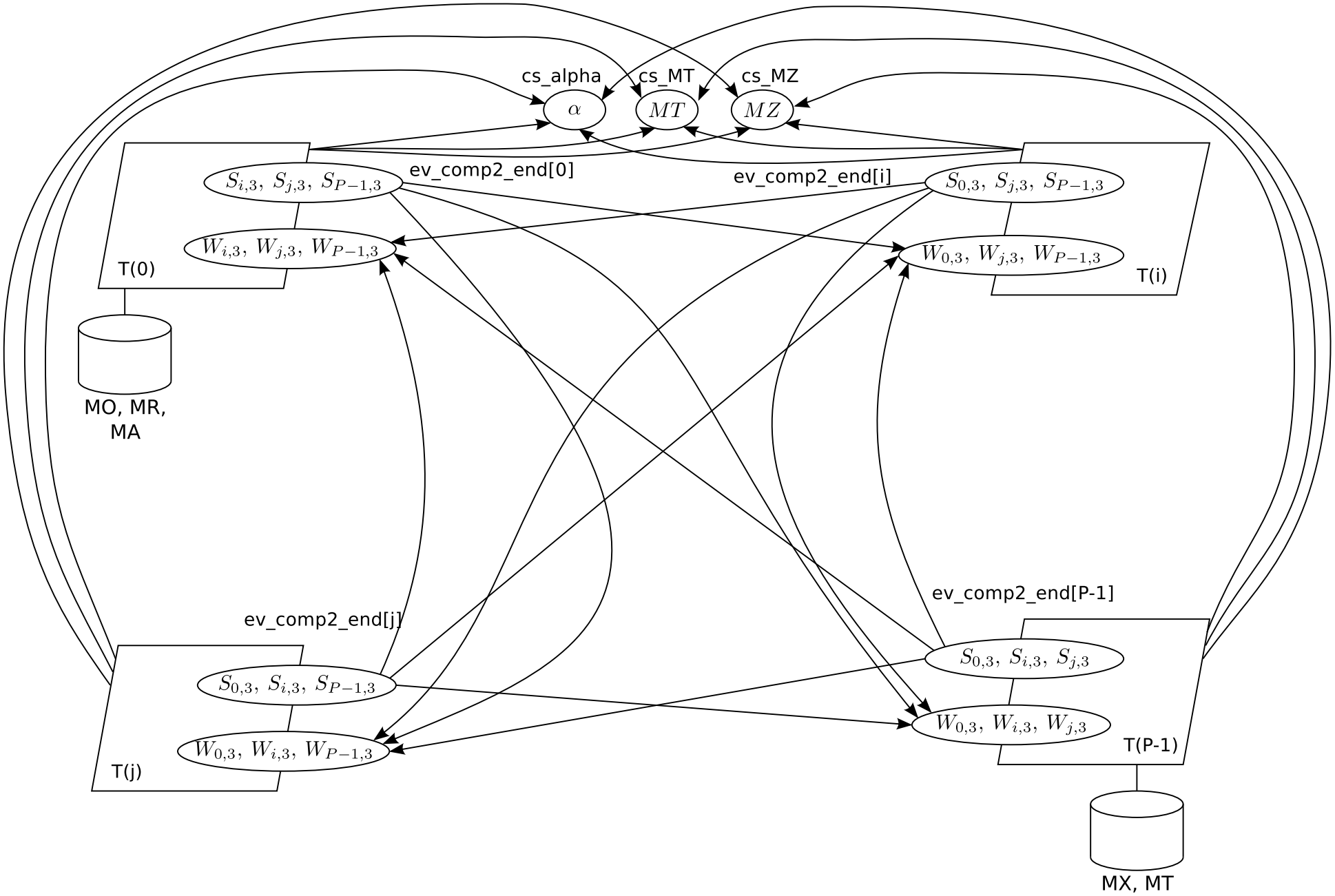

Рис. 2.2 – Структурна схема взаємодії задач під час кроків алгоритму 7-15

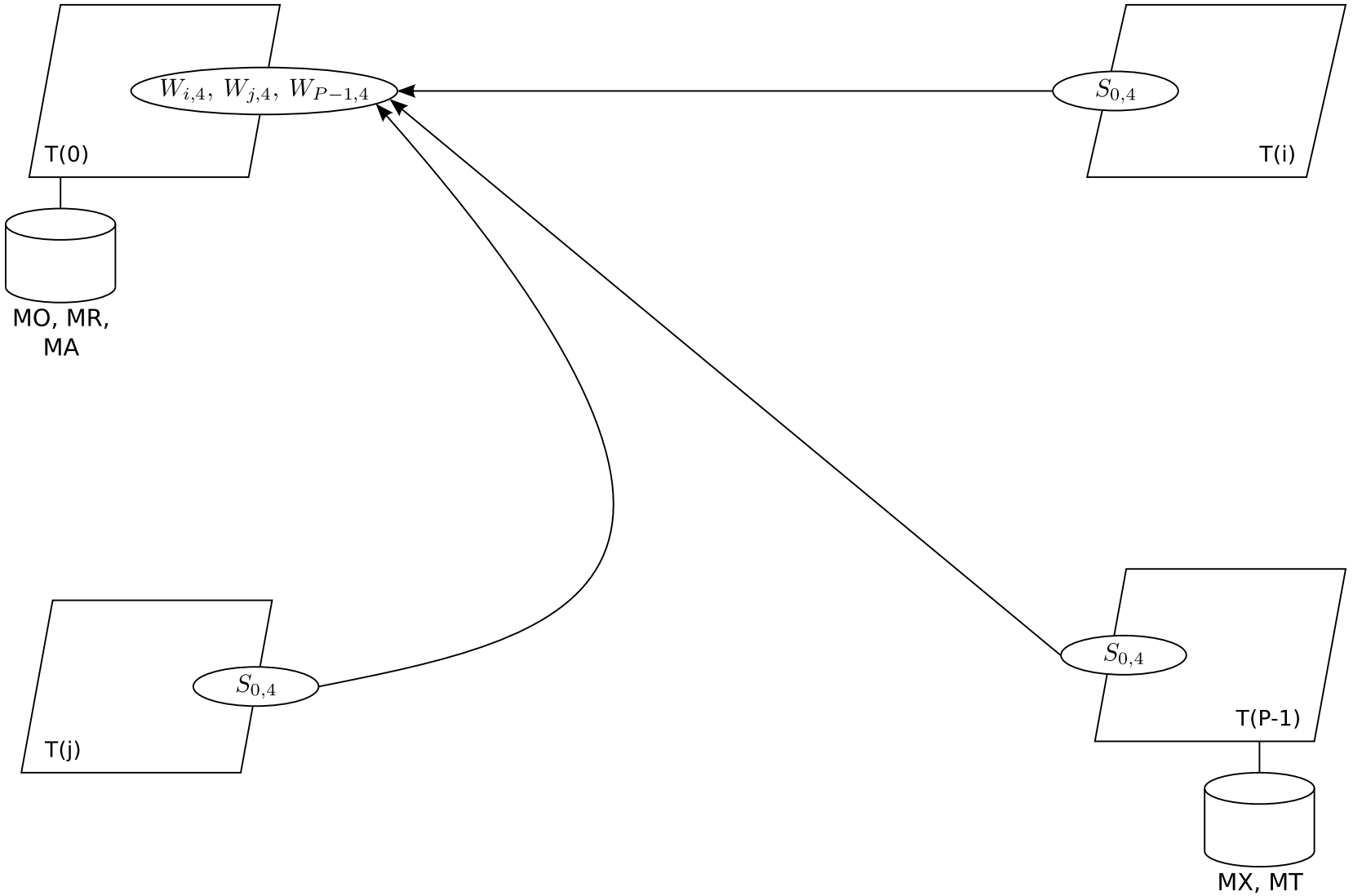

Рис. 2.3 – Структурна схема взаємодії задач під час кроків алгоритму 16-17

2.1.4 Розробка програми

Програма реалізована у вигляді трьох модулів: main.cc, matrix.cc, functions.cc. Останні два модуля є допоміжними, а main.cc є основним. В ньому реалізовано дві функції:

– main — основна підпрограма, що формує ідентифікатори задач та запускає задачі. Основна підпрограма також вимірює час виконання задач;

– thread_proc — підпрограма, що запускається у кожній з задач.

Крім того, в основному модулі знаходиться константа P, змінюючи яку можна виконати налаштування програми під конкретну комп'ютерну систему.

Для експериментального дослідження впливу копіювання спільних ресурсів, що тільки читаються задачами, на час роботи програми, введена константа copy_shared_readonly. Коли ця константа встановлена в true, спільні ресурси, що тільки читаються копіюються кожною з задач. Коли значення константи false, спільні ресурси, що тільки читаються, не копіюються.

Повний код програми знаходится у додатку Г.

2.2 Розробка програмного забезпечення на Java

Так як математична задача та паралельна комп'ютерна система співпадають з тими, що розглядались в розділі 2.1, то перші два кроки по рекомендованій методиці [31] не матимуть відмінностей від вже виконаних. Тому розділи «Розробка паралельного математичного алгоритму» і «Розробка алгоритмів задач» пропущені.

2.2.1 Розробка схеми взаємодії задач

Під час виконання даного етапу була розроблена структура класу-монітора TaskControl. На даному етапі визначався набір захищених елементів, що будуть знаходитись у моніторі, а також множина захищених операцій. Набір захищених елементів визначається множиною спільних ресурсів (див. паралельний математичний алгоритм) та множиною змінних, що використовуються в якості умов. Семантика захищених операцій обиралася виходячи з завдання мінімізації кількості захищених операцій.

Клас TaskControl містить поля MT, MZ, MA, alpha для зберігання відповідних спільних ресурсів, а також поля inputCount, comp2Count, comp3Count для організації умов виконання методів монітору. В класі містяться наступні методи:

– waitInput — для очікування введення даних в потоках T(0) та T(P-1);

– waitComp2 — для очікування закінчення обчислень2;

– waitComp3AndOutputMA — для очікування закінчення обчислень3 та виводу MA;

– inputMT — для введення MT (так як тільки один спільний ресурс є вхідними даними програми, то для уникнення зайвого копіювання введення перенесено у монітор);

– copyMT — для копіювання спільного ресурсу MT;

– copyMZ — для копіювання спільного ресурсу MZ;

– copyAlpha — для копіювання спільного ресурсу α;

– signalInputDone — для сигналу про завершення вводу даних;

– computeMaxAlpha — для виконання операції над спільним ресурсом α;

– signalComp2AndSetMZPart — для сигналу про завершення обчислень2 та запису спільного ресурсу MZ;

– signalComp3AndSetMAPart — для сигналу про завершення обчислень3 та запису спільного ресурсу MA.

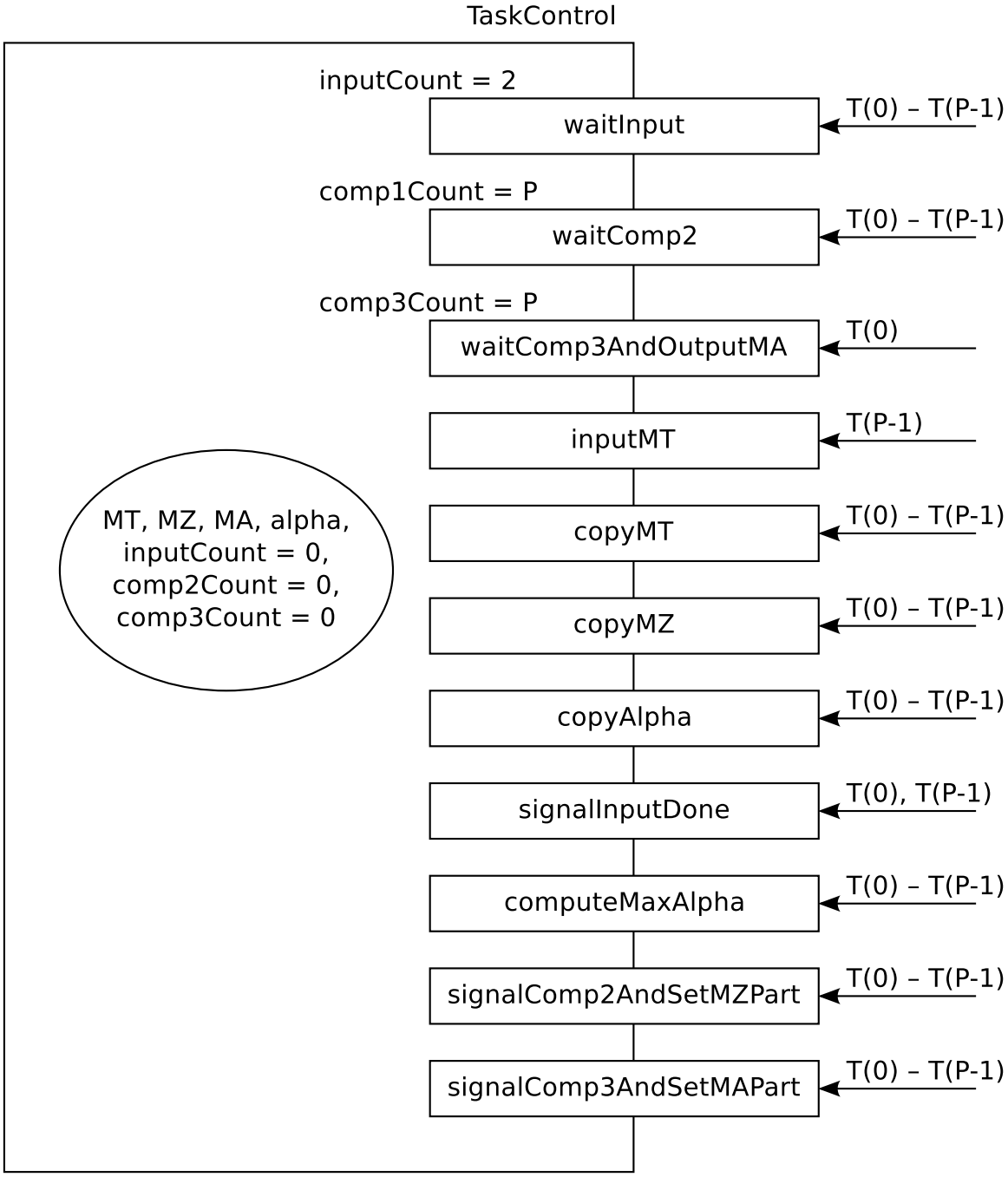

Рис. 2.4 – Структура класу TaskControl

2.2.2 Розробка програми

Програма реалізована у вигляді чотирьох класів:

– Main — основний клас. Містить головний метод, що запускається JVM при старті програми. Головний метод формує ідентифікатори потоків, запускає потоки та вимірює час їх виконання. В основному класі знаходиться константа P, змінюючи яку можна виконати налаштування програми під конкретну комп'ютерну систему;

– TT — задачний тип, дочірній по відношенню до класу Thread;

– TaskControl — клас-монітор, який вирішує задачі синхронізації та взаємного виключення, а також зберігає спільні ресурси;

– Functions — допоміжний клас, який містить методи, що безпосередньо виконують обчислення.

Повний код програми знаходится у додатку Д.

2.3 Висновки

– Рекомендована методика [31] дозволяє уникати помилок при розробці паралельних програмного забезпечення для виконання лінійної векторної алгебри.

– Дана задача в цілому є частково паралельною, хоча її кроки 2 та 3 є повністю паралельними.

– З точки зору концепції необмеженого паралелізму, в системі з необмеженою (будь-якою потрібною) кількістю процесорів час виконання операції знаходження максимального елементу матриці більше, ніж час виконання операції множення матриць.

– Розробка програмного забезпечення для виконання елементарних операцій лінійної алгебри для масштабованих систем зі спільною пам'яттю не складніше розробки для жорстко заданих систем.

– Програмне забезпечення на C++ не використовує монітори і тому може не копіювати спільні ресурси, які тільки зчитуються. Програмне забезпечення на Java вирішує задачу взаємного виключення за допомогою монітору і тому копіювання спільних ресурсів обов'язкове навіть для ресурсів, що тільки зчитуються.

3 Тестування програмного забезпечення

3.1 Методика тестування

В якості критеріїв порівняння двох реалізацій одного алгоритму обрано коефіцієнт прискорення $K_{\text{П}}$ та коефіцієнт ефективності $K_{\text{Е}}$.

Коефіцієнт прискорення показує, у скільки разів менше часу займає виконання паралельної програми в паралельній обчислювальній системі з P процесорами у порівнянні з виконанням послідовної програми в однопроцесорній системі:

$$K_{\text{П}}(P) = \frac{T_1}{P}.$$

Коефіцієнт ефективності показує середній рівень завантаження процесорів при виконанні програми в паралельній обчислювальній системі:

$$K_{\text{Е}}(P) = \frac{K_{\text{П}}(P)}{T_P}$$

Для тестування використовувалась паралельна обчислювальна система з наступним апаратним забезпеченням:

– процесор: Intel Core i5-750 (2,66 ГГц, 4 ядра, 8 Мб кешу третього рівня);

– оперативна пам'ять: DDR3 1600 МГц, 2 $\times$ 2048 Мб.

В якості програмного забезпечення використовувались:

– операційна система: Debian GNU/Linux «Squeeze» 6.0 x86_64;

– компілятор C++: g++ 4.4.4;

– реалізація WinAPI: Wine 1.1.32;

– компілятор та віртуальна машина Java: Sun Java 1.6.0_20 64-бітна версія.

Для виконання програми на заданій кількості ядер використовувалась програма schedtool, призначена для налаштування параметрів планувальника ОС Linux.

Для вимірювання часу виконання використовувались засоби, доступні у вибраній мові програмування. Всі виміри часу повторювались тричі та обчислювалось середнє значення часу виконання для заданих N та P.

3.2 Тестування програмного забезпечення на C++

Програмне забезпечення на C++ підтримує два режими роботи: з копіюванням спільних ресурсів, що тільки читаються, та без копіювання. Обидва режими тестуються окремо.

3.2.1 Тестування в режимі з копіюванням CP

Табл. 3.1 – Час виконання програми на C++ в режимі з копіюванням CP

N	T_1, мс	T_2, мс	T_3, мс	T_4, мс
1000	14257	7334	5646	4276
2000	123484	61996	46696	35408
2500	258262	130752	96606	73480
3000	459146	232891	173961	131069

Табл. 3.2 – Коефіцієнт прискорення програми на C++ в режимі з копіюванням CP

N	$K_{\text{П}}(1)$	$K_{\text{П}}(2)$	$K_{\text{П}}(3)$	$K_{\text{П}}(4)$
1000	1	1,94	2,53	3,33
2000	1	1,99	2,64	3,49
2500	1	1,98	2,67	3,51
3000	1	1,97	2,64	3,50

Табл. 3.3 – Коефіцієнт ефективності програми на C++ в режимі з копіюванням CP

N	$K_{\text{E}}(1)$	$K_{\text{E}}(2)$	$K_{\text{E}}(3)$	$K_{\text{E}}(4)$
1000	1	0,97	0,84	0,83
2000	1	0,99	0,88	0,87
2500	1	0,99	0,89	0,88
3000	1	0,99	0,88	0,88

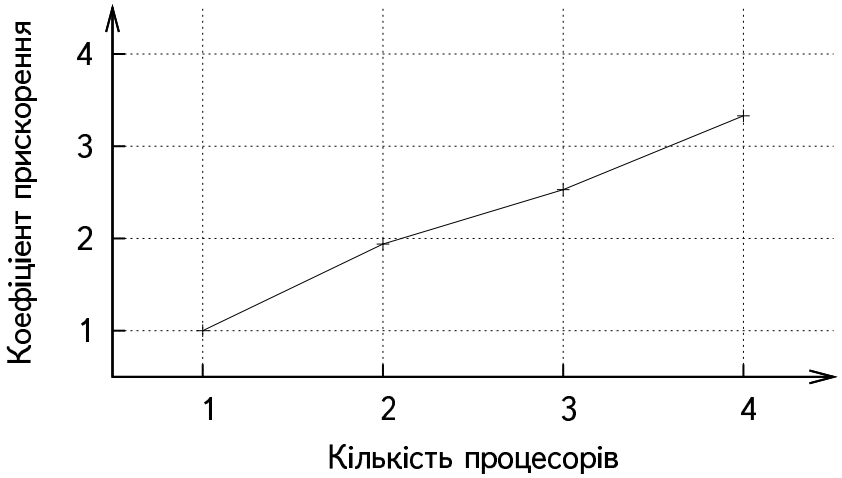

Рис. 3.1 – Коефіцієнт прискорення програми на C++ в режимі з копіюванням CP при $N = 1000$

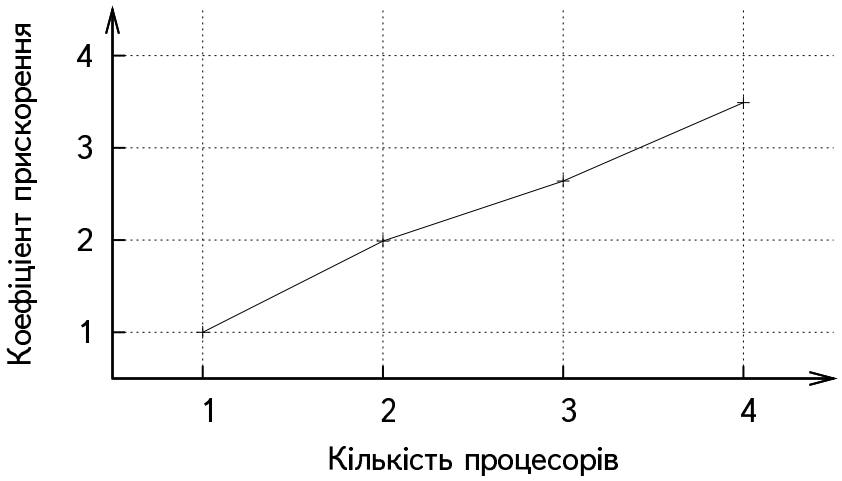

Рис. 3.2 – Коефіцієнт прискорення програми на C++ в режимі з копіюванням CP при $N = 2000$

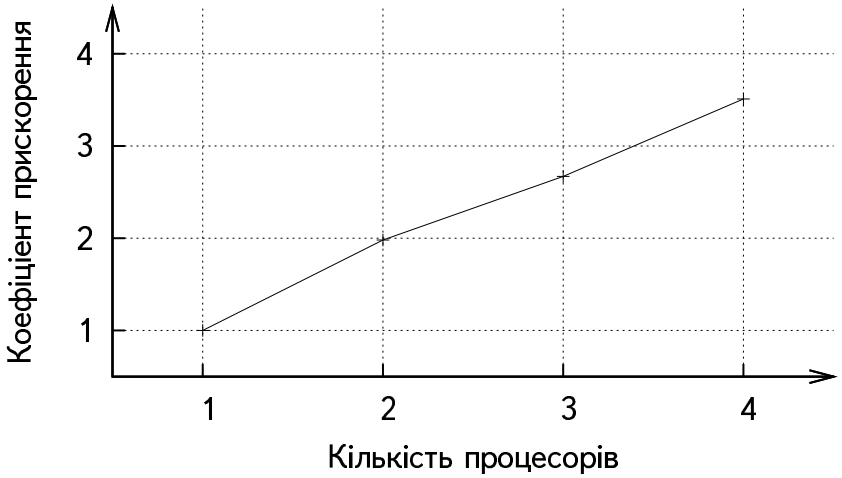

Рис. 3.3 – Коефіцієнт прискорення програми на C++ в режимі з копіюванням CP при $N = 2500$

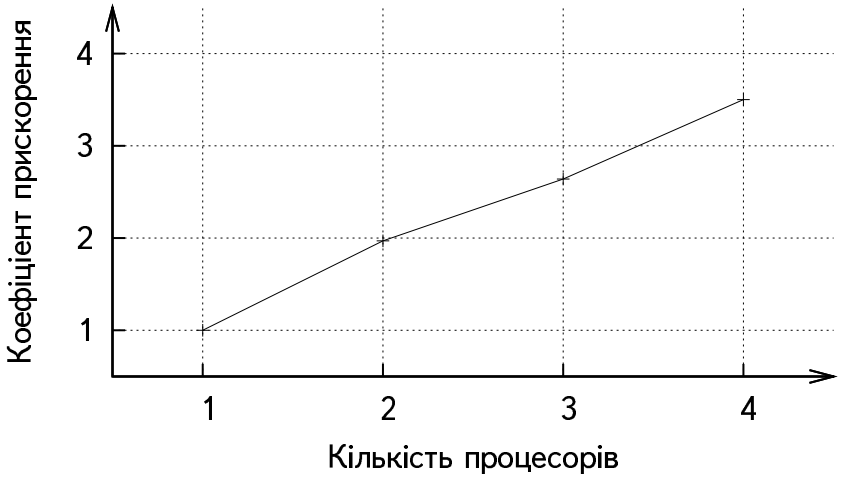

Рис. 3.4 – Коефіцієнт прискорення програми на C++ в режимі з копіюванням CP при $N = 3000$

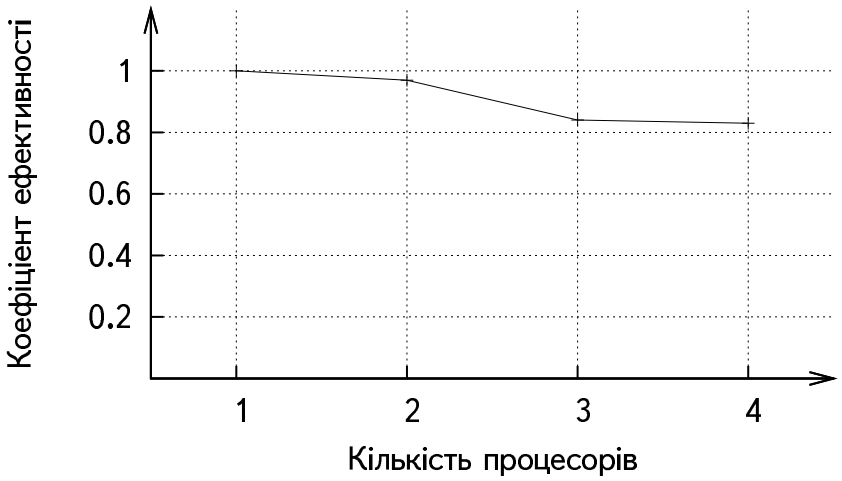

Рис. 3.5 – Коефіцієнт ефективності програми на C++ в режимі з копіюванням CP при $N = 1000$

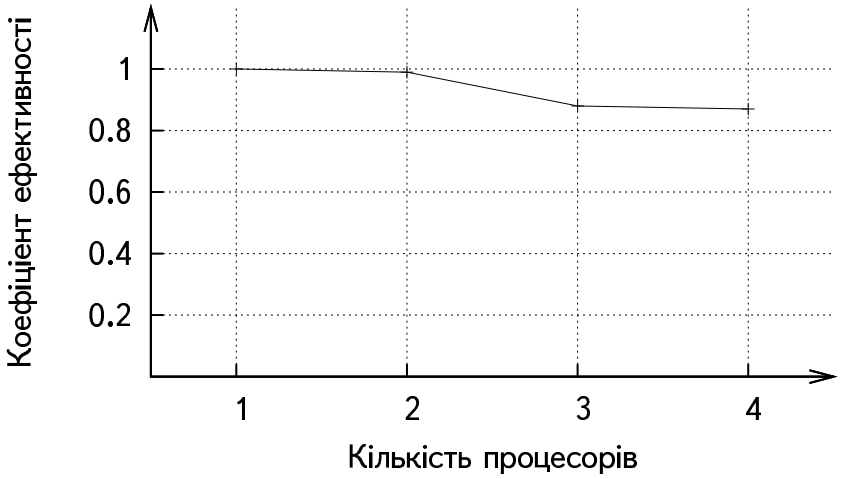

Рис. 3.6 – Коефіцієнт ефективності програми на C++ в режимі з копіюванням CP при $N = 2000$

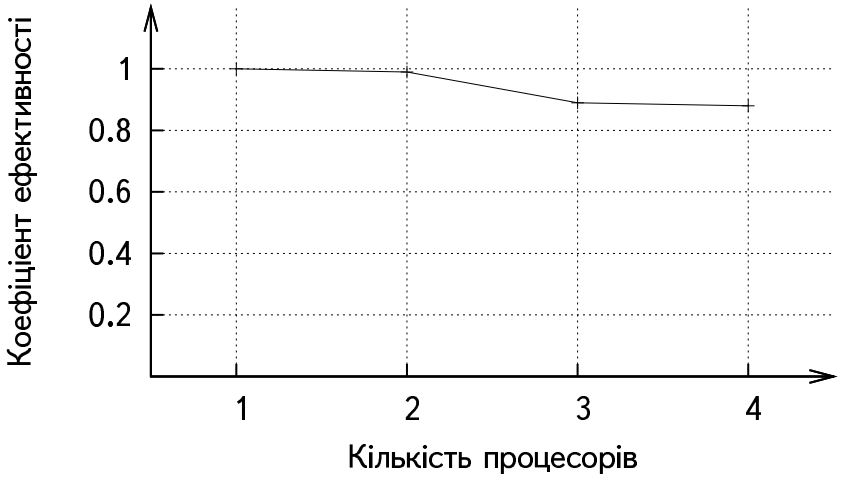

Рис. 3.7 – Коефіцієнт ефективності програми на C++ в режимі з копіюванням CP при $N = 2500$

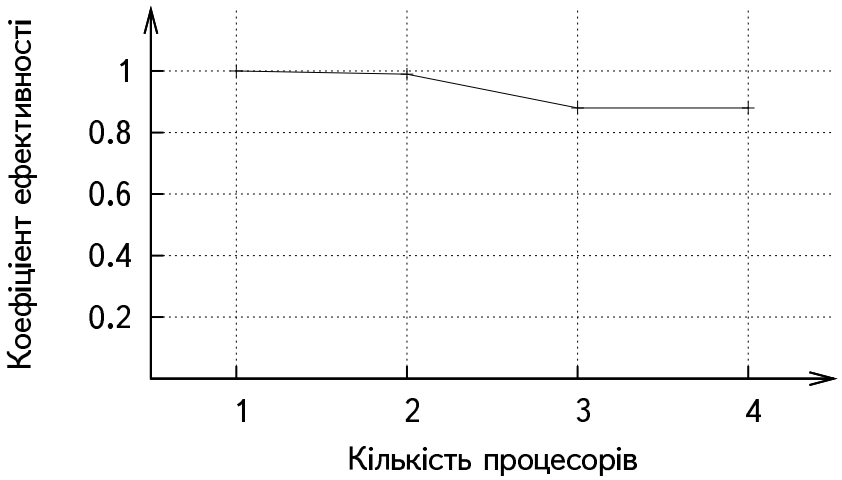

Рис. 3.8 – Коефіцієнт ефективності програми на C++ в режимі з копіюванням CP при $N = 3000$

3.2.2 Тестування в режимі без копіювання CP

Табл. 3.4 – Час виконання програми на C++ в режимі без копіювання CP

N	T_1, мс	T_2, мс	T_3, мс	T_4, мс
1000	14239	7159	5419	4069
2000	123332	62029	46533	34860
2500	256712	129560	95198	71783
3000	457806	230066	171188	128708

Табл. 3.5 – Коефіцієнт прискорення програми на C++ в режимі без копіювання CP

N	$K_{\text{П}}(1)$	$K_{\text{П}}(2)$	$K_{\text{П}}(3)$	$K_{\text{П}}(4)$
1000	1	1,99	2,63	3,50
2000	1	1,99	2,65	3,54
2500	1	1,98	2,70	3,58
3000	1	1,99	2,67	3,56

Табл. 3.6 – Коефіцієнт ефективності програми на C++ в режимі без копіювання CP

N	$K_{\text{E}}(1)$	$K_{\text{E}}(2)$	$K_{\text{E}}(3)$	$K_{\text{E}}(4)$
1000	1	0,97	0,88	0,88
2000	1	0,99	0,88	0,88
2500	1	0,99	0,90	0,90
3000	1	0,99	0,90	0,89

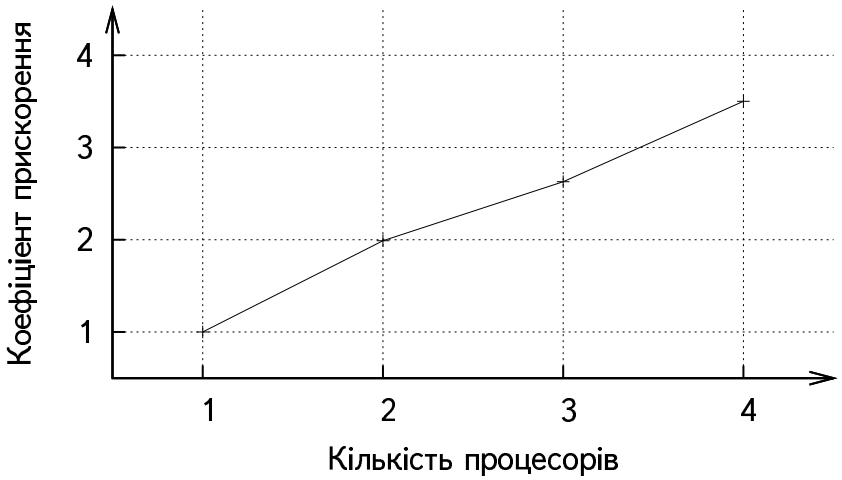

Рис. 3.9 – Коефіцієнт прискорення програми на C++ в режимі без копіювання CP при $N = 1000$

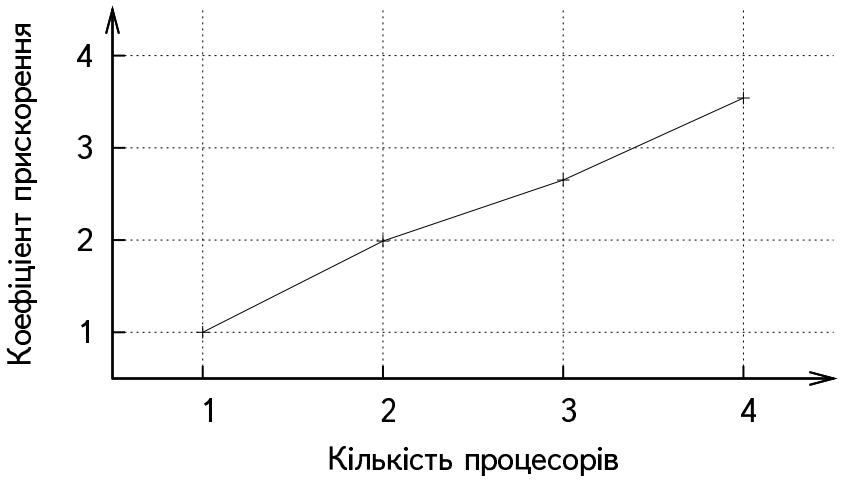

Рис. 3.10 – Коефіцієнт прискорення програми на C++ в режимі без копіювання CP при $N = 2000$

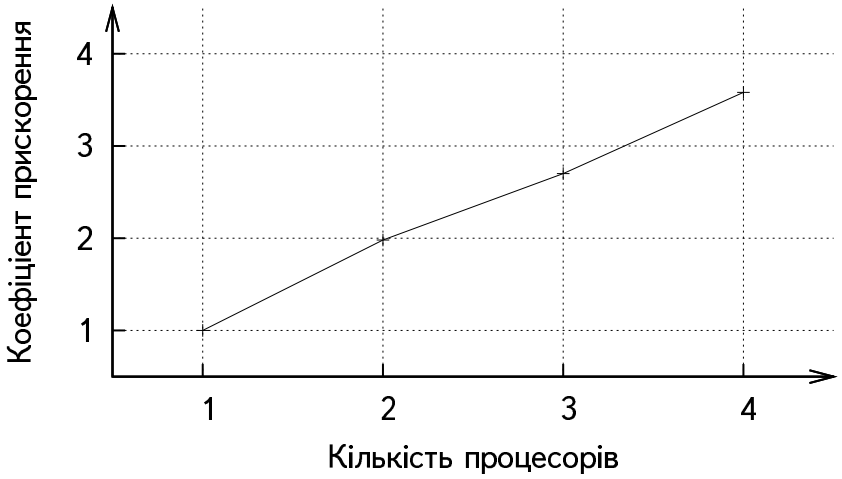

Рис. 3.11 – Коефіцієнт прискорення програми на C++ в режимі без копіювання CP при $N = 2500$

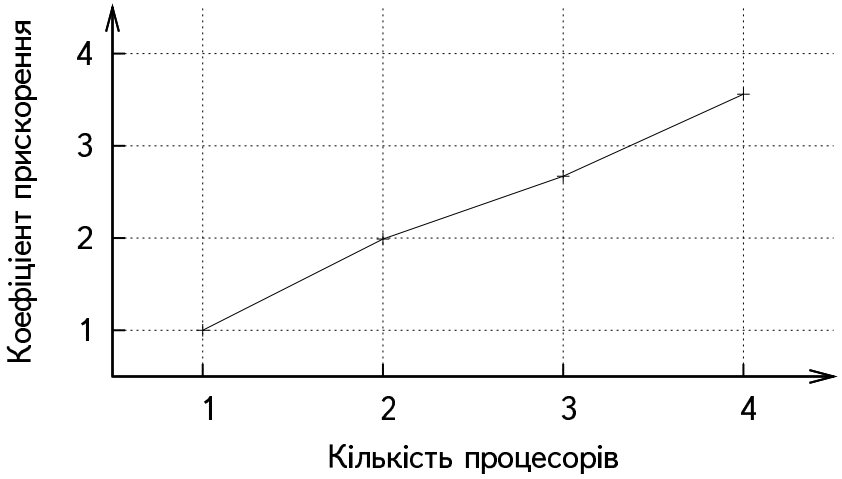

Рис. 3.12 – Коефіцієнт прискорення програми на C++ в режимі без копіювання CP при $N = 3000$

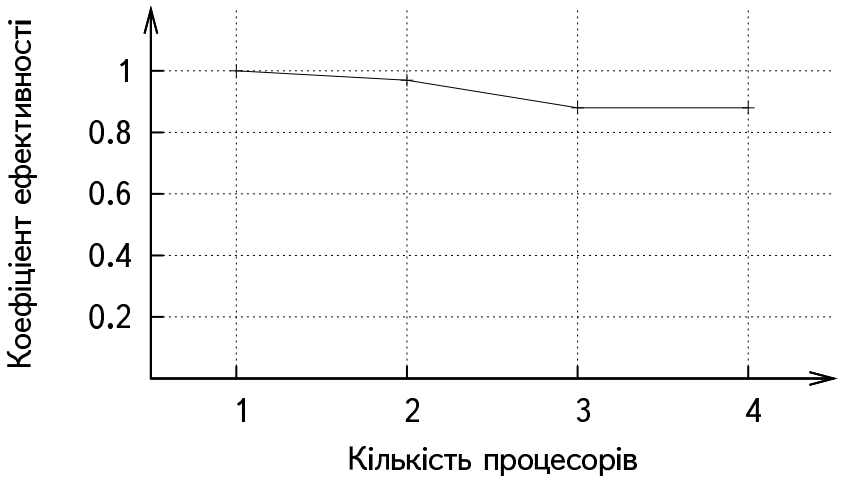

Рис. 3.13 – Коефіцієнт ефективності програми на C++ в режимі без копіювання CP при $N = 1000$

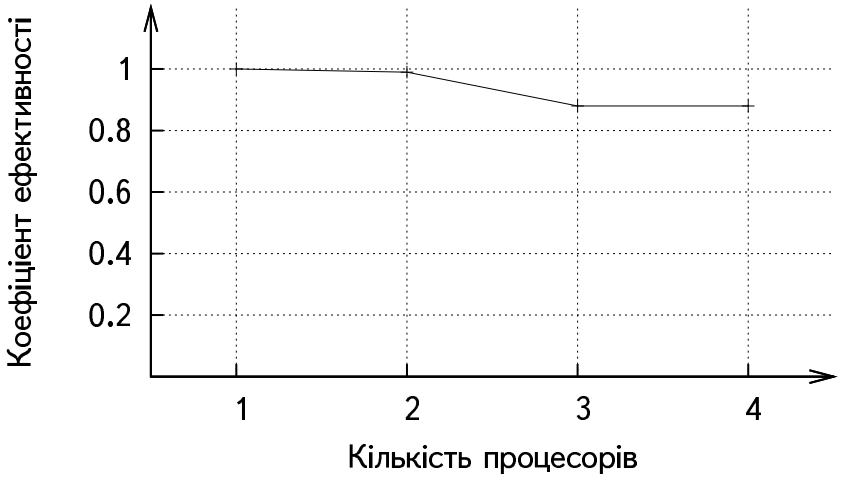

Рис. 3.14 – Коефіцієнт ефективності програми на C++ в режимі без копіювання CP при $N = 2000$

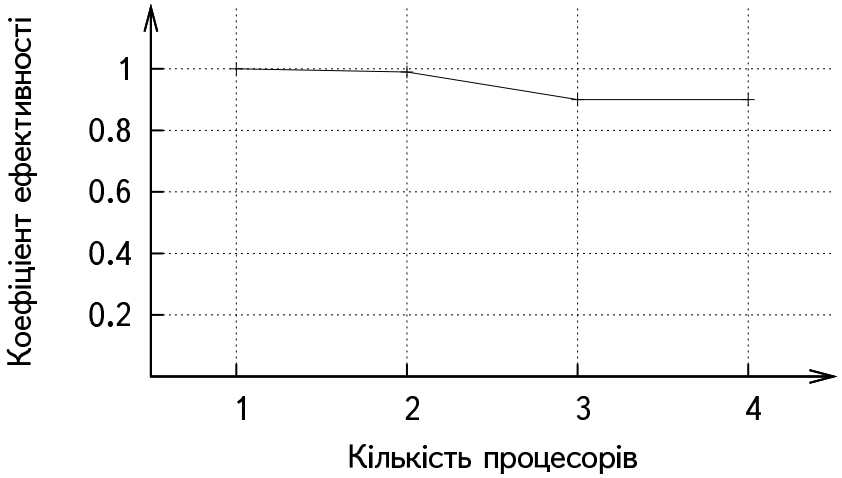

Рис. 3.15 – Коефіцієнт ефективності програми на C++ в режимі без копіювання CP при $N = 2500$

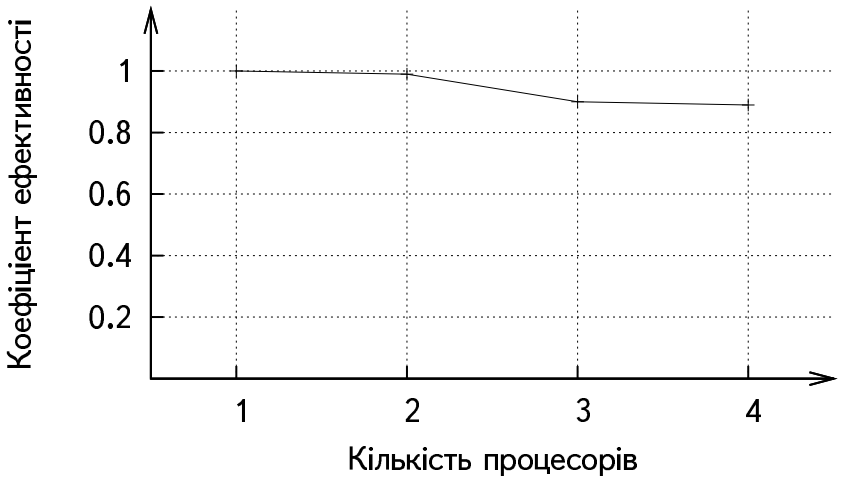

Рис. 3.16 – Коефіцієнт ефективності програми на C++ в режимі без копіювання CP при $N = 3000$

3.3 Тестування програмного забезпечення на Java

Табл. 3.7 – Час виконання програми на Java

N	T_1, мс	T_2, мс	T_3, мс	T_4, мс
1000	15241	7695	5881	4464
2000	127457	64617	49080	36828
2500	255633	129667	97717	74030
3000	466055	233835	178175	152355

Табл. 3.8 – Коефіцієнт прискорення програми на Java

N	$K_{\text{П}}(1)$	$K_{\text{П}}(2)$	$K_{\text{П}}(3)$	$K_{\text{П}}(4)$
1000	1	1,98	2,59	3,41
2000	1	1,97	2,60	3,46
2500	1	1,97	2,62	3,45
3000	1	1,99	2,62	3,06

Табл. 3.9 – Коефіцієнт ефективності програми на Java

N	$K_{\text{E}}(1)$	$K_{\text{E}}(2)$	$K_{\text{E}}(3)$	$K_{\text{E}}(4)$
1000	1	0,99	0,86	0,85
2000	1	0,99	0,87	0,87
2500	1	0,99	0,87	0,86
3000	1	0,99	0,87	0,76

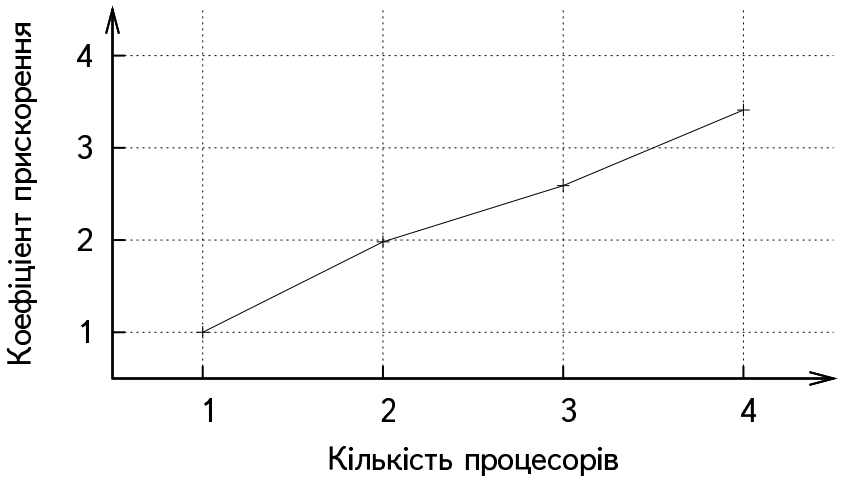

Рис. 3.17 – Коефіцієнт прискорення програми на Java при $N = 1000$

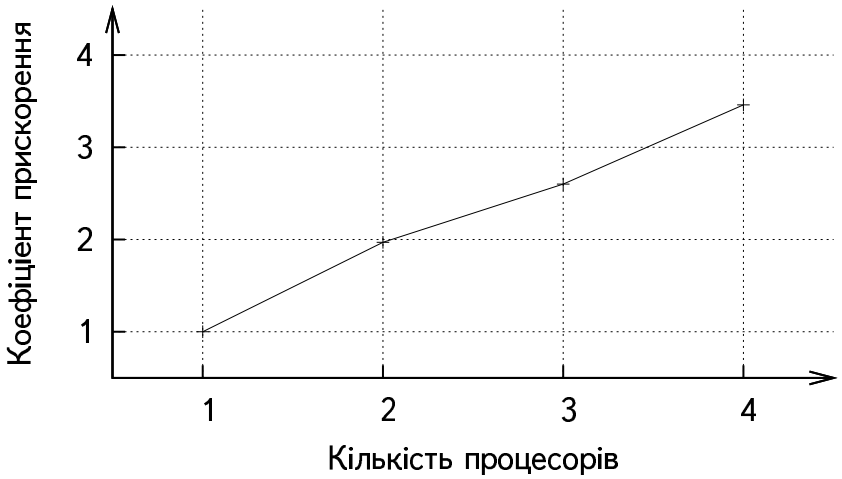

Рис. 3.18 – Коефіцієнт прискорення програми на Java при $N = 2000$

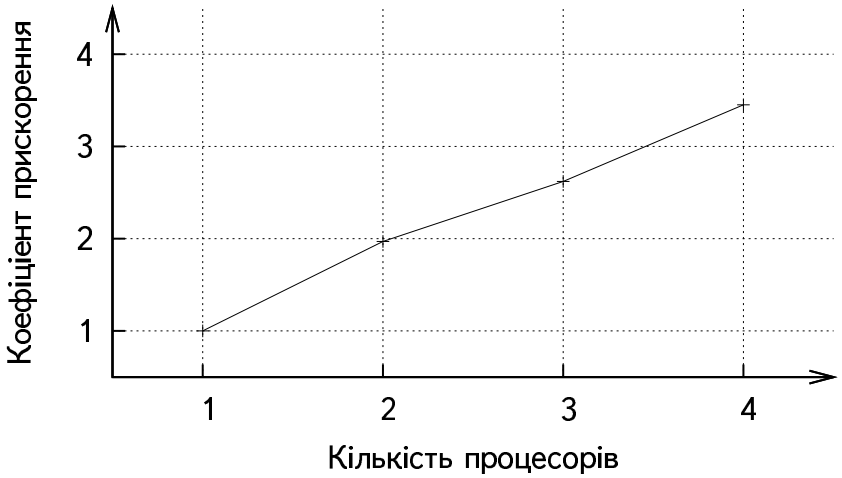

Рис. 3.19 – Коефіцієнт прискорення програми на Java при $N = 2500$

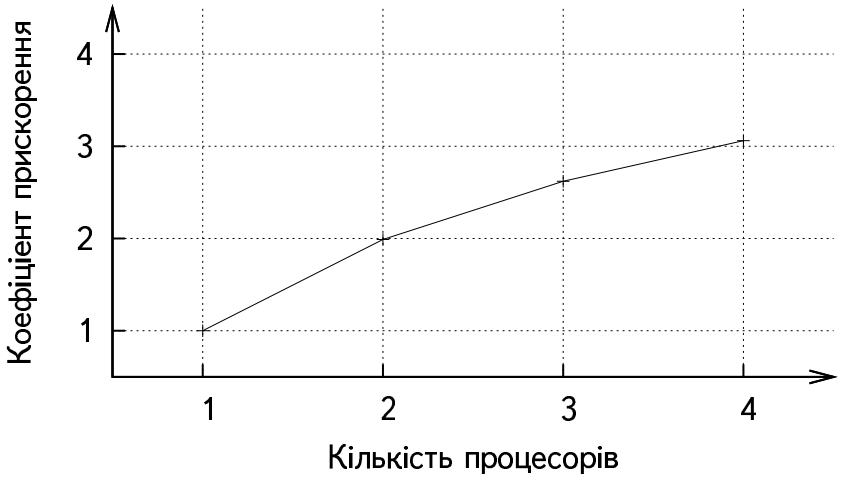

Рис. 3.20 – Коефіцієнт прискорення програми на Java при $N = 3000$

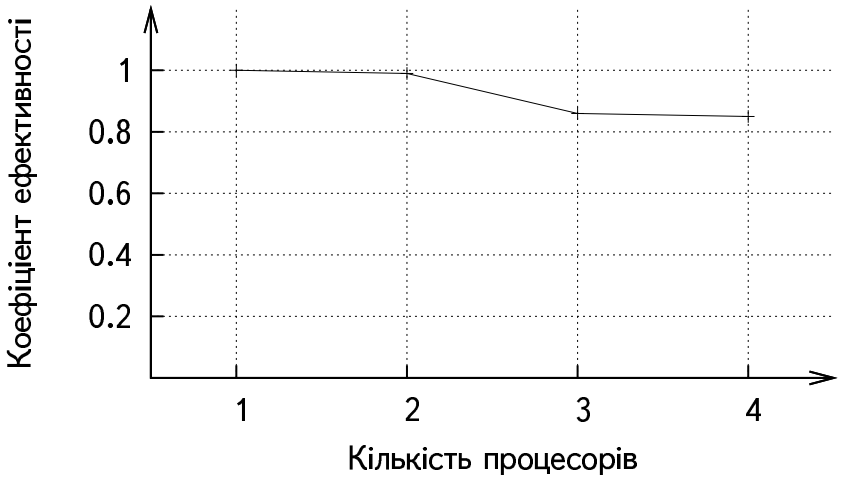

Рис. 3.21 – Коефіцієнт ефективності програми на Java при $N = 1000$

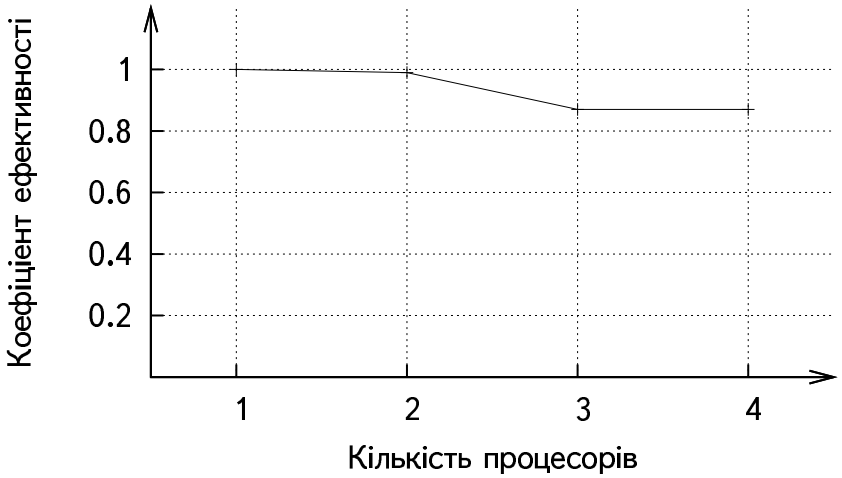

Рис. 3.22 – Коефіцієнт ефективності програми на Java при $N = 2000$

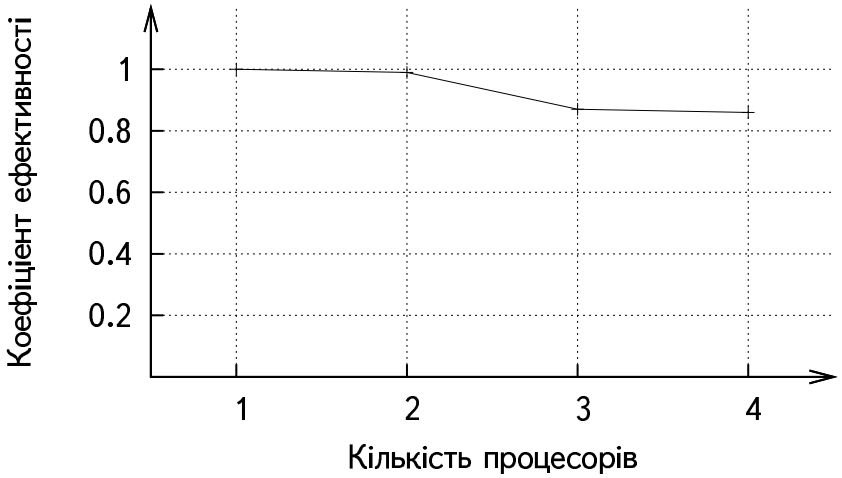

Рис. 3.23 – Коефіцієнт ефективності програми на Java при $N = 2500$

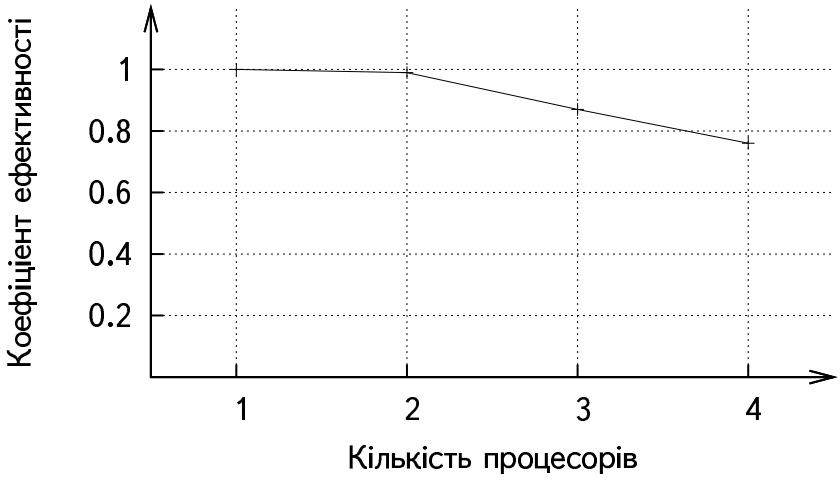

Рис. 3.24 – Коефіцієнт ефективності програми на Java при $N = 3000$

3.4 Висновки

– Тестування програмного забезпечення показало можливість прискорення виконання заданих векторно-матричних операцій в паралельній обчислювальній системі, при чому коефіцієнт прискорення K_Π досягав 3,58 при $P = 4$.

– Коефіцієнти прискорення програми на C++ з копіюванням спільних ресурсів виявились нижче, ніж у програми без копіювання. Цей результат можна пояснити так: всі ядра процесору Intel Core i5-750 мають доступ до спільної пам'яті через спільний контролер в північному мосту, а тому комп'ютерна система є дійсною SMP-системою з рівномірним доступом до пам'яті (на відміну від розглянутих у другому розділі багатоядерних процесорів AMD). Тому копіювання спільних ресурсів, що тільки читаються, призводить тільки до збільшення використання пам'яті в даній системі.

– Високі значення коефіцієнту прискорення $K_\Pi \approx P$ для $P = 2$ підтверджують результати теоретичного аналізу даної задачі: окремі кроки задачі є повністю паралельними, а кількість дій, зв'язаних з синхронізацією, є дуже малою.

– При збільшенні кількості ядер $P > 2$ коефіцієнт прискорення зменшується. Це в основному пов'язано з тим, що пропускна здатність пам'яті ділиться між всіма ядрами та під час запитів до пам'яті ядра простоюють. Крім того, деякий час витрачається на синхронізацю, але в даній програмі цей час є дуже малим.

– Так як кількість дій в програмі, зв'язаних з синхронізацією, є дуже малою, то порівняти ефективність бібліотеки Win32 та моніторів Java не є можливим.

– Час виконання програм на C++ та Java сильно не відрізняться. Це можна пояснити тим, що JVM від Sun компілює байт код в процесорні команди замість інтерпретації (Just in time compiler), а з-за специфіки програми збирач сміття не виконує ніякої роботи і тому не навантажує процесор.

Висновки

- Подальше підвищення тактової частоти мікропроцесорів стало неможливим з-за досягнення деяких фізичних обмежень. Випуск багатоядерних процесорів є природнім рішенням даної проблеми.
- Кеш-пам'ять багатоядерних процесорів AMD є невключаючою, тому її корисний розмір рівний її реальному розміру.
- Багатоядерні процесори AMD базуються на архітектурі DirectConnect, що призводить до того, що в рамках одного кристалу створюється NUMA-система.
- Розробка програми, що вирішує задачі взаємного виключення та синхронізації за допомогою моніторів, в цілому є простішою через централізацію в моніторі контролю за спільними ресурсами та організації взаємодії процесів.
- Коефіцієнт прискорення розробленого програмного забезпечення є досить високим з-за того, що задані векторно-матричні операції в цілому є частково-паралельними, але окремі кроки є повністю паралельними.
- Програма, що копіює спільні ресурси, які тільки читаються, працює повільніше в SMP-системах, ніж програма, яка не копіює.
- Падіння коефіцієнту прискорення зі збільшенням кількості ядер зв'язано в основному з недостатньою пропускною здатністю пам'яті.

Література

1 *Moore G. E.* Cramming more components onto integrated circuits [Текст] // *Electronics*. — 1965. — Т. 38, № 8.

2 The nanoelectronic road ahead: Despite challenges, silicon offers 20 more years of semiconductor progress [Електронний ресурс]. — Режим доступу: http://gtresearchnews.gatech.edu/newsrelease/FUTURECHIP.html. — Останнє звернення: 01.04.2010. — Назва з екрану.

3 Multi-core processors — the next evolution in computing [Електронний ресурс]. — Режим доступу: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/Multi-Core_Processing_33211A.pdf. — Останнє звернення: 01.04.2010. — Назва з екрану.

4 From a few cores to many: a tera-scale computing research overview [Електронний ресурс]. — Режим доступу: http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf. — Останнє звернення: 01.04.2010. — Назва з екрану.

5 Двухъядерные процессоры Intel и AMD: теория, часть 1 [Електронний ресурс]. — Режим доступу: http://www.ferra.ru/online/processors/s25920/. — Останнє звернення: 01.04.2010. — Назва з екрану.

6 Auto-vectorization in GCC [Electronic resource]. — Access mode: http://gcc.gnu.org/projects/tree-ssa/vectorization.html. — Last access: 04.04.2010. — Title from the screen.

7 *Naishlos Dorit*. Autovectorization in GCC [Text] // *GCC developers' summit 2004*. — 2004.

8 Power4 system microarchitecture [Електронний ресурс]. — Режим доступу: http://www-03.ibm.com/systems/resources/systems_p_hardware_whitepapers_power4.pdf. — Останнє звернення: 01.04.2010. — Назва з екрану.

9 *Parallax Inc*. — Propeller P8X32A Datasheet. http://www.parallax.com/Portals/0/Downloads/docs/prod/prop/PropellerDatasheet-v1.2.pdf.

10 AMD's dual-core Opteron processors [Electronic resource]. — Access mode: http://techreport.com/articles.x/8236/1. — Last access: 04.04.2010. — Title from the screen.

11 AMD unveils 64-bit Opteron processor [Electronic resource]. — Access mode: http://www.pcpro.co.uk/news/40992/amd-unveils-64-bit-opteron-processor. — Last access: 06.04.2010. — Title from the screen.

12 AMD64 architecture programmer's manual volume 1: Application programming [Electronic resource]. — Access mode: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf. — Last access: 06.04.2010. — Title from the screen.

13 AMD64 architecture programmer's manual volume 2: System programming [Electronic resource]. — Access mode: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf. — Last access: 06.04.2010. — Title from the screen.

14 Inside AMD64 architecture [Electronic resource]. — Access mode: http://www.hardwaresecrets.com/article/324. — Last access: 07.04.2010. — Title from the screen.

15 *Drepper Ulrich*. What every programmer should know about memory [Text]. — 2007. http://people.redhat.com/drepper/cpumemory.pdf.

16 Inside AMD K10 architecture [Electronic resource]. — Access mode: http://www.hardwaresecrets.com/article/480. — Last access: 05.04.2010. — Title from the screen.

17 Intel 64 and ia-32 architectures software developer's manual volume 1: Basic architecture [Electronic resource]. — Access mode: http://www.intel.com/Assets/PDF/manual/253665.pdf. — Last access: 06.04.2010. — Title from the screen.

18 AMD K10 micro-architecture [Electronic resource]. — Access mode: http://www.xbitlabs.com/articles/cpu/display/amd-k10.html. — Last access: 07.04.2010. — Title from the screen.

19 AMD64 architecture programmer's manual volume 3: General purpose and system instructions [Electronic resource]. — Access mode: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf. — Last access: 06.04.2010. — Title from the screen.

20 «Magny-Cours» and Direct Connect architecture 2.0 [Electronic resource]. — Access mode: http://developer.amd.com/documentation/articles/pages/Magny-Cours-Direct-Connect-Architecture-2.0.aspx. — Last access: 05.04.2010. — Title from the screen.

21 HyperTransport [Electronic resource]. — Access mode: http://offline.computerra.ru/2004/547/34188/. — Last access: 06.04.2010. — Title from the screen.

22 Desktop processor solutions [Electronic resource]. — Access mode: http://products.amd.com/en-us/DesktopCPUResult.aspx. — Last access: 08.04.2010. — Title from the screen.

23 Phenom ii x3 - enable and unlock the 4th core [Electronic resource]. — Access mode: http://www.guru3d.com/news/phenom-ii-x3–enable-the-4th-core/. — Last access: 07.04.2010. — Title from the screen.

24 AMD Phenom II key architectural features [Electronic resource]. — Access mode: http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-key-architectural-features.aspx. — Last access: 08.04.2010. — Title from the screen.

25 AMD's Turbo Core technology [Electronic resource]. — Access mode: http://www.pcper.com/article.php?aid=897. — Last access: 08.04.2010. — Title from the screen.

26 AMD Phenom II X6 1090T and 1055T six-core processor review [Electronic resource]. — Access mode: http://www.legitreviews.com/article/1289/1/. — Last access: 09.04.2010. — Title from the screen.

27 AMD Cool'n'Quiet technology [Electronic resource]. — Access mode: http://www.amd.com/us/products/technologies/cool-n-quiet/Pages/cool-n-quiet.aspx. — Last access: 09.04.2010. — Title from the screen.

28 HyperTransport technology I/O link [Електронний ресурс]. — Режим доступу: http://www.hypertransport.org/docs/wp/25012A_HTWhite_Paper_v1.1.pdf. — Останнє звернення: 01.04.2010. — Назва з екрану.

29 HyperTransport I/O technology overview [Електронний ресурс]. — Режим доступу: http://www.hypertransport.org/docs/wp/HT_Overview.pdf. — Останнє звернення: 01.04.2010. — Назва з екрану.

30 Meeting the I/O bandwidth challenge: How HyperTransport technology accelerates performance in key applications [Electronic resource]. — Access mode: http://www.hypertransport.org/docs/wp/HT_Meeting_IO_Challenge.pdf. — Last access: 02.04.2010. — Title from the screen.

31 *Жуков І. А. Корочкін О. В.* Паралельні та розподілені обчислення [Текст]. — Корнійчук, 2005. — ISBN 996-7599-36-1.

Додаток А. Структура ПОС зі спільною пам'яттю

Додаток Б. Алгоритм основної процедури

Додаток В. Алгоритм задач

Додаток Г. Лістинг програми на C++

```
g++ (Debian 4.4.4-1) 4.4.4
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

main.cc

```
  1. /* ----------------------------------------------------------------------
  2.  * Паралельні та розподілені обчислення
  3.  * Бібліотека Win32
  4.  * Завдання:
  5.  *   MA = max(MO) * (MR * (MX * MT))
  6.  * Виконав: Грибенко Дмитро В'ячеславович
  7.  * Sat May  1 21:42:06 EEST 2010
  8.  * ---------------------------------------------------------------------- */
  9.
 10. #include <iostream>
 11. #include <iomanip>
 12.
 13. #include <limits.h>
 14.
 15. #include <windows.h>
 16.
 17. #include "matrix.h"
 18. #include "functions.h"
 19.
 20. using std::cout;
 21. using std::cin;
 22. using std::endl;
 23. using std::flush;
 24.
 25. const size_t N = 10;
 26. const unsigned int P = 4;
 27. const size_t H = N / P;
 28.
 29. const bool copy_shared_readonly = true;
 30.
 31. matrix MA(N);
 32. matrix MO(N);
 33. matrix MR(N);
 34. matrix MX(N);
 35. matrix MT(N);
 36. matrix MZ(N);
 37.
 38. int alpha = INT_MIN;
 39.
 40. HANDLE ev_input_1;
 41. HANDLE ev_input_2;
 42. CRITICAL_SECTION cs_alpha;
 43. CRITICAL_SECTION cs_MT;
 44. CRITICAL_SECTION cs_MZ;
 45. HANDLE ev_comp2_end[N];
 46. HANDLE sem_comp3_end;
 47.
 48. /* ---------------------------------------------------------------------- */
 49. DWORD WINAPI thread_proc(void *lpParameter)
 50. {
 51.   int tid = (int) lpParameter;
 52.   int start_index = tid * H;
 53.   int end_index = (tid != P - 1) ? (tid + 1) * H : N;
 54.   cout << "T" << tid << ": Started." << endl;
 55.   if(tid == 0)
 56.   {
 57.     cout << "T" << tid << ": Reading data..." << endl;
 58.     MO.fill();
 59.     MR.fill();
 60.     SetEvent(ev_input_1);
 61.   }
 62.   if(tid == P - 1)
 63.   {
 64.     cout << "T" << tid << ": Reading data..." << endl;
 65.     MX.fill();
 66.     MT.fill();
 67.     SetEvent(ev_input_2);
 68.   }
 69.
 70.   cout << "T" << tid << ": Waiting for data..." << endl;
 71.   if(tid != 0)
```

```
72.   {
73.     WaitForSingleObject(ev_input_1, INFINITE);
74.   }
75.   if(tid != P - 1)
76.   {
77.     WaitForSingleObject(ev_input_2, INFINITE);
78.   }
79.
80.   cout << "T" << tid << ": comp1" << endl;
81.   int alpha_local = comp1(start_index, end_index, MO);
82.
83.   EnterCriticalSection(&cs_alpha);
84.   if(alpha_local > alpha)
85.   {
86.     alpha = alpha_local;
87.   }
88.   LeaveCriticalSection(&cs_alpha);
89.
90.   cout << "T" << tid << ": comp2" << endl;
91.   if(copy_shared_readonly)
92.   {
93.     EnterCriticalSection(&cs_MT);
94.     matrix MT_copy(MT);
95.     LeaveCriticalSection(&cs_MT);
96.
97.     comp2(start_index, end_index, MZ, MX, MT_copy);
98.   }
99.   else
100.  {
101.    comp2(start_index, end_index, MZ, MX, MT);
102.  }
103.
104.  SetEvent(ev_comp2_end[tid]);
105.  WaitForMultipleObjects(P, ev_comp2_end, TRUE, INFINITE);
106.
107.  cout << "T" << tid << ": comp3" << endl;
108.  if(copy_shared_readonly)
109.  {
110.    EnterCriticalSection(&cs_MZ);
111.    matrix MZ_copy(MZ);
112.    LeaveCriticalSection(&cs_MZ);
113.
114.    EnterCriticalSection(&cs_alpha);
115.    int alpha_copy = alpha;
116.    LeaveCriticalSection(&cs_alpha);
117.
118.    comp3(start_index, end_index, MA, alpha_copy, MR, MZ_copy);
119.  }
120.  else
121.  {
122.    comp3(start_index, end_index, MA, alpha, MR, MZ);
123.  }
124.
125.  if(tid == 0)
126.  {
127.    for(size_t i = 0; i < P - 1; i++)
128.    {
129.      WaitForSingleObject(sem_comp3_end, INFINITE);
130.    }
131.
132.    if(N <= 40)
133.    {
134.      cout << "T" << tid << ": Result:" << endl
135.           << MA << endl;
136.    }
137.  }
138.  else
139.  {
140.    ReleaseSemaphore(sem_comp3_end, 1, NULL);
141.  }
142.
143.  cout << "T" << tid << ": Finished." << endl;
144.  return 0;
145. }
146. /* ---------------------------------------------------------------------- */
147.
148. int main(int argc, char* argv[])
149. {
150.   ev_input_1 = CreateEvent(NULL, TRUE, FALSE, NULL);
```

```
151.   ev_input_2 = CreateEvent(NULL, TRUE, FALSE, NULL);
152.   InitializeCriticalSection(&cs_alpha);
153.   InitializeCriticalSection(&cs_MT);
154.   InitializeCriticalSection(&cs_MZ);
155.   sem_comp3_end = CreateSemaphore(NULL, 0, P - 1, NULL);
156.
157.   for(size_t i = 0; i < P; i++)
158.   {
159.     ev_comp2_end[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
160.   }
161.
162.   LARGE_INTEGER performace_frequency;
163.   LARGE_INTEGER start_time;
164.   LARGE_INTEGER end_time;
165.
166.   QueryPerformanceFrequency(&performace_frequency);
167.   QueryPerformanceCounter(&start_time);
168.
169.   alpha = INT_MIN;
170.
171.   SIZE_T stack_size = 0;
172.   HANDLE T[P];
173.   for(size_t i = 0; i < P; i++)
174.   {
175.     T[i] = CreateThread(NULL, stack_size, thread_proc, (void *) i, 0, NULL);
176.   }
177.
178.   WaitForMultipleObjects(P, T, TRUE, INFINITE);
179.
180.   QueryPerformanceCounter(&end_time);
181.
182.   double elapsed_time =
183.       ((double) (end_time.QuadPart - start_time.QuadPart))
184.         / (double) performace_frequency.QuadPart;
185.
186.   cout << "main(): all tasks finished execution" << endl;
187.   cout << "main(): computations took "
188.        << (int) (elapsed_time * 1000) << " ms" << endl;
189.
190.   for(size_t i = 0; i < P; i++)
191.   {
192.     CloseHandle(T[i]);
193.   }
194.
195.   CloseHandle(ev_input_1);
196.   CloseHandle(ev_input_2);
197.   DeleteCriticalSection(&cs_alpha);
198.   DeleteCriticalSection(&cs_MT);
199.   DeleteCriticalSection(&cs_MZ);
200.   CloseHandle(sem_comp3_end);
201.
202.   for(size_t i = 0; i < P; i++)
203.   {
204.     CloseHandle(ev_comp2_end[i]);
205.   }
206. }
207.
No errors.


g++ (Debian 4.4.4-1) 4.4.4
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

matrix.h
  1. /* ------------------------------------------------------------------------
  2.  * Параллельные и распределённые вычисления
  3.  * Лабораторная работа 8.  MPI
  4.  * Задание:
  5.  *   MA = MB + MC * MO
  6.  * Выполнил: Грибенко Дмитрий Вячеславович
  7.  * Вск Апр 25 16:00:10 EEST 2010
  8.  * ------------------------------------------------------------------------ */
  9.
 10. #ifndef _MATRIX_H_
 11. #define _MATRIX_H_ 1
 12.
 13. #include <istream>
```

```
 14. #include <ostream>
 15. #include <assert.h>
 16.
 17. class matrix
 18. {
 19.   public:
 20.     matrix(size_t N);
 21.     matrix(const matrix &other);
 22.     virtual ~matrix();
 23.
 24.     int get(size_t i, size_t j) const
 25.     {
 26.       assert(i < this->N);
 27.       assert(j < this->N);
 28.       return this->data[i * this->N + j];
 29.     }
 30.
 31.     void set(size_t i, size_t j, int value)
 32.     {
 33.       assert(i < this->N);
 34.       assert(j < this->N);
 35.       this->data[i * this->N + j] = value;
 36.     }
 37.
 38.     void fill();
 39.
 40.     const size_t N;
 41.     int *data;
 42.
 43.   private:
 44.     friend std::istream &operator>>(std::istream &istr, matrix &m);
 45.     friend std::ostream &operator<<(std::ostream &ostr, const matrix &m);
 46.
 47. };
 48.
 49. std::ostream &operator<<(std::ostream &ostr, const matrix &m);
 50.
 51. #endif
 52.
No errors.


g++ (Debian 4.4.4-1) 4.4.4
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

matrix.cc
  1. /* ----------------------------------------------------------------------
  2.  * Паралельні та розподілені обчислення
  3.  * Бібліотека Win32
  4.  * Завдання:
  5.  *   MA = max(MO) * (MR * (MX * MT))
  6.  * Виконав: Грибенко Дмитро В'ячеславович
  7.  * Sat May  1 21:42:06 EEST 2010
  8.  * ---------------------------------------------------------------------- */
  9.
 10. #include "matrix.h"
 11.
 12. matrix::matrix(size_t N):
 13.   N(N),
 14.   data(new int[N * N])
 15. {
 16. }
 17.
 18. matrix::matrix(const matrix &other):
 19.   N(other.N),
 20.   data(new int[N * N])
 21. {
 22.   for(size_t i = 0; i < this->N * this->N; i++)
 23.   {
 24.     this->data[i] = other.data[i];
 25.   }
 26. }
 27.
 28. matrix::~matrix()
 29. {
 30.   delete [] data;
 31. }
```

```
 32.
 33. void matrix::fill()
 34. {
 35.   for(size_t i = 0; i < this->N; i++)
 36.   {
 37.     for(size_t j = 0; j < this->N; j++)
 38.     {
 39.       this->data[i * this->N + j] = 1;
 40.     }
 41.   }
 42. }
 43.
 44. std::ostream &operator<<(std::ostream &ostr, const matrix &m)
 45. {
 46.   for(size_t i = 0; i < m.N; i++)
 47.   {
 48.     for(size_t j = 0; j < m.N; j++)
 49.     {
 50.       ostr << m.data[i * m.N + j] << " ";
 51.     }
 52.     ostr << std::endl;
 53.   }
 54.   return ostr;
 55. }
 56.
No errors.


g++ (Debian 4.4.4-1) 4.4.4
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

functions.h
  1. /* ----------------------------------------------------------------------------
  2.  * Параллельные и распределённые вычисления
  3.  * Лабораторная работа 2.  Библиотека Win32
  4.  * Задание:
  5.  *   a = max(MB * alpha - beta * MO * MR)
  6.  * Выполнил: Грибенко Дмитрий Вячеславович
  7.  * Вск Фев 28 19:18:52 EET 2010
  8.  * ---------------------------------------------------------------------------- */
  9.
 10. #ifndef _FUNCTIONS_H_
 11. #define _FUNCTIONS_H_ 1
 12.
 13. #include "matrix.h"
 14.
 15. /* max(MO_H) */
 16. int comp1(
 17.     size_t start,
 18.     size_t end,
 19.     const matrix &MO);
 20.
 21. /* MZ_H = MX_H * MT */
 22. void comp2(
 23.     size_t start,
 24.     size_t end,
 25.     matrix &MZ,
 26.     const matrix &MX,
 27.     const matrix &MT);
 28.
 29. /* MA_H = alpha * MR_H * MZ */
 30. void comp3(
 31.     size_t start,
 32.     size_t end,
 33.     matrix &MA,
 34.     int alpha,
 35.     const matrix &MR,
 36.     const matrix &MZ);
 37.
 38. #endif
 39.
No errors.


g++ (Debian 4.4.4-1) 4.4.4
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
```

warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

functions.cc

```
/* ----------------------------------------------------------------------
 * Паралельні та розподілені обчислення
 * Бібліотека Win32
 * Завдання:
 *   MA = max(MO) * (MR * (MX * MT))
 * Виконав: Грибенко Дмитро В'ячеславович
 * Sat May  1 21:42:06 EEST 2010
 * ---------------------------------------------------------------------- */

#include <limits.h>

#include "functions.h"

/* max(MO_H) */
int comp1(
    size_t start,
    size_t end,
    const matrix &MO)
{
  int result = INT_MIN;
  for(size_t i = start; i < end; i++)
  {
    for(size_t j = 0; j < MO.N; j++)
    {
      int x = MO.get(i, j);
      if(x > result)
      {
        result = x;
      }
    }
  }
  return result;
}

/* MZ_H = MX_H * MT */
void comp2(
    size_t start,
    size_t end,
    matrix &MZ,
    const matrix &MX,
    const matrix &MT)
{
  for(size_t i = start; i < end; i++)
  {
    for(size_t j = 0; j < MZ.N; j++)
    {
      int acc = 0;
      for(size_t k = 0; k < MZ.N; k++)
      {
        acc += MX.get(i, k) * MT.get(k, j);
      }
      MZ.set(i, j, acc);
    }
  }
}

/* MA_H = alpha * MR_H * MZ */
void comp3(
    size_t start,
    size_t end,
    matrix &MA,
    int alpha,
    const matrix &MR,
    const matrix &MZ)
{
  for(size_t i = start; i < end; i++)
  {
    for(size_t j = 0; j < MA.N; j++)
    {
      int acc = 0;
      for(size_t k = 0; k < MA.N; k++)
      {
        acc += MR.get(i, k) * MZ.get(k, j);
      }
      acc *= alpha;
      MA.set(i, j, acc);
```

```
 77.     }
 78.   }
 79. }
 80.
No errors.
```

Додаток Д. Лістинг програми на Java

```
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) 64-Bit Server VM (build 16.3-b01, mixed mode)
```

Main.java

```
 1 /* ---------------------------------------------------------------------
 2  * Паралельні та розподілені обчислення
 3  * Java. Монітори
 4  * Завдання:
 5  *   MA = max(MO) * (MR * (MX * MT))
 6  * Виконав: Грибенко Дмитро В'ячеславович
 7  * Sat May  1 21:42:06 EEST 2010
 8  * --------------------------------------------------------------------- */
 9
10 public class Main
11 {
12   public static final int N = 3000;
13   public static final int P = 4;
14   public static final int H = N / P;
15
16   private static Matrix MO = new Matrix(N);
17   private static Matrix MR = new Matrix(N);
18   private static Matrix MX = new Matrix(N);
19
20   static class TaskControl
21   {
22     private int inputCount = 0;
23     private int comp2Count = 0;
24     private int comp3Count = 0;
25
26     private int alpha = Integer.MIN_VALUE;
27     private Matrix MT = new Matrix(N);
28     private Matrix MZ = new Matrix(N);
29     private Matrix MA = new Matrix(N);
30
31     public synchronized void waitInput()
32     {
33       try
34       {
35         while(inputCount != 2)
36         {
37           wait();
38         }
39       }
40       catch(InterruptedException e)
41       {
42       }
43     }
44
45     public synchronized void waitComp2()
46     {
47       try
48       {
49         while(comp2Count != P)
50         {
51           wait();
52         }
53       }
54       catch(InterruptedException e)
55       {
56       }
57     }
58
59     public synchronized void waitComp3AndOutputMA()
60     {
61       try
62       {
63         while(comp3Count != P)
64         {
65           wait();
66         }
67       }
68       catch(InterruptedException e)
69       {
70       }
71
```

```
72        if(N <= 40)
73        {
74          MA.write(System.out);
75        }
76      }
77
78      public synchronized void inputMT()
79      {
80        MT.fill();
81      }
82
83      public synchronized void copyMT(Matrix copy)
84      {
85        MT.copyTo(copy);
86      }
87
88      public synchronized void copyMZ(Matrix copy)
89      {
90        MZ.copyTo(copy);
91      }
92
93      public synchronized int copyAlpha()
94      {
95        return alpha;
96      }
97
98      public synchronized void signalInputDone()
99      {
100       inputCount++;
101       if(inputCount == 2)
102       {
103         notifyAll();
104       }
105     }
106
107     public synchronized void computeMaxAlpha(int alpha_i)
108     {
109       if(alpha_i > alpha)
110       {
111         alpha = alpha_i;
112       }
113     }
114
115     public synchronized void signalComp2AndSetMZPart(Matrix M, int start, int end)
116     {
117       comp2Count++;
118       if(comp2Count == P)
119       {
120         notifyAll();
121       }
122
123       for(int i = start; i < end; i++)
124       {
125         for(int j = 0; j < N; j++)
126         {
127           MZ.set(i, j, M.get(i, j));
128         }
129       }
130     }
131
132     public synchronized void signalComp3AndSetMAPart(Matrix M, int start, int end)
133     {
134       comp3Count++;
135       if(comp3Count == P)
136       {
137         notifyAll();
138       }
139
140       for(int i = start; i < end; i++)
141       {
142         for(int j = 0; j < N; j++)
143         {
144           MA.set(i, j, M.get(i, j));
145         }
146       }
147     }
148   }
149
150   /* ------------------------------------------------------------------------ */
```

```
151   static class TT extends Thread
152   {
153     private TaskControl mon;
154     private int tid;
155
156     public TT(TaskControl mon, int tid)
157     {
158       this.mon = mon;
159       this.tid = tid;
160     }
161
162     @Override
163     public void run()
164     {
165       int startIndex = tid * H;
166       int endIndex = (tid != P - 1) ? (tid + 1) * H : N;
167
168       System.out.println("T" + tid + ": Started.");
169
170       if(tid == 0)
171       {
172         System.out.println("T" + tid + ": Reading data...");
173         MO.fill();
174         MR.fill();
175         mon.signalInputDone();
176       }
177       if(tid == P - 1)
178       {
179         System.out.println("T" + tid + ": Reading data...");
180         MX.fill();
181         mon.inputMT();
182         mon.signalInputDone();
183       }
184
185       mon.waitInput();
186
187       System.out.println("T" + tid + ": comp1");
188       int alpha_local = Functions.comp1(startIndex, endIndex, MO);
189       mon.computeMaxAlpha(alpha_local);
190
191       System.out.println("T" + tid + ": comp2");
192       Matrix MZlocal = new Matrix(N);
193       Matrix MTcopy = new Matrix(N);
194       mon.copyMT(MTcopy);
195       Functions.comp2(startIndex, endIndex, MZlocal, MX, MTcopy);
196
197       mon.signalComp2AndSetMZPart(MZlocal, startIndex, endIndex);
198       mon.waitComp2();
199
200       System.out.println("T" + tid + ": comp3");
201       Matrix MAlocal = new Matrix(N);
202       mon.copyMZ(MZlocal);
203       int alphaCopy = mon.copyAlpha();
204       Functions.comp3(startIndex, endIndex, MAlocal, alphaCopy, MR, MZlocal);
205
206       mon.signalComp3AndSetMAPart(MAlocal, startIndex, endIndex);
207
208       if(tid == 0)
209       {
210         System.out.println("T" + tid + ": Result:");
211         mon.waitComp3AndOutputMA();
212       }
213
214       System.out.println("T" + tid + ": Finished.");
215     }
216   }
217   /* --------------------------------------------------------------------- */
218
219   public static void main(String[] args) throws InterruptedException
220   {
221     TaskControl mon = new TaskControl();
222     TT[] threads = new TT[P];
223     for(int i = 0; i < P; i++)
224     {
225       threads[i] = new TT(mon, i);
226     }
227
228     long startTime = System.currentTimeMillis();
229
```

```
   230     for(int i = 0; i < P; i++)
   231     {
   232       threads[i].start();
   233     }
   234
   235     for(int i = 0; i < P; i++)
   236     {
   237       threads[i].join();
   238     }
   239     long endTime = System.currentTimeMillis();
   240
   241     System.out.println("main(): all tasks finished execution");
   242     System.out.println("main(): computations took " + (endTime - startTime) + " ms");
   243   }
   244 }
   245
No errors.
```

Matrix.java

```
     1 /* ---------------------------------------------------------------------
     2  * Паралельні та розподілені обчислення
     3  * Java. Монітори
     4  * Завдання:
     5  *   MA = max(MO) * (MR * (MX * MT))
     6  * Виконав: Грибенко Дмитро В'ячеславович
     7  * Sat May  1 21:42:06 EEST 2010
     8  * --------------------------------------------------------------------- */
     9
    10 import java.io.*;
    11
    12 public class Matrix
    13 {
    14   public Matrix(int N)
    15   {
    16     this.N = N;
    17     this.data = new int[this.N * this.N];
    18   }
    19
    20   public int get(int i, int j)
    21   {
    22     return this.data[i * this.N + j];
    23   }
    24
    25   public void set(int i, int j, int value)
    26   {
    27     this.data[i * this.N + j] = value;
    28   }
    29
    30   public void fill()
    31   {
    32     for(int i = 0; i < this.N; i++)
    33     {
    34       for(int j = 0; j < this.N; j++)
    35       {
    36         this.data[i * this.N + j] = 1;
    37       }
    38     }
    39   }
    40
    41   public void copyTo(Matrix result)
    42   {
    43     for(int i = 0; i < this.N; i++)
    44     {
    45       for(int j = 0; j < this.N; j++)
    46       {
    47         result.data[i * this.N + j] = this.data[i * this.N + j];
    48       }
    49     }
    50   }
    51
    52   public void write(OutputStream out)
    53   {
    54     Writer w = new OutputStreamWriter(out);
    55     PrintWriter pw = new PrintWriter(new BufferedWriter(w));
    56     for(int i = 0; i < this.N; i++)
    57     {
    58       for(int j = 0; j < this.N; j++)
```

```
    59       {
    60         pw.print(this.data[i * this.N + j]);
    61         pw.print(" ");
    62       }
    63       pw.println();
    64     }
    65     pw.println();
    66     pw.flush();
    67   }
    68
    69   int[] data;
    70   int N;
    71 }
    72
No errors.
```

Functions.java

```
     1 /* ------------------------------------------------------------------------
     2  * Паралельні та розподілені обчислення
     3  * Java. Монітори
     4  * Завдання:
     5  *   MA = max(MO) * (MR * (MX * MT))
     6  * Виконав: Грибенко Дмитро В'ячеславович
     7  * Sat May  1 21:42:06 EEST 2010
     8  * ------------------------------------------------------------------------ */
     9
    10 public class Functions
    11 {
    12   /* max(MO_H) */
    13   public static int comp1(
    14       int start,
    15       int end,
    16       Matrix MO)
    17   {
    18     int result = Integer.MIN_VALUE;
    19     for(int i = start; i < end; i++)
    20     {
    21       for(int j = 0; j < MO.N; j++)
    22       {
    23         int x = MO.get(i, j);
    24         if(x > result)
    25         {
    26           result = x;
    27         }
    28       }
    29     }
    30     return result;
    31   }
    32
    33   /* MZ_H = MX_H * MT */
    34   public static void comp2(
    35       int start,
    36       int end,
    37       Matrix MZ,
    38       Matrix MX,
    39       Matrix MT)
    40   {
    41     for(int i = start; i < end; i++)
    42     {
    43       for(int j = 0; j < MZ.N; j++)
    44       {
    45         int acc = 0;
    46         for(int k = 0; k < MZ.N; k++)
    47         {
    48           acc += MX.get(i, k) * MT.get(k, j);
    49         }
    50         MZ.set(i, j, acc);
    51       }
    52     }
    53   }
    54
    55   /* MA_H = alpha * MR_H * MZ */
    56   public static void comp3(
    57       int start,
    58       int end,
    59       Matrix MA,
    60       int alpha,
```

```
    61       Matrix MR,
    62       Matrix MZ)
    63   {
    64     for(int i = start; i < end; i++)
    65     {
    66       for(int j = 0; j < MA.N; j++)
    67       {
    68         int acc = 0;
    69         for(int k = 0; k < MA.N; k++)
    70         {
    71           acc += MR.get(i, k) * MZ.get(k, j);
    72         }
    73         acc *= alpha;
    74         MA.set(i, j, acc);
    75       }
    76     }
    77   }
    78 }
    79
No errors.
```