НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Кафедра обчислювальної техніки\_\_\_\_\_\_\_\_\_\_\_\_

(повна назва кафедри, циклової комісії)

**КУРСОВА РОБОТА**

з дисципліни «Паралельні та розподілені обчислення»

(назва дисципліни)

на тему: «Розробка програмного забезпечення для паралельних комп’ютерних систем»

Студента (ки) 3 курсу \_ІО-24\_ групи

напряму підготовки 050102 «Комп’ютерна інженерія»

\_\_Ружевський М.С.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище та ініціали)

Керівник доцент Корочкін О.В.

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_

Оцінка: ECTS \_\_\_\_\_\_\_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали

Київ- 2015 рік

Національний технічний університет України

“Київський політехнічний інститут”

Факультет (інститут) інформатики та обчислювальної техніки

( повна назва )

Кафедра обчислювальної техніки

( повна назва )

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки 6.050102 «Комп’ютерна інженерія»

# (шифр і назва)

З А В Д А Н Н Я

НА КУРСОВУ РОБОТУ СТУДЕНТУ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ружевський Микита Сергійович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім’я, по батькові)

1. Тема роботи «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

керівник роботи Корочкін Олександр Володимирович к.т.н.**,** доцент

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи 18 травня 2014 р.

3. Вхідні дані до роботи

- засоби роботи з процесами в OpenMP

- математична задача A=B\*a + C\*(MO\*MK)\*max(Z)

- структури ПКС ОП та ПКС ЛП

- мови і бібліотеки програмування: OpenMP

- засоби організації взаємодії процесів: MPI

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- Огляд двонадцятиядерних процесорів AMD

5. Перелік графічного матеріалу

- структурна схема ПКС ОП

- структурна схема ПКС ЛП

- схеми алгоритмів процесів і головної програми для ПРГ1

- схеми алгоритмів процесів і головної програми для ПРГ2.

6. Дата видачі завдання \_\_\_\_19.02.2015\_\_\_\_\_

КАЛЕНДАРНИЙ ПЛАН

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання Розділу 1 | 23.02.2015 |
| 2 | Виконання Розділу 2 | 23.03.2015 |
| 3 | Виконання Розділу 3 | 23.04.2015 |
| 4 | Тестування програм ПРГ1 та ПРГ2 | 10.05.2015 |
| 7 | Оформлення КР | 17.05.2015 |
| 8 | Захист КР | 18.05.2015 |

**Студент \_\_\_\_\_\_\_\_\_ \_\_\_\_Ружевський М.С.\_\_\_\_\_**

( підпис ) (прізвище та ініціали)

**Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_Корочкін О.В.\_\_\_\_\_\_\_\_\_**

( підпис ) (прізвище та ініціали)

**ЗМІСТ**

ВСТУП………………………………………………………………………………..5

РОЗДІЛ 1. ОГЛЯД ДВОНАДЦЯТИЯДЕРНИХ ПРОЦЕССОРІВ AMD …….…..6

[1.1. Відомості про модельний ряд AMD Opteron 6100 серії……………………6](#_Toc419755580)

[1.2 Загальні відомості про модельний ряд AMD Opteron 6200 серії…………..9](#_Toc419755581)

[1.3. Загальні відомості про модельний ряд AMD Opteron 6300 серії………...11](#_Toc419755582)

[1.4. Огляд 12-ядерних процесорів AMD………………………………………..15](#_Toc419755583)

1.5. Висновки до розділу 1 22

[РОЗДІЛ 2. РОЗРОБКА ПЗ ДЛЯ ПКС З СП………………………………………23](#_Toc419755585)

2.1 Розробка паралельного алгоритму 23

[2.2. Розробка алгоритмів процесів](#_Toc419755589) 23

[2.3. Розробка схеми взаємодії процесів](#_Toc419755590) 25

[2.4. Розробка програми ПРГ1](#_Toc419755591) 27

[2.5. Тестування програми ПРГ1](#_Toc419755592) 27

[2.6. Висновки до розділу 2](#_Toc419755592) 27

[РОЗДІЛ 3. РОЗРОБКА ПЗ ДЛЯ ПКС З ЛП………………………………………31](#_Toc419755588)

3.1 Розробка паралельного алгоритму 31

[3.2. Розробка алгоритмів процесів 31](#_Toc419755589)

[3.3. Розробка схеми взаємодії процесів 34](#_Toc419755590)

[3.4. Розробка програми ПРГ2 37](#_Toc419755591)

[3.5. Тестування програми ПРГ 2 38](#_Toc419755592)

[3.6. Висновки до розділу 2](#_Toc419755592) 39

[ВИСНОВКИ](#_Toc419755588)………………………………………………………………………...40

[Список літератури………………………………………………………………….41](#_Toc419755588)

[ДОДАТОК 1. Лістинг коду………………………………………………………...4](#_Toc419755588)2

**ВСТУП**

В даному курсовому проекті буде виконаний огляд двонадцятиядерних процесорів виробництва компанії AMD та розробка прикладної багатопоточної програми засобами, що вказані у технічному завданні.

Метою даної курсової роботи є закріплення навичок у написанні багатопоточних програм та ознайомлення з існуючими багатоядерними та багатопоточними паралельними процесорами (в даному випадку – 12-ядерні процесори виробництва AMD).

Актуальність роботи полягає у тому, що проаналізовані процесори на даний час є у виробництві а також у тому, що у даній курсовій роботі використовуються засоби багатопоточності, що також є актуальними.

У першому розділі будуть розглянуті 12-ядерні процесори виробництва AMD. Усі с перелічених моделей випускаються, та використовуються.

У другому розділі буде розроблена програма для паралельної комп’ютерної системи з спільною пам’яттю.

У третьому розділі буде розроблена програма для паралельної комп’ютерної системи з локальною пам’яттю.

**РОЗДІЛ 1. ОГЛЯД ДВОНАДЦЯТИЯДЕРНИХ ПРОЦЕССОРІВ AMD**

Двонадцятиядерні процесори на даний момент часу представлені у сегменті високопродуктивних серверних процесорів декількома моделями, об’єднаними декількома модельними рядами [8].

Цими процесорами є моделі Opteron 6100, 6200 та 6300 серії, що були випущені впродовж 2010-2014 років. Загальною характеристикою для них є використання спільної платформи (сокету) G34 з оперативною пам’яттю стандарту DDR3 DIMM, вбудованого графічного ядра Matrox G200-серії та апаратна підтримка операцій AMD64/EMT64, SSE2, SSE3, SSE4, NXBit та Intel Virtualization Technology [15] [8]; у серії 6100 також є апаратна реалізація технології 3DNow. Для усіх модельних рядів, що будуть розглянуті є більш потужні та многоядерні версії цих процесорів, що мають складаються з 16 ядер (за виключенням 6100 серії), «бюджетні» економічні дешеві версії з 4 ядрами, процесори з аналогічною архітектурою для стаціонарних комп’ютерів, мобільні версії процесорів для ноутбуків.

## **1.1. Відомості про модельний ряд AMD Opteron 6100 серії**

Лінійка процесорів серії 6100 має у своєму арсеналі 8 модифікацій процесору с 12 ядрами. Усього дана модель налічує 13 модифікацій, з яких інші 5 мають лише 8 ядер. Дані чіпи побудовані на кристалах Magny-Cours за 45-нм технологією, на відміну від інших представників серверних процесорів, що розглядаються у даному курсовому проекті мають 3DNow для забезпечення сумісності з програмами, що її використовують, наприклад, для рендерінгу відео. Усі процесори, що розглядаються, мають 128 КБ кешу першого рівня на кожне ядро, 512 КБ кешу другого рівня на ядро, що між собою з’єднані шиною AMD HyperTransport (виходить 6МБ загального кешу 2 рівня), та 12 МБ кешу третього рівня (що єдиний для усіх ядер), таким чином даний процесор чудово застосовується як для програмування програм із загальною пам’яттю, так і для програм, що мають локальну пам'ять (в залежності від того, чи використовується кеш 3-го рівня, чи 2-го рівня з шиною HyperTransport). Але, це не повноцінні 12-ядерні процесори, усі процесори даної серії є «спареними» у один блок двома процесорами по 6 ядер[10]. Процесори працюють на платформі G34, використовують модулі пам’яті DDR3 DIMM частотою до 3200 МГц та побудовані на архітектурі AMD K10. Усі представники були випущені у 2010-2011 роках та на даний момент зняті з виробництва [15].

**Огляд архітектури AMD K10 та її модифікації К10.5**

Архітектура AMD K10 та її модифікація К10.5 полягли у основу процесорів AMD Opteron 6100 серії.

Основною відмінністю процесорів покоління K10 від своїх попередників на базі AMD K8 є об'єднання чотирьох ядер на одному кристалі, поновлення протоколу Hyper-Transport до версії 3.0, загальний для всіх ядер кеш L3, а також перспективна підтримка контролером пам'яті DDR3. Самі ядра також були модернізовані в порівнянні з ядрами AMD K8. AMD K10.5 є подальшою модифікацією AMD K10, у якій кількість ядер на одному кристалі було збільшено з 4 до 6.

Архітектура Direct Connect Architecture дозволяє збільшити продуктивність і ефективність шляхом прямого з'єднання контролера пам'яті і каналу вводу-виведення з ядром, розроблена для одночасного виконання як 32-бітових, так і 64-бітових обчислень. Потребує інтеграції контролера пам'яті стандарту DDR2 (аж до режиму 533 (1066) МГц) або DDR3 (до 1333 МГц). Ця архітектура має свої переваги:

* Збільшення продуктивності додатків шляхом скорочення затримок при зверненні до пам'яті
* Розподіляє смугу пропускання пам'яті залежно від запитів
* Технологія Hyper-Transport забезпечує з'єднання на пікової швидкості до 16,0 ГБ / с для запобігання затримок
* До 33,1 ГБ / с сумарної пропускної спроможності між процесором і системою (з урахуванням шини Hyper-Transport і контролера пам’яті)

Одно з нововведень - загальний для всіх ядер кеш L3 об'ємом 2 МБ на додаток до 512 КБ кешу L2 для кожного ядра. Перевага - скорочення затримок при зверненні до часто використовуваних даних для збільшення продуктивності.

128-бітний FPU для кожного ядра прискорює вибірку та обробку даних в обчисленнях з плаваючою комою.

З'єднання Hyper-Transport з піковою швидкістю до 16,0 ГБ/с при роботі в режимі Hyper-Transport 3.0 (виключно в К10.5) дає до 33,1 ГБ/с сумарної пропускної спроможності між процесором і системою (з урахуванням шини Hyper-Transport і контролера пам'яті).

Інтегрований контролер пам'яті з високою пропускною здатністю і низькими затримками, що підтримує пам’ять стандарту DDR3 DIMM з пропускною здатністю до 17.1 Гб/с.

AMD-V дає програмісту апаратний набір функцій розроблених для збільшення продуктивності, надійності і безпеки в існуючих і майбутніх середовищах віртуалізації, що дозволяє віртуальним машинам безпосередньо звертатися до виділеної пам'яті.

Удосконалена система управління живленням Cool'n'Quiet 2.0, що автоматично регулює продуктивність процесора залежно від навантаження знижує споживання енергії і швидкості обертання кулера в режимі простою, дозволяє знижувати енергоспоживання шляхом відключення невикористовуваних частин процесора. Роздільна система для контролера пам'яті і логіки процесора дозволяє керувати напругою і відключати їх незалежно один від одного. Працює автоматично без необхідності підтримки з боку драйвера або BIOS. Дозволяє незалежно управляти частотами кожного ядра. Швидкість перемикання режимів роботи дорівнює одному такту процесорного ядра.

Процесори серії Opteron також отримали можливість роботи в старих материнських платах, побудованих на платформі Socket F. В обох випадках потрібно лише оновлення BIOS материнської плати. Всі дані процесори побудовані на архітектурі AMD64, вони здатні працювати з 32-бітовим x86, 16-бітним і AMD64 кодом.

## **1.2 Загальні відомості про модельний ряд AMD Opteron 6200 серії**

Серія Opteron 6200 почала виходити в кінці 2011-початку 2012 року. Основні відмінності цього покоління відносно серії 6100: зменшений техпроцесс з 45 до 32 нм, завдяки якому кількість ядер на чіп у топових конфігураціях було збільшено з 12 до 16, відмова від підтримки технології 3DNow у серверних процесорах як від невикористовуємої дозволила зменшити затрати енергії на працю процесора, також був зроблений перехід з архітектури К10 на абсолютно нову архітектуру Bulldozer.

**Огляд архітектури Bulldozer**

Однією з ключових особливостей процесорів на базі мікроархітектури Bulldozer став перехід на 32-нм літографічний процес, який протягом майже двох років досить успішно використовується основним конкурентом - компанією Intel. Крім потенційного зменшення енергоспоживання та покращення розгінного потенціалу, цей факт позитивно позначилося на вартості виробництва напівпровідникових кристалів.

Головні нововведення, які були зроблені в мікроархітектурі Bulldozer, торкнулися алгоритму виконання багатопоточних обчислень. Довгий час центральні процесори були здатні виконувати єдиний обчислювальний потік в один момент часу. Так звана одночасна робота декількох програм здійснювалася за допомогою обробника переривань, тобто обчислювальні завдання різних додатків по черзі отримували короткочасний доступ до ресурсів процесора. Саме завдяки цьому стала можливою робота багатозадачних операційних систем. Чи варто говорити, що швидкість роботи в такому режимі була невисокою. У той же час, розробники CPU стали помічати, що в навантаженні різні функціональні блоки процесора можуть простоювати без роботи, поки інші зайняті обробкою обчислювального потоку. Саме це і наштовхнуло їх на спільне використання одних і тих же ресурсів процесора для обробки декількох обчислювальних потоків. Компанія Intel впровадила таку можливість під назвою Hyper-Threading в свої процесори ще в далекому 2002 році. Даний принцип дає деякий приріст в певному типі завдань. У той же час, підхід AMD до реалізації багатопоточних обчислень довгий час залишався незмінним: кожен потік повинен виконуватися на окремому ядрі. Тепер же, після оптимізації продуктивності окремих вузлів процесора і ретельного аналізу навантаження, розробники AMD порахували, що швидкодії деяких вузлів цілком достатньо для обслуговування відразу двох незалежних обчислювальних потоків. Такий підхід дозволив здорово скоротити кількість транзисторів, але зберегти високу продуктивність. Тепер же, в світлі збільшення вимог до швидкодії при збереженні прийнятних параметрів енергоспоживання, розробники змушені шукати шляхи збільшення кількості виконуваних за такт інструкцій.

В основі всіх процесорів AMD Opteron 6200 серії лежить напівпровідниковий кристал, що складається з чотирьох обчислювальних модулів, кожен з яких забезпечений власним масивом кеш-пам'яті 2-го рівня, загального кеша 3-го рівня об'ємом 12 Мбайт, двухканального контролера пам'яті DDR3, контролерів шини HyperTransport і вбудованого північного мосту.

До складу обчислювального модуля входять два блоки цілочисельних обчислень (ALU), кожен з яких здатний виконувати до чотирьох інструкцій за такт, забезпечених власною кеш-пам'яттю 1-го рівня для зберігання даних. Всі інші блоки, такі як провісник розгалужень, декодер інструкцій, буферна пам'ять для зберігання інструкцій і масив кеш-пам'яті 2-го рівня розмірів 2 Мбайт, представлені в одиничному екземплярі. Очевидно, розробники вважали, що продуктивності цих блоків достатньо для обслуговування двох ALU. Крім того, кожен з обчислювальних модулів розпорядженні блоком обчислень з плаваючою точкою (FPU), який також зазнав значних доопрацюванням. Так до стандартних SIMD-розширень додалися набори SSE4.1 і SSE4.2, а також специфічні інструкції XOP, AES і AVX, які дозволяють значно підвищити швидкодію за умови їх підтримки з боку програмного забезпечення. Цікаво виглядає можливість виконання 256-бітних інструкцій AVX, для цього задіюються ресурси відразу двох блоків, кожен з яких здатний обробляти 128-бітові команди FMAC. При цьому блок FPU здатний виконувати дві коротких інструкції AVX одночасно.

Як видно з короткого опису дизайну ядра, мікроархітектура Bulldozer, незважаючи на всі свої нововведення, не позбавлена ​​деяких недоліків. Все-таки на кожен обчислювальний модуль доводиться тільки один провісник розгалужень, блок вибірки інструкцій і один декодер інструкцій, який, до слова, здатний обробляти не більше чотирьох інструкцій за такт[7][8].

## **1.3. Загальні відомості про модельний ряд AMD Opteron 6300 серії**

Так як вже вказано раніше, спроба AMD зробити процесор з надвисокою кількістю ядер при спрощенні та їх об’єднанні не здобула успіхів. Але, не дивлячись на це, AMD презентували виправлену версію архітектури Bulldozer, що названа Piledriver.

Компактність цієї архітектури дозволила створити потужні 16-ядерні та 12-ядерні серверні процесори для сокету G34 (хоча, як вже було сказано, ці ядра не повноцінні).

**Огляд архітектури Piledriver**

Як відомо, при впровадженні Bulldozer процесорна команда AMD змінила пріоритети. Ця мікроархітектура в порівнянні зі Stars (K10) зменшила кількість виконуваних за такт інструкцій, але зате дозволила розвивати більш високі тактові частоти. Однак досягнутим ефектом залишилися задоволені далеко не всі, тому через три квартали з моменту появи на ринку перших версій Bulldozer AMD підготувала оновлення мікроархітектури - своєрідну роботу над помилками - Piledriver. Втім, не варто покладати на Piledriver якісь особливі очікування. Структурно ця мікроархітектура повністю повторює Bulldozer, тобто ґрунтується на використанні умовно-двоядерних модулів, в яких є по два набори цілочисельних виконавчих пристроїв, але частина ресурсів представлена лише в єдиному на два ядра екземплярі. До числа таких поділюваних компонентів відноситься кеш-пам'ять, блок вибірки інструкцій, їх декодер і блок операцій з плаваючою крапкою. В результаті, модуль може обробляти два потоки одночасно, але його пікова продуктивність обмежена пропускною здатністю об'єднаного декодера, здатного декодувати не більше чотирьох інструкцій за такт на два ядра. Для порівняння: процесорах виробництва Intel сімейства Core декодер має приблизно такий самий темп роботи, але там він для кожного ядра індивідуальний. Це означає, що число оброблюваних за такт інструкцій у Piledriver серйозно збільшитися не могло. Якісні зміни відбудуться тільки в наступному поколінні мікроархітектури, Steamroller: передбачається, що в майбутньому AMD забезпечить власним декодером інструкцій кожне з двох ядер в модулі. Поки ж всі поліпшення Piledriver грунтуються на оптимізацію в алгоритмі роботи окремих внутрішніх блоків, але не зачіпають дизайн в цілому. Список виправлень та вдосконалень відносно попередньої архітектури вказано нижче:

* За рахунок впровадження гібридного дворівневого провісника поліпшена точність передбачення переходів;
* Набір інструкцій розширено трьохоперандними 128- і 256-бітними інструкціями FMA3 (fused multiply-add) та інструкціями з класу SSE5 - перетворенням речових даних з половинною точністю F16C;
* Оптимізована робота планувальників;
* За рахунок переробки відповідного виконавчого пристрою прискорено виконання операцій ділення;
* Збільшена L1 TLB;
* Алгоритми попередньої вибірки даних у L1- та L2-кеши поліпшені, тепер вони дозволяють працювати з патернами змінної довжини, у тому числі і що знаходяться на кордонах сторінок;
* Збільшена ефективність L2-кеша за рахунок більш агресивного його звільнення від невикористовуваних даних, помилково завантажених внаслідок роботи алгоритмів попередньої вибірки[13].

Темп декодування інструкцій всі перераховані нововведення наростити не в змозі, але, тим не менш, певний прискорення вони забезпечити можуть. Для того щоб уявити собі, наскільки мікроархітектура Piledriver ефективніше своєї попередниці, я провів невелике порівняння на практичних бенчмарках.

Були використані наступні процесори:

* Intel Core i7-3770K
* Intel Core i5-3570K
* AMD FX-8350
* AMD FX-8150

Також тестові експонати були розігнані до 4.7ГГц (FX-8350) та його конкурент - Intel Core i5-3570K – до 4.6ГГц.

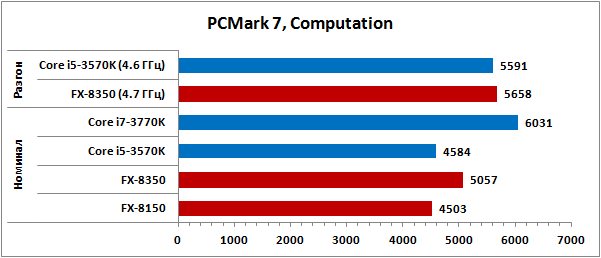


Рис.1.1 Графік порівняння потужності представника архітектури Piledriver з попередниками та конкурентами у бенчмарку PCMark 7

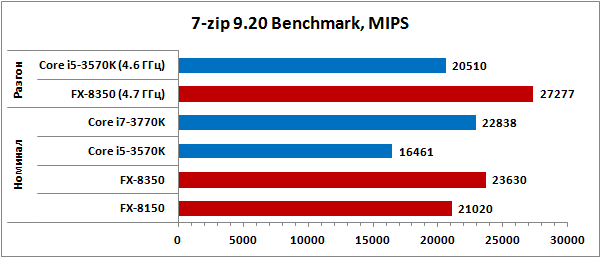


Рис.1.2 Графік порівняння потужності представника архітектури Piledriver з попередниками та конкурентами у бенчмарку 7zip

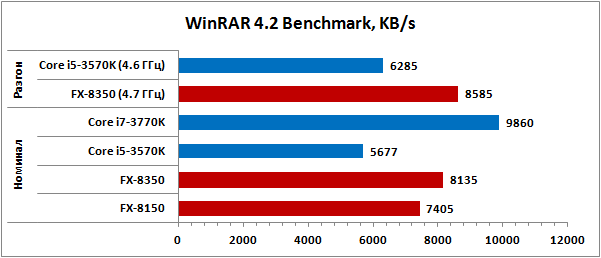


Рис.1.1 Графік порівняння потужності представника архітектури Piledriver з попередниками та конкурентами у бенчмарку WinRAR

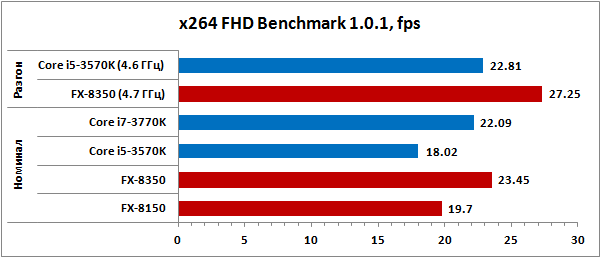


Рис.1.1 Графік порівняння потужності у бенчмарку х264

На тлі свого попередника з мікроархітектурою Bulldozer процесор FX-8350 як представник архітектури Piledriver виглядає дуже непогано. З анонсом перших CPU сімейства Bulldozer AMD обіцяла 10-15-відсотковий приріст продуктивності власних процесорів щорічно, і, як бачимо, плану вона чітко дотримується[6][7]. По факту, середній приріст швидкості склав 13 відсотків, і складається він, нагадаємо, з двох складових: невеликих мікроархітектурних поліпшень і збільшення тактової частоти. Звичайно, це зовсім не виглядає принциповим ривком, але справедливості заради зазначимо, що перехід конкурента від дизайну Sandy Bridge до Ivy Bridge дав ще менший ефект.

А якщо до цього додати той факт, що підтримка багатопоточності сучасними програмами за останній рік тільки покращилася, то дуже гідний виступ FX-8350 на тлі Core i5-3570K зовсім не дивно. Іншими словами, у всяких ресурсоємних додатках, які здатні распараллелівать свою навантаження на велике число ядер, FX-8350 буде явно краще за свого прямого інтеловського конкурента. При цьому до рівня продуктивності Core i7-3770K, який посилено технологією Hyper-Threading, він не дотягує, але не забувайте про ціну: AMD FX-8350 коштує навіть дешевше Core i5-3570K[6].

Не міняє розстановку сил і оверклокінг. Хоча робочу частоту AMD FX-8350 нам вдалося підвищити всього на 18 відсотків, а частота Core i5-3570K при розгоні може бути збільшена приблизно вдвічі сильніше, вісім обчислювальних ядер процесора з мікроархітектурою Piledriver все одно дозволяють йому домінувати в більшості бенчмарків, що створюють серйозну багатопоточне навантаження.

## **1.4. Огляд 12-ядерних процесорів AMD**

Найкращий спосіб зрозуміти, чим виділяються 12-ядерні процесори від AMD це порівняти результати потужності цих процесорів з їх попередниками на конкурентами. В умовах конкуренції з Intel до усіх процесорів є свій аналог від AMD.

**1.4.1 Процессор AMD Opteron 6174 vs Intel Xeon X5670 vs Intel Xeon MP X7560**

У жовтні автор власного блогу з сайту infoworld.com Paul Venezia вирішив провести стрес-тест трьох вищевказаних процесорів. Особливою умовою проведення тесту став бюджет серверних станцій. Як систему з процесорами AMD Opteron став Dell PowerEdge R815, обладнаний 4-ма процесорами AMD Opteron 6174. Йому були протипоставлені аналогічні за ціною системи на базі двох Intel Xeon MP X7560 та також двох Intel Westmere-EP X5670. Не дивлячись на те, що процесорів у Dell PowerEdge R815 аж 4, він усе одно коштував на значно дешевше за своїх суперників. Нагадаємо, що AMD Opteron 6174 є «бюджетним» серверним 12-ядерним чіпом, так як коштував на час виходу близько $980[8], його ж конкуренти Intel Xeon MP X7560 на час релізу коштував $3692[8], Intel Xeon X5670 - близько $1450. Тобто кількість ядер у сервері від AMD (48-ядерна система) була повністю адекватна для даного експерименту з точки зору бюджетності моделі. Тести передбачали собою виконання монотонної задачі у 8, 12, 24, 48, 96 та 128 потоків, як це звичайно відбувається під час звичайної експлуатації серверу. Результати тестування вказані на діаграмі нижче (вказаний час у секундах, нижче – краще)[1]:

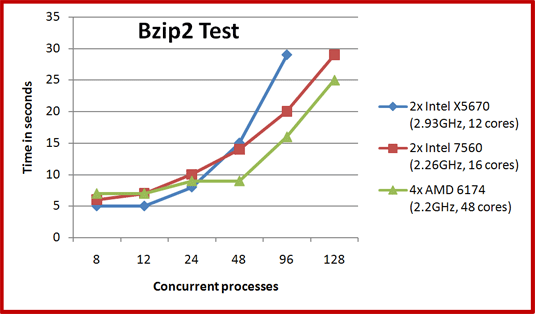


Рис.1.6 Графік порівняння потужності у бенчмарку BZip2

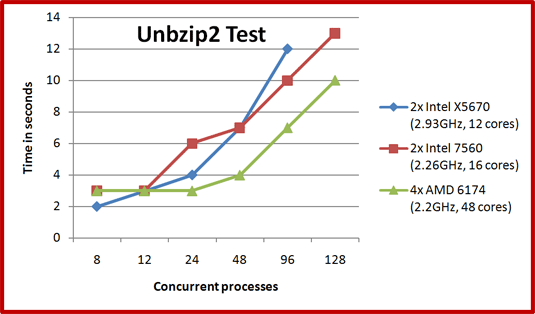


Рис.1.7 Графік порівняння потужності у бенчмарку UnBZip2

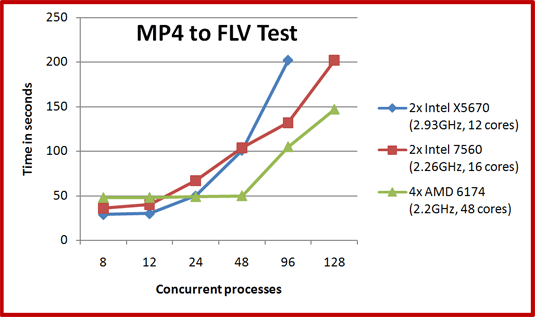


Рис.1.8 Графік порівняння швидкості рекодування відео

Як ми бачимо з результатів зверху, AMD вдало впорався із завданням відносно його конкурентів з Intel. У всіх тестах він однаково потужний з конкурентами на 24 потоках та значно кращий за конкурентів у випадку, коли потоків 48 або більше. Тому саме серія Opteron 6100 була найбільш вдалою у проекції «ціна-продуктивність» у 2010-2011 роках, до релізу серії Opteron 6200.

**1.5.2. AMD Opteron 6174 vs AMD Opteron 6272**

У пресс-релізі нової архітектури Bulldozer AMD запевняла у значному зрості продуктивності на ватт, швидкодії та інших перевагах нової архітектури, тому огляд цих архітектур та їх порівняння є інтересним та познавчим для нас. В якості тестових систем був використаний сервер HP ProLiant DL385 G7, люб'язно наданий компанією Hewlett Packard і в якому якраз розташовуються два процесори AMD Opteron 6272 з тієї самої новою архітектурою [3].

Для можливості порівняти отримані результати з іншими рішеннями був використаний аналогічний сервер HP ProLiant DL385 G7, але зі «старими» процесорами AMD Opteron 6174 (відомими під назвою Magnu-Cours). А для повноти картини всі тести також випробовувались і на тестовому сервері ручної збірки на базі Intel Xeon E3-1230, який за характеристиками практично повністю еквівалентний моделі HP ProLiant DL120 G7.

Щоб оцінити обчислювальні можливості нової архітектури було вибрано п'ять груп тестів, сумарно покривають такі області застосування як чисельні методи в аєро-гідродинаміки, робота з щільними матрицями, молекулярна динаміка та обробка відео-потоків. Нижче дана просто коротка характеристика для кожного тесту:

Тест 1. Рішення рівнянь Лапласа. Для цього використовувався пакет OpenMP з SSE (що додає підтримку векторних розширень

Тест 2. Використання бібліотек BLAS. Метою була оцінка досягнутої продуктивності при використанні готових реалізацій стандарту BLAS. Для цього були використані бібліотеки Intel MKL і AMD CML.

Тест 3. Обробка відео. У даному тесті використовувалася відома бібліотека FFMPEG, призначена для перекодування відео-фалів зі всіляких форматів у всілякі. Для солідності варто відзначити, що, згідно з деякими джерелами, саме з її допомогою обробляються всі відео-ролики, що завантажуються на найпопулярніший портал YouTube.

Тест 4. Використання пакета LAMMPS. Даний популярний пакет призначений для проведення моделювання різних процесів в такій області як молекулярна динаміка. Для тестування використовувався вбудований в нього бенчмарк, який проводить обчислення потенціалу Леннарда-Джонса, моделювання білка родопсину і ряд інших тестів.

Тест 5. Використання пакета OpenFOAM. В останньому тесті оцінювалася швидкість роботи двох вирішувачей з пакету (interFoam та simpleFoam), за допомогою яких моделювалися процес руйнування дамби і горіння суміші газів відповідно.

Варто підкреслити, що в тестах 1-2 паралелізм реалізований через потоки, а в тестах 3-5 - через процеси.

Найбільш цікаві результати були отримані в тесті Laplace, в якому оцінювалася продуктивність «самописних» програм при їх перенесення на багатоядерні системи. Так, при запуску «просто паралельної» програми більш високу продуктивність показали старі процесори AMD Opteron 6174, однак при збільшенні розміру сітки з 512x512 до 4096x4096 вузлів швидкість їх роботи впала більш ніж у 7 разів. З іншого боку, 32-ядерна система на базі AMD Opteron 6272 хоч мала і трохи меншу продуктивність, але показала більш стабільні результати - при аналогічному переході швидкість знизилася всього в 2.8 рази.

При спробі провести додаткові оптимізації у вигляді розпаралелювання внутрішніх циклів через інструкції SSE і розбиття даних на блоки з метою підвищення ефективності використання кеша на всіх системах вдалося підняти продуктивність приблизно в 5-10 разів. Причому «бульдозери» знову показали більш стабільну роботу - при різних вхідних даних і різних видах оптимізації їх швидкість роботи виявлялася набагато більш передбачуваною і очікуваною, ніж у їхніх попередників.

Інший тест - запуск функцій з BLAS-бібліотек. Сказати по правді, його результати виявилися трохи бентежить. Так на тесті SYMV 4-ядерна система на базі Intel Xeon E3 обійшла приблизно в 10 разів всі системи від AMD. І це при тому, що в останніх обчислювальних ядер більше разів в 6-8. З іншого боку, на «важких» тестах типу перемноження матриць (GEMM) багатоядерність дала про себе знати, і рішення від AMD без проблем обігнали одиночний процесор від Intel в 2-2,5 рази. А що ще більш цікаво - обчислювальна ефективність систем виявилася рівною 79 – 90,2%, що можна пояснити закритістю бібліотек, в результаті чого захований в них алгоритм виконує трохи менше операцій, ніж припускали автори.

Останні три тести описують продуктивність при використанні промислових пакетів. Так, при перекодуванні відео-даних використання великої кількості ядер суттєво підвищує швидкість обчислень, в результаті чого нові процесори AMD Opteron 6272 опиняються поза конкуренцією. У тесті пакету LAMMPS аналогічна ситуація спостерігалася лише на ряді випадків (chute і lj). В інших тестах (chain, eam, rhodo) вузьким місцем ставала оперативна пам'ять, і зайві 8 ядер особливої ​​ролі не зіграли. Нарешті, в пакеті OpenFOAM беззастережну перемогу здобули старі процесори AMD Opteron 6174, що виявилося повною несподіванкою. Подібна ситуація спостерігалася при використанні двох різних вирішувачей, різних алгоритмів інвертування матриць і різних передбачателей, тому причина з великою ймовірністю криється в надрах самого пакету, поточна версія якого просто «не дружить» з новими процесорами. Поки «не дружить» [16].

Також варто сказати пару слів про масштабованість (тобто залежність продуктивності від кількості використовуваних ядер). Так от, в більшості випадків на процесорах від AMD вона була, м'яко кажучи, не найкращою. У тесті rhodo з пакету LAMMPS використання, наприклад, 31 ядра замість 32 знижувало швидкість роботи в 13,5 разів. А в тесті OpenFOAM на процесорах AMD Opteron 6272 мало сенс використовувати не більше 16 ядер. При спробі додати хоч ще одне ядро ​​швидкість обчислень відразу ж падала в 2,5-3 рази і ставала рівній продуктивності, що досягається при використанні всього 4 ядер. На жаль, подібні приклади далеко не єдині. Справедливості заради варто відзначити, що система від Intel подібних проблем не відчувала.

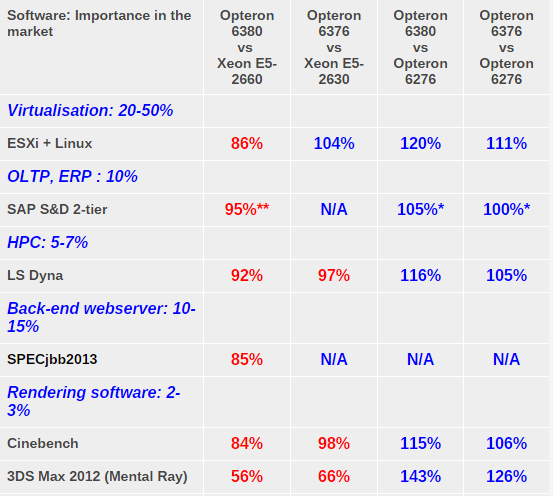
Якщо говорити чесно, то для будь-якої системи можна підібрати такий тест і такі вхідні дані, на яких вона легко і з великим відривом обійде дві інші системи. Однак при цьому варто сформулювати пару-трійку тез, які так чи інакше підходять для всіх груп тестів. По-перше, одне ядро ​​процесора Intel Xeon E3 по продуктивності відповідає приблизно 2-3 ядер процесорів від AMD. По-друге, нові процесори AMD Opteron 6272 виявилися більш стабільними, показали трохи кращу масштабованість і в середньому обійшли своїх попередників на 10-20%. І по-третє, для процесорів AMD рекомендованіше виявилася реалізація паралелізму через процеси, ніж через потоки, а також робота з даними невеликого розміру, ніж з багатомегабайтними масивами. У той же час процесор від Intel в цьому плані виявився «всеїдним».

**1.5.3: Opteron 6376 проти Opteron 6276**

Також цікаво як саме змінювалися процесори в рамках однієї архітектурної концепції (Bulldozer > Piledriver). В работі [4] виконано дослідження…..

Саме тому дослідники з anandtech.com зробили швидкий огляд того, як саме змінилась лінійка серверних процесорів AMD внаслідок удосконалення архітектури. Результати їх досліджень ми розглянемо далі, результати тестування наведено в таблиці 1.1.

Новий Opteron 6376 пропонує 5% до 11% вищу продуктивність за такт, 8% більш низьке споживання енергії, на 6% нижче нижче пікове єнергоспоживання і на 11% нижчу ціну, ніж Opteron 6276. Це все добре, але це не все. Ведення G34 платформа жива має дуже позитивний вплив на ціноутворення OEM: сервери Opteron є відчутно дешевше. Ви можете отримати сервер з 6380 по ціні сервера на основі Xeon E5-2640.



Таблиця 1.1 Порівняння продуктивності процесорів 6300 та 6200 серії

Все це є маленькі кроки вперед, щоб зробити AMD Opteron знову привабливим для клієнтів ринку серверних процесорів. Opteron мають більше за Intel енергоспоживання, але усе ж з більшою продуктивністю на доллар, ніж пропозицій від Intel середнього рівня.

## **1.6. Висновки до частини 1**

1. Були розглянуті усі покоління процесорів AMD, що мають 12-ядерну будову. На даний час це виключно процесори серії Opteron 6100/6200/6300, що є дещо специфічними версіями процесорів для серверів.

2. Якщо розглядати положення AMD у сегменті серверних процесорів, то на даний час у цій сфері домінує Intel із процесорами серії Xeon. Це їм вдалося завдяки технології Hyper-Threading, що дозволяє розпаралелювати непаралельні задачі. У свою чергу AMD робить ставку на надвисоку кількість ядер на низьку ціну (що іноді у 5-8 разів нижча, ніж у конкурента) та шину власної розробки HyperTransport, що зазвичай працює на частоті ядра та дозволяє ядрам обмінюватись інформацією зі швидкості до 33.1 ГБ/с. Також цю шину за власним бажанням можуть реалізовувати виробники материнських плат для багатопроцесорних систем.

3. Взагалі, використання вищерозглянутої продукції AMD буде економічно вигідніше, так як 64-ядерна система на базі AMD Opteron 6300 серії буде коштувати дешевше, ніж використання однопроцесорної системи на базі Intel Xeon з 8 ядрами. Звісно, якщо розглядати 4-процесорні системи в обох випадках, рішення AMD програють. Але у показниках потужності/ціна однозначно лідером є AMD.

# РОЗДІЛ 2. РОЗРОБКА ПЗ ДЛЯ ПКС З СП

В розділі розглядається розробка програми ПРТ1 для паралельної комп’ютерної системи зі спільною пам’яттю.

Математична задача: A = B\*a + C\*(MO\*MK)\*max(Z).

Мова програмування – С++ з бібліотекою OpenMP.

Структура ПКС СП представлена на рис. 2.1



Рис.2.1 Структура ПКС

**2.1 Розробка паралельного алгоритму**

Перш за все розроблюється математичний алгоритм, аналізується наявність спільних ресурсів.

1. zi = max(ZH), де і = 1..P
2. m = max (m, zi)
3. A = B\*a + C\*(MO\*MK)\*m

**2.2 Розробка алгоритмів задач**

Т1 КД,ТС

1. Читання B, C, MK
2. Бар’єр для усіх задач, синхронізація по вводу Бар’єр
3. Множення MT = MO\*MK з синхронізацією Бар’єр  
   виконуємих блоків
4. tz­H = max(ZH)
5. maxz = max(tz) КД, СР: maxz
6. AH = C \* MTH;
7. Бар’єр для проміжної синхронізації Бар’єр
8. AH = AH\*maxz + BH\*alpha КД, СР:maxz,alpha
9. Бар’єр перед виводом даних Бар’єр
10. Вивід А КД

Тi (i=2…p-1) КД,ТС

1. Бар’єр для усіх задач, синхронізація по вводу Бар’єр

2. Множення MT = MO\*MK з синхронізацією Бар’єр  
виконуємих блоків

3. tz­H = max(ZH)

4. maxz = max(tz) КД, СР: maxz

5. AH = C \* MTH;

6. Бар’єр для проміжної синхронізації Бар’єр

AH = AH\*maxz + BH\*alpha КД, СР:maxz,alpha

Тp КД,ТС

1. Читання MO, Z

2. Бар’єр для усіх задач, синхронізація по вводу Бар’єр

3. Множення MT = MO\*MK з синхронізацією Бар’єр  
виконуємих блоків

4. tz­H = max(ZH)

5. maxz = max(tz) КД, СР: maxz

6. AH = C \* MTH;

7. Бар’єр для проміжної синхронізації Бар’єр

AH = AH\*maxz + BH\*alpha КД, СР:maxz,alpha

2**.3 Розробка схеми взаємодії задач**

На підставі даних алгоритмів будується схема взаємодії процесів. Як спосіб синхронізації використовуються базові конструкції бібліотеки OpenMP, такі як критичні ділянки та бар’єри. Також, у деяких випадках, у яких використання паралелізму не має сенсу використовуються непаралельні блоки коду (наприклад, пошук максимуму у масиві з кількістю елементів, що дорівнює кількості процесорів у системі). Схема взаємодії представлена на рис. 2.2.

Були використані наступні засоби бібліотеки OpenMP:

#pragma omp parallel – об’ява блоку програми, що повинна виконуватись паралельно

#pragma omp barrier – використовується як засіб синхронізації потоків. При досягненні даного блоку поток блокується та очікує, коли інші потоки також довиконуються до бар’єру

#pragma omp for – використовується для розпаралелювання циклів for

#pragma omp single – використовується для виконання коду у одному потоці

omp\_set\_num\_threads(p) – використовується для вказання кількості потоків

omp\_get\_thread\_num() – використовується для визначення номеру потоку.

Лістинг програми розміщений у додатку А «Лістинг програми для ПКС з СП».

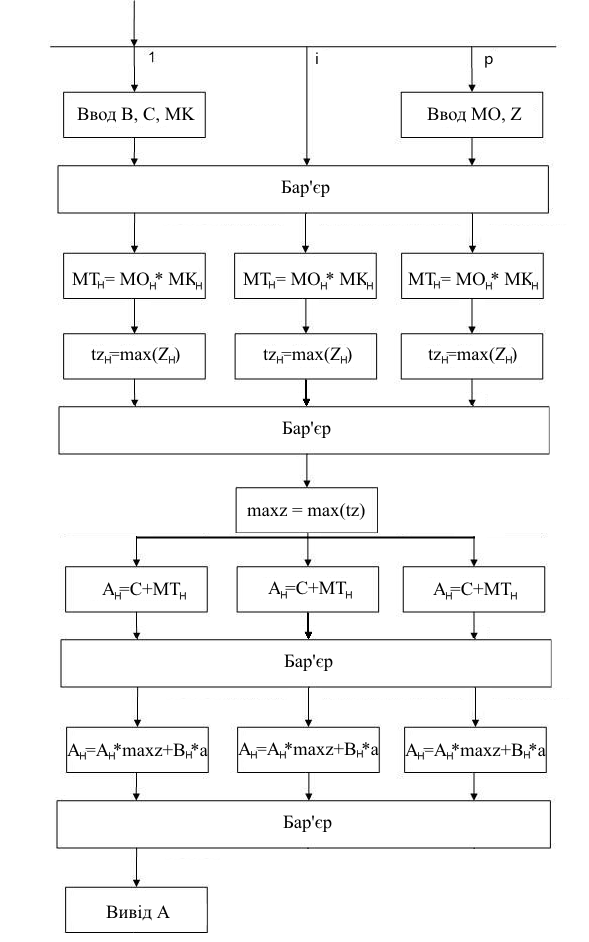


Рис. 2.2 – Схема взаємодії процесів.

**2.5 Дослідження ефективності розробленого ПЗ на реальній Р-ядерної ПКС**

Тестування програми виконувалось на 8-ядерному AMD FX-8150.

Для дослідження необхідно визначити час виконання розробленої програми в реальній ПКС. Тому при тестуванні послідовно використовуються 1, 2, 3, 4 процесорів, для яких визначаються час виконання Програми 1. При цьому встановлюється декілька значень розмірності векторів (матриць) N = 900, 1800, 2400. В таблиці 2.1 відображаються значення часу для різних N та Р.

Таблиця 2.1 - Час виконання програми в ПКС з СП (в секундах)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N//P | 1 | 2 | 4 | 8 |
| 900 | 14,303 | 7,407 | 4,248 | 2,727 |
| 1800 | 165,525 | 85,154 | 44,665 | 24,89 |
| 2400 | 553,409 | 283,735 | 148,227 | 81,582 |

Базуючись на отриманих результатах, вираховуємо значення коефіцієнтів прискорення . У таблиці 2.2 приведено результати цих розрахунків.

Таблиця 2.2. Значения Кп для ПКС з СП

|  |  |  |  |
| --- | --- | --- | --- |
| N//P | 2 | 4 | 8 |
| 900 | 1,931 | 3,36 | 5,245 |
| 1800 | 1,943 | 3,70 | 6,65 |
| 2400 | 1,95 | 3,73 | 6,783 |

На основі отриманих коефіцієнтів прискориння будується графік змінення КП в залежності від Р та N. Графік наведено на рис. 2.2. На осі абцис разташована розмірність векторів та матриць, ординатами зазначено коефіцієнт прискорення. Для позначення результатів різних багатопроцесорних систем використовуються різні типи ліній та маркеров.

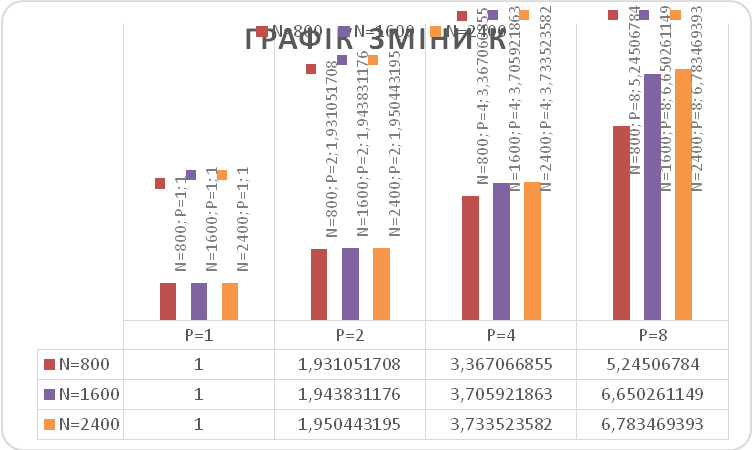


Рис. 2.3. Графік залежності коефіцієнта прискорення від розмірності матриць і векторів для ПКС з СП.

Розрахуємо коефіцієнти ефективності та заповнимо їми таблицю 2.3. Для розрахунку використовуються формули  .

Таблиця 2.3 - Значення Ке для програми ПКС з СП

|  |  |  |  |
| --- | --- | --- | --- |
| N//P | 2 | 4 | 8 |
| 800 | 0,965 | 0,841 | 0,655 |
| 1800 | 0,971 | 0,926 | 0,831 |
| 2400 | 0,975 | 0,933 | 0,847 |

На основі отриманих коефіцієнтів ефективності будується графік змінення Ке в залежності від Р та N. Графік наведено на рис. 2.3. На осі абцис разташована розмірність векторів та матриць, ординатами зазначено коефіцієнт ефективності. Для позначення результатів різних багатопроцесорних систем використовуються різні типи ліній та маркеров.

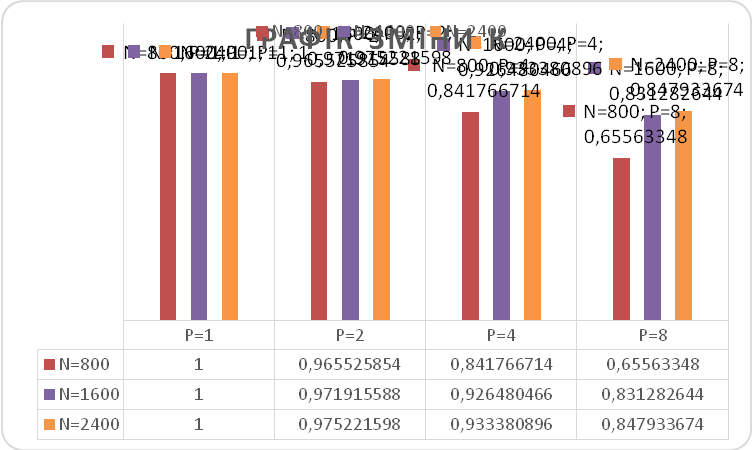


Рис. 2.3 - Графік залежності коефіцієнта прискорення від розмірності матриць і векторів для ПКС з СП.

**2.6 Висновки до розділу 2**

Виконано розробку програми ПРГ1 для ПКС із СП. При розробці використовувалась мова програмування Ада. Для взаємодії процесів використовувався механізм монітора.

На основі даних тестування програми ПРГ1 заповнено таблиці 2.1-2.3 та побудовано відповідні графіки на яких показано наступні результати:

* Використання багатоядерної ПКС забезпечує скорочення часу обчислення заданої математичної задачі.
* Максимальне значення *n K* = 6,783 забезпечує ПКС з Р = 8 та N=2400.
* Мінімальне значення *n K* = 1,93 забезпечує ПКС з Р = 2 та N = 800.
* По графіку з рис. 2.4 видно, що коефіцієнт прискорення збільшується пропорційно до кількості використовуваних ядер.
* Коефіцієнти ефективності коливаються в межах від 84-97%, це свідчить про високу ефективність розробленого паралельного алгоритму.
* Максимальне значення *e K* = 97% для ПКС з Р = 2 та N = 2400;
* Мінімальне значення *e K* = 84,9% для ПКС з Р = 8 та N = 800.
* Нерівномірність коефіцієнтів ефективності на графіку з рис. 2.5 пояснюється похибкою вимірювання часу. Так, похибка виміру часу для N = 800 найбільша, тому графік коефіцієнта ефективності при такій розмірності може бути відносно неточним.

Оцінивши значення *n K* та *e K* можна зробити висновок про доцільність

використання ресурсів ПКС для роз`в’язання задач такого типу.

# РОЗДІЛ 2. РОЗРОБКА ПЗ ДЛЯ ПКС З ЛП

В розділі розглядається розробка програми ПРТ2 для паралельної комп’ютерної системи з локальною пам’яттю.

Математична задача: A = B\*a + C\*(MO\*MK)\*max(Z).

Структура ПКС ЛП зображена на рис. 3.1. Ввід і видів даних відбувається в процесах з номерами 1 і Q.

 Рис. 3.1 Структура ПКС із локальною пам’яттю

**3.1. Розробка паралельного математичного алгоритму**

Відповідно до технічного завдання необхідно виконати розробку паралельного алгоритму. Паралельний математичний алгоритм можна подати у вигляді наступних етапів:

1. mz = min (ZH)



1. minz = min (minz, mzi) ЗР: minz
2. AH = BH\*a + C\*(MO\*MKH)\*minz

## **3.2. Розробка алгоритмів процесів**

**Задача Т1:**

1. Введення B, MK, Z
2. Передати BN-2H, MKN-2H, ZN-2H в Т2.
3. Прийняти a, C, MO від T2.
4. Передати BH, a, C, MO, MKH, ZH в Т1.1.
5. Обчислення mz1 = min (ZH)
6. Прийняти mz1.1 від Т1.1.
7. Обчислення mz1 = min (mz1, mz1.1)
8. Передати mz1 в Т2.
9. Прийняти minzвід Т2.
10. Передати minzв Т1.1.
11. Обчислення AH = BH\*a + C\*(MO\*MKH)\*minz
12. Прийняти АН від Т1.1.
13. Прийняти АN-2H від Т2.
14. Виведення А.

**Задача ТQ, Q = (P/4)+1, Q** ≠ **P/2:**

1. Прийняти a, C, MO від ТQ+1.
2. Передати a, C, MO в ТQ-1.
3. Прийняти B(N-2Q)H, MK(N-2Q)H, Z(N-2Q)H від ТQ-1.
4. Передати B(N-2Q-2)H, MK(N-2Q-2)H, Z(N-2Q-2)H ­в ТQ+1.
5. Передати a, BH, MKH, C, MO, ZH в ТQ.1.
6. Обчислення mzQ = min (ZH)
7. Прийняти mzQ.1 від ТQ.1.
8. Прийняти mzQ+1 від ТQ+1.
9. Прийняти mzQ-1 від ТQ-1.
10. Обчислення mzQ=min(mzQ, mzQ.1, mz Q+1, mz Q-1)
11. Передати mz1 в ТQ+1.
12. Передати mz1 в ТQ-1.
13. Передати mz1 в ТQ.1.
14. Обчислення AH = BH\*a + C\*(MO\*MKH)\*minz
15. Прийняти АН від ТQ.1.
16. Прийняти А2H(Q-1) від ТQ+1.
17. Передати А2HQ в ТQ-1.

**Задача ТQ, Q = P/2, Q** ≠ **2:**

1. Введення a, C, MO.
2. Передати a, C, MO в ТQ-1.
3. Прийняти B2H, MK2H, Z2H від ТQ-1.
4. Передати a, BH, C, MO, MKH, ZH в ТQ.1.
5. Обчислення mzQ = min (ZH)
6. Прийняти mzQ.1 від ТQ.1.
7. Обчислення mzQ=min(mzQ, mz1.1)
8. Передати mzQ в ТQ-1.
9. Прийняти mzQ від ТQ-1.
10. Передати mzQ в ТQ.1.
11. Обчислення AH = BH\*a + C\*(MO\*MKH)\*minz
12. Прийняти АН від ТQ.1.
13. Передати А2H в ТQ-1.

**Задача ТQ, Q < (P/4)+1, Q**≠1**:**

1. Прийняти B(N-2Q)H, MK(N-2Q)H, Z(N-2Q)H від ТQ-1.
2. Передати B(N-2Q-2)H, MK(N-2Q-2)H, Z(N-2Q-2)H ­в ТQ+1.
3. Прийняти a, C, MO від ТQ+1.
4. Передати a, C, MO в ТQ-1.
5. Передати a, C, MO, MKH, BH, ZH в ТQ.1.
6. Обчислення mzQ = min(ZH)
7. Прийняти mzQ.1 від ТQ.1.
8. Прийняти mzQ-1 від ТQ-1.
9. Обчислення mzQ=min(mzQ, mzQ.1, mzQ-1)
10. Передати mzQ в ТQ+1.
11. Прийняти minzQ від ТQ+1.
12. Передати minzQ в ТQ-1.
13. Передати minzQ в ТQ.1.
14. Обчислення AH = BH\*a + C\*(MO\*MKH)\*minz
15. Прийняти АН від ТQ.1.
16. Прийняти А2H(Q-1) від ТQ+1.
17. Передати А2HQ в ТQ-1.

**Задача ТQ, Q > (P/4)+1, Q**≠ **P/2:**

1. Прийняти a, C, MO від ТQ+1.
2. Передати a, C, MO в ТQ-1.
3. Прийняти B(N-2Q)H, MK(N-2Q)H, Z(N-2Q)H від ТQ-1.
4. Передати B(N-2Q-2)H, MK(N-2Q-2)H, Z(N-2Q-2)H ­в ТQ+1.
5. Передати a, C, MO, MKH, BH, ZH в ТQ.1.
6. Обчислення mzQ = min(ZH)
7. Прийняти mzQ.1 від ТQ.1.
8. Прийняти mzQ-1 від ТQ+1.
9. Обчислення minzQ=min(mzQ, mzQ.1, mz Q+1)
10. Передати minzQ в ТQ-1.
11. Прийняти minzQ від ТQ-1.
12. Передати minzQ в ТQ+1.
13. Передати minzQ в ТQ.1.
14. Обчислення AH = BH\*a + C\*(MO\*MKH)\*minz
15. Прийняти АН від ТQ.1.
16. Прийняти А2H(Q-1) від ТQ+1.
17. Передати А2HQ в ТQ-1.

**Задача ТQ.1, Q= 1..P/2:**

1. Прийняти a, C, MO, MKH, BH, ZH від ТQ.
2. Обчислення mzQ = min(ZH)
3. Передати mzQ в ТQ.
4. Прийняти minzQ від ТQ.
5. Обчислення AH = BH\*a + C\*(MO\*MKH)\*minz
6. Передати АH в ТQ.
   1. **Розробка схеми взаємодії процесів**

На основі алгоритму для всіх типів задач розроблено структурні схеми взаємодії задач (рис. 3.2). За допомогою цих схем можна наочно спостерігати як саме відбувається пересилка даних. Таке графічне зображення дозволяє простіше зрозуміти механізм пересилки повідомлень та взаємодії процесів.

Усі задачі логічно розбиваються на декілька типів:

1. Перша задача («крайня зліва»), в якій вводяться дані (*B, MK, Z*).



1. Остання задача («крайня зправа»), в якій вводяться дані (*a, C, MO*).



1. Середня «проміжна» задача ТQ, Q = (P/4)+1, в якій збирається результат обчислення і яка слугує посередником при початковій пересилці даних (щоб не виникало взаємного блокування при рандеву).



1. «Проміжні» задачі ліворуч від середньої - ТQ, Q < (P/4)+1, Q≠1.
2. «Проміжні» задачі праворуч від середньої - ТQ, Q > (P/4)+1, Q≠ P/2.
3. «Нижні» задачі ТQ.1, Q= 1..P/2.

Таке логічне розбиття на типи гарантує, що пересилка даних та збір і розповсюдження проміжних результатів буде виковуватися за кількість тактів, що дорівнює діаметру системи. Тобто, найбільш оптимально.

Передача повідомлень між процесами та синхронізація відбувається за допомогою викликів таких функцій:

MPI\_Recv – функція блокує процес поки не будуть отриманні всі данні.

MPI\_Send – функція блокує процес поки не будуть передані всі дані.

MPI\_Isend – функція викликає передачу даних без блокування процесу.

MPI\_Wait – очікування виконання умови (наприклад, очікування завершення передачі даних).



Рис. 3.2. Структурна схема взаємодії процесів ПРГ2 для ПКС ЛП

**3.4 Розробка програми ПРГ2**

Програма ПРГ2 згідно технічного завдання розроблена на мові програмування C++ із використанням засобів бібліотеки MPI.

Програма складається з файлів Matrix.h, Matrix.cpp, CourseWork(mpi).cpp. У файлі Matrix.h розміщується об’явлення класу Matrix та його конструкторів і функцій. У файлі Matrix.cpp розміщується визначення конструкторів і функцій класу Matrix:

getRows() – повертає кількість рядків у матриці;

getColumns() – повертає кількість стовпців у матриці;

getPtrToArray() – вказівник на перший елемент матриці;

input() – заповнення елементів матриці значенням константи FILLER;

copy(Matrix \*\*copyMatrix) – копіює матрицю;

mult(Matrix \*\*res, Matrix \*\*multMatr) – множення матриць;

getMin() – пошук мінімуму по всій матриці;

getMin(int from, int to) – пошук мінімуму в заданому участку;

delExcessive(int from, int to) – видалення усіх елементів, окрім діапазону, вказаному в аргументах. Це необхідно для зменшення використання пам’ятті при пересилках матриць. Після закінчення передачі матриці, виконується видалення зайвих елементів.

merge(Matrix \*\*MAh) – об’єднання двох матриць.

У класі Source(mpi).cpp визначений метод main(), в якому запускаються процеси, а також константи *N, P, H,* допоміжні функції та процеси.

Лістинг розробленої програми наведено у додатку Б.

Алгоритм роботи задач наведено у додатку B.

**3.5 Тестування програми ПРГ 2**

Метою проведення тестування є оцінка коефіцієнтів прискорення і коефіцієнтів ефективності для розроблених програм при їх виконанні на реальній паралельній обчислювальній системі. Для визначення вищевказаних коефіцієнтів проведений ряд експериментів із різними розмінностями операндів (N = 1200, 1500, 1800, 2100) і різною кількістю працюючих ядер ( P = 1, 2, 3, 4).

Для виміру часу використовується процедура clock() бібліотеки time.h мови програмування С++, що повертає поточний час системи. Різниця між часом системи після завершення обчислення і часом при запуску програми дає час виконання обчислення, що вимірюється в мілісекундах.

Для оцінки ефективності програми використовуються коефіцієнти прискорення та ефективності.

Коефіцієнт прискорення показує скорочення часу виконання паралельної програми в паралельній системі з процесорами в порівнянні з часом виконання послідовної програми в однопроцесорній системі :



Коефіцієнт ефективності застосування комп’ютерної системи показує ступінь використання процесорів системи:



Результати тестування і проведених досліджень ефективності розробленої програми наведено в табл. 3.1-3.3.

Таблиця 3.1. Час виконання програми ПРГ2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 1200 | 15 | 8 | 6 | 4 |
| 1500 | 29 | 15 | 11 | 9 |
| 1800 | 50 | 26 | 19 | 15 |
| 2100 | 85 | 44 | 32 | 24 |

На основі даних із табл. 3.1 виконано розрахунок значень коефіцієнтів прискорення, які наведені в табл. 3.2.

Таблиця 3.2. Коефіцієнти прискорення для програми ПРГ2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | P | | | |
| 1 | 2 | 3 | 4 |
| 1200 | 1 | 1,875 | 2,5 | 3,75 |
| 1500 | 1 | 1,933333 | 2,636364 | 3,222222 |
| 1800 | 1 | 1,923077 | 2,631579 | 3,333333 |
| 2100 | 1 | 1,931818 | 2,65625 | 3,541667 |

Коефіцієнти ефективності табл. 3.3 обчислено за даними табл. 3.2.

Таблиця 3.3. Коефіцієнти ефективності для програми ПРГ2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 1200 | 100 | 93,75 | 83,33333 | 93,75 |
| 1500 | 100 | 96,66667 | 87,87879 | 80,55556 |
| 1800 | 100 | 96,15385 | 87,7193 | 83,33333 |
| 2100 | 100 | 96,59091 | 88,54167 | 88,54167 |

Використовуючи табл. 3.1‑3.3 побудовано графіки зміни часу виконання обчислень та коефіцієнтів прискорення і ефективності в залежності від *N* і (рис. 3.3 – 3.5).

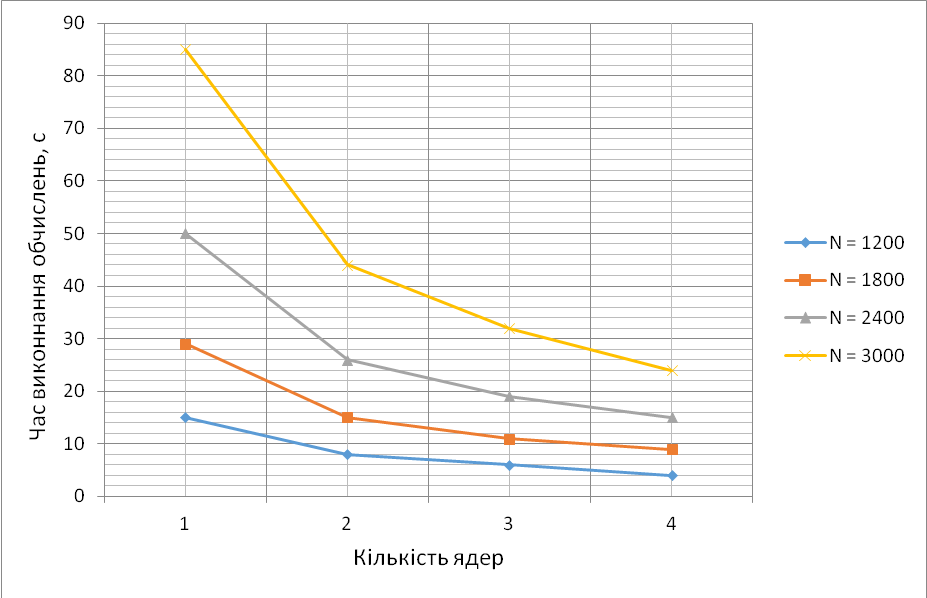


Рис. 3.4. Графік зміни часу виконання обчислень програми ПРГ1 в залежності від кількості ядер

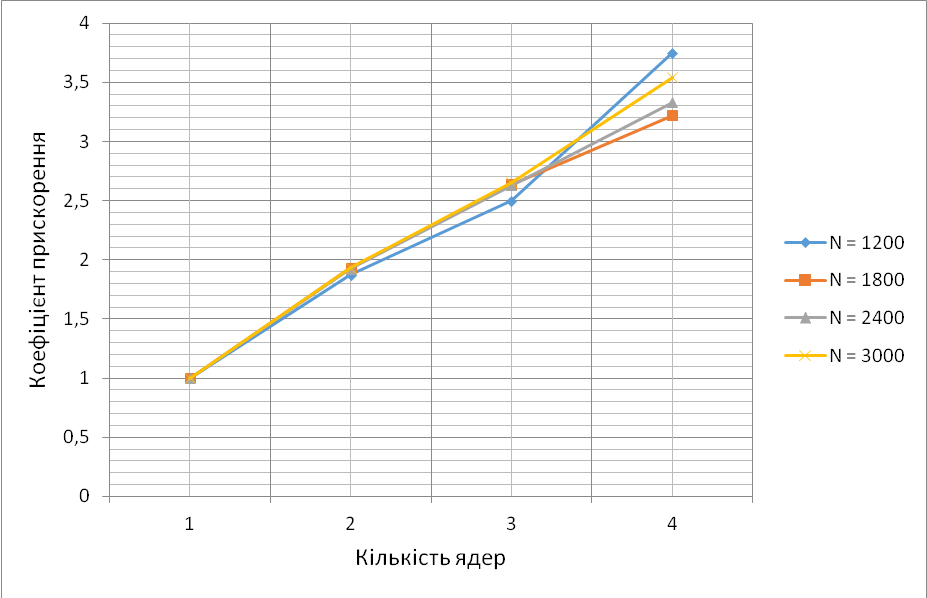


Рис. 3.5. Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер

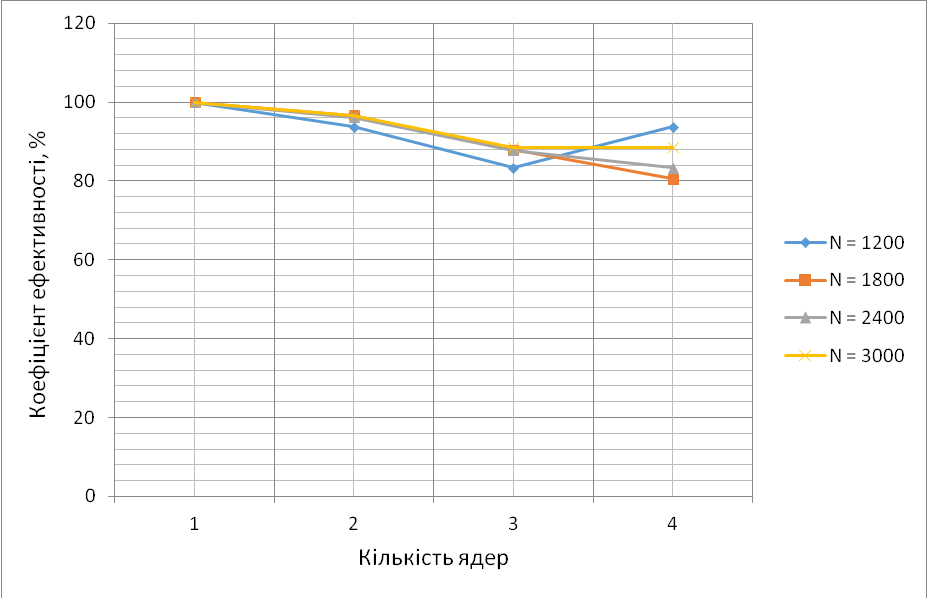


Рис. 3.6. Графік зміни коефіцієнту ефективності програми ПРГ2 в залежності від кількості ядер

**3.6 Висновки до Розділу 3**

Виконано розробку програми ПРГ2 для ПКС із ЛП. При розробці використовувалась мова програмування C++. Для передачі повідомлень між процесами використовувалася бібліотека MPI.

На основі даних тестування програми ПРГ2 заповнено таблиці 3.1-3.3 та побудовано відповідні графіки, на яких показано наступні результати:

* Використання багатоядерної ПКС забезпечує скорочення часу обчислення заданої математичної задачі.
* Максимальне значення = 3,75 забезпечує ПКС з Р = 4 та N=1200.



* Мінімальне значення = 1,875 забезпечує ПКС з Р = 2 та N=1200.



* По графіку з рис. 3.5 видно, що коефіцієнт прискорення збільшується прапорційно до кількості використовуваних ядер.
* Коефіцієнти ефективності коливаються в межах від 80-100%, це свідчить про ефективність розробленого паралельного алгоритму.
* Максимальне значення = 96,66667% для ПКС з Р = 2 та N = 1500;



* Мінімальне значення = 80,55556% для ПКС з Р = 4 та N = 1500.



Не зважаючи на те, що пересилка даних між процессами – досить затратна операція, реалізація задачі з допомогою бібліотеки MPI демострує досить ефективне використання ядер.

Оцінивши значення та можна зробити висновок про доцільність використання бібліотеки MPI для роз`в’язання задач такого типу.



**ВИСНОВКИ**

В данній курсовій роботі було розглянуто та порівняно існуючі 12-ядерні процесори виробництва AMD. Вони відрізняються від інших існуючих аналогів найкращим співвідношенням ціна/вартість, універсальністю сокету та спеціальною шиною для сполучення між ядрами зі швидкістю до 33.1ГБ/с.

Щодо ефективності використання паралельних засобів програмування то OpenMP зарекомендувало себе значно кращим за MPI, наприклад, для 4 ядер коефіцієнти прискорення в нього 3.74 на відміну від 3.5 в конкуренту. Щодо доцільності використання даних засобів, найбільш ефективно використовувати комбіновані варіанти розгортання кластерних систем з багатопотоковими процесорами.

Також були вдосконалені навички з розробки програмних засобів для паралельних обчислень зі спільною та розподіленою пам’яттю.**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Stress test: AMD Opteron Magny-Cours versus Intel Xeon Nehalem-EX [Електронний ресурс] - <http://www.infoworld.com/article/2624367/servers/stress-test--amd-opteron-magny-cours-versus-intel-xeon-nehalem-ex.html?page=2>
2. Intel Xeon MP X7560 [Електронний ресурс] - <http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20MP%20X7560%20-%20AT80604004869AA%20(BX80604X7560).html>
3. Теоретический осмотр архитектуры AMD Bulldozer [Електронний ресурс] - [http://supercomputers.ru/](http://supercomputers.ru/index.php?option=com_k2&view=item&id=308:%D0%B0%D1%82%D0%B0%D0%BA%D0%B0-%D0%B1%D1%83%D0%BB%D1%8C%D0%B4%D0%BE%D0%B7%D0%B5%D1%80%D0%BE%D0%BC&tmpl=component&print=1)
4. The new Opteron 6300: Finally Tested [Електронний ресурс] - <http://www.anandtech.com/show/6508/the-new-opteron-6300-finally-tested/14>
5. Bulldozer, как он есть. Обзор и тестирование процессора AMD FX 8120 [Електронний ресурс] - <http://www.overclockers.ua/cpu/amd-bulldozer-fx-8120/all/>
6. Обзор процессора AMD FX-8350 [Електронний ресурс] - <http://www.3dnews.ru/636955>
7. AMD Trinity для десктопа [Електронний ресурс] - <http://www.3dnews.ru/cpu/635804/>
8. AMD Opteron 6300 series microprocessor family [Електронний ресурс] - <http://www.cpu-world.com/CPUs/Bulldozer/TYPE-Opteron%206300%20series.html>
9. BEST SERVER CPUS [Електронний ресурс] - <http://cpuboss.com/cpus/Server-CPUs-870880>
10. AMD Official Website [Електронний ресурс] - <http://www.amd.com/>
11. AMD K10[Електронний ресурс] - <https://ru.wikipedia.org/wiki/AMD_K10>
12. Семейство AMD Opteron 6300 ("Abu Dhabi") - Piledriver для серверов [Електронний ресурс] - <http://www.hardwareluxx.ru/index.php/news/hardware/prozessoren/23778-amd-opteron-6300-serie-piledriver-.html>
13. AMD Opteron 6174 vs Intel Xeon X5650 Review [Електронний ресурс] - <http://www.bit-tech.net/hardware/cpus/2010/03/31/amd-opteron-6174-vs-intel-xeon-x5650-review/4>
14. PassMark - CPU Benchmarks - List of Benchmarked CPUs [Електронний ресурс] - <https://cpubenchmark.net/cpu_list.php>

Додаток А

Лістинг ПРГ1  
Додаток 1: лістинг коду

//============================================================================

// Name : Project1.cpp

// Author : Ruzhevsky Nikita

// Copyright : Your copyright notice

// Description : PRO courseproject, part2

//============================================================================

#include <iostream>

#include <omp.h>

#include <limits.h>

using namespace std;

const int n = 800;

const int p = 8;

int \*\*createMatrix() {

int\*\* array = new int\*[n];

for (int i = 0; i < n; i++) {

array[i] = new int[n];

for (int j = 0; j < n; j++) {

array[i][j] = 1;

}

}

return array;

}

int \*createVector() {

int\* arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = 1;

}

return arr;

}

void printMatrix(int\*\* arr) {

for (int i = 0; i < n; i++){

for (int j = 0; j < n; j++) {

cout << arr[i][j] << " ";

}

printf("\n");

}

}

void printVector(int\* arr) {

for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

printf("\n");

}

int main() {

int\*\* MO;

int\*\* MK;

int\*\* MT;

int\* A = new int[n];

int\* B;

int\* C;

int\* Z;

int alpha;

int maxz;

getchar();

omp\_set\_num\_threads(p);

double tbeg = omp\_get\_wtime();

#pragma omp parallel shared(MO, MK, MT)

{

switch (omp\_get\_thread\_num()) //Enter data

{

case 0:

cout << "task 0 enters data!\n";

B = createVector();

C = createVector();

MK = createMatrix();

break;

case p - 1:

cout << "task p-1 enters data!\n";

MO = createMatrix();

Z = createVector();

alpha = 5;

break;

default:

break;

}

MT = createMatrix();

int s = n / p; //size of block

for (int l = 0; l < p; l++) { //s - num of lent

#pragma omp barrier

#pragma omp for //MO\*MK

for (int bl = 0; bl < p; bl++) { // b - num of block

for (int i = bl\*s; i < (bl + 1)\*s; i++) {

for (int j = s \* (bl + l); j < s \* (bl + l + 1); j++) {

MT[i][j%n] = 0;

for (int k = 0; k < n; k++) {

MT[i][j%n] = MT[i][j%n] + (MO[i][k] \* MK[k][j%n]);

}

}

}

}

}

int\* tz = new int[p]; //maxZ

#pragma omp for

for (int i = 0; i < p; i++) {

tz[i] = INT\_MIN;

}

#pragma omp for

for (int i = 0; i < n; i++) {

if (Z[i] > tz[omp\_get\_thread\_num()]) {

tz[omp\_get\_thread\_num()] = Z[i];

}

}

#pragma omp single

{

maxz = tz[0];

for (int i = 1; i < p; i++) {

if (tz[i] > maxz) {

maxz = tz[i];

}

}

}

#pragma omp for //C \* (MO\*MK)

for (int i = 0; i < n; i++){

A[i] = 0;

for (int j = 0; j < n; j++){

A[i] += MT[i][j] \* C[j];

}

}

#pragma omp barrier

#pragma omp for // finish the calc

for (int i = 0; i < n; i++) {

A[i] = A[i] \* maxz;

B[i] = B[i] \* alpha;

A[i] = A[i] + B[i];

}

#pragma omp barrier

switch (omp\_get\_thread\_num()) //Enter data

{

case 0:

//printVector(A);

break;

default:

break;

}

}

double tend = omp\_get\_wtime();

cout << tend - tbeg;

getchar();

return 0;

}

#pragma once

#include <iostream>

class Matrix {

private:

static const int FILLER = 1;

//int size;

int rows;

int columns;

public:

int \*matrix;

Matrix(int size);

Matrix(int rows, int columns);

~Matrix();

int getRows();

int getColumns();

void\* getPtrToArray();//--

int\* operator [] (int row);

void input();

void copy(Matrix& copyMatrix);

void mult(Matrix& res, Matrix& multMatr);

int getMin();

int getMin(int from, int to);

void delExcessive(int from, int to);

void merge(Matrix& MAh);

};

std::ostream& operator << (std::ostream &out, Matrix &matrix);

Клас Matrix.cpp:

#include "Matrix.h"

#include <iostream>

Matrix::Matrix(int size) {

this->rows = size;

this->columns = size;

matrix = new int [size\*size];

}

Matrix::Matrix(int rows, int columns) {

this->rows = rows;

this->columns = columns;

matrix = new int [rows\*columns];

}

Matrix::~Matrix() {

delete(matrix);

}

int Matrix::getRows() {

return rows;

}

int Matrix::getColumns() {

return columns;

}

void\* Matrix::getPtrToArray(){

return (void\*)(&(\*matrix));

}

int\* Matrix::operator[](int row) {

return matrix + row\*columns;

}

void Matrix::input() {

for ( int i = 0; i < rows\*columns; i++ ) {

matrix[i] = FILLER;

}

}

void Matrix::copy(Matrix& copyMatrix) {

for ( int i = 0; i < rows; i++ ) {

for ( int j = 0; j < columns; j++ ) {

copyMatrix[i][j] = matrix[i\*columns + j];

}

}

}

void Matrix::mult(Matrix& res, Matrix& multMatr){

int\* r;

int\* mm;

int\* tm;

int rColumns = res.getColumns();

int mmColumns = multMatr.getColumns();

int tmColumns = columns;

for (int i = 0; i < mmColumns; i++)

{

r = res.matrix;

tm = matrix;

for (int j = 0; j < rows; j++)

{

r[i] = 0;

mm = multMatr.matrix;

for (int k = 0; k < rows; k++)

{

r[i] += tm[k] \* mm[i];

mm += mmColumns;

};

r += rColumns;

tm += tmColumns;

}

}

}

int Matrix::getMin(){

int min = matrix[0];

for(int i = 1; i < rows\*columns; i++){

if(min > matrix[i]){

min = matrix[i];

}

}

return min;

}

int Matrix::getMin(int from, int to){

int min = matrix[from];

for(int i = 0; i < rows; i++){

for(int j = from; j < to; j++){

if(min > matrix[i]){

min = matrix[i];

}

}

}

return min;

}

void Matrix::delExcessive(int from, int to){

int\* accum = new int[(to-from)\*rows];

for(int i = 0; i < rows; i++){

for(int j = from; j < to; j++){

accum[i\*(to-from) - from + j] = matrix[i\*columns + j];

}

}

int\* toDel = matrix;

matrix = accum;

delete[] toDel;

columns = to-from;

}

void Matrix::merge(Matrix& MAh){

int size1 = columns\*rows;

int size2 = MAh.getColumns()\*MAh.getRows();

int\* accum = new int[size1 + size2];

for(int i = 0; i < rows; i++){

for(int j = 0; j < (columns + MAh.getColumns()); j++){

if(j < columns){

accum[i\*(columns + MAh.getColumns()) + j] = matrix[i\*columns + j];

} else {

accum[i\*(columns + MAh.getColumns()) + j] = MAh[i][j-columns];

}

}

}

int\* toDel = matrix;

matrix = accum;

delete[] toDel;

columns += MAh.getColumns();

}

std::ostream& operator<<(std::ostream &out, Matrix &matrix) {

int last = matrix.getColumns() - 1;

out << "Matrix:" << std::endl;

for (int row = 0; row < matrix.getRows(); row++) {

for (int col = 0; col < last; col++)

out << matrix[row][col] << " ";

out << matrix[row][last] << std::endl;

}

return out;

};

int N = 1000;

int P = 10;

int H = N/P;

int FILLER = 1;

int first = 0;

int last = P/2 - 1;

void\* shiftPtrLeft(void\* inp, int s){

return (void\*)((int\*)inp + s\*H\*N);

}

void copyBuf(void\* from, void\* to, int size){

for(int i = 0; i < size; i++){

((int\*)to)[i] = ((int\*)from)[i];

}

}

bool isCentral(int rank){

if(rank == (P/4))

return true;

return false;

}

int findAndSendMinGlobal(int rank, int localMin){

int ei;

int bottomRank = rank + P/2;

MPI\_Status st;

if(rank != last){

MPI\_Recv(&ei, 1, MPI\_INT, rank+1, 21, MPI\_COMM\_WORLD, &st);

localMin = (ei < localMin) ? ei : localMin;

}

MPI\_Recv(&ei, 1, MPI\_INT, rank-1, 21, MPI\_COMM\_WORLD, &st);

localMin = (ei < localMin) ? ei : localMin;

MPI\_Recv(&ei, 1, MPI\_INT, bottomRank, 21, MPI\_COMM\_WORLD, &st);

localMin = (ei < localMin) ? ei : localMin;

MPI\_Request rqb, rql;

MPI\_Isend(&localMin, 1, MPI\_INT, bottomRank, 21, MPI\_COMM\_WORLD, &rqb);

MPI\_Isend(&localMin, 1, MPI\_INT, rank-1, 21, MPI\_COMM\_WORLD, &rql);

if(rank != last){

MPI\_Request rqr;

MPI\_Isend(&localMin, 1, MPI\_INT, rank+1, 21, MPI\_COMM\_WORLD, &rqr);

MPI\_Wait(&rqr, MPI\_STATUS\_IGNORE);

}

MPI\_Wait(&rqb, MPI\_STATUS\_IGNORE);

MPI\_Wait(&rql, MPI\_STATUS\_IGNORE);

return localMin;

}

int getMinGlobal(int rank, int localMin){

int globalMin;

if(isCentral(rank)){

globalMin = findAndSendMinGlobal(rank, localMin);

} else {

int ei;

MPI\_Status st;

if(rank < P/4){

MPI\_Recv(&ei, 1, MPI\_INT, rank + P/2, 21, MPI\_COMM\_WORLD, &st);

localMin = (ei < localMin) ? ei : localMin;

if(rank != first){

MPI\_Recv(&ei, 1, MPI\_INT, rank - 1, 21, MPI\_COMM\_WORLD, &st);

localMin = (ei < localMin) ? ei : localMin;

}

MPI\_Send(&localMin, 1, MPI\_INT, rank+1, 21, MPI\_COMM\_WORLD);

MPI\_Recv(&globalMin, 1, MPI\_INT, rank+1, 21, MPI\_COMM\_WORLD, &st);

if(rank != first){

MPI\_Send(&globalMin, 1, MPI\_INT, rank-1, 21, MPI\_COMM\_WORLD);

}

MPI\_Send(&globalMin, 1, MPI\_INT, rank+P/2, 21, MPI\_COMM\_WORLD);

} else {

MPI\_Recv(&ei, 1, MPI\_INT, rank + P/2, 21, MPI\_COMM\_WORLD, &st);

localMin = (ei < localMin) ? ei : localMin;

if(rank != last){

MPI\_Recv(&ei, 1, MPI\_INT, rank + 1, 21, MPI\_COMM\_WORLD, &st);

localMin = (ei < localMin) ? ei : localMin;

}

MPI\_Send(&localMin, 1, MPI\_INT, rank-1, 21, MPI\_COMM\_WORLD);

MPI\_Recv(&globalMin, 1, MPI\_INT, rank-1, 21, MPI\_COMM\_WORLD, &st);

if(rank != last){

MPI\_Send(&globalMin, 1, MPI\_INT, rank+1, 21, MPI\_COMM\_WORLD);

}

MPI\_Send(&globalMin, 1, MPI\_INT, rank+P/2, 21, MPI\_COMM\_WORLD);

}

}

return globalMin;

}

void calcMAh(Matrix& MA, int e, Matrix& MKh, int l, Matrix& ME, Matrix& MT, Matrix& MMh){

Matrix Res(N, H);

MT.mult(Res, MMh);

ME.mult(MA, Res);

for(int i = 0; i < H; i++){

for(int j = 0; j < N; j++){

MA[j][i] = e\*MKh[j][i] + l\*MA[j][i];

}

}

}

int size(Matrix& MA){

return MA.getColumns()\*MA.getRows();

}

void matrOut(Matrix& M){

for(int i = 0; i < M.getRows(); i++){

for(int j = 0; j < M.getColumns(); j++){

cout << M[i][j] << " ";

}

cout << endl;

}

}

void sendLeftMAh(int rank, Matrix& MAh){

Matrix MAh2(N, H);

MPI\_Recv(MAh2.getPtrToArray(), N\*H, MPI\_INT, rank + P/2, 31, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MAh.merge(MAh2);

if(rank != last){

Matrix MAh1(N, H\*(P - (rank+1)\*2));

MPI\_Recv(MAh1.getPtrToArray(), size(MAh1), MPI\_INT, rank+1, 31, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MAh.merge(MAh1);

}

MPI\_Send(MAh.getPtrToArray(), size(MAh), MPI\_INT, rank-1, 31, MPI\_COMM\_WORLD);

}

void threadFuncFirst(){

long transferStart1 = clock();

Matrix MO(N), ME(N);

MO.input();

ME.input();

int rightSS = N\*N - 2\*H\*N;

int rightRS = 2\*N\*H;

int fullSize = N\*N;

int bottomRank = 0 + P/2;

int flag = -1;

MPI\_Request req0;

MPI\_Isend(&flag, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, &req0);

MPI\_Request req4;

MPI\_Request req5;

MPI\_Isend(shiftPtrLeft(MO.getPtrToArray(), 2), rightSS, MPI\_INT, 1, 4, MPI\_COMM\_WORLD, &req4);

MPI\_Isend(ME.getPtrToArray(), fullSize, MPI\_INT, 1, 5, MPI\_COMM\_WORLD, &req5);

Matrix MKh(N,2\*H), MMh(N,2\*H);

Matrix MT(N);

int li;

MPI\_Status st1;

MPI\_Status st2;

MPI\_Status st3;

MPI\_Recv(&flag, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MKh.getPtrToArray(), rightRS, MPI\_INT, 1, 1, MPI\_COMM\_WORLD, &st1);

MPI\_Recv(MMh.getPtrToArray(), rightRS, MPI\_INT, 1, 2, MPI\_COMM\_WORLD, &st2);

MPI\_Recv(MT.getPtrToArray(), fullSize, MPI\_INT, 1, 3, MPI\_COMM\_WORLD, &st3);

MPI\_Recv(&li, 1, MPI\_INT, 1, 6, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Wait(&req4, MPI\_STATUS\_IGNORE);

MPI\_Wait(&req5, MPI\_STATUS\_IGNORE);

MPI\_Request reql1, reql2, reql3, reql4, reql5, reql6;

MPI\_Isend(shiftPtrLeft(MKh.getPtrToArray(), 1), N\*H, MPI\_INT, bottomRank, 11, MPI\_COMM\_WORLD, &reql1);

MPI\_Isend(shiftPtrLeft(MMh.getPtrToArray(), 1), N\*H, MPI\_INT, bottomRank, 12, MPI\_COMM\_WORLD, &reql2);

MPI\_Isend(shiftPtrLeft(MO.getPtrToArray(), 1), N\*H, MPI\_INT, bottomRank, 14, MPI\_COMM\_WORLD, &reql4);

MPI\_Isend(MT.getPtrToArray(), N\*N, MPI\_INT, bottomRank, 13, MPI\_COMM\_WORLD, &reql3);

MPI\_Isend(ME.getPtrToArray(), N\*N, MPI\_INT, bottomRank, 15, MPI\_COMM\_WORLD, &reql5);

MPI\_Isend(&li, 1, MPI\_INT, bottomRank, 16, MPI\_COMM\_WORLD, &reql6);

MPI\_Wait(&reql1, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql2, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql3, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql4, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql5, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql6, MPI\_STATUS\_IGNORE);

MO.delExcessive(0, H);

MKh.delExcessive(0, H);

MMh.delExcessive(0, H);

long transferFinish1 = clock();

long cStart1 = clock();

int ei = MO.getMin();

long cFinish1 = clock();

int e = getMinGlobal(0, ei);

Matrix MAh(N, H);

long cStart2 = clock();

calcMAh(MAh, e, MKh, li, ME, MT, MMh);

long cFinish2 = clock();

long transferStart2 = clock();

Matrix MAh1(N, H);

MPI\_Recv(MAh1.getPtrToArray(), N\*H, MPI\_INT, bottomRank, 31, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MAh.merge(MAh1);

Matrix MAh2(N, N-2\*H);

MPI\_Recv(MAh2.getPtrToArray(), N\*(N-2\*H), MPI\_INT, 1, 31, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MAh.merge(MAh2);

long transferFinish2 = clock();

cout <<"Time to transfer: ";

cout << ((transferFinish1-transferStart1) + (transferFinish2 - transferStart2))/1000 << endl;

cout <<"Time to calculate1: ";

cout << (cFinish1 - cStart1)/1000 << endl;

cout <<"Time to calculate2: ";

cout << (cFinish2 - cStart2)/1000 << endl;

if(N < 13){

cout << "Matrix MA: \n";

for(int i = 0; i < MAh.getRows(); i++){

for(int j = 0; j < MAh.getColumns(); j++){

cout << MAh[i][j] << " ";

}

cout << endl;

}

}

}

void threadFuncLast(){

int bottomRank = P-1;

Matrix MK(N), MT(N), MM(N);

MK.input();

MT.input();

MM.input();

int li = FILLER;

int flag = 1;

MPI\_Request req0;

MPI\_Isend(&flag, 1, MPI\_INT, P/2-2, 0, MPI\_COMM\_WORLD, &req0);

MPI\_Request req1;

MPI\_Request req2;

MPI\_Request req3;

MPI\_Request req6;

MPI\_Isend(MK.getPtrToArray(), N\*N - 2\*H\*N, MPI\_INT, P/2-2, 1, MPI\_COMM\_WORLD, &req1);

MPI\_Isend(MM.getPtrToArray(), N\*N - 2\*H\*N, MPI\_INT, P/2-2, 2, MPI\_COMM\_WORLD, &req2);

MPI\_Isend(MT.getPtrToArray(), N\*N, MPI\_INT, P/2-2, 3, MPI\_COMM\_WORLD, &req3);

MPI\_Isend(&li, 1, MPI\_INT, P/2-2, 6, MPI\_COMM\_WORLD, &req6);

Matrix MOh(N, 2\*H), ME(N);

MPI\_Status st4;

MPI\_Status st5;

MPI\_Recv(&flag, 1, MPI\_INT, P/2-2, 0, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MOh.getPtrToArray(), 2\*N\*H, MPI\_INT, P/2-2, 4, MPI\_COMM\_WORLD, &st4);

MPI\_Recv(ME.getPtrToArray(), N\*N, MPI\_INT, P/2-2, 5, MPI\_COMM\_WORLD, &st5);

MPI\_Wait(&req1, MPI\_STATUS\_IGNORE);

MPI\_Wait(&req2, MPI\_STATUS\_IGNORE);

MPI\_Wait(&req3, MPI\_STATUS\_IGNORE);

MPI\_Wait(&req6, MPI\_STATUS\_IGNORE);

MPI\_Request reql1, reql2, reql3, reql4, reql5/\*, reql6\*/;

MPI\_Isend(shiftPtrLeft(MK.getPtrToArray(), P - 1 ), N\*H, MPI\_INT, bottomRank, 11, MPI\_COMM\_WORLD, &reql1);

MPI\_Isend(shiftPtrLeft(MM.getPtrToArray(), 1), N\*H, MPI\_INT, bottomRank, 12, MPI\_COMM\_WORLD, &reql2);

MPI\_Isend(shiftPtrLeft(MOh.getPtrToArray(), 1), N\*H, MPI\_INT, bottomRank, 14, MPI\_COMM\_WORLD, &reql4);

MPI\_Isend(MT.getPtrToArray(), N\*N, MPI\_INT, bottomRank, 13, MPI\_COMM\_WORLD, &reql3);

MPI\_Isend(ME.getPtrToArray(), N\*N, MPI\_INT, bottomRank, 15, MPI\_COMM\_WORLD, &reql5);

MPI\_Isend(&li, 1, MPI\_INT, bottomRank, 16, MPI\_COMM\_WORLD, &reql5);

MPI\_Wait(&reql1, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql2, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql3, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql4, MPI\_STATUS\_IGNORE);

MPI\_Wait(&reql5, MPI\_STATUS\_IGNORE);

MK.delExcessive((P-2)\*H, (P-1)\*H);

MM.delExcessive((P-2)\*H, (P-1)\*H);

MOh.delExcessive(0, H);

int ei = MOh.getMin(0, H);

int e = getMinGlobal(last, ei);

Matrix MAh(N, H);

calcMAh(MAh, e, MK, li, ME, MT, MM);

sendLeftMAh(last, MAh);

}

void threadFuncMed(int rank){

int bottomRank = rank + P/2;

int rightSS = (P - rank\*2 - 2)\*H\*N;

int rightSR = (rank\*2 + 2)\*H\*N;

int leftSS = rank\*H\*2\*N;

int leftSR = (P-2\*rank)\*H\*N;

int flag;

MPI\_Request rq[7];

MPI\_Request rqf;

MPI\_Recv(&flag, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

Matrix MKh(N, rightSR/N), MMh(N, rightSR/N), MT(N);

int li;

Matrix MOh(N, leftSR/N), ME(N);

if(flag == 1){

MPI\_Recv(MKh.getPtrToArray(), rightSR, MPI\_INT, rank+1, 1, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MMh.getPtrToArray(), rightSR, MPI\_INT, rank+1, 2, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MT.getPtrToArray(), size(MT), MPI\_INT, rank+1, 3, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(&li, 1, MPI\_INT, rank+1, 6, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Isend(&flag, 1, MPI\_INT, rank-1, 0, MPI\_COMM\_WORLD, &rq[0]);

MPI\_Isend(MKh.getPtrToArray(), leftSS, MPI\_INT, rank-1, 1, MPI\_COMM\_WORLD, &rq[1]);

MPI\_Isend(MMh.getPtrToArray(), leftSS, MPI\_INT, rank-1, 2, MPI\_COMM\_WORLD, &rq[2]);

MPI\_Isend(MT.getPtrToArray(), size(MT), MPI\_INT, rank-1, 3, MPI\_COMM\_WORLD, &rq[3]);

MPI\_Isend(&li, 1, MPI\_INT, rank-1, 6, MPI\_COMM\_WORLD, &rq[6]);

MPI\_Recv(&flag, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MOh.getPtrToArray(), leftSR, MPI\_INT, rank-1, 4, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(ME.getPtrToArray(), size(ME), MPI\_INT, rank-1, 5, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Isend(&flag, 1, MPI\_INT, rank+1, 0, MPI\_COMM\_WORLD, &rqf);

MPI\_Isend(shiftPtrLeft(MOh.getPtrToArray(), 2), rightSS, MPI\_INT, rank+1, 4, MPI\_COMM\_WORLD, &rq[4]);

MPI\_Isend(ME.getPtrToArray(), size(ME), MPI\_INT, rank+1, 5, MPI\_COMM\_WORLD, &rq[5]);

} else {

MPI\_Recv(MOh.getPtrToArray(), leftSR, MPI\_INT, rank-1, 4, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(ME.getPtrToArray(), size(ME), MPI\_INT, rank-1, 5, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Isend(&flag, 1, MPI\_INT, rank+1, 0, MPI\_COMM\_WORLD, &rqf);

MPI\_Isend(shiftPtrLeft(MOh.getPtrToArray(), 2), rightSS, MPI\_INT, rank+1, 4, MPI\_COMM\_WORLD, &rq[4]);

MPI\_Isend(ME.getPtrToArray(), size(ME), MPI\_INT, rank+1, 5, MPI\_COMM\_WORLD, &rq[5]);

MPI\_Recv(&flag, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MKh.getPtrToArray(), rightSR, MPI\_INT, rank+1, 1, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MMh.getPtrToArray(), rightSR, MPI\_INT, rank+1, 2, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(MT.getPtrToArray(), size(MT), MPI\_INT, rank+1, 3, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Recv(&li, 1, MPI\_INT, rank+1, 6, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

MPI\_Isend(&flag, 1, MPI\_INT, rank-1, 0, MPI\_COMM\_WORLD, &rq[0]);

MPI\_Isend(MKh.getPtrToArray(), leftSS, MPI\_INT, rank-1, 1, MPI\_COMM\_WORLD, &rq[1]);

MPI\_Isend(MMh.getPtrToArray(), leftSS, MPI\_INT, rank-1, 2, MPI\_COMM\_WORLD, &rq[2]);

MPI\_Isend(MT.getPtrToArray(), size(MT), MPI\_INT, rank-1, 3, MPI\_COMM\_WORLD, &rq[3]);

MPI\_Isend(&li, 1, MPI\_INT, rank-1, 6, MPI\_COMM\_WORLD, &rq[6]);

}

for(int i = 0; i < 7; i++){

MPI\_Wait(&rq[i], MPI\_STATUSES\_IGNORE);

}

MPI\_Wait(&rqf, MPI\_STATUSES\_IGNORE);

MKh.delExcessive(size(MKh)/N-2\*H, size(MKh)/N);

MMh.delExcessive(size(MMh)/N-2\*H, size(MMh)/N);

MOh.delExcessive(0, 2\*H);

MPI\_Request rq1[6];

MPI\_Isend(shiftPtrLeft(MKh.getPtrToArray(), 1), N\*H, MPI\_INT, bottomRank, 11, MPI\_COMM\_WORLD, &rq1[0]);

MPI\_Isend(shiftPtrLeft(MMh.getPtrToArray(), 1), size(MMh) - N\*H, MPI\_INT, bottomRank, 12, MPI\_COMM\_WORLD, &rq1[1]);

MPI\_Isend(MT.getPtrToArray(), size(MT), MPI\_INT, bottomRank, 13, MPI\_COMM\_WORLD, &rq1[2]);

MPI\_Isend(shiftPtrLeft(MOh.getPtrToArray(), 1), size(MOh) - N\*H, MPI\_INT, bottomRank, 14, MPI\_COMM\_WORLD, &rq1[3]);

MPI\_Isend(ME.getPtrToArray(), size(ME), MPI\_INT, bottomRank, 15, MPI\_COMM\_WORLD, &rq1[4]);

MPI\_Isend(&li, 1, MPI\_INT, bottomRank, 16, MPI\_COMM\_WORLD, &rq1[5]);

for(int i = 0; i < 6; i++){

MPI\_Wait(&rq1[i], MPI\_STATUSES\_IGNORE);

}

MKh.delExcessive(0, H);

MMh.delExcessive(0, H);

MOh.delExcessive(0, H);

int ei = MOh.getMin(0, H);

int e = getMinGlobal(rank, ei);

Matrix MAh(N, H);

calcMAh(MAh, e, MKh, li, ME, MT, MMh);

sendLeftMAh(rank, MAh);

}

void threadFuncBottom(int rank){

Matrix MOh(N, H), MKh(N, H), MMh(N, H), MT(N), ME(N);

int li;

MPI\_Status st1;

MPI\_Status st2;

MPI\_Status st3;

MPI\_Status st4;

MPI\_Status st5;

MPI\_Status st6;

MPI\_Recv(MKh.getPtrToArray(), N\*H, MPI\_INT, rank - P/2, 11, MPI\_COMM\_WORLD, &st1);

MPI\_Recv(MMh.getPtrToArray(), N\*H, MPI\_INT, rank - P/2, 12, MPI\_COMM\_WORLD, &st2);

MPI\_Recv(MT.getPtrToArray(), N\*N, MPI\_INT, rank - P/2, 13, MPI\_COMM\_WORLD, &st3);

MPI\_Recv(MOh.getPtrToArray(), N\*H, MPI\_INT, rank - P/2, 14, MPI\_COMM\_WORLD, &st4);

MPI\_Recv(ME.getPtrToArray(), N\*N, MPI\_INT, rank - P/2, 15, MPI\_COMM\_WORLD, &st5);

MPI\_Recv(&li, 1, MPI\_INT, rank - P/2, 16, MPI\_COMM\_WORLD, &st6);

int ei = MOh.getMin();

MPI\_Send(&ei, 1, MPI\_INT, rank - P/2, 21, MPI\_COMM\_WORLD);

int e;

MPI\_Recv(&e, 1, MPI\_INT, rank - P/2, 21, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

Matrix MAh(N, H);

calcMAh(MAh, e, MKh, li, ME, MT, MMh);

MPI\_Send(MAh.getPtrToArray(), N\*H, MPI\_INT, rank - P/2, 31, MPI\_COMM\_WORLD);

}

int main(int argc, char\* argv[])

{

MPI\_Init(&argc, &argv);

long tStart = clock();

int rank;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &P);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

H = N/P;

last = P/2 - 1;

cout << "Thread " << rank << " started.\n";

if(rank == first){

threadFuncFirst();

} else {

if(rank == last){

threadFuncLast();

} else {

if(rank > (last)){

threadFuncBottom(rank);

} else {

threadFuncMed(rank);

}}

}

cout << "Thread " << rank << " finish.\n";

MPI\_Finalize();

if(rank == first){

long tFinish = clock();

cout << endl << "time: " << (tFinish - tStart)/1000 << "s " << (tFinish - tStart)%1000 << "ms" << endl;

char ch;

cin >> ch;

}

return 0;

}