НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Кафедра обчислювальної техніки
(повна назва кафедри, циклової комісії)

КУРСОВА РОБОТА

з дисципліни «Паралельне програмування»
(назва дисципліни)

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

Студента (ки) 3 курсу групи ІП-42
напряму підготовки 050103 «Програмна інженерія»

Дзюби В.В._____
(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- 2017 р.

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 6.050103 «Програмна інженерія»

(шифр і назва)

З А В Д А Н Н Я

НА КУРСОВУ РОБОТУ СТУДЕНТУ

Дзюби Влада Володимировича
(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 11 травня 2017 р.

3. Вхідні дані до роботи
- засоби роботи з потоками в бібліотеці OpenMP
- математична задача $A = max(Z) \cdot E \cdot (MO \cdot MT) + sort(S)$
- структури ПКС ОП та ПКС ЛП
- мови і бібліотеки програмування: C++, OpenMP.
- засоби організації взаємодії процесів: цикли for.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд засобів роботи з процесами в мові Ада
- розробка і тестування програми ПРГ1 для ПКС ОП
- розробка і тестування програми ПРГ2 для ПКС ЛП

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.

7. Дата видачі завдання _____9.03.2017_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання КР	Строк виконання етапів КР
1	Виконання розділу 1	13.03.2017
2	Виконання розділу 2	3.04.2017
3	Виконання розділу 3	23.04.2016
4	Оформлення КР	8.05.2016
5	Перевірка КР викладачем	11.05.2016
6	Захист КР	18.05.2016

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

ЗМІСТ

Вступ

Метою курсовою роботи є аналіз та використання засобів розробки параллельних програм з використання OpemMP. Прикладом використання є параллельне обчислення математичної задачі.

З ростом виробництва комп’ютерних процесорів вдосконалення засобів комунікації між багатьма процессорами стає важлиівшою за вдосконалення окремих процессорів. OpenMP є тим стандартом, який дозволяє писати параллельни програми максимально коротко та зрозуміло.

Не менш важливим є питання вибору системи пам’яті для обчислювальної техніки: спільна пам’ять або локальна пам’ять. Система зі спільною пам’яттю є більш швидкими. Системи ж з локальною пам’яттю є більш структурованими та простішими для сприйняття та написання програм.

У першому розділі розглянуто засоби роботи з потокми бібліотеки OpenMP. У другому розділі описан процес розробки програм для систем зі спільною пам’яттю. У третьому розділі описан процес розробки програм для систем з локальною пам’яттю.

РОЗДІЛ 1. ОГЛЯД БІБЛІОТЕКИ OPENMP

1.1. Загальний опис.

OpenMP — API, призначений для розробки багатопоточних додатків для багатопроцесорних систем з спільною пам’яттю. Розробку специфікації ведуть декілька великих виробників обчислювальної техніки та програмного забезпечення. *OpenMP* підтримується основними компіляторами *C*/*C++*/*Fortran*.

В *OpenMP* немає явного задання потоків у коді. Натомість є можливість вказати компілятору з допомогою директив *#pragma*, що блок кода може бути розпаралелен. Знаючи цю інформацію, компілятор в стані згенерувати додаток, яке складається з одного головного потока, який створює багато інших потоків для паралельного блоку коду. Ці потоки синхронізуються у кінці паралельного блоку кода, вертаючись к головному потоку.

Так як *OpenMP* контролюється прагамами, то код на *C++* коректно скомпілюється будь-яким компілятором *C++*, тому що прагми, які не підтримуються, повинні ігноруватись. Проте *OpenMP API* містить також декілька функцій, та, щоб їх використати, необхідно підключити заголовочний файл. Найпростіший спосіб визначити, чи підтримує компілятор *OpenMP* — спробувати підключити *omp.h*.

Якщо *OpenMP* підтримується, потрібно його ввімкнути за допомогою спеціального флагу компілятора: *g++ -fopenmp.*

1.2. Синтаксис

Директиви *OpenMP* починаються з *#pragma omp*.

Parallel

Директива *#pragma omp parallel* створює групу з *N* потоків. *N* визначається під час виконання, загалом це кількість ядер процесора, але також можно задать *N* вручну.

Кожен з потоків у групі виконує наступну за директивою команду (або блок команд, визначений в фігурних дужках). Після виконання, потоки “зливаються” в один.

Такий прозорий спосіб розмітки паралельної програми дозволяє мінімізувати написання коду для породження *N* потоків, задання їм функції виконання та синхронізації закінчення їх виконання. В наслідок того, що така задача є поширеною, *OpenMP* пропонує простий синтаксис для її вирішення.

Директива for

Директива *for* розділяє цикл між поточною групою потоків, так що кожен потік у группі оброблює свою частину цикла. Таким чином, кожен потів оброблює свою частину цикла параллельно з усіма потоками. При цьому послідовність виконання ітерацій цикла довільна.

Важливо, що директива *#pragma omp л*ише делегує порції цикла різним потокам у поточній групі потоків. В момент запуска програми група з одиничного (головного) потока. Щоб створити нову групу потоків, необхідно використати ключове слово *parallel*.

Для того, щоб задати кількість потоків у групі, можна скористатись параметром *num_threads*.

В *OpenMP* 2.5, ітераційна змінна цикла повинна бути типа *signed*. В *OpenMP* 3.0 вона також може мати тип *unsigned integer,* може бути вказівником або *constant-time random access* ітератором. В останньому випадку, для визначення кількості ітерацій цикла буде використовуватись *std::distance()*.

Перевагою синтаксису *OpenMP* для паралельного проходу цикла є автоматичне розбиття ітерацій між наявною групою потоків. При цьому в коді модифікується тільки директиви, що призводить до того, що без *OpenMP*, отримуємо простий цикл, який працює таким самим чином проте без паралельності.

1.2.3. Планування

Програміст може контролювати то, яким чином потоки будуть завантажуватись роботою при обробці цикла. Існують декілька варіантів.

Static є варіантом за замовчуванням. Ще до входу в цикл кожний поток “знає”, які частини цикла від буде обробляти. Такий варіант підходить, коли варіація часу виконання частини ітерації одним потоком мінімальна. При цьому балансування, яке проводить *OpenMP*, повністю задоволняє необхідність в ній. Цей варіант поширен настільки, що розробники *OpenMP* зробили цю стратегію балансування за замовчуванням.

Використовуючи d*ynamic*, неможливо передбачити порядок, в якому ітерації цикла будуть призначені потокам. Кожен потік виконує вказану кількість ітерацій. Якщо це число не задане, за замовчуванням воно дорівнює 1. Після того, як потік завершить виконання заданих ітерацій, він переходить до наступного набору ітерацій. Так продовжується, поки не будуть пройдені всі ітерації. Останній набір ітерацій може бути меншим, ніж заданий на початку. Такий варіант дуже корисний, коли різні ітерації цикла обраховуються різний час. Можна також вказати кількість ітерацій, після виконання яких поток “попросить” у *OpenMP* наступні.

Існує також варіант *guided*. Він схожий на *dynamic*, проте розмір порції зменьшується експоненційно. Така стратегія упорядкування дозволяє боротись з проблемою, яка виникає, коли відношення потоків до ітерацій циклу велике. Тоді видатки на створення різних потоків перевищують дохід від розпаралелення. Така стратегія схожа на тип алгоритмів “жадібні алгоритми”. Вона не гарантує найкраший результат, проте є оптимальною в багатьох випадках.

Динамічне та кероване планування добре підходять, коли при кожній ітерації виконується різні об’єми роботи або якщо процесори більш виробничими, ніж інші. При статичному плануванні немає способу, який дозволяє сбалансувати нагрузку на різні потоки. При динамічному та керованому плануванні нагрузка розподіляється автоматично — такова сама суть цих підходів. Як правило, при керованому плануванні

код виконується швидше, ніж при динамічному, в наслідок менших витрат на планування.

***Ordering* (впорядкування)**

Порядок, в якому будуть обрабляти ітерації цикла непердбачуваним. Проте можливо “змусити” *OpenMP* виконувати вирази в циклі в порядку. Для цього існує ключеве слово *ordered*[1].

Хоча більшість ітеративних задач є аккумулятивними, тобто такими, для яких немає різниці у порядку виконання ітерацій, існують і ті, що потребують порядку. Для задач таких, як сумування, множення, перетворення кожного елементу масиву таким чином, який не залежить від значень інших елементів масиву, видбір елементів масиву, що відповідають певному крітерію, немає різниці в порядку. Проте для копіювання масиву, створення нового масиву з зворотнім порядком, виводу в один і той же *stream* грає роль порядок. Саме для таких задач є можливість вказання порядку для паралелювання цикла в *OpenMP*.

Директива *sections*

Неітеративна паралельна конструкція. Визначає набір незалежних секцій кода (“кінечний паралелізм”). Секції відділяються друг від друга директивой. На відміну від директиви *parallel* секції кода виконується не всіма потоками, а кожна секція виконується окрмемим потоком. Це гарна заміна використанню функції *OPM_GET_THREADS_NUM* з конструкцією *switch*, яка теж дозволяє задати, що робить поток, основуючись на його номеру по порядку. Директива *sections* менш об’ємна, проте не залежить від кількості наявних потоків. Якщо *switch* приймає у клаузи *case* лише константні вирази, що говорить про те, що кількість потоків не можливо змінити динамічно, то *OpenMP* сам проводить розрахунок логіки, за якою розподіляються паралельні секція по потах. Це показує, що якщо потоків більше ніж секція або навпаки, *OpenMP* має готовий алгоритм для таких ситуацій.

Директива *single*

Визначає блок кода, який виконається тільки одним потоком (першим, який дійде до цього блока). Така можливість дає змогу легко проводити ініціалізації різних структур або ресурсів. В деяких випадках повторна ініціалізація змінних може затерти результати, які вже були отримані іншим потоком. Ще більш неприйнятна ситуація трапляється, коли ініціалізованою структурою не проста зміна типу число, буква, рядок чи флаг, а структурою є багаторозмірний масив, дерево чи масив, який потребує початкового сортування. Тоді обчислювальноємкі задачі будуть повторно витрачати час процесора, що є дуже неефективно. Ця неефективність походить від багатьох непотрібних обчислень, та існує неефективність, пов’язана з витратою великої кількості пам’яті. Якщо робота з одним і тим же зовнішним ресурсом, такими як файл, база даних, віддалений сервер, спричинятиме відкриття цього ресурса в кожному потоці, це призвиде до великої кількості відкритих дискрипторів в операційній системи, кожен з яких через паралельність може не закриватись або закриватись двічі. Саме ці проблеми вирішує директива *single*.

Явне керування розподіленням роботи

За допомогою функції *OMP_GET_THREAD_NUM* та *OMP_GET_NUM_THREADS* потік може визначити свій номер та загальну кількість потоків, а далі виконати свою частину роботи в залежності від свого номеру (цей підхід широко виспользується у програмах на базі інтерфейса).

Директиви синхронізації

Master — визначає блок коду, який виконається тільки *master*-ом (нульовим потоком). Ця директива схоже на директиву *single*, проте несе інший логічний зміст. Ця директива виконується нульовим потоком, тоже дії в серединні блока коду повинні бути направленні на керування потоками, ніж на обчислення. До того ж на відміну від директиви *single* код блока виконається раніше, бо нульовий потік в загальному випадку має менш задач ніж будь-який інший.

Critical — визначає критичну секцію, блок кода, яикй не повинен виконуватись одночасно двумя або більшою кількістю потоків. Критична секція — розповсюджений паттерн паралельного програмування. ЇЇ реалізують різноманітними засобами: мьютексами, семафорами, моніторами. Розробники *OpenMP* винесли цей паттерн в окрему абстракцію, що призвело до зручності написання коду: не потрібно створювати додаткові об’єкти, *OpenMP* автоматизує цей процес.

Barrier — визначає точку бар’єрної синхронизації, в якій кожний потік чекає всіх інших. Ця директива відіграє роль функції *join* в моделі *fork/join*. При цьому *fork*-ом є початок паралельної секції. Такий елемент синхронізації є зручним для неідеально паралельних задач або поєднання двох паралельних задач так, щоб початок першого потоку другої задачі знаходився після кінця останього потоку першої задачі. Ця директива аналогічна послідовної об’яві двох паралельних секцій.

TaskWait — визначає очікування для закінчення задач потомків потоків, які були сгенеровані після початку поточної задачі.

Зважаючи на те, що конструкція *taskwait* немає виразу мови *C*, як чистини свого синтаксису, присутні деякі заборони для вибору місця використання у програмі. Директива *taskwait* може бути розміщена тілька в точці, де вирази базової мови дозволені. Директива *taskwait* не може бути використана в місці, яке слідує за виразами *if, while, do, switch* або *label*.

Atomic — визначає змінну в лівій частині оператора “атомарного” присвоювання, який повинен коректно оновлюватись декількаами потоками. Ця директива дозволяє замінити мьютекс, який потрібен був би для зміни змінної декількаами потоками. Потрібно зважати на те, що *atomic* змінні працюють повільніше за звичайні змінні. Це означає, що, якщо змінна у програмі не змінюється, то краще використовувати звичайні змінні.

Flush — явно визначає точку, в якій реалізації повинна забезпечувати однаковий вид пам’яті для всіх потоків. Неявно *FLUSH* присутній в наступних директивах:

BARRIER, CRITICAL, END CRITICAL, END DO, END PARALLEL, END SECTIONS, END SINGLE, ORDERED, END ORDERED.

В цілях синхронизації можна також використовувати механізмом замків (*locks*).

1.2.9. Класи змінних

В *OpenMP* змінних у паралельних областях програми розділяються на два основних класа:

- *SHARED* (загальні; з ім’ям А всі потоки бачать одну змінну) та

- *PRIVATE* (приватні; з ім’ям А кожен потік бачить свою змінну).

Окремі правила визначають поведінку змінних при вході та виходів з паралельної області або паралельного цикла: *REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN*.

За замовчанням, всі *COMMON*-блоки, а також змінні, які породжені поза паралельної області, при вході в цю область залишаються спільними (*SHARED*). Виключенням є змінні — лічильники ітерацій в циклі. Змінні, які породженні у паралельній області є приватними (*PRIVATE)*. Явно назначити клас змінних за замовченням можна за допомогою клаузи *DEFAULT.*

SHARED — використовується до змінних, які необхідно зробити спільними.

PRIVATE — використовується до змінних, які необхідно зробити приватними. При вході в паралельну область для кожного потоку створюється окремий екземпляр змінної, який не має ніякого зв’язку з оригінаьною змінною поза паралельною областю.

THREADPRIVATE — використовується к *COMMON*-блокам, які необхідно зробити приватними. Директива повинна бути після кожної декларації *COMMON*-блока.

FIRSTPRIVATE — приватні копії змінної при вході в паралельну область ініціалізуються значенням оригінаьної змінної.

LASTPRIVATE — після кінця паралельного цикла або блока паралельних секцій, потік, яка виконала останню ітерацію цикла або останню сецкію блока, оновлює значення оригінальної змінної.

REDUCATION(+ :A) — означає змінну, з якою в циклі проводиться *reduction*-операція (наприклад, сумування). При виході з цикла, данна операція проводиться над копіями змінної в усіх потоках, та результат присвоються оригінальной змінній.

COPYIN — застосовується к *COMMON*-блокам, які помічені як *THREADPRIVATE*. При вході в паралельну область приватні копії цих данних ініціалізуються оригінальними значеннями.

***Runtime*-процедури та змінні середовища**

В цілях створення середовища запуска паралельних програм, які можна перенести, в *OpenMP* визначен ряд змінних середовища, які контролюють поведінку додатка.

В *OpenMP* передбачен також набір бібліотечних процедур, які дозволяють:

- під час виконання контролювати та запитувати різні параметри, які визначають поведінку додатку (такі як кількість потоків та процесорів, можливість вкладеного паралелізма); процедури назначення параметрів мають пріоритет над відповідними змінними середи.

- використовувати синхронізацію на базі замків (*locks*).

Змінні середовища

OMP_SCHEDULE — визначає розподілення ітерацій в циклі, якщо в директиві *DO* використана клауза *SCHEDULE(RUNTIME)*.

OMP_NUM_THREADS — визначає кількість потоків для виконання паралельних областей додатка.

OMP_DYNAMIC — дозволяє або забороняє динамічне зміну кількості потоків.

OMP_NESTED — дозволяє або заюороняє вкладений паралелізм.

Процедури для контроля/запроса параметрів середовища виконання

OMP_SET_NUM_THREADS — дозволяє назначити максимальну кількість потоків для використання в наступній паралельній області (якщо цю кількість дозволено динамічно змінювати). Визивається з послідовної області програми.

OMP_GET_MAX_THREADS — повертає максимальну кількість потоків.

OMP_GET_NUM_THREAD — повертає фактичне число потоків в паралельній області програми.

OMP_GET_NUM_PROCS — повертає кількість процесорів, які доступні додатку.

OMP_IN_PARALLEL — повертає *TRUE*, якщо викликана з паралельної області програми.

OMP_SET_SYNAMIC / OMP_GET_DYNAMIC — встановлює / запитує стан флага, який дозволяє динамічно змінювати кількість потоків.

OMP_GET_NESTED / OMP_SET_NESTED — встановлює / запитує стан флага, який дозволяє вкладений паралелізм.

Процедури синхронізації на базі замків

В якості замків використовується загальні змінні типа *INTEGER* (розмір повинен бути достатнім для зберегання адреси). Данні змінні повині використовуватись тільки параметри примітивів синхронізації.

OMP_INIT_LOCK(var) / OMP_DESTROY_LOCK(var) — ініціалізує замок, який пов’язан зі змінною var.

OMP_SET_LOCK — змушує потік, який викликав цю процедуру, дочекатись замка, а далі захоплює його.

OMP_UNSET_LOCK — звільняє замок, якщо він був захоплений потіком, який визвав цю процедуру.

OMP_TEST_LOCK — пробує захопити вказаний замок. Якщо це неможливо, повертає *FALSE*.

1.3. Специфікація *OpenMPI для мов C/C++*

Специфікація *OpenMP* для мов *C/C++*, яка випущена на рік пізніше фортраної, містить в основному аналогічну функціональність.

Необхідно лише відмітити наступні моменти:

1) Замість спецкоментарів використовуються директиви компілятора “#pragma omp”.

2) Компілятор з підтримкою *OpenMP* визначає макрос *“_OPENMP”*, який може використовуватись для умовної компіляції окремих блоків, які характерні для паралельної версії програми.

3) Розпаралелення застосовується к *for*-циклам, для цього використовується директива *“#pragma omp for”*. В паралельних циклах забороняється використовувати оператор *break.*

4) Статичні (*static*) змінні, які визначенні в паралельній області програми є спільними (*shared*).

5) Пам’ять, яка виділена за допомогою *malloc(),* є спільною (проте вказівник на неї може бути, як спільним, так і приватним).

6) Типи та функції *OpenMP* визначені в файлі <*omp.h*>.

7) Крім звичайних, можливі також “вкладені” (*nested*) замки — замість логічних змінних використовуються цілі числа, та потік, яка вже захопила замок, при повторному захопленні може збільшити це число[2].

1.4. Порівняння спільної та розподіленої пам’яті

Машини, що можуть проводити паралельни обчислення, можуть мати різні типи пам’яті: розподілену та спільну.

Розподілена пам’ять

Цей тип характеризується такими ознаками:

- кожен процесор має свій адресний простір пам’яті.

- значення змінних незалжені. На одному процесорі *x* може дорівнювати 2, і водночас в іншому — 3.

- приклади: кластера лінукс, *Blue Gene/L*.

Спільна пам’ять

Цей тип характерезується такими ознаками:

- також називається *Symmetric Multiprocessing (SMP)*.

- один адресний простір для всіх процесорів. Якщо один процесор присвоює *x* значення 2, то *x* дорівнює 2 на інших процесорах.

- приклади: *IBM p-series*, багатоядерні персональні комп’ютери.

Використання різних типів пам’яті

Декілька процесів:

- кожен процесор виконує незалежну задачу з його власним адресним пространстов пам’яті.

Декілька потоків:

- процес створює додаткові задачі (потоки) з тим самим адресним простором пам’яті[3].

1.5. Розмір стеку та прив’язка потоків

Розмір стеку

Стандарт *OpenMP* не вказує об’єм пам’яті стеку, який має мати кожен потік. Саме тому різні реалізації можуть мати різний розмір стеку.

За замовчуванням розмір стеку є досить малим. Для прикладу приведено розміри стеку та розмріи доступних масивів типу *double* в табл. 1.5.1.1.

Компілятор	**Приблизний розмір стеку**	**Приблизний розмір масиву**
Linux icc, ifort	4 *MB*	700 x 700
Linux pgcc, pgf90	8 *MB*	1000 x 1000
Linux gcc, gfortran	2 *MB*	500 x 500

Табл. 1.5.1.1

Потоки, які використають повністю їх стек, мають непердбачену поведінку. Вони можуть завершитись з помилкою *seg fault*, а можуть і ні. Додаток загалом може продовжувати роботу навіть попри те, що данні спотворено.

Статично злінкований код може бути опорною причиною для подальших меж об’єму стека.

Також обмеження поточного користувача можуть зменшити розмір стека.

Якщо реалізація *OpenMP* підтримує стандарт *OpenMP* 3.0, можливо використати *OMP_STACKSIZE* змінну середовища для того, щоб задати розмір стеку для кожного потоку перед виконанням програми. Наприклад:

- *setenv OMP_STACKSIZE 2000500B*
- *setenv OMP_STACKSIZE “300 k “*
- *setenv OMP_STACKSIZE 10M*
- *setenv OMP_STACKSIZE “ 10 M “*
- *setenv OMP_STACKSIZE “20 m “*
- *setenv OMP_STACKSIZE “ 1G”*
- *setenv OMP_STACKSIZE 20000*

Прив’язка потоків

В деяких випадках програма працює краще, якщо потоки прив’язані до процсорів/ядер.

“Прив’язка” потоку до процесора означає, що потік буде запланован операційною системою завжди запускатись на тому ж самому процесорі. Інакше, потоки можуть бути заплановані виконуватись на будь-якому процесорі та “скакати” між процесорами з кожним інтервалом часу.

Також це називається “потокова близькість” або “процесорна близкість”.

Прив’зка потоків до процесорів може привести до кращою утилізації кешей, що призвиде до зменшення дорогого доступу до пам’яті. Це головна мотивація для прив’язки потоків до процесорів.

Зважаючи на платформу, операційну систему, компілятор та *OpenMP* реалізацію, прив’язка потоків до процесорів може бути зроблена декількома різними шляхами.

OpenMP 3.1 API впроваджує змінні середовища для ввімкнення або вимкнення процесорної прив’язки. Наприклад:

setenv OMP_PROC_BIND TRUE

setenv OMP_PROC_BIND FALSE

На більшом високому рівні абстракії до процесорів можуть прив’язуватись процеси замість потоків.

1.6. Нагляд, налагодження та інструменти аналізу ефективності для *OpenMP*

Нагяд та налогодження потоків

Налагоджувачі різняться в їхній здібності оброблювати потоки. *TotalView debugger* є рекомендованим налагоджувачем для паралельних програм. Він добре підходить водночас і для наглядом та налагодженням поточних програм.

TotalView має такі елементи:

1. Панель st*ack trace* головного потоку.
2. Панель для розрізнення процесів/потоків.
3. Панель stack frame головного потоку для показу спільних змінних.
4. Панель для показу *stack trace* для неголовних потоків.
5. Панель *stack frame* для неголовних потоків.
6. Головне вікно, яке показує всі потоки.
7. Панель потоків, яка показує всі потоки та обраний потік.

Команда *ps* в лінукс забезпечує деякі прапори для просмотру інформації потоку. Деякі приклади наведено нижче:

% ps -Lf

UID

PID

PPID LWP C

NLWP STIME TTY TIME

CMD

blaise

22529 28240 22529 0 5 11:31 pts/53 00:00:00 a.out

blaise 22529 28240 22530 99 5 11:31 pts/53 00:01:24 a.out

blaise 22529 28240 22531 99 5 11:31 pts/53 00:01:24 a.out

blaise 22529 28240 22532 99 5 11:31 pts/53 00:01:24 a.out

blaise 22529 28240 22533 99 5 11:31 pts/53 00:01:24 a.out

% *ps -T*

PID SPID TTY TIME CMD

22529 22529 pts/53 00:00:00 a.out

22529 22530 pts/53 00:01:49 a.out

22529 22531 pts/53 00:01:49 a.out

22529 22532 pts/53 00:01:49 a.out

22529 22533 pts/53 00:01:49 a.out

% ps -Lm

PID LWP TTY TIME CMD

22529 - pts/53 00:18:56 a.out

- **22529** - **00:00:00** -

- **22530** - **00:04:44** -

- **22531** - **00:04:44** -

- **22532** - **00:04:44** -

- **22533** - **00:04:44** -

Кластера лінукс також забезпечують команду *top* для нагяду за процесами на елементі кластера. Якщо використовувати флаг *-H*, потоки, які містяться у данному процесі.

Інструмент для аналізу ефективності

Є великиа кількість інструментів для аналізу ефективності, які можна використати з *OpenMP*. Приклади наведено нижче:

- *Open | SpeedShop*

- *TAU*

- *PAPI*
- *Intel Vtune Amplifier*
- *ThreadSpotter*[5]

Висновки:

1. Виконан аналіз засобів написання програм, які використовують паралельну систему обчислення, за допомогою бібліотеки *OpenMP*. Показано, що паралельність програм досягається за рахунок використання директив *#pragma omp*, що дозволяє компілювати код програми, який написан для паралельного виконаня, з компілятором, який не підтримує *OpenMP*, проте з утратою паралельності.

2. Розглянуто директиви для розпаралелення циклів та директиви для балансування навантаження при цьому. Показано, що різні задачі потребують різних стратегій навантаження: статичної, динамочної або керованої.

3. Розглянуто типи змінних в залежності від їх поведінки у різних потоках. Показано, що зміна типу змінної за допомогою спеціальних директив може змістовно змінити роботу програми.

4. Проведено аналіз систем з спільною пам’яттю та розподіленою. Показано, що використання кожного типа на відповіному йому рівні краще за використання тільки одного типу пам’ятті.

5. Наведені приклади інструментів для роботи за паралельними програми. Показано, які засоби є можливи для аналізу роботи потоків. Наведено приклади використання цих інструментів.

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП

2.1. Розробка паралельного математичного алгоритму.

Структурна схема для ПКС СП знаходиться у Додатку А.

1. $z_i = max(Z_h), i \in [0..P-1]$
2. $z = max(z, z_i), i \in [0..P-1]$
3. $C_h = sort\,1(S_h)$
4. $C = sort\,2(C_h)$
5. $A_h = z \cdot E \cdot (MO \cdot MT_h) + C_h$

2.2. Розроботка алгоритмів процесів.

T_1:		$T_i, i \leftarrow [2..P-1]$:	
1. Ввести Z, MO.		1. Чекати сигнала про завершення введення від T_1 та T_p .	W_1
2. Сигнал про завершення введення до всіх потоків.	S_1	2. Обрахувати $z_h = max(Z_h)$.	
3. Чекати сигнала про завершення введення від потока T_p .	W_1	3. Обрахувати $z = max(z, z_h)$.	$КУ_1$
4. Обрахувати . $z_h = max(Z_h)$.		4. Сигнал про завершення обрахунку z до всіх потоків.	S_2
5. Обрахувати $z = max(z, z_h)$.	$КУ_1$	5. Чекати сигнал про завершення обрахунку z від всіх потоків.	W_2
6. Сигнал про завершення обрахунку z до всіх потоків.	S_2	6. Копіювати $z_i = z, E_i = E, MO_i = MO$.	$КУ_2$
7. Чекати сигнал про завершення обрахунку z від всіх потоків.	W_2	7. Обрахувати $B_h = z_i \cdot E_i \cdot (MO_i \cdot MT_h)$.	
8. Копіювати $z_1 = z, E_1 = E, MO_1 = MO$.	$КУ_2$	8. Сигнал про завершення обрахунку B_h до всіх потоків.	S_3
9. Обрахувати $B_h = z_1 \cdot E_1 \cdot (MO_1 \cdot MT_h)$.		9. Чекати сигнал про завершення обрахунку B_h від всіх потоків.	W_3
10. Сигнал про завершення обрахунку B_h до всіх потоків.	S_3	10. Обрахувати $C_h = sort\,1(S_h)$	
11. Чекати сигнал про завершення обрахунку B_h від всіх потоків.	W_3	11. Сигнал про завершення обрахунку C_h до всіх потоків.	S_4
12. Обрахувати $C_h = sort\,1(S_h)$		12. Чекати сигнал про завершення обрахунку C_h від всіх потоків.	W_4
13. Сигнал про завершення обрахунку C_h до всіх потоків.	S_4	13. Обрахувати $C = sort\,2(C_h)$.	
14. Чекати сигнал про завершення обрахунку C_h від всіх потоків.	W_4	14. Сигнал про завершення обрахунку C до всіх потоків.	S_5
15. Обрахувати $C = sort\,2(C_h)$.		15. Чекати сигнал про завершення обрахунку C від всіх потоків.	W_5
16. Сигнал про завершення обрахунку C до всіх потоків.	S_5	16. Обрахувати $A_h = B_h + C_h$.	
17. Чекати сигнал про завершення обрахунку C від всіх потоків.	W_5	17. Сигнал про завершення обрахунку A_h до T_1 .	S_6
18. Обрахувати $A_h = B_h + C_h$.			
19. Чекати сигнал про завершення обрахунку A_h від всіх потоків.	W_6		
20. Вивести A .			

T_p:	
1. Ввести E, S, MT.	
2. Сигнал про завершення введення до всіх потоків.	S_1
3. Чекати сигнала про завершення введення від потока T_p .	W_1
4. Обрахувати . $z_h = max(Z_h)$.	
5. Обрахувати $z = max(z, z_h)$.	$КУ_1$
6. Сигнал про завершення обрахунку z до всіх потоків.	S_2
7. Чекати сигнал про завершення обрахунку z від всіх потоків.	W_2
8. Копіювати $z_p = z, E_p = E, MO_p = MO$.	$КУ_2$
9. Обрахувати $B_h = z_p \cdot E_p \cdot (MO_p \cdot MT_h)$.	
10. Сигнал про завершення обрахунку B_h до всіх потоків.	S_3
11. Чекати сигнал про завершення обрахунку B_h від всіх потоків.	W_3
12. Обрахувати $C_h = sort\,1(S_h)$.	S_4
13. Сигнал про завершення обрахунку C_h до всіх потоків.	
14. Чекати сигнал про завершення обрахунку C_h від всіх потоків.	W_4
15. Обрахувати $C = sort\,2(C_h)$.	
16. Сигнал про завершення обрахунку C до всіх потоків.	S_5
17. Чекати сигнал про завершення обрахунку C від всіх потоків.	W_5
18. Обрахувати $A_h = B_h + C_h$.	
19. Чекати сигнал про завершення обрахунку A_h від всіх потоків.	W_6
20. Вивести A .	

2.3. Розробка схеми взаємодії процесів

Схема взаємодії процесів для ПКС СП знаходиться у Додатку В.

Критичні секції zUpdate та dataCopy використовуються для доступу к спільним ресурсам. При цьому zUpdate використовується, як для читання, так і для запису в змінну z. Критична секція dataCopy використовується лише для читання зі змінних z, E, MO.

Наступний бар’єр потрібен для синхронізації потоків після вводу данних.

Наступний *for* використовується для обрахунку максимального *z*.

Наступний бар’єр потрібен для синхронізації потоків після обрахунку максимального z.

Наступний *for* обчислення результату множення матриць.

Наступний бар’єр потрібен для синхронізації потоків після обчислення результата множення матриць.

Секція s*ingle* використовується для сортування вектору *S*.

Наступний бар’єр, потрібен для синхронізації потоків після сортування вектору S.

Кінець паралельної секції (*parallel end*) потрібен для синхронізації потоків для виводу результату загальної задачі.

2.4. Розробка програми ПРГ1

Вхідні данні: Z, MO, E, S, MT.

Вихідні данні: A.

Тимчасові данні: z, B, C.

mval — максимальне значення серед Z_h .

z — максимальне значення серед Z.

MO1, E1, z1 — копії MO, E та z для обробки потоком відповідно.

X — елемент матриці, яка є результатом множення MO та MT.

C — відсортований S.

ind — масив індексів для сортування цільного масиву S.

mini — індекс частини S, яка має максимальний елемент у місці, вказаному індексом масиву ind.

ReadArr — функція для створення масиву заданої довжини, заповненого одиницями.

ReadVec — створення масиву довжиною N, заповненого одиницями.

ReadMat — створення масиву довжиною N*N, заповненого одиницями.

Arrcpy — функція для копіювання значень з іншого масиву заданої довжини.

Veccpy — функція для копіювання масиву довжиною N.

Matcpy — функція для копіювання масиву довжиною N*N.

get_timestamp — функція для отримання поточного часу.

2.5. Тестування програми ПРГ1

Таблиця 2.1 Час виконання програми для ПРГ1

N	T1	T10	T20	T30	T40
960	4.61	0.6987	0.479	0.4559	0.4723
1920	49.44	7.1341	4.1266	4.104	4.0984
2880	212.46	25.5548	14.4855	15.5079	14.5219

Таблиця 2.2 Значення Кп для ПРГ1

N	Кількість процесорів (P)				
	1	10	20	30	40
960	1	6.6	9.63	10.12	9.77
1920	1	6.93	11.98	12.05	12.09
2880	1	8.31	14.67	13.7	14.63

Таблиця 2.3 Значення Ке для ПРГ1

N	Кількість процесорів (P)				
	1	10	20	30	40
960	100	66	48.15	33.73	24.43
1920	100	69.3	59.9	40.17	30.23
2880	100	83.1	73.35	45.67	35.16

В таблиці 2.1 наведені часи виконання програми ПРГ1 за різних N та P.

В таблиці 2.2 наведені коефіцієнти прискорення програми ПРГ1 за різних N та P.

В таблиці 2.3 наведені коефіцієнти ефективності програми ПРГ1 за різних N та P.

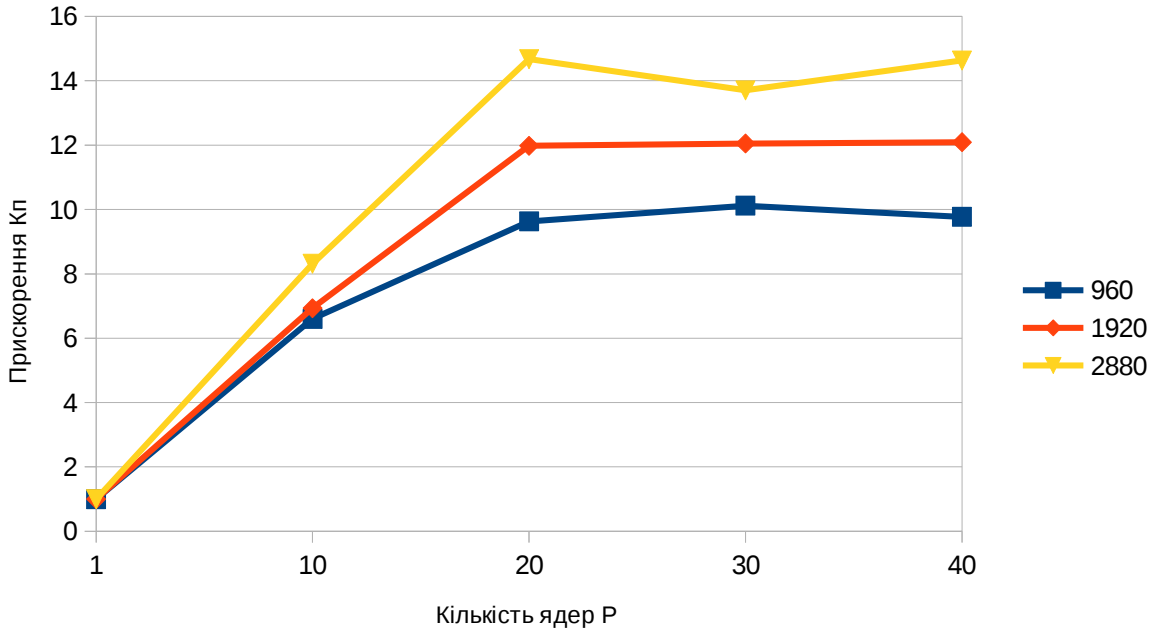


Рис 2.4 Программа ПРГ1. Графік зміни коефіцієнту прискорення Кп в залежності від кількості ядер

На Рис. 2.4 наведен графік залежності коефіцієнта прискорення від кількості ядер при різному N.

2.6. Висновки до розділу 2

Виконано розробку програми ПРГ1 для ПКС ОП з використанням мови C++ та засобів синхронізації бібліотеки openMP. Тестування програми показало наступне:

- використання багатоядерної ПКС та програми ПРГ1 забезпучує скорочення часу обчислення заданої математичної задачі. Значення Кп лежить в межах 6,6 та 14,67;
- максимальне значення Кп забезпечує ПКС з P=20 та N=2880;
- мінімальне значення Кп забезпечує ПКС з P=10 та N=960;
- з ростом N зміна Ку є додатковою;
- з ростом P зміна Ку є від’ємною;
- використання від 15 до 25 процесорів є оптимальним.

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП

3.1. Розробка паралельного математичного алгоритму.

Структурна схема ПКС ЛП наведена у Додатку Б.

1. $z_i = max(Z_h), i \in [0..P-1]$
2. $z = max(z, z_i), i \in [0..P-1]$
3. $C_h = sort\,1(S_h)$
4. $C = sort\,2(C_h)$
5. $A_h = z \cdot E \cdot (MO \cdot MT_h) + C_h$

3.2. Розроботка алгоритмів процесів.

Сортвуання та пошук максимума на останньому етапі прохродять у одному або в двох (в залежності від парності половини кількості процесорів) центральних процесах. Хай $mid = [P/2], quarter = [mid/2]$, тоді якщо quarter — непарна кількість, то центральний процес - $T_{(quarter+1)}$: , інакше центральні процеси - $T_{quarter}$: та $T_{(quarter+1)}$: .

Результати сортвуання та пошуку максимума, C_h та z , приходять з центрального процесу. Коли центральних процесів два, результат приходить з ближчого. Алгоритм визначення процесу, з якого приходить результати, чітко визначений і для того, щоб описувати його в кожному потоці, введемо поток T_c , який буде найближчим центральним процесом, та T_f , які є центральними процесами у випадку, коли їх два.

В залежності від розташування процесу визначається напрям передачі результатів. Для i < mid: i <= quarter, dir = 1, для i > quarter, dir = -1. З допомогою напрямо можливо визначити індекс попередьного та наступного процесів у ланцюгу передачі результатів. Це відповідно $T_{(i-dir)}$: та $T_{(i+dir)}$: . Для i >= mid процеси мають один напрям передачі.

Розмірність масиву C сожним кроком збільшується. У поток T_i передається масив з розмірністю залежною від i. Якщо i <= quarter, то передається $C_{((i-1)*2h)}$. Якщо i > quarter та i < mid, то передається $C_{((mid-i)*2h)}$. Для спрощення хай до T_i передається $C_{(k(i))}$.

T_1:		$T_i, i \in [2..mid-1], T_i \neq T_c, T_i \neq T_f$	
1. Ввести Z, MO, E, S, MT.		1. Чекати сигнал з передачою Z_h та S_h з T_1 .	W_1
2. Сигнал з передачою Z_h та S_h до всіх потоків.	S_1	2. Обрахувати $z_i = max(Z_h)$.	
3. Обрахувати . $z_1 = max(Z_h)$.		3. Обрахувати $C_h = sort\,1(S_h)$	
4. Обрахувати $C_h = sort\,1(S_h)$		4. Чекати сигналу з $z_{(i+mid)}$ та C_h з $T_{(i+mid)}$	W_2
5. Чекати сигналу з $z_{(1+mid)}$ та C_h з $T_{(1+mid)}$	W_1	5. Чекати сигналу з $z_{(i-dir)}$ та C_{2h} з $T_{(i+mid)}$	W_3
6. Обрахувати $z_1 = max(z_1, z_{(1+mid)})$		6. Обрахувати $z_i = max(z_i, z_{(i-1)}, z_{(i+mid)})$	
7. Обрахувати $C_{2h} = sort\,2(C_h, C_h)$		7. Обрахувати $C_{(k(i)+2h)} = sort\,2(C_h, C_{(k(i))}, C_h)$	
8. Сигнал з передачею z_1 та C_{2h} до T_2	S_2	8. Сигнал з передачею z_i та $C_{(k(i)+2h)}$ до $T_{(i+dir)}$	S_1
9. Чекати сигналу з z та C_h з T_c	W_2	9. Чекати сигналу з z та C_h з T_c	W_4
10. Сигнал з E , MO та MT_h до всіх потоків.	S_3	10. Чекати сигналу з E_i, MO_i, MT_h з T_1 .	W_5
11. Копіювати $z_1 = z, E_1 = E, MO_1 = MO$		11. Обрахувати $A_h = z_i \cdot E_i \cdot (MO_i \cdot MT_h) + C_h$.	
12. Обрахувати $A_h = z_1 \cdot E_1 \cdot (MO_1 \cdot MT_h) + C_h$		12. Сигнал з A_h до T_{mid}	S_2
13. Сигнал з A_h до T_{mid}	S_4		

T_c:		T_f	
1. Чекати сигнал з передачою Z_h та S_h з T_1 .	W_1	1. Чекати сигнал з передачою Z_h та S_h з T_1 .	W_1
2. Обрахувати . $z_c = max(Z_h)$.		2. Обрахувати $z_f = max(Z_h)$.	
3. Обрахувати $C_h = sort\,1(S_h)$		3. Обрахувати $C_h = sort\,1(S_h)$	
4. Чекати сигналу з $z_{(c+mid)}$ та C_h з $T_{(c+mid)}$	W_2	4. Чекати сигналу з $z_{(f+mid)}$ та C_h з $T_{(f+mid)}$	W_2
5. Чекати сигналу з $z_{(c-dir)}$ та $C_{((mid-1)\cdot h)}$ з T_P	W_3	5. Чекати сигналу з $z_{(f-dir)}$ та $C_{((mid-1)\cdot h)}$ з T_P	W_3
6. Чекати сигналу з $z_{(c+dir)}$ та $C_{((mid-1)\cdot h)}$ з T_P	W_4	6. Обрахувати $z_f = max(z_f, z_{(f+mid)}, z_{(f-dir)})$	
7. Обрахувати $z_c = max(z_c, z_{(c+mid)}, z_{(c-dir)}, z_{(c+dir)})$		7. Обрахувати $C_{(mid\cdot h)} = sort\,2(C_h, C_h, C_{((mid-2)\cdot h)})$	
8. Обрахувати $C_N = sort\,2(C_h, C_h, C_{((mid-1)\cdot h)}, C_{((mid-1)\cdot h)})$	S_1	8. Сигнал з передачею z_f та $C_{(mid\cdot h)}$ до $T_{(f+dir)}$	S_1
9. Сигнал з передачею z_{mid} та C_H до всіх потоків.		9. Чекати сигнал з передачею $z_{(f+dir)}$ та $C_{(mid\cdot h)}$ з $T_{(f+dir)}$	W_4
10. Чекати сигналу з E_c, MO_c, MT_h з T_1 .	W_5	10. Обрахувати $z_f = max(z_f, z_{(f+dir)})$	
11. Обрахувати $A_h = z_c \cdot E_c \cdot (MO_c \cdot MT_h) + C_h$		11. Обрахувати $C_N = sort\,2(C_{(mid\cdot h)}, C_{(mid\cdot h)})$	
12. Сигнал з A_h до T_{mid}	S_2	12. Сигнал з передачею z_{mid} та C_H до всіх потоків	S_2
		13. Чекати сигналу з E_i, MO_i, MT_h з T_1 .	W_5
		14. Обрахувати $A_h = z_f \cdot E_f \cdot (MO_f \cdot MT_h) + C_h$.	
		15. Сигнал з A_h до T_{mid}	S_3

T_{mid}: 1. Чекати сигнал з передачою Z_h та S_h з T_1 . 2. Обрахувати . $z_{mid}=max(Z_h)$. 3. Обрахувати $C_h=sort\,1(S_h)$ 4. Чекати сигналу з z_P та C_h з T_P 5. Обрахувати $z_{mid}=max(z_{mid},z_P)$ 6. Обрахувати $C_{2h}=sort\,2(C_h,C_h)$ 7. Сигнал з передачею z_1 та C_{2h} до $T_{(mid-1)}$ 8. Чекати сигналу з z та C_h з T_c 9. Чекати сигналу з E_{mid},MO_{mid},MT_h з T_1 . 10. Обрахувати $A_h=z_{mid}\cdot E_{mid}\cdot(MO_{mid}\cdot MT_h)+C_h$ 11. Чекти сигнал з A_h з усіх потоків. 12. Вивести А.	W_1 W_2 S_1 W_3 W_4 W_5	$T_i, i\in[mid+1..N]$: 1. Чекати сигнал з передачою Z_h та S_h з T_1 . 2. Обрахувати . $z_{mid}=max(Z_i)$. 3. Обрахувати $C_h=sort\,1(S_h)$ 4. Сигнал з передачею z_i та C_h до $T_{(i-mid)}$ 5. Чекати сигналу з z та C_h з T_c 6. Чекати сигналу з E_i,MO_i,MT_h з T_1 . 7. Обрахувати $A_h=z_i\cdot E_i\cdot(MO_i\cdot MT_h)+C_h$ 8. Сигнал з A_h до T_{mid}	W_1 S_1 W_2 W_3 S_2

3.3. Розробка схеми взаємодії процесів

Схема взаємодії процесів для ПКС ЛП знаходиться у Додатку Г.

Процес Ti, де i >= mid взаємодіє лише з потоками: T(i-mid), T1, Tc та Tmid. З T1 він отримує вхідні данні, до T(i-mid) та з Tc він відповідно посилає та отримує відсортований Sh та zMax. До Tmid він посилає Ah.

Процес Ti, де i > 1, i < mid — 1, i != c, отримує відсоровані Sh та zMax з T(i-dir) та T(i+mid) та посилає ці та свій Sh та zMax до T(i+dir). З T1, Tc та Tmid працює таким самим чином, як і Ti, де i >=mid.

2.4. Розробка програми ПРГ2

Вхідні данні: Z, MO, E, S, MT.

Вихідні данні: A.

Тимчасові данні: z, B, C.

mid=[P/2], quarter = [mid/2]

graph_comm, topo_comm — комутатори для основного графу та взяття інформації про граф.

mpiRun — процедура запуску обчислень.

maxSort — процедура непаралельного знаходження максимума та сортування.

mergeSend — збору відсортованих частин масиву та локальних максимумів.

mergeN — фукнція для спільного сортування N відсортованих масивів.

index, edges — масиви для задання кількості суміжних ребер, початків та кінців ребер графа.

initVectror, initMatrix — читання векторів та матриць.

Ei, Moi, Mth, Sh, Ah — данні для обчислень у конкретному потоці.

ZH_SENDING — мітка передачі Zh при знаходженні максимума.

SH_SENDING — мітка передачі Sh при сортуванні.

MAX_Z - мітка при передачі локального максимума.

SORTED_S — мітка при передачі відсортованої частини.

PART_C — мітка при передачі частини C.

TOTAL_MAX_Z — мітка при передачі занального максимума.

E_SENDING — мітка при передачі E.

MO_SENDING — мітка при передачі MO.

MTH_SENDING — мітка при передачі частини MT.

AH_SENDING — мітка при передачі частини A.

2.5. Тестування програми ПРГ2

Таблиця 3.1 Час виконання програми для ПРГ2

N	T1	T10	T20	T30	T40
960	4.37	0.5345	0.6229	0.4964	0.4427
1920	46.81	5.9212	5.3042	4.3175	4.3098
2880	201.13	25.5474	16.9096	16.5257	16.4143

Таблиця 3.2 Значення Кп для ПРГ1

N	Кількість процесорів (P)				
	1	10	20	30	40
960	1	8.63	7.41	9.29	10.42
1920	1	8.35	9.32	11.45	11.47
2880	1	8.32	12.56	12.86	12.94

Таблиця 3.3 Значення Ке для ПРГ1

N	Кількість процесорів (P)				
	1	10	20	30	40
960	100	86.3	37.05	37.67	26.05
1920	100	83.5	46.6	38.17	28.88
2880	100	83.2	62.8	42.67	32.35

В таблиці 3.1 наведені часи виконання програми ПРГ1 за різних N та P.

В таблиці 3.2 наведені коефіцієнти прискорення програми ПРГ1 за різних N та P.

В таблиці 3.3 наведені коефіцієнти ефективності програми ПРГ1 за різних N та P.

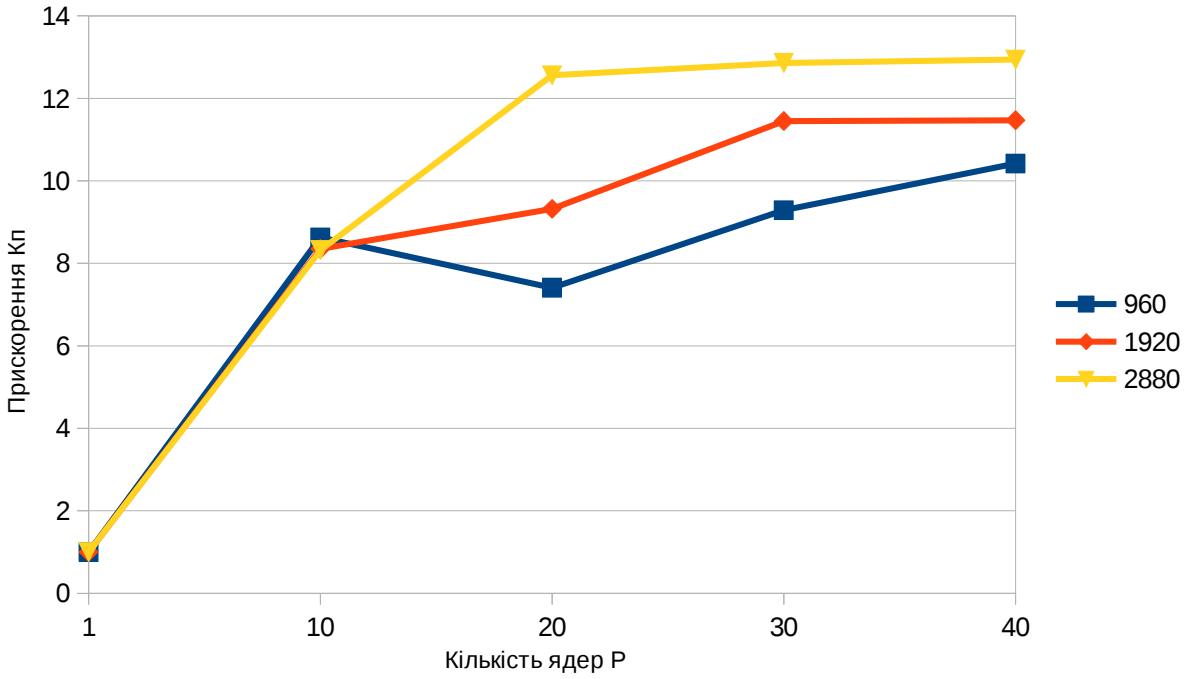


Рис 3.4 Программа ПРГ2. Графік зміни коефіцієнту прискорення Кп в залежності від кількості ядер

На Рис. 3.4 наведен графік залежності коефіцієнта прискорення від кількості ядер при різному N.

2.6. Висновки до розділу 3

Виконано розробку програми ПРГ2 для ПКС ЛП з використанням мови С++ та засобів синхронізації бібліотеки MPI. Тестування програми показало наступне:

- використання багатоядерної ПКС та програми ПРГ2 забезпучує скорочення часу обчислення заданої математичної задачі. Значення Кп лежить в межах 8,32 та 12,94;
- максимальне значення Кп забезпечує ПКС з P=40 та N=2880;
- мінімальне значення Кп забезпечує ПКС з P=10 та N=2880;
- з ростом N зміна Ку є додатковою для всіх P крім 10;
- з ростом P зміна Ку є від’ємною;
- використання від 15 до 25 процесорів є оптимальним для N=2880, від 25 до 35 для N=1920 та від 35 до 45 для N=960.

4. ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ДО РОБОТИ

Виконано розробку програми ПРГ1 для ПКС ОП з використанням мови С++ та засобів синхронізації бібліотеки openMP та програми ПРГ2 для ПКС ЛП з використанням мови С++ та засобів синхронізації бібліотеки MP. Тестування програм показало наступне:

- використання багатоядерної ПКС з програми ПРГ1 або ПРГ2 забезпучує скорочення часу обчислення заданої математичної задачі. Діапазон значень Кп для ПРГ1 є більшим за діапазон для ПРГ2, а середні значення приблизно однакові. Це показує, що *openMP* ефективніший при великих P та N, а MPI при малих;
- максимальне значення Кп забезпечує ПКС з більшим N та великим P;
- мінімальне значення Кп забезпечує ПКС з P=10, для ПРГ2 при великому N, для ПРГ1 — при малому;
- з ростом N зміна Ку є додатковою;
- з ростом P зміна Ку є від’ємною;
- використання від 15 до 25 процесорів є оптимальним для ПРГ1, діапазон змінюється в залженості від N для ПРГ2;
- в середньому абсолютний час виконання програми більший для ПРГ2. Також на ПРГ2 більше впливає збільшення N та P, проте на малих N та P дає кращі результати.

5. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kos66, Введение в OpenMP: параллельное программирование на C++ // Intel Developer Zone, 2011. [Електорнний ресурс] — Режим доступу: https://software.intel.com/ru-ru/blogs/2011/11/21/openmp-c
2. Что такое OpenMP? // Лаборатория Паралельніх Информационних Технологий, НИВЦ МГУ. [Електорнний ресурс] — Режим доступу: https://parallel.ru/tech/tech_dev/openmp.html
3. Sondak D. Parallel Processing with OpenMP // Boston University, Scientific Computing and Visualization Office of Information Technology. [Електорнний ресурс] — Режим доступу: http://www.compunity.org/training/tutorials/openmp_Boston.pdf
4. Blaise B. OpenMP // Lawrence Livermore National Laboratory. [Електорнний ресурс] — Режим доступу: https://computing.llnl.gov/tutorials/openMP/

ДОДАТОК А

Структурна схема ПКС СП

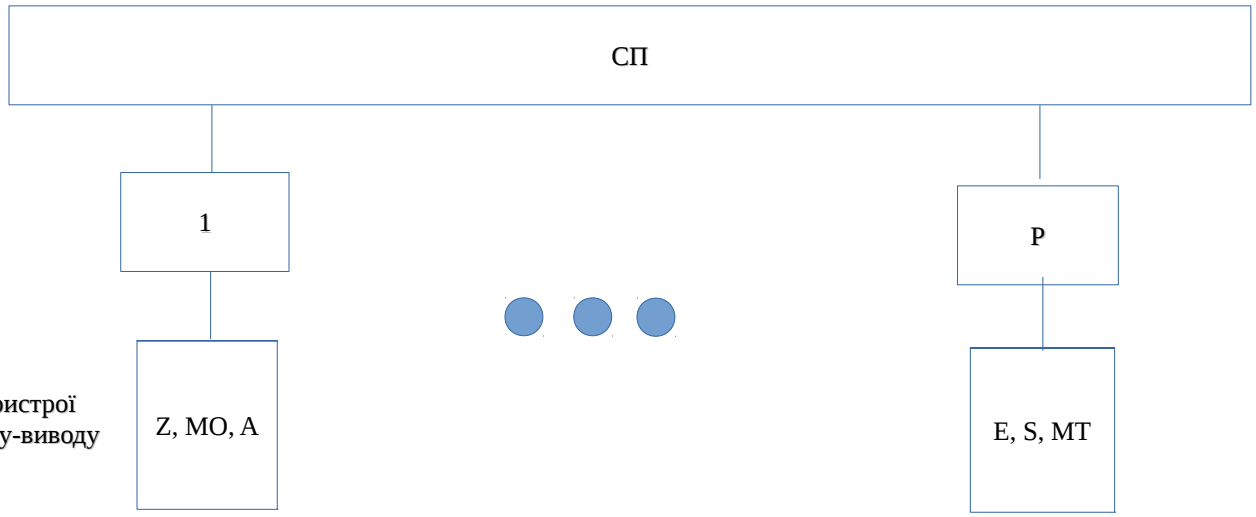


Рис. 2.1. Структура ПКС ОП

ДОДАТОК Б

Структурна схема ПКС ЛП

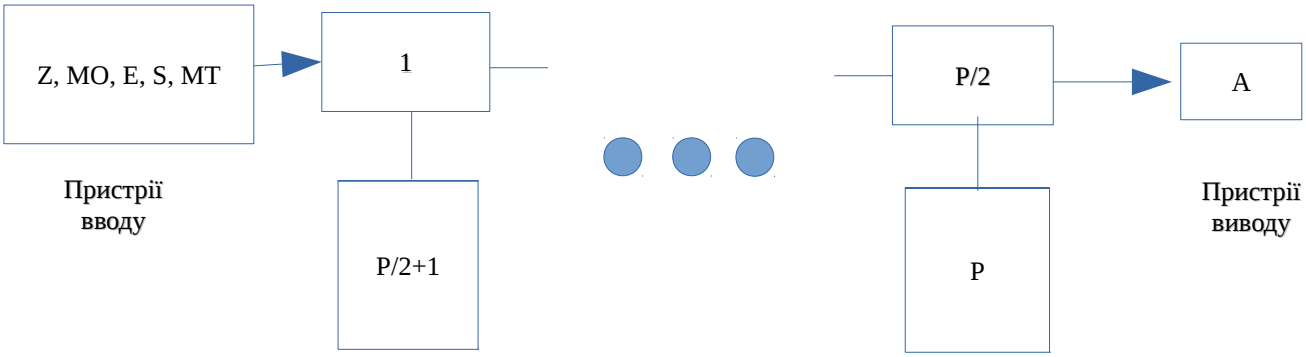
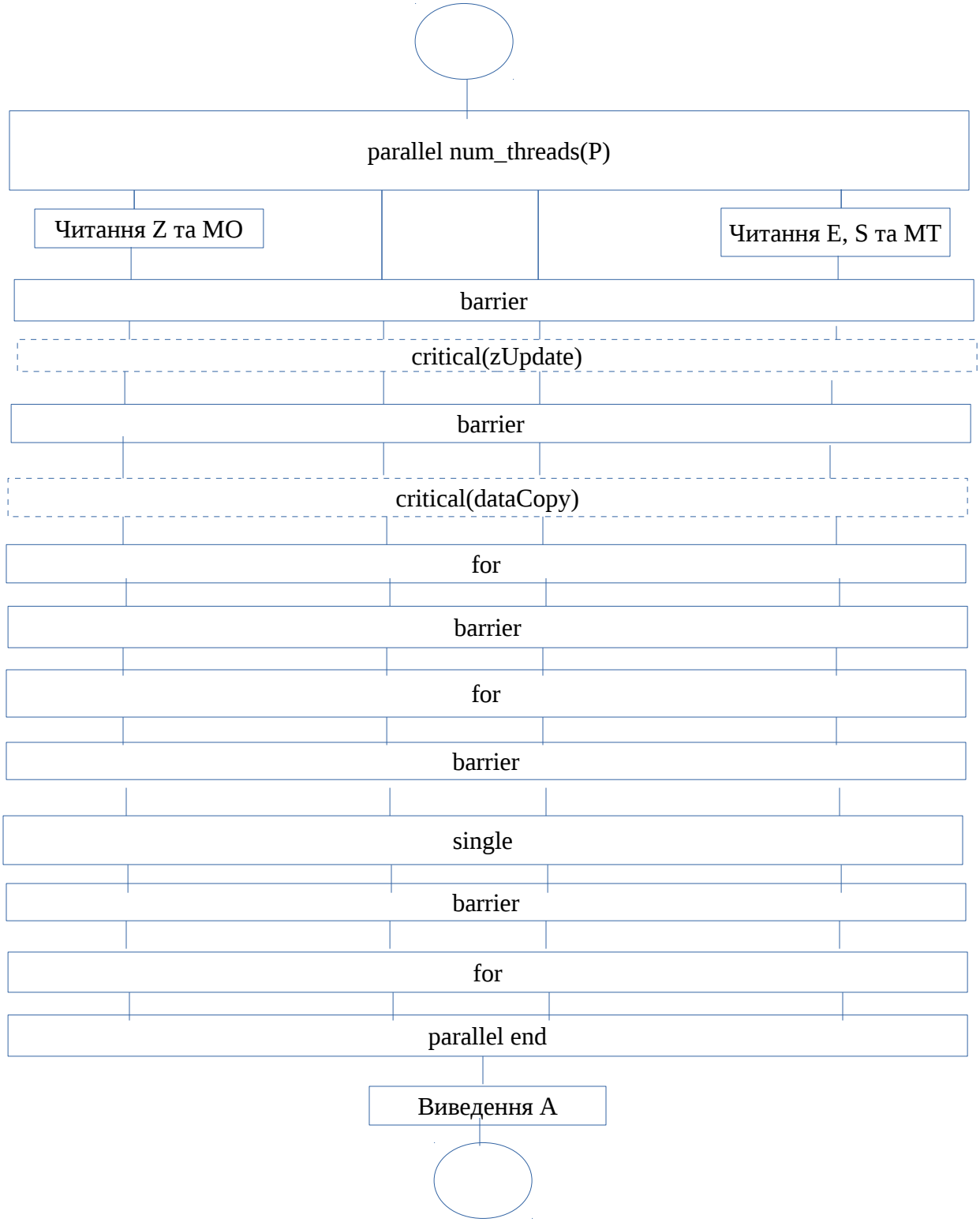


Рис. 3.1. Структура ПКС ЛП

ДОДАТОК В

Схема алгоритму програми ПРГ1



ДОДАТОК Г

Схема алгоритму програми ПРГ2

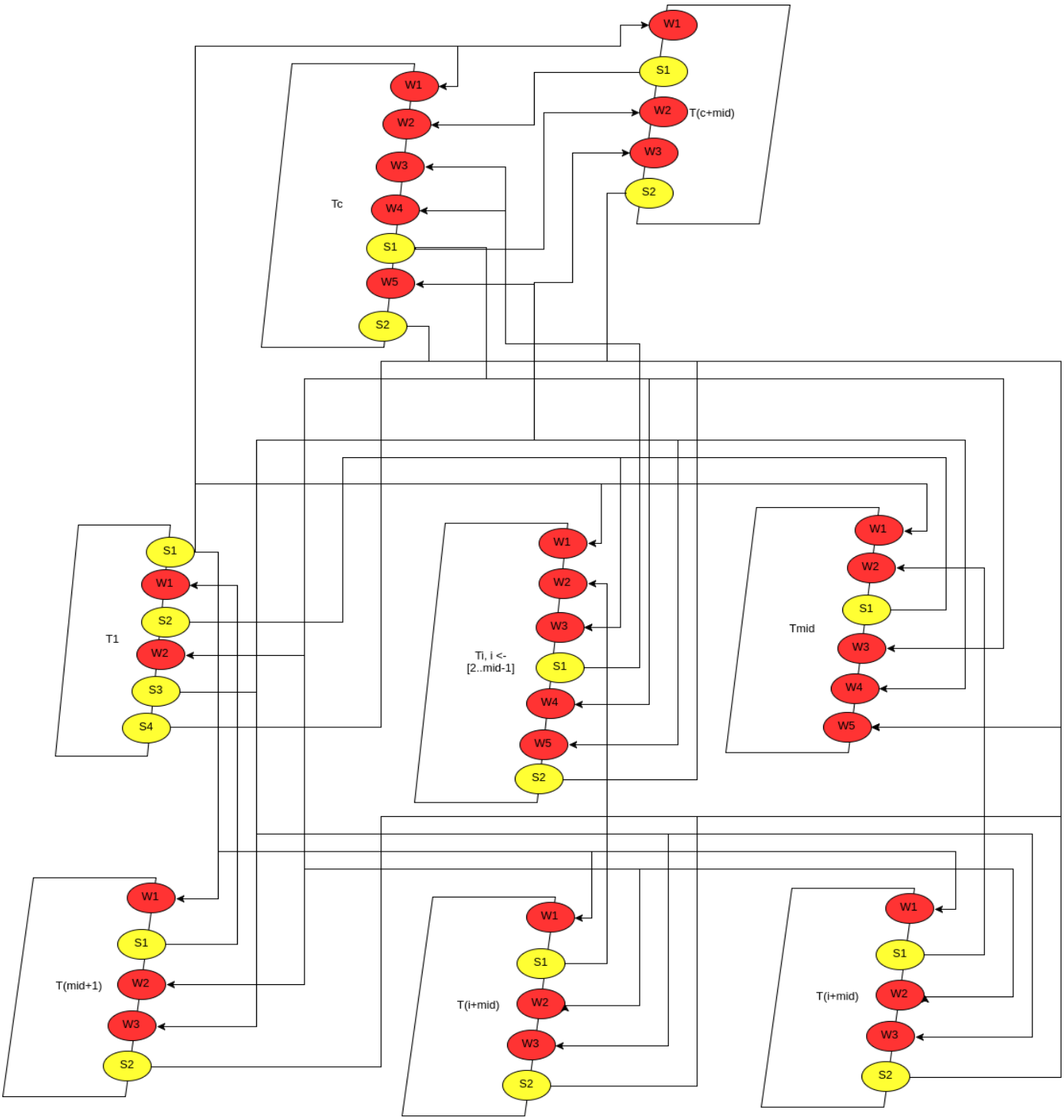


Рисунок 3.2. Схема взаємодії потоків

ДОДАТОК Ґ

Лістінг ПРГ1

```
1 /*
2 // main
3 // Author:
4 //    Dzyuba Vlad, IP-42
5 */
6 #include <iostream>
7 #include <omp.h>
8 #include <string.h>
9 #include <stdlib.h>
10 #include <sys/time.h>
11
12 typedef unsigned long long timestamp_t;
13
14 timestamp_t get_timestamp () {
15  struct timeval now;
16  gettimeofday (&now, NULL);
17  return  now.tv_usec + (timestamp_t)now.tv_sec * 1000000;
18 }
19
20 using namespace std;
21
22 int N, P, H;
23
24 // functions for generating structures
25 int* readVec();
26 int* readMat();
27 // functions for copying structures
28 int* veccpy(int *src);
29 int* matcpy(int *src);
30
31 int main(int argc, char* argv[]) {
32  N = atoi(argv[1]);
```

```
33  P = atoi(argv[2]);
34  H = N / P;
35  // input data
36  int *Z, *MO, *E, *S, *MT;
37  // output data
38  int *A = new int[N];
39
40  // intermidiate data
41  int z = -65536, *B = new int[N], *C = new int[N];
42  timestamp_t startTime = get_timestamp();
43  #pragma omp parallel num_threads(P)
44  {
45   int tid = omp_get_thread_num();
46   // generate data if first or last thread
47   if (tid == 0) {
48    Z = readVec();
49    MO = readMat();
50   }
51   if (tid == P - 1) {
52    E = readVec();
53    S = readVec();
54    MT = readMat();
55   }
56   // finding maximum of Z
57   #pragma omp barrier
58   for (int i = 0; i < P; i++) {
59    int mval = Z[i*H];
60    for (int j = i*H+1; j < (i+1)*H; j++) {
61     mval = Z[j] > mval ? Z[j] : mval;
62    }
63    #pragma omp critical(zUpdate)
64    if (mval > z) {
65     z = mval;
66    }
67   }
68   #pragma omp barrier
69   // calculating B
```

```
70   int *MO1, *E1, z1;
71   #pragma omp critical(dataCopy)
72   {
73    MO1 = matcpy(MO);
74     E1 = veccpy(E);
75     z1 = z;
76   }
77   #pragma omp for
78   for (int i = 0; i < P; i++) {
79    for (int j = i * H; j < (i + 1) * H; j++) {
80     B[j] = 0;
81     for (int k = 0; k < N; k++) {
82      int x = 0;
83      for (int l = 0; l < N; l++) {
84       x += MO1[k*N+l] * MT[l*N+j];
85      }
86      B[j] += x * E1[k];
87     }
88     B[j] *= z1;
89    }
90   }
91   #pragma omp barrier
92   // sorting parts of S
93   #pragma omp for
94   for (int i = 0; i < P; i++) {
95    for (int j = 1; j < H; j++) {
96     for (int k = i * H; k < (i + 1) * H - j; k++) {
97      if (S[k] > S[k + 1]) {
98       int x = S[k];
99       S[k] = S[k + 1];
100       S[k + 1] = x;
101      }
102     }
103    }
104   }
105   #pragma omp barrier
106   // sorting whole S and stores itto C
```

```
  #pragma omp single
  {
   int* ind = new int[N];
   for (int i = 0; i < N; i++) {
    ind[i] = 0;
   }
   for (int i = 0; i < N; i++) {
    int mini = -1;
    for (int j = 0; j < N; j++) {
     if (ind[j] < N) {
      if (mini == -1 || S[ind[j]] < S[ind[mini]]) {
       mini = j;
      }
     }
    }
    C[i] = S[ind[mini]];
    ind[mini]++;
   }
  }
  #pragma omp barrier
  // calculating A
  #pragma omp for
  for (int i = 0; i < N; i++) {
   A[i] = B[i] + C[i];
  }
 }
 // print A
 if (N <= 20) {
  for (int i = 0; i < N; i++) {
   cout << A[i] << " ";
  }
  cout << endl;
 } else {
  timestamp_t endTime = get_timestamp();
  cout << (endTime - startTime) / 1000000.0 << endl;
 }
 delete[] A;
```

```
144 }
145
146 int* readArr(int n) {
147  int* res = new int[n];
148  for (int i = 0; i < n; i++) {
149   res[i] = 1;
150  }
151  return res;
152 }
153
154 int* readVec() { return readArr(N); }
155 int* readMat() { return readArr(N*N); }
156
157 int* arrcpy(int *src, int n) {
158  int *res = new int[n];
159  for (int i = 0; i < n; i++) {
160   res[i] = src[i];
161  }
162  return res;
163 }
164
165 int* veccpy(int *src) { return arrcpy(src, N); }
166 int* matcpy(int *src) { return arrcpy(src, N * N); }
```

ДОДАТОК Д

Лістінг ПРГ2

```
1 #include <iostream>
2 #include <mpi.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <sys/time.h>
6
7 using namespace std;
8
9 typedef unsigned long long timestamp_t;
10
11 timestamp_t get_timestamp () {
12  struct timeval now;
13  gettimeofday (&now, NULL);
14  return  now.tv_usec + (timestamp_t)now.tv_sec * 1000000;
15 }
16
17 #define EDGE_COUNT (2 * (P - 1))
18 #define ZH_SENDING 0
19 #define SH_SENDING 1
20 #define MAX_Z 2
21 #define SORTED_S 3
22 #define PART_C 4
23 #define TOTAL_MAX_Z 5
24 #define E_SENDING 6
25 #define MO_SENDING 7
26 #define MTH_SENDING 8
27 #define AH_SENDING 9
28
29 void mpiRun(MPI_Comm &graph_comm, MPI_Comm &topo_comm, int* index, int* edges);
30 void mergeSend(int rank, MPI_Comm graph_comm, int *Z, int *S, int *maxZ);
31 void maxSort(int &maxZ, int* Z, int* S);
```

```
32 int* mergeN(int n, int lengths[], int* arrays[]);
33
34 int N;
35 int P;
36 int H;
37 int mid;
38 int quarter;
39
40 int main(int argc, char* argv[]) {
41   MPI_Init(&argc, &argv);
42   MPI_Comm_size(MPI_COMM_WORLD, &P);
43  N = atoi(argv[1]);
44   H = N / P;
45   mid = P / 2;
46   quarter = mid / 2;
47
48   MPI_Comm graph_comm, topo_comm;
49   int* index = new int[P];
50   int* edges = new int[EDGE_COUNT];
51
52   mpiRun(graph_comm, topo_comm, index, edges);
53
54   MPI_Finalize();
55   delete[] index;
56   delete[] edges;
57 }
58
59 int* initVector(int n) {
60   int *res = new int[n];
61   for (int i = 0; i < n; i++) {
62     res[i] = 1;
63   }
64   return res;
65 }
66
67 int* initMatrix(int n) {
68   return initVector(n * n);
```

```
69 }
70
71 void mpiRun(MPI_Comm &graph_comm, MPI_Comm &topo_comm, int* index, int* edges) {
72  timestamp_t startTime = get_timestamp();
73   index[0] = 2;
74   for (int i = 1; i < mid - 1; i++) {
75     index[i] = index[i - 1] + 3;
76   }
77   index[mid - 1] = index[mid - 2] + 2;
78
79   for (int i = mid; i < P; i++) {
80     index[i] = index[i - 1] + 1;
81   }
82
83   for (int i = 0; i < mid - 1; i++) {
84     edges[i * 2] = i;
85     edges[i * 2 + 1] = i + 1;
86     edges[i * 2 + 2] = i;
87     edges[i * 2 + 3] = mid + i;
88   }
89   edges[EDGE_COUNT - 2] = mid - 1;
90   edges[EDGE_COUNT - 2] = P - 1;
91
92   MPI_Graph_create(MPI_COMM_WORLD, P, index, edges, 0, &graph_comm);
93
94   MPI_Comm_dup(graph_comm, &topo_comm);
95
96   int topo_type;
97   MPI_Topo_test(topo_comm, &topo_type);
98
99   if (topo_type != MPI_GRAPH) {
100     cout << "Topo type error" << endl;
101     return;
102   }
103
104   int rank;
105   MPI_Comm_rank(graph_comm, &rank);
```

```
106
107   int H = N / P;
108   MPI_Status status;
109   int maxZ, *Ei, *MOi, *MTh, *Sh, *Ah;
110   Ei = new int[N];
111   MOi = new int[N*N];
112   MTh = new int[N*H];
113   Sh = new int[H];
114   Ah = new int[H];
115   if (rank == 0) {
116     int *Z, *E, *MO, *MT, *S;
117     Z = initVector(N);
118     E = initVector(N);
119     MO = initMatrix(N);
120     MT = initMatrix(N);
121     S = initVector(N);
122
123     for (int i = 1; i < P; i++) {
124       MPI_Send(Z + i * H, H, MPI_INT, i, ZH_SENDING, graph_comm);
125       MPI_Send(S + i * H, H, MPI_INT, i, SH_SENDING, graph_comm);
126     }
127
128     maxSort(maxZ, Z, S);
129     mergeSend(rank, graph_comm, Z, S, &maxZ);
130
131     for (int i = 1; i < P; i++) {
132       MPI_Send(E, N, MPI_INT, i, E_SENDING, graph_comm);
133       MPI_Send(MO, N * N, MPI_INT, i, MO_SENDING, graph_comm);
134       MPI_Send(MT + i * H * N, H * N, MPI_INT, i, MTH_SENDING, graph_comm);
135     }
136
137     for (int i = 0; i < H; i++) {
138       Sh[i] = S[i];
139     }
140     for (int i = 0; i < N; i++) {
141       Ei[i] = E[i];
142     }
```

```
143     for (int i = 0; i < N * N; i++) {
144       MOi[i] = MO[i];
145     }
146     for (int i = 0; i < N * H; i++) {
147       MTh[i] = MT[i];
148     }
149
150     delete[] Z;
151     delete[] E;
152     delete[] MO;
153     delete[] MT;
154     delete[] S;
155   } else {
156     int *Z, *S;
157     Z = new int[H];
158     S = new int[H];
159     MPI_Recv(Z, H, MPI_INT, 0, ZH_SENDING, graph_comm, &status);
160     MPI_Recv(S, H, MPI_INT, 0, SH_SENDING, graph_comm, &status);
161
162     maxSort(maxZ, Z, S);
163
164     int rank1 = rank < mid ? rank : rank - mid;
165     int sourceRank = (mid & 1) ? quarter : rank1 < quarter ? quarter - 1 : quarter;
166
167     if (rank < mid) {
168       mergeSend(rank, graph_comm, Z, S, &maxZ);
169       for (int i = 0; i < H; i++) {
170         Sh[i] = S[i];
171       }
172     } else {
173       MPI_Send(&maxZ, 1, MPI_INT, rank - mid, MAX_Z, graph_comm);
174       MPI_Send(S, H, MPI_INT, rank - mid, SORTED_S, graph_comm);
175
176       MPI_Recv(&maxZ, 1, MPI_INT, sourceRank, TOTAL_MAX_Z, graph_comm, &status);
177       MPI_Recv(Sh, H, MPI_INT, sourceRank, PART_C, graph_comm, &status);
178     }
179     delete[] Z;
```

```
180     delete[] S;
181
182     MPI_Recv(Ei, N, MPI_INT, 0, E_SENDING, graph_comm, &status);
183     MPI_Recv(MOi, N * N, MPI_INT, 0, MO_SENDING, graph_comm, &status);
184     MPI_Recv(MTh, H * N, MPI_INT, 0, MTH_SENDING, graph_comm, &status);
185   }
186
187   for (int i = 0; i < H; i++) {
188     Ah[i] = Sh[i];
189     for (int j = 0; j < N; j++) {
190       int mot = 0;
191       for (int k = 0; k < N; k++) {
192         mot += MOi[k * N + j] * MTh[i * N + k];
193       }
194       Ah[i] += maxZ * mot * Ei[j];
195     }
196   }
197
198
199   if (rank == mid - 1) {
200     int *A = new int[N];
201     for (int i = 0; i < H; i++) {
202       A[(mid - 1) * H + i] = Ah[i];
203     }
204     for (int i = 0; i < P; i++) {
205       if (i != mid - 1) {
206         MPI_Recv(A + i * H, H, MPI_INT, i, AH_SENDING, graph_comm, &status);
207       }
208     }
209     if (N <= 20) {
210       for (int i = 0; i < N; i++) {
211         cout << A[i] << " ";
212       }
213       cout << endl;
214     } else {
215     timestamp_t endTime = get_timestamp();
216     cout << (endTime - startTime) / 1000000.0 << endl;
```

```
217
218     }
219     delete[] A;
220   } else {
221     MPI_Send(Ah, H, MPI_INT, mid - 1, AH_SENDING, graph_comm);
222   }
223
224   delete[] Ei;
225   delete[] MOi;
226   delete[] MTh;
227   delete[] Sh;
228 }
229
230 void maxSort(int &maxZ, int* Z, int* S) {
231   maxZ = -2000000000;
232   for (int i = 0; i < H; i++) {
233     if (Z[i] > maxZ) {
234       maxZ = Z[i];
235     }
236   }
237
238   for (int i = H; i > 1; i--) {
239     for (int j = 1; j < i; j++) {
240       if (S[j - 1] > S[j]) {
241         int c = S[j - 1];
242         S[j - 1] = S[j];
243         S[j] = c;
244       }
245     }
246   }
247 }
248
249 void mergeSend(int rank, MPI_Comm graph_comm, int *Z, int *S, int *maxZP) {
250   int maxZ = *maxZP;
251   MPI_Status status;
252   int maxAnotherZ;
253   MPI_Recv(&maxAnotherZ, 1, MPI_INT, rank + mid, MAX_Z, graph_comm, &status);
```

```
254   if (maxAnotherZ > maxZ) {
255     maxZ = maxAnotherZ;
256   }
257
258   int* sortedS2 = new int[H];
259   MPI_Recv(sortedS2, H, MPI_INT, rank + mid, SORTED_S, graph_comm, &status);
260
261   int targetRank = rank < quarter ? rank + 1 : rank - 1;
262   int sourceRank = 2 * rank - targetRank;
263
264   if (rank < mid - 1 && rank > 0) {
265     MPI_Recv(&maxAnotherZ, 1, MPI_INT, sourceRank, MAX_Z, graph_comm, &status);
266     if (maxAnotherZ > maxZ) {
267       maxZ = maxAnotherZ;
268     }
269     int sortedSSize = (rank < quarter ? 2 * rank : 2 * (mid - 1 - rank)) * H;
270     int* sortedS3 = new int[sortedSSize];
271     MPI_Recv(sortedS3, sortedSSize, MPI_INT, sourceRank,
272         SORTED_S, graph_comm, &status);
273
274     if ((mid & 1) == 1 && rank == quarter) {
275       int maxAnotherZ;
276       MPI_Recv(&maxAnotherZ, 1, MPI_INT, targetRank, MAX_Z, graph_comm, &status);
277       if (maxAnotherZ > maxZ) {
278         maxZ = maxAnotherZ;
279       }
280       int *sortedS4 = new int[sortedSSize];
281       MPI_Recv(sortedS4, sortedSSize, MPI_INT, targetRank, SORTED_S, graph_comm,
&status);
282
283       int lengths[] = { H, H, sortedSSize, sortedSSize };
284       int* arrays[] = { S, sortedS2, sortedS3, sortedS4 };
285       int *merged = mergeN(4, lengths, arrays);
286
287       for (int i = 0; i < P; i++) {
288         if (i != rank) {
289           *maxZP = maxZ;
```

```
290           MPI_Send(maxZP, 1, MPI_INT, i, TOTAL_MAX_Z, graph_comm);
291           MPI_Send(merged + i * H, H, MPI_INT, i, PART_C, graph_comm);
292         }
293       }
294
295       for (int i = 0; i < H; i++) {
296         S[i] = merged[i + rank * H];
297       }
298
299       delete[] merged;
300       delete[] sortedS4;
301       return;
302     }
303     int mergedSize = sortedSSize + 2 * H;
304     int* merged;
305     int lengths[] = { H, H, sortedSSize };
306     int* arrays[] = { S, sortedS2, sortedS3 };
307     merged = mergeN(3, lengths, arrays);
308     int anotherMaxZ;
309
310     if ((mid & 1) == 0 && (rank == quarter - 1 || rank == quarter)) {
311       int maxAnotherZ;
312       int *sortedS2 = new int[mid * H];
313
314       if (rank == quarter) {
315         MPI_Send(&anotherMaxZ, 1, MPI_INT, targetRank, MAX_Z, graph_comm);
316         MPI_Send(merged, mergedSize, MPI_INT, targetRank, SORTED_S, graph_comm);
317
318         MPI_Recv(&maxAnotherZ, 1, MPI_INT, targetRank, MAX_Z, graph_comm, &status);
319         if (maxAnotherZ > maxZ) {
320           maxZ = maxAnotherZ;
321         }
322         MPI_Recv(sortedS2, mid * H, MPI_INT, targetRank, SORTED_S, graph_comm,
&status);
323
324       } else {
325         MPI_Recv(&maxAnotherZ, 1, MPI_INT, targetRank, MAX_Z, graph_comm, &status);
```

```
326         if (maxAnotherZ > maxZ) {
327           maxZ = maxAnotherZ;
328         }
329         MPI_Recv(sortedS2, mid * H, MPI_INT, targetRank, SORTED_S, graph_comm,
&status);
330
331         MPI_Send(&anotherMaxZ, 1, MPI_INT, targetRank, MAX_Z, graph_comm);
332         MPI_Send(merged, mergedSize, MPI_INT, targetRank, SORTED_S, graph_comm);
333       }
334
335       int* totalS;
336       int lengths[] = { mid * H, mid * H };
337       int* arrays[] = { merged, sortedS2 };
338       totalS = mergeN(2, lengths, arrays);
339       *maxZP = maxZ;
340       if (rank == quarter - 1) {
341         for (int i = 0; i < quarter - 1; i++) {
342           MPI_Send(maxZP, 1, MPI_INT, i, TOTAL_MAX_Z, graph_comm);
343           MPI_Send(totalS + i * H, H, MPI_INT, i, PART_C, graph_comm);
344         }
345         for (int i = mid; i < mid + quarter; i++) {
346           MPI_Send(maxZP, 1, MPI_INT, i, TOTAL_MAX_Z, graph_comm);
347           MPI_Send(totalS + i * H, H, MPI_INT, i, PART_C, graph_comm);
348         }
349       } else {
350         for (int i = quarter + 1; i < mid; i++) {
351           MPI_Send(maxZP, 1, MPI_INT, i, TOTAL_MAX_Z, graph_comm);
352           MPI_Send(totalS + i * H, H, MPI_INT, i, PART_C, graph_comm);
353         }
354         for (int i = mid + quarter; i < P; i++) {
355           MPI_Send(maxZP, 1, MPI_INT, i, TOTAL_MAX_Z, graph_comm);
356           MPI_Send(totalS + i * H, H, MPI_INT, i, PART_C, graph_comm);
357         }
358       }
359       for (int i = 0; i < H; i++) {
360         S[i] = totalS[i + rank * H];
361       }
```

```
362
363       delete[] merged;
364       delete[] totalS;
365       delete[] sortedS2;
366       return;
367     }
368     MPI_Send(&anotherMaxZ, 1, MPI_INT, targetRank, MAX_Z, graph_comm);
369     MPI_Send(merged, mergedSize, MPI_INT, targetRank, SORTED_S, graph_comm);
370
371     delete[] sortedS3;
372   } else {
373     int lengths[] = { H, H };
374     int* arrays[] = { S, sortedS2 };
375     int* merged = mergeN(2, lengths, arrays);
376     *maxZP = maxZ;
377     MPI_Send(maxZP, 1, MPI_INT, targetRank, MAX_Z, graph_comm);
378     MPI_Send(merged, 2 * H, MPI_INT, targetRank, SORTED_S, graph_comm);
379
380     delete[] merged;
381   }
382   delete[] sortedS2;
383
384   int sortedSourceRank = (mid & 1) == 0
385     ? rank < quarter ? quarter - 1 : quarter
386     : quarter;
387   MPI_Recv(maxZP, 1, MPI_INT, sortedSourceRank, TOTAL_MAX_Z, graph_comm, &status);
388   MPI_Recv(S, H, MPI_INT, sortedSourceRank, PART_C, graph_comm, &status);
389 }
390
391 int* mergeN(int n, int lengths[], int* arrays[]) {
392   int* coeffs = new int[n];
393   int totalLength = 0;
394   for (int i = 0; i < n; i++) {
395     coeffs[i] = 0;
396     totalLength += lengths[i];
397   }
398   int* merged = new int[totalLength];
```

```
399
400   for (int i = 0; i < totalLength; i++) {
401     int max_i = -1;
402     int max;
403     for (int j = 0; j < n; j++) {
404       if (coeffs[j] < lengths[j] && (max_i == -1 || arrays[j][coeffs[j]] > max)) {
405         max_i = j;
406         max = arrays[j][coeffs[j]];
407         coeffs[j]++;
408       }
409     }
410     merged[i] = max;
411   }
412
413   delete[] coeffs;
414   return merged;
415 }
```