тувНАЦІОНАЛЬНИЙ 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»

Студентки 3 курсу групи \_\_\_ІВ-71\_\_\_

спеціальності

123 «Комп’ютерна інженерія»

\_\_\_\_\_\_\_\_\_Молчанова В.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 – 2020 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

Спеціальність 123 «Комп’ютерна інженерія»

(шифр і назва)

### *З А В Д А Н Н Я*

**НА КУРСОВУ РОБОТУ СТУДЕНТУ**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Молчановій Варварі Сергіївні\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(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 2020 р.

3. Вхідні дані до роботи

* засоби роботи з процесами в бібліотеці OpenMP
* математична задача A= sort(Z)\*MX+ min(Z)\*R\*(MD\*ME)
* структури ПКС ОП та ПКС ЛП
* мови і бібліотеки програмування: Ада
* засоби організації взаємодії процесів: захищений модуль

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

- огляд засобів роботи з процесами в бібліотеці OpenMP

- 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 \_\_\_\_\_\_12.02.2020\_\_\_\_\_\_\_

### КАЛЕНДАРНИЙ ПЛАН

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання Розділу 1 | 01.03.2020 |
| 2 | Виконання Розділу 2 | 23.03.2020 |
| 3 | Виконання Розділу 3 | 23.04.2020 |
| 4 | Оформлення КР | 10.05.2020 |
| 5 | Здача КР керівнику | 11.05.2015 |
| 6 | Перевірка КР викладачем | 17.05.2015 |
| 7 | Захист КР | 18.05.2020 |

**Студент \_\_\_\_\_\_\_\_\_ \_\_\_**Молчанова В.С.\_\_\_\_\_\_\_\_\_\_\_

( підпис ) (прізвище та ініціали)

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

( підпис ) (прізвище та ініціали)

### ЗМІСТ

[ВСТУП 5](#_Toc38911014)

[РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ С ПРОЦЕСАМИ В БІБЛІОТЕЦІ OPENMP 6](#_Toc38911015)

[1.1. Основні принципи роботи бібліотеки OpenMP. 6](#_Toc38911016)

[1.2. Засоби програмування процесів в бібліотеці OpenMP. 9](#_Toc38911017)

[1.3. Синхронізація процесів в бібліотеці OpenMP 19](#_Toc38911023)

[1.4. Висновки до розділу 1 23](#_Toc38911028)

[РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП 25](#_Toc38911029)

[2.1. Розробка паралельного математичного алгоритму 25](#_Toc38911030)

[2.2. Розробка алгоритмів процесів 27](#_Toc38911031)

[2.3. Розробка схеми взаємодії процесів 29](#_Toc38911032)

[2.4. Розробка програми ПРГ1 31](#_Toc38911033)

[2.5. Тестування програми ПРГ1 32](#_Toc38911034)

[2.6. Висновки до розділу 2 36](#_Toc38911035)

[РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП 37](#_Toc38911036)

[3.1. Розробка паралельного математичного алгоритму 38](#_Toc38911037)

[3.2. Розробка алгоритмів процесів 39](#_Toc38911038)

[3.3. Розробка схеми взаємодії процесів 43](#_Toc38911039)

[3.4. Розробка програми ПРГ2 45](#_Toc38911040)

[3.5. Тестування програми ПРГ2 46](#_Toc38911041)

[3.6. Висновки до розділу 3 49](#_Toc38911042)

[ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ДО РОБОТИ 50](#_Toc38911043)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 51](#_Toc38911044)

[ДОДАТКИ 52](#_Toc38911045)

### ВСТУП

Курсова робота по дисципліні «Паралельні і розподілені обчислення» складається з трьох розділів.

В першому розділі наведено загальний опис роботи паралельних програм та огляд засобів роботи с процесами в бібліотеці OpenMP.

Другий та третій розділи присвячені розробці програми для обчислення математичної задачі в паралельній комп’ютерній системі зі спільною та локальною пам’яттю відповідно. Програмне забезпечення для комп’ютерної системи зі спільною та локальною пам’яттю розроблено на мові Аdа із використанням механізму захищених модулів та Рандеву. Проведено тестування отриманих програмних продуктів і зроблено висновки по їх ефективності.

Лістинги розроблених програм наведено у додатках.

### РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ С ПРОЦЕСАМИ В БІБЛІОТЕЦІ OPENMP

OpenMP (Open Multi-Processing) — це набір директив компілятора, бібліотечних процедур та змінних середовища, які призначені для програмування багатопоточних застосунків на багатопроцесорних системах із спільною пам'яттю на мовах C, C++ та Fortran.

Розробку специфікації OpenMP ведуть кілька великих виробників обчислювальної техніки та програмного забезпечення, робота яких регулюється некомерційною організацією, названою OpenMP Architecture Review Board (ARB). Детальна специфікація OpenMP міститься на сторінці. Специфікації для мов Fortran і C/C++ з'явилися відповідно в жовтні 1997 року і жовтні 1998 року. [1].

#### Основні принципи роботи бібліотеки OpenMP.

Будь-яка програма, послідовна або паралельна, складається з набору областей двох типів: послідовних областей і областей розпаралелювання. При виконанні послідовних областей породжується тільки один головний потік (процес). У цьому ж потоці ініціюється виконання програми, а також відбувається її завершення. У послідовній програмі в областях розпаралелювання породжується також тільки один, головний потік, і цей потік є єдиним протягом виконання всієї програми. У паралельній програмі в областях розпаралелювання породжується ціла низка паралельних потоків. Породжені паралельні потоки можуть виконуватися як на різних процесорах, так і на одному процесорі обчислювальної системи. В останньому випадку паралельні процеси (потоки) конкурують між собою за доступ до процесора. Управління конкуренцією здійснюється планувальником операційної системи за допомогою спеціальних алгоритмів. В операційній системі Linux планувальник завдань здійснює обробку процесів за допомогою стандартного карусельного (round-robin) алгоритму. При цьому тільки адміністратори системи мають можливість змінити або замінити цей алгоритм системними засобами. Таким чином, в паралельних програмах в областях розпаралелювання виконується ряд паралельних потоків. Принципова схема паралельної програми зображена на рис. 1.1.

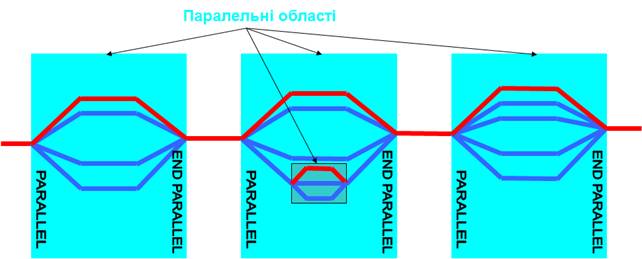


Рис. 1.1. Принципова схема паралельної програми.

При виконанні паралельної програми робота починається з ініціалізації та виконання головного потоку (процесу), який у міру необхідності створює і виконує паралельні потоки, передаючи їм необхідні дані. Паралельні потоки з однієї паралельної області програми можуть виконуватися як незалежно один від одного, так і з пересилкою та отриманням повідомлень від інших паралельних потоків. Остання обставина ускладнює розробку програми, оскільки в цьому випадку програмісту доводиться займатися плануванням, організацією і синхронізацією посилки повідомлень між паралельними потоками. Таким чином, при розробці паралельної програми бажано виділяти такі області розпаралелювання, в яких можна організувати виконання незалежних паралельних потоків. Для обміну даними між паралельними процесами (потоками) в OpenMP використовуються загальні змінні. При зверненні до загальних змінним в різних паралельних потоках можливе виникнення конфліктних ситуацій при доступі до даних. Для запобігання конфліктів можна скористатися процедурою синхронізації. При цьому треба мати на увазі, що процедура синхронізації - дуже дорога операція по тимчасових витратах і бажано по можливості уникати її або застосовувати якомога рідше. Для цього необхідно дуже ретельно продумувати структуру даних програми.

Виконання паралельних потоків в паралельній області програми починається з їх ініціалізації. Воно полягає у створенні дескрипторів породжуваних потоків і копіювання всіх даних з області даних головного потоку в області даних створюваних паралельних потоків. Ця операція надзвичайно трудомістка - вона еквівалентна приблизно трудомісткості 1000 операцій. Ця оцінка надзвичайно важлива при розробці паралельних програм c допомогою OpenMP, оскільки її ігнорування веде до створення неефективних паралельних програм, які виявляються найчастіше повільніше їх послідовних аналогів. Справді: для того щоб отримати виграш у швидкодії паралельної програми, необхідно, щоб трудомісткість паралельних процесів в областях розпаралелювання програми істотно перевершувала б трудомісткість породження паралельних потоків. В іншому випадку ніякого виграшу за швидкодією отримати не вдасться, а часто можна опинитися навіть і в програші.

Завершення виконання паралельних потоків управління програмою знову передає виконання головному потоку. При цьому виникає проблема коректної передачі даних від паралельних потоків головного. Тут важливу роль відіграє синхронізація завершення роботи паралельних потоків, оскільки в силу цілого ряду обставин час виконання навіть однакових по трудомісткості паралельних потоків непередбачувано (воно визначається як історією конкуренції паралельних процесів, так і поточним станом обчислювальної системи). При виконанні операції синхронізації паралельні потоки, вже завершили своє виконання, простоюють і чекають завершення роботи самого останнього потоку. Природно, при цьому неминуча втрата ефективності роботи паралельної програми. Крім того, операція синхронізації має трудомісткість, порівнянну з трудомісткістю ініціалізації паралельних потоків, тобто еквівалентна приблизно трудомісткості виконання 1000 операцій. [2].

На підставі викладеного вище можна зробити наступний важливий висновок: при виділенні паралельних областей програми та розробці паралельних процесів необхідно, щоб трудомісткість паралельних процесів була не менш 2000 операцій ділення. В іншому випадку паралельний варіант програми буде програвати у швидкодії послідовній програмі. Для ефективної працюючої паралельної програми ця межа має бути істотно перевищена.

#### Засоби програмування процесів в бібліотеці OpenMP.

За рахунок ідеї "часткового розпаралелювання" OpenMP ідеально підходить для розробників, бажаючих швидко розпаралелити свої обчислювальні програми з великими паралельними циклами. Розробник не створює нову паралельну програму, а просто додає в текст послідовної програми OpenMP директиви. Передбачається, що OpenMP-програма на однопроцесорній платформі може бути використана в якості послідовної програми, тобто немає необхідності одночасно підтримувати послідовну і паралельну версії. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs), текст яких наведено в специфікаціях.

Специфікації паралелізму в OpenMP представляють собою директиви виду

#pragma omp [назва директиви] [опції] {тіло}

Наприклад:

#pragma omp parallel default(none) shared(i,j)

Існують вконувані директиви (*barrier, taskwait, flush*) та описові (*threadprivate*). Решта директив розповсюджується на структурний блок. [3].

##### Директива parallel

Директива *parallel* створює паралельну область для наступного за нею структурованого блоку. ЇЇ можливі опції:

* *if* (умова) - виконання паралельної області за умовою. Входження в паралельну область здійснюється тільки при виконанні деякої умови. Якщо умова не виконана, то директива не спрацьовує і триває обробка програми в колишньому режимі;
* *num\_threads* (цілочисленний вираз) - явне завдання кількості потоків, які будуть виконувати паралельну область; за замовчуванням вибирається останнє значення, встановлене за допомогою функції *omp\_set\_num\_threads* (), або значення змінної OMP\_NUM\_THREADS;
* *default* (shared | none) - всім змінним у паралельній області, яким явно не призначений клас, буде призначений клас shared; none означає, що всім змінним у паралельній області клас повинен бути призначений явно;
* *private* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* *firstprivate* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* *shared* (список) - задає список змінних, загальних для всіх потоків;
* *copyin* (список) - задає список змінних, оголошених як threadprivate, які при вході в паралельну область ініціалізуються значеннями відповідних змінних в потоці-майстра;
* *reduction* (оператор: список) - задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно типу оператора (для адитивних операцій - 0 або його аналоги, для мультиплікативних операцій - 1 або її аналоги); над локальними копіями змінних після виконання всіх операторів паралельної області виконується заданий оператор; оператор це: +, \*, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску. [4]

Ця директива повідомляє компілятор, що структурований блок коду(тіло) повинен бути виконаний паралельно, в декількох потоках. Кожен потік буде виконувати один і той же потік команд, але не один і той же набір команд - все залежить від операторів, керуючих логікою програми, таких як *if-else*.

При вході в паралельну область породжуються нові OMP\_NUM\_THREADS-1 потоків, кожний потік отримує свій унікальний номер, причому породжуючий потік отримує номер 0 і стає основним потоком групи ("майстром"). Решта потоків отримують в якості номера цілі числа з 1 до OMP\_NUM\_THREADS-1. Кількість потоків, що виконують дану паралельну область, залишається незмінним до моменту виходу з області. При виході з паралельної області виробляється неявна синхронізація і знищуються всі потоки, крім майстра.

Якщо один з потоків паралельної області зустрічає іншу директиву parallel, то він створює нову групу потоків, згідно з правилами, і стає основним потоком нової групи.

Якщо виконання потоку аварійно переривається всередині паралельної області, то також переривається виконання всіх потоків у всіх групах [2]. Порядок переривання роботи потоків не визначений. Вся робота, виконана групою до останньої бар'єрної синхронізації, гарантовано буде виконана. Обсяг виконаної роботи, виконаної кожним потоком після останньої бар'єрної синхронізації, до аварійного завершення роботи потоків не визначений.

Всі породжені потоки виконують один і той же код, відповідний до паралельної області. Передбачається, що в SMP-системі потоки будуть розподілені по різним процесорам (однак це, як правило, перебуває у віданні операційної системи).

Під час виконання будь потік може призупинити виконання своє неявне завдання в точці планування завдань і переключитися на виконання будь-якої явно-згенерованої задачі перш ніж відновити виконання неявної задачі.

Потік може дізнатися свій номер за допомогою виклику бібліотечної функції *omp\_get\_thread\_num*.

Дуже часто паралельна область не містить нічого, крім конструкції поділу роботи (тобто конструкція поділу роботи тісно вкладена в паралельну область). У цьому випадку можна вказувати не дві директиви, а вказати одну комбіновану.

Наприклад, якщо всередині паралельної області міститься тільки один паралельний цикл або одна конструкція *sections*, то можна використовувати укорочену запис: *parallel for* або *parallel sections*. При цьому допустима вказівка ​​всіх опцій цих директив, за винятком опції *nowait*.

Обмеження для директиви *parallel* наступні:

* Програма не повинна залежати від будь-якого порядку визначення опцій паралельної директиви, або від будь-яких побічних ефектів визначення опцій;
* Тільки одна опція if може бути присутня в директиві;
* Тільки одна опція num\_threads може бути присутня в директиві. Вираження в опції num\_threads має бути цілочисловим;
* Виклик винятку виконаний всередині паралельної області повинен викликати обробку виключення в рамках однієї паралельної області, і того ж потоку, який викликав виняток. [5].

##### Оператори циклу

Якщо в паралельній області зустрівся оператор циклу, то, згідно загальним правилам, він буде виконаний всіма потоками поточної групи, тобто кожний потік виконає всі ітерації даного циклу. Для розподілу ітерацій циклу між різними потоками можна використовувати директиву *for*.

#pragma omp for [*опція*[,*опція*] ... ]{*тіло циклу*}

Ця директива відноситься до блоку що йде слідом за даною директивою.

Можливі опції:

* *private* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* *firstprivate* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* *lastprivate* (список) - змінним, перерахованим у списку, присвоюється результат з останнього витка циклу;
* *reduction* (оператор: список) - задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно типу оператора (для адитивних операцій - 0 або його аналоги, для мультиплікативних операцій - 1 або її аналоги); над локальними копіями змінних після завершення всіх ітерацій циклу виконується заданий оператор; оператор це: +, \*, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску;
* *schedule (type [, chunk])* - опція задає, яким чином ітерації циклу розподіляються між потоками;
* *collapse* (n) - опція вказує, що n послідовних вкладених циклів асоціюються з даною директивою; для циклів утворюється загальний простір ітерацій, яке ділиться між потоками; якщо опція collapse не задана, то директива відноситься тільки до одного безпосередньо наступному за нею циклу;
* *ordered* - опція, що говорить про те, що в циклі можуть зустрічатися директиви *ordered*; в цьому випадку визначається блок всередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть в послідовному циклі;
* *nowait* - наприкінці паралельного циклу відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається тільки тоді, коли всі вони досягнуть даної точки; якщо в подібній затримці немає необхідності, опція *nowait* дозволяє потокам, що вже дійшли до кінця циклу, продовжити виконання без синхронізації з іншими.

На вигляд паралельних циклів накладаються досить жорсткі обмеження. Зокрема, передбачається, що коректна програма не повинна залежати від того, який саме потік яку ітерацію паралельного циклу виконає. Не можна використовувати побічний вихід з паралельного циклу. Розмір блоку ітерацій, зазначений в опції schedule, не повинен змінюватися в рамках циклу.

Ці вимоги введені для того, щоб OpenMP міг при вході в цикл точно визначити число ітерацій. Якщо директива паралельного виконання стоїть перед гніздом циклів, що завершуються одним оператором, то директива діє тільки на самий зовнішній цикл. Ітеративна змінна розподіленого циклу за змістом повинна бути локальною, тому в разі, якщо вона специфікована загальною, то вона неявно робиться локальною при вході в цикл. Після завершення циклу значення ітеративної змінної циклу не визначено, якщо не вказано в опції *lastprivate*.

Наступний приклад демонструє використання директиви for. У послідовній області ініціалізуються три вихідних масивів A, B, C. В паралельній області дані масиви оголошені загальними. Допоміжні змінні i і n оголошені локальними. Кожний потік присвоїть змінній n свій порядковий номер. Далі за допомогою директиви *for* визначається цикл, ітерації якого будуть розподілені між існуючими потоками. На кожній i-ій ітерації даний цикл складе i-ті елементи масивів A і B і результат запише в i-ий елемент масиву C. Також на кожній ітерації буде надрукований номер потоку, що виконав дану ітерацію.

Приклад 1.2.1

#include <stdio.h>   
#include <omp.h>    
int main(int argc, char \*argv[])   
{   
    int A[10], B[10], C[10], i, n;   
// заповнимо початковий масив   
    for (i = 0; i < 10; i++)   
    {   
        A[i] = i;   
        B[i] = 2 \* i;   
        C[i] = 0;   
    }   
    #pragma omp parallel shared(A, B, C) private(i, n)   
    {   
// отримаємо номер потоку   
        n = omp\_get\_thread\_num();   
        #pragma omp for   
        for (i = 0; i < 10; i++)   
        {   
            C[i] = A[i] + B[i];   
            printf("Потік \%d додав елементи з номером %d\n", n, i);   
        }   
    }   
}

##### Директива single

Якщо в паралельній області який-небудь ділянку коду повинен бути виконаний лише один раз, то його потрібно виділити директивою *single*.

#pragma omp single [опція[, опція]...]{тіло}

Можливі опції:

* *private* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* *firstprivate* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* *copyprivate* (список) - після виконання нитки, що містить конструкцію одно-, нові значення змінних списку будуть доступні всім однойменною приватним змінним (private та *firstprivate*), описаним на початку паралельної області і використовуваним усіма її потоками; опція не може використовуватися спільно з опцією *nowait*; змінні списку не повинні бути перераховані в опціях *private* і *firstprivate* даної директиви *single*;
* *nowait* - після виконання виділеної ділянки відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається тільки тоді, коли всі вони досягнуть даної точки; якщо в подібній затримці немає необхідності, опція *nowait* дозволяє потокам, що вже дійшли до кінця ділянки, продовжити виконання без синхронізації з іншими.

Доволі часто зустрічаються блоки коду, доступ до яких бажано надавати тільки одному потоку, - наприклад, блоки коду, що відповідають за запис даних у файл. У багатьох таких ситуаціях не має значення, який потік виконає код, важливо лише, щоб цей потік був єдиним. Для цього в OpenMP служить директива *single*.

Обмеження для директиви *single* наступні:

* Один потік буде виконувати фрагмент, а всі інші потоки будуть очікувати завершення його роботи, якщо тільки не вказана опція *nowait*.
* Опція *copyprivate* не повинна використовуватися разом з опцією *nowait*;
* Тільки одна опція *nowait* може бути використана в директиві *single*;
* Виняток викликаний в області *single* має бути оброблений в рамках однієї області *single*, одним і тим же ж потоком.

##### Директива sections

Директива *sections* використовується для задання кінцевого (неітеративного) паралелізму. Директива *sections* містить набір структурованих блоків, які розподіляються по потокам в групі. Кожен структурований блок виповнюється один раз, одним з потоків в групі.

#pragma omp sections [опція[, опція] ...]   
{   
#pragma omp section

--тіло   
#pragma omp

--тіло

...   
}

При виконанні цього коду OpenMP спочатку створює групу потоків, а потім розподіляє між ними обробку ітерацій циклу, після виконання якого потоки починають паралельну обробку розділів коду, що залишилися. Якщо кількість розділів програмного коду буде більше числа потоків, обробка декількох розділів буде відкладена до тих пір, поки не з'являться вільні потоки. На відміну від планування циклів, розподіл навантаження між потоками при обробці паралельних розділів коду здійснюється і контролюється OpenMP. Програмісту залишається тільки вибрати, які змінні будуть загальними, а які - індивідуальними, і передбачити вираження зменшення аналогічно сегменту з організацією циклів.

Можливі опції:

* *private* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* *firstprivate* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* *lastprivate* (список) - змінним, перерахованим у списку, присвоюється результат, отриманий в останній секції;
* *reduction* (оператор: список) - задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно типу оператора (для адитивних операцій - 0 або його аналоги, для мультиплікативних операцій - 1 або її аналоги); над локальними копіями змінних після завершення всіх секцій виконується заданий оператор; оператор це: +, \*, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску;
* *nowait* - наприкінці блоку секцій відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається тільки тоді, коли всі вони досягнуть даної точки; якщо в подібній затримці немає необхідності, опція *nowait* дозволяє потокам, що вже дійшли до кінця своїх секцій, продовжити виконання без синхронізації з іншими.

Директива *section* задає ділянку коду, усередині секції *sections*, для виконання одним потоком.

#pragma omp section

Перед першою ділянкою коду в блоці *sections* директива *section* не обов'язкова. Які саме потоки будуть задіяні для виконання якої секції не визначається. Якщо кількість потоків більше кількості секцій, то частина потоків, для виконання секцій, не буде задіяна. Якщо кількість потоків менше кількості секцій, то деяким (або усім) ниткам дістанеться більше однієї секції.

##### Директива task

Директива *task* застосовується для виділення окремої незалежної завдання.

#pragma omp task опція[[[,] опція] ...]

--тіло

Поточний потік виділяє як завдання асоційований з директивою блок операторів. Завдання може виконуватися негайно після створення або бути відкладено на невизначений час і виконуватися по частинах. Розмір таких частин, а також порядок виконання частин різних відкладених завдань визначається реалізацією.

Можливі опції:

* *if* (умова) - породження нового завдання тільки при виконанні деякої умови; якщо умова не виконується, то завдання буде виконано поточним потоком і негайно;
* *untied* - опція означає, що у разі відкладання завдання може бути продовжена будь-яким потоком з числа виконуючих дану паралельну область; якщо дана опція не вказана, то завдання може бути продовжено тільки потоком-майстром;
* *default* (*shared* | *none*) - всім змінним в задачі, яким явно не призначений клас, буде призначений клас *shared*; *none* означає, що всім змінним в задачі клас повинен бути призначений явно;
* *private* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
* *firstprivate* (список) - задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в потоці-майстра;
* *shared* (список) - задає список змінних, загальних для всіх потоків.

Наступний приклад показує як пройти деревоподібну структуру використовуючи директиву *task.* Відзначимо, що функція траверс (*traverse*) повинна бути викликана з паралельної області для різних зазначених завдань, які будуть виконуватися паралельно. Також відзначимо, що завдання виконуються не в зазначеному порядку, оскільки тут не використовуються директиви синхронізації. Таким чином, припущення, що обхід буде зроблений в тому ж порядку, що і в послідовному коді, є невірним.

Приклад 1.2.2

struct node   
{   
    struct node \*left;   
    struct node \*right;   
};    
extern void process(struct node \*);   
void traverse( struct node \*p )   
{   
    if (p->left)   
        #pragma omp task // p є firstprivate-змінною за замовченням   
            traverse(p->left);   
    if (p->right)   
        #pragma omp task // p є firstprivate-змінною за замовченням  
            traverse(p->right);   
    process(p);   
}

#### Синхронізація процесів в бібліотеці OpenMP

У попередніх розділах при згадуванні директив роботи з циклами проблема синхронізації вже зачіпалася. По-перше, було зазначено, що ця процедура є дуже трудомісткою і порівнянна з трудомісткістю ініціалізації паралельних потоків. Тому бажано користуватися синхронізацією якомога рідше.

По-друге, було відзначено, що неявно (за замовчуванням) синхронізація паралельних процесів забезпечується при виконанні циклів в паралельному режимі. Була згадана директива nowait для усунення неявній синхронізації при завершенні циклів. Проте користуватися цією директивою слід вельми і вельми акуратно, попередньо проаналізувавши порядок роботи програми і переконавшись, що скасування синхронізації не приведе до псування даних і непередбачуваних результатів.

Механізм роботи синхронізації можна описати таким чином. При ініціалізації набору паралельних процесів в програмі встановлюється контрольна точка (аналогічна контрольній точці у відладчику), в якій програма очікує завершення всіх породжених паралельних процесів. Відзначимо, що поки всі паралельні процеси свою роботу не завершили, програма не може продовжити роботу за точкою синхронізації. А оскільки всі сучасні високопродуктивні процесори є процесорами конвеєрного типу, стає зрозумілою і висока трудомісткість процедури синхронізації. Справді, поки не завершені всі паралельні процеси, програма не може розпочати підготовку завантаження конвеєрів процесорів. Ось це-то і веде до великих втрат при синхронізації процесів, аналогічних втрат при роботі умовних операторів у звичайній послідовній програмі.

##### Директива critical

За допомогою директив *critical* оформляється критична секція програми. Критична секція забороняє одночасне виконання структурованого блоку більш ніж одним потоком.

#pragma omp critical ім’я[()]

--тіло

В один момент часу в критичній секції може перебувати не більше одного потоку. Якщо критична секція вже виконується будь-яким потоком, то всі інші потоки, які виконали директиву для секції з даним ім'ям, будуть заблоковані, поки потік не закінчить виконання даної критичної секції. Як тільки потік що працював вийде з критичної секції, один із заблокованих на вході потоків увійде в неї. Якщо на вході в критичну секцію стояло кілька потоків, то випадковим чином вибирається один з них, а решта заблокованих потоків продовжують очікування.

Всі неіменовані критичні секції умовно асоціюються з одним і тим же ім'ям. Всі критичні секції, що мають одне і теж ім'я, розглядаються єдиної секцією, навіть якщо знаходяться в різних паралельних областях. Побічні входи і виходи з критичної секції заборонені.

Наступний приклад ілюструє застосування директиви *critical*. Змінна n оголошена поза паралельної області, тому за замовчуванням є спільною. Критична секція дозволяє розмежувати доступ до змінної n. Кожна нитка по черзі присвоїть n свій номер і потім надрукує отримане значення.

Приклад 1.3.1

#include <stdio.h>   
#include <omp.h>   
int main(int argc, char \*argv[])   
{   
    int n;   
    #pragma omp parallel   
    {   
        #pragma omp critical   
        {   
            n = omp\_get\_thread\_num();   
            printf("Потік %d\n", n);   
        }   
    }   
}

Якби в прикладі не була вказана директива *critical*, результат виконання програми був би непередбачуваний. З директивою *critical* порядок виведення результатів може бути довільним, але це завжди буде набір одних і тих же чисел від 0 до OMP\_NUM\_THREADS-1. Звичайно, подібного ж результату можна було б добитися іншими способами, наприклад, оголосивши змінну n локальною, тоді кожний потік працював би зі своєю копією цієї змінної. Однак у виконанні цих фрагментів суттєва різниця.

Якщо є критична секція, то в кожен момент часу фрагмент буде оброблятися лише якимось одним потоком. Решта потоків, навіть якщо вони вже підійшли до даної точки програми і готові до роботи, чекатимуть своєї черги. Якщо критичної секції немає, то всі потоки можуть одночасно виконати дану ділянку коду. З одного боку, критичні секції надають зручний механізм для роботи із загальними змінними. Але з іншого боку, користуватися ним потрібно обачно, оскільки критичні секції додають послідовні ділянки коду в паралельну програму, що може знизити її ефективність.

##### Директива barrier

Найпоширеніший спосіб синхронізації в OpenMP - бар'єр. Він оформляється за допомогою директиви *barrier*. Директива *barrier* дає всім потокам вказівку очікувати один одного перед тим, як вони продовжать виконання за бар'єром.

#pragma omp barrier

Потоки, які виконують поточну паралельну область, дійшовши до цієї директиви, зупиняються і чекають, поки всі потоки не дійдуть до цієї точки програми, після чого розблокуються і продовжують працювати далі. Крім того, для розблокування необхідно, щоб всі потоки, які синхронізуються завершили всі породжені ними завдання (*task*).

##### Директива atomic

Частим випадком використання критичних секцій на практиці є оновлення загальних змінних. Наприклад, якщо змінна *sum* є загальною і оператор виду *sum* = *sum* + *expr* знаходиться в паралельній області програми, то при одночасному виконанні даного оператора декількома потоками можна отримати некоректний результат. Щоб уникнути такої ситуації можна скористатися механізмом критичних секцій або спеціально передбаченою для таких випадків директивою *atomic*.

#pragma omp atomic capture

--тіло

Дана директива відноситься до оператора присвоювання, що йде безпосередньо за нею (на використовувані в якому конструкції накладаються досить зрозумілі обмеження), гарантуючи коректну роботу із загальною змінної, що стоїть в його лівій частині. На час виконання оператора блокується доступ до даної змінної, всім запущеним в даний момент потокам, крім потоку, що виконує операцію. Атомарної є тільки робота зі змінною в лівій частині оператора присвоєння, при цьому обчислення в правій частині не зобов'язані бути атомарними.

Наступний приклад ілюструє застосування директиви *atomic*. У даному прикладі проводиться підрахунок загальної кількості породжених потоків. Для цього кожний потік збільшує на одиницю значення змінної *count*. Для того, щоб запобігти одночасній зміні декількома потоками значення змінної, що стоїть в лівій частині оператора присвоювання, використовується директива *atomic*.

Приклад 1.3.2

#include <stdio.h>   
#include <omp.h>   
int main(int argc, char \*argv[])   
{   
    int count = 0;   
    #pragma omp parallel   
    {   
        #pragma omp atomic   
        count++;   
    }   
    printf("Кількість потоків: %d\n", count);   
}

##### Директива ordered

Директиви ordered визначають блок всередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть в послідовному циклі.

#pragma omp ordered

--тіло

Блок операторів відноситься до самого внутрішнього з осяжниг циклів, а в паралельному циклі повинна бути задана опція ordered. Потік, що виконує перший ітерацію циклу, виконує операції даного блоку. Потік, що виконує будь-яку наступну ітерацію, повинен спочатку дочекатися виконання всіх операцій блоку усіма потоками, які виконують попередні ітерації. Може використовуватися, наприклад, для впорядкування висновку від паралельних потоків. [6].

#### Висновки до розділу 1

1. Виконано аналіз принципів роботи паралельних програм та бібліотека OpenMP. Досліджено, що паралелізм є доволі важливим в умовах вирішення, великих за обсягом, задач. Проте доволі часто використання паралелізму не є виправданим. Оскільки, при виділенні паралельних ділянок програми і розробці паралельних процесів необхідно, щоб трудомісткість паралельних процесів була не менше 2000 операцій поділу. В іншому випадку паралельний варіант програми буде програвати у швидкодії послідовної програмі. Ресурсозатратність не завжди позитивно впливає на роботу програми, яка використовує паралелізм.
2. Виявлено, що проблема синхронізації паралельних потоків важлива не тільки для паралельного програмування з використанням OpenMP, а й для всього паралельного програмування в цілому. Проблема полягає в тому, що структурний паралельний блок за визначенням має одну точку виходу, за якою зазвичай знаходиться послідовний структурний блок. Обчислення в послідовному блоці, як правило, можуть бути продовжені, якщо завершені всі процеси в паралельному структурному блоці і їх результати коректно передані в послідовний блок. Саме для забезпечення такої коректної передачі даних і необхідна процедура синхронізації паралельних потоків.

### РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС СП

У даному розділі розглянуто розробку та дослідження програми ПРГ1 для ПКС зі СП.

Математична задача:

A= sort(Z)\*MX+ min(Z)\*R\*(MD\*ME)

Мова програмування: Ada (з використанням захищених модулів).

Структура ПКС СП показана на рис. 2.1 та у Додатку А. Введення даних відбувається в процесах 1, 4, 6, 8. Виведення даних відбувається у процесі 8. На схемі СП – спільна пам’ять; П1, П4, П6, П8 – процеси.

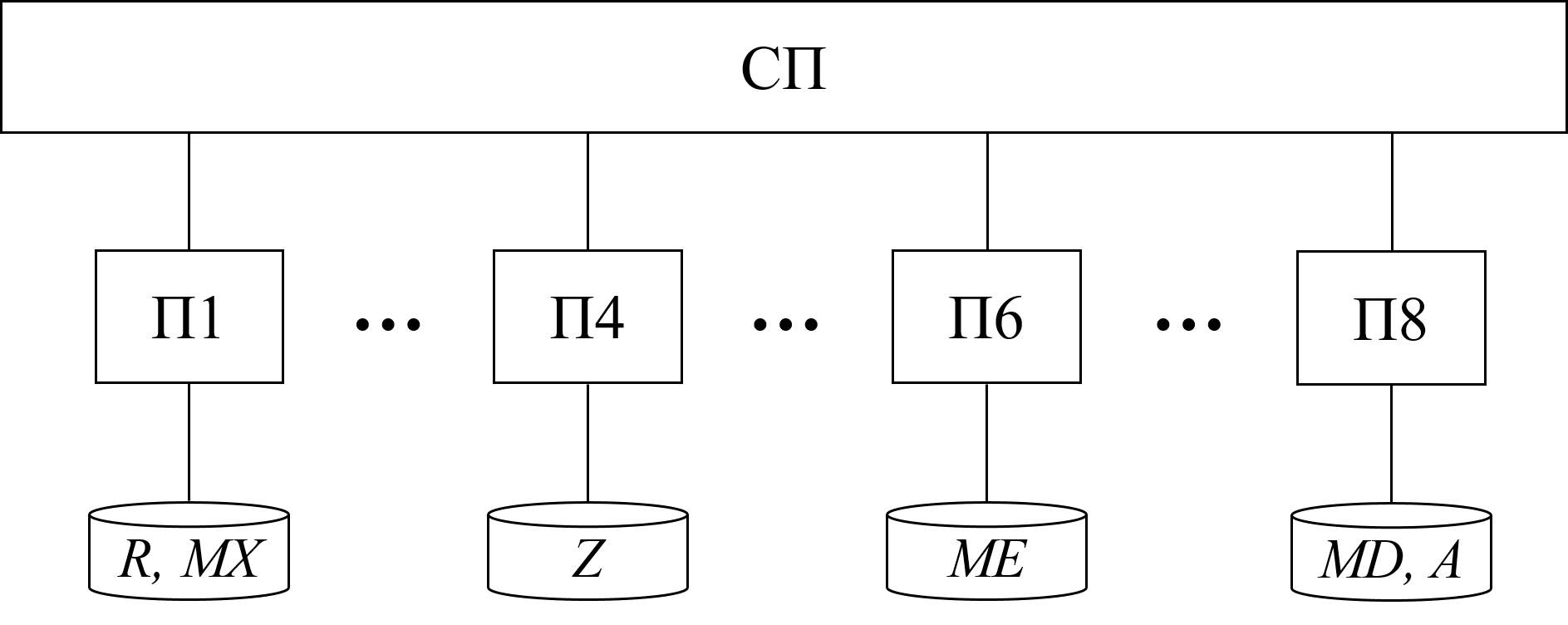


Рис. 2.1. Структура ПКС з СП

#### Розробка паралельного математичного алгоритму

Згідно з технічним завданням необхідно розробити паралельний алгоритм. Кількість процесорів P = 8. Його можна розділити на наступні етапи:

1. 
2. 
3. 
4. 
5. 

Спільні ресурси: 

Пояснення до використовуваних констант позначень:

* розмірність векторів і матриць;
*  кількість ядер;
* 
*  - сортування злиттям
* 
* 

Оцінка прискорення і ефективності розробленого алгоритму, згідно з теоремою Мунро-Петерсона:



Де кількість бінарних операцій, час розв’язання задачі на  вузлах.

1. Для сортування вектору *Z* потрібно  операцій порівняння. Тому час порівняння буде:



1. Для обчислення одного елементу добутку матриць  необхідно виконати *N* операцій множення та *N –* 1додавання. Тому час виконання буде:



1. Для обчислення одного елементу вектору  необхідно виконати *N* операцій множення та *N –* 1додавання. Тому час виконання буде:



1. Для обчислення одного елементу вектору  необхідно виконати 2*N* операцій множення та *N –* 1додавання. Тому час виконання буде:



1. Операція додавання виконується за 1 такт. Отже, сумарний час виконання операції:



#### Розробка алгоритмів процесів

|  |  |  |
| --- | --- | --- |
|  | **Т1** | ТС КД |
| 1. | Ввести дані: *R, MX* |  |
| 2. | Сигнал Т2-Т8 про введення даних |  |
| 3. | Очікувати введення даних в Т4, Т6, Т8 |  |
| 4. |  |  |
| 5. | Очікувати завершення сортування в Т2 |  |
| 6. |  |  |
| 7. | Очікувати завершення сортування\* в Т4 |  |
| 8. |  |  |
| 9. | Очікувати завершення сортування\* в Т8 |  |
| 10. |  |  |
| 11. | Сигнал Т2-Т8 про завершення сортування |  |
| 12. | Копіювати | КД |
| 13. | Обчислити |  |
| 14. | Сигнал Т8 про завершення обчислення |  |

|  |  |  |
| --- | --- | --- |
|  | **Т4** | ТС КД |
| 1. | Ввести дані: *Z* |  |
| 2. | Сигнал Т1-Т3, Т5-Т8 про введення даних |  |
| 3. | Очікувати введення даних в Т1, Т6, Т8 |  |
| 4. |  |  |
| 5. | Очікувати завершення сортування в Т3 |  |
| 6. |  |  |
| 7. | Сигнал Т1 про завершення сортування\* |  |
| 8. | Очікувати завершення сортування\* в Т1 |  |
| 9. | Копіювати | КД |
| 10. | Обчислити |  |
| 11. | Сигнал Т8 про завершення обчислення |  |

|  |  |  |
| --- | --- | --- |
|  | **Т6** | ТС КД |
| 1. | Ввести дані: *ME* |  |
| 2. | Сигнал Т1-Т5, Т7-Т8 про введення даних |  |
| 3. | Очікувати введення даних в Т1, Т4, Т8 |  |
| 4. |  |  |
| 5. | Очікувати завершення сортування в Т5 |  |
| 6. |  |  |
| 7. | Сигнал Т8 про завершення сортування\* |  |
| 8. | Очікувати завершення сортування\* в Т1 |  |
| 9. | Копіювати | КД |
| 10. | Обчислити |  |
| 11. | Сигнал Т8 про завершення обчислення |  |

|  |  |  |
| --- | --- | --- |
|  | **Т8** | ТС КД |
| 1. | Ввести дані: *MD* |  |
| 2. | Сигнал Т1-Т7 про введення даних |  |
| 3. | Очікувати введення даних в Т1, Т4, Т6 |  |
| 4. |  |  |
| 5. | Очікувати завершення сортування в Т7 |  |
| 6. |  |  |
| 7. | Очікувати завершення сортування\* в Т6 |  |
| 8. |  |  |
| 9. | Сигнал Т1 про завершення сортування\* |  |
| 10. | Очікувати завершення сортування\* в Т1 |  |
| 11. | Копіювати | КД |
| 12. | Обчислити |  |
| 13. | Очікувати завершення обчислення в Т1-Т7 |  |
| 14. | Вивести *А* |  |

Задачі Т2, Т3, Т5, Т7 мають аналогічний алгоритм з використанням  заміни індексу *i* на відповідний номер задачі.

|  |  |  |
| --- | --- | --- |
|  | **Т*i*** | ТС КД |
| 1. | Очікувати введення даних в Т1, Т4, Т6, Т8 |  |
| 2. |  |  |
| 3. | Сигнал (*i*=2 – T1, *i*=3 – T4, *i*=5 – T6, *i*=7 – T8) про завершення сортування |  |
| 4. | Очікувати завершення сортування\* в Т1 |  |
| 5. | Копіювати | КД |
| 6. | Обчислити |  |
| 7. | Сигнал Т8 про завершення обчислення |  |

#### Розробка схеми взаємодії процесів

У програмі використовуються два захищених модулі. ResourceModule відповідає за вирішення задачі взаємного виключення, SynchronizeModule відповідає за синхронізацію задач.

Захищені функції на процедури в  ResourceModule використвуються для копіюванні даних при доступі до СР.

Захищені функції для копіювання:

* CopyS (копіювання вектора S)
* CopyR (копіювання вектора R)
* CopyMD (копіювання матриці MD).

Для запису в монітор використовуються захищені функції:

* WriteS (запис вектора S)
* WriteR (запис вектора R)
* WriteMD (запис матриці MD).

В моніторі SynchronizeModule відбувається синхронізація потоків за допомогою захищених процедур та захищених входів.

Захищені процедури:

* SignalInput (сигнал про введення в потоках Т1,Т4,T6,Т8),
* SignalSort(index, val: Integer) (сигнал про завершення сортування певної частини вектора Z),
* SignalCalculate (Cигнал про обчислення A).

Захищені входи:

* WaitInput (очікування вводу)
* WaitSort*K*H\_*I* (очікування сортування частини довжиною *,* щоб почати сортування злиттям двох частин у процесі *I*)
* WaitAllSort (очікування сортування усього вектору *Z*)
* WaitCalculate (очікування обчислення *А*)

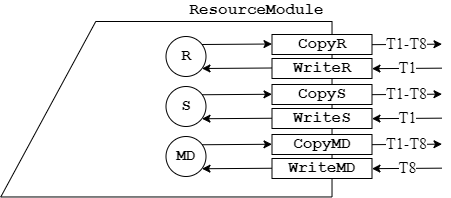


Рис. 2.2. Структурна схема монітора для вирішення задачі взаємного виключення

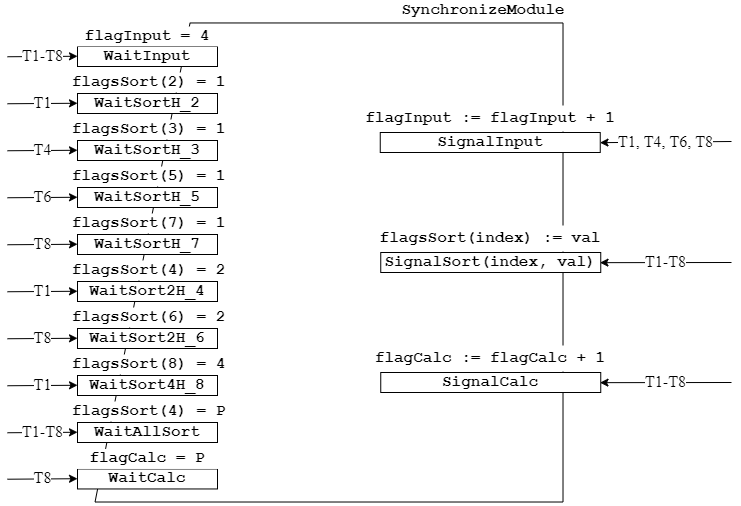


Рис. 2.3. Структурна схема монітора для вирішення задачі синхронізації

#### Розробка програми ПРГ1

Програма ПРГ1 згідно технічного завдання розроблена на мові програмування Ада. Для взаємодії процесів використовується концепція моніторів (захищених модулів).

Програма складається з модулів *Main* та *Data*, які складається з:

* *Data*
  + Визначення констант *N, P, H.*
  + Визначення типів *Vector* та *Matrix*.
  + Процедур введення/виведення вектору та введення матриці.
  + Процедури множення матриці на *Н* рядків іншої матриці.
  + Процедур сортування *Н* чисел вектору та сортування злиттям двох частин вектору.
  + Функції обчислення першого та останнього індексу для визначення частин довжиною *Н* векторів або матриць.
* *Main*
  + Специфікації та реалізації захищених модулів *ResourceModule* та *SynchronizeModule*.
  + Специфікації та реалізації задачного типу *CalculateTask*, що має дискримінант *taskNumber* – ідентифікатор (номер) процесу.
  + Визначення типу вказівника на задачу для динамічної ініціалізації задач з допомогою дискримінанта *CalculateTaskPointer*.
  + Визначення типу масиву вказівників *CalculateTaskPointer* на задачі для збереження всіх задач.

Лістинг розробленої програми наведено у додатку Г.

#### Тестування програми ПРГ1

В якості критеріїв порівняння двох реалізацій одного алгоритму обрано коефіцієнт прискорення  та коефіцієнт ефективності .

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



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



Для тестування використовувалась паралельна обчислювальна система з наступним апаратним забезпеченням:

* процесор: Intel Core i5-7200U, 4 ядра;
* оперативна пам’ять: DDR3 2133 МГц, 4 × 2048 Мб;
* Операційна система Linux 5.5.10-arch1-1;

Для виконання програми на заданій кількості ядер використовувався *Windows Task Manager*. Для вимірювання часу виконання використовувалася функція *Clock* мови *Ada*. Усі виміри часу повторювались тричі та обчислювалось середнє значення часу виконання для заданих *N* та *Р.*

Табл. 2.1. Час виконання програми

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* | *T1*,мс | *T2*,мс | *T3*,мс | *T4*,мс |
| 1000 | 7333 | 5562 | 4679 | 3613 |
| 1600 | 31277 | 18974 | 16614 | 15036 |
| 2400 | 138094 | 79088 | 69847 | 61646 |

Табл. 2.2. Коефіцієнт прискорення програми

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* |  |  |  |  |
| 1000 | 1 | 1,32 | 1,57 | 2,03 |
| 1600 | 1 | 1,65 | 1,88 | 2,08 |
| 2400 | 1 | 1,75 | 1,98 | 2,24 |

Табл. 2.3. Коефіцієнт ефективності програми

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* |  |  |  |  |
| 1000 | 100 | 66 | 52 | 50 |
| 1600 | 100 | 82 | 63 | 52 |
| 2400 | 100 | 87 | 66 | 56 |

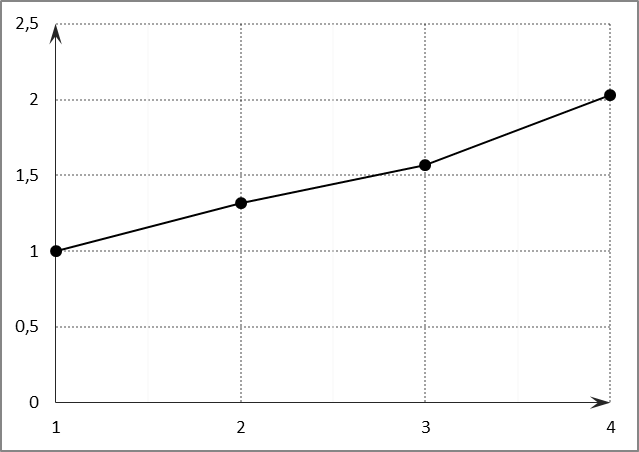


Рис. 2.4. Коефіцієнт прискорення програми при *N* = 1000

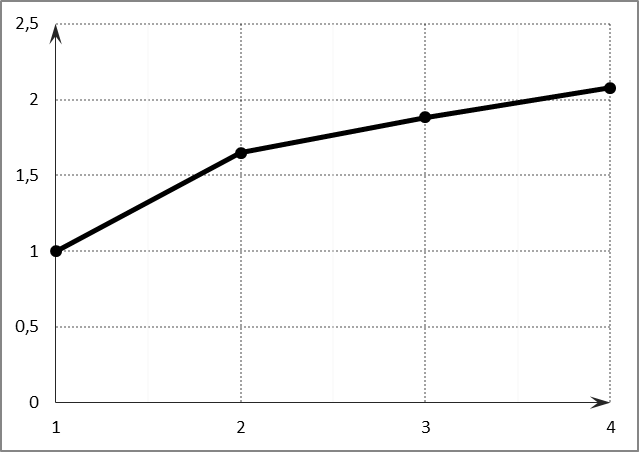


Рис. 2.5. Коефіцієнт прискорення програми при *N* = 1600

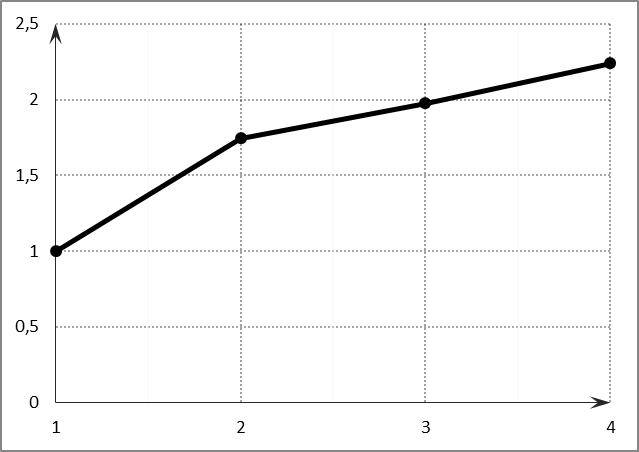


Рис. 2.6. Коефіцієнт прискорення програми при *N* = 2400

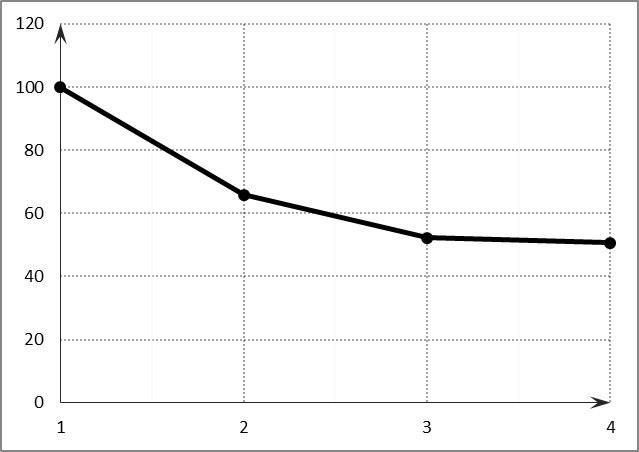


Рис. 2.7. Коефіцієнт ефективності програми при *N* = 1000

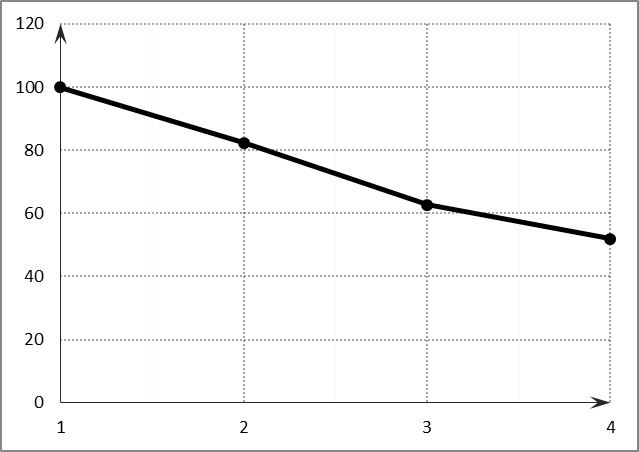


Рис. 2.8. Коефіцієнт ефективності програми при *N* = 1600



Рис. 2.9. Коефіцієнт ефективності програми при *N* = 2400

#### Висновки до розділу 2

Виконано розробку програми ПРГ1 з використанням мови Ада, зокрема захищених модулів. Тестування програми показало наступне:

* використання багатоядерної ПКС та програми ПРГ1 забезпечує скорочення часу обчислення заданої математичної задачі. Значення  лежать в межах від 1,32 до 2,24. Значення  лежать в межах від 50% до 87% ;
* максимальне значення  забезпечує ПКС з *P* = 4 при *N* = 2400;
* мінімальне значення забезпечує ПКС з *P* = 2 при *N* = 1000*;*
* максимальне значення  забезпечує ПКС з *P* = 2 при *N* = 2400;
* мінімальне значення забезпечує ПКС з *Р = 4* при *N* = 1000;

### РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП

У даному розділі розглянуто розробку та дослідження програми ПРГ2 для ПКС з ЛП.

Математична задача:

A= sort(Z)\*MX+ min(Z)\*R\*(MD\*ME)

Мова програмування: Ada (з використанням механізму рандеву).

Структура ПКС ЛП показана на рис. 3.1. Введення даних відбувається в процесах 1, 4, 6, 8. Виведення даних відбувається у процесі 8. На схемі П1-П8 – процеси.

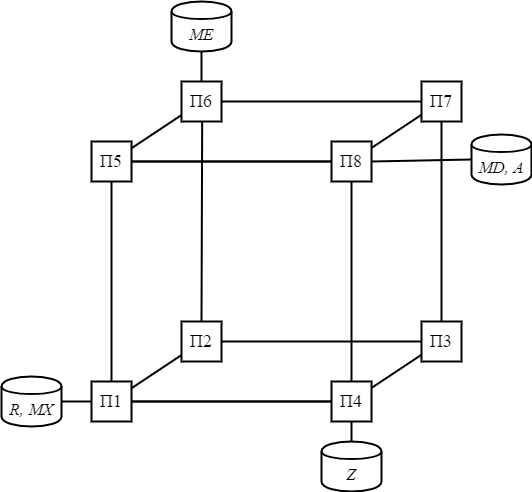


Рис. 3.1. Структура ПКС з ЛП

#### Розробка паралельного математичного алгоритму

Згідно з технічним завданням необхідно розробити паралельний алгоритм. Кількість процесорів P = 8. Його можна розділити на наступні етапи:

1. 
2. 
3. 
4. 
5. 

Спільні ресурси: 

Пояснення до використовуваних констант позначень:

* розмірність векторів і матриць;
*  кількість ядер;
* 
*  - сортування злиттям
* 
* 

Оцінка прискорення і ефективності розробленого алгоритму, згідно з теоремою Мунро-Петерсона:



Де кількість бінарних операцій, час розв’язання задачі на  вузлах.

1. Для сортування вектору *Z* потрібно  операцій порівняння. Тому час порівняння буде:



1. Для обчислення одного елементу добутку матриць  необхідно виконати *N* операцій множення та *N –* 1додавання. Тому час виконання буде:



1. Для обчислення одного елементу вектору  необхідно виконати *N* операцій множення та *N –* 1додавання. Тому час виконання буде:



1. Для обчислення одного елементу вектору  необхідно виконати 2*N* операцій множення та *N –* 1додавання. Тому час виконання буде:



1. Операція додавання виконується за 1 такт. Отже, сумарний час виконання операції:



#### Розробка алгоритмів процесів

|  |  |
| --- | --- |
|  | **Т1** |
| 1. | Ввести дані: *R, MX* |
| 2. | Отримати  від Т4 |
| 3. | Відправити  до Т2 |
| 4. |  |
| 5. | Отримати від Т2 |
| 6. |  |
| 7. | Відправити  до Т4 |
| 8. | Отримати  від Т2 та  від Т5 |
| 9. | Відправити  до Т5 |
| 10. | Відправити  до Т2 |
| 11. | Відправити  до Т4 |
| 12. | Обчислити |
| 13. | Отримати  від Т2 |
| 14. | Відправити  до Т4 |

|  |  |
| --- | --- |
|  | **Т2** |
| 1. | Отримати  від Т1 |
| 2. |  |
| 3. | Відправити  до Т1 |
| 4. | Отримати  від Т6 |
| 5. | Відправити  до Т1 |
| 6. | Отримати  від Т1 |
| 7. | Відправити до Т3 |
| 8. | Обчислити |
| 9. | Відправити  до Т1 |

|  |  |
| --- | --- |
|  | **Т3** |
| 1. | Отримати  від Т4 |
| 2. |  |
| 3. | Відправити  до Т4 |
| 4. | Отримати  від Т2 |
| 5. | Обчислити |
| 6. | Відправити  до Т4 |

|  |  |
| --- | --- |
|  | **Т4** |
| 1. | Ввести дані: *Z* |
| 2. | Відправити  до Т8,  до Т1,  до Т3 |
| 3. |  |
| 4. | Отримати від Т3 |
| 5. |  |
| 6. | Отримати  від Т1 |
| 7. |  |
| 8. | Відправити  до Т8 |
| 9. | Отримати  від Т1 |
| 10. | Обчислити |
| 11. | Отримати  від Т3 |
| 12. | Отримати  від Т1 |
| 13. | Відправити  до Т8 |

|  |  |
| --- | --- |
|  | **Т5** |
| 1. | Отримати  від Т8 |
| 2. | Відправити  до Т6 |
| 3. |  |
| 4. | Отримати від Т6 |
| 5. |  |
| 6. | Відправити  до Т8 |
| 7. | Отримати  від Т8 |
| 8. | Відправити  до Т1 |
| 9. | Отримати  від Т6 та від Т1 |
| 10. | Відправити  до Т6 |
| 11. | Відправити  до Т8 |
| 12. | Обчислити |
| 13. | Отримати  від Т6 |
| 14. | Відправити  до Т8 |

|  |  |
| --- | --- |
|  | **Т6** |
| 1. | Ввести дані: *ME* |
| 2. | Отримати  від Т5 |
| 3. |  |
| 4. | Відправити  до Т5 |
| 5. | Відправити  до Т2  до Т5 |
| 6. | Отримати  від Т5 |
| 7. | Відправити  до Т7 |
| 8. | Обчислити |
| 9. | Відправити  до Т5 |

|  |  |
| --- | --- |
|  | **Т7** |
| 1. | Отримати  від Т8 |
| 2. |  |
| 3. | Відправити  до Т8 |
| 4. | Отримати  від Т6 |
| 5. | Обчислити |
| 6. | Відправити  до Т8 |

|  |  |
| --- | --- |
|  | **Т8** |
| 1. | Ввести дані: *MD* |
| 2. | Отримати  від Т4 |
| 3. | Відправити  до Т5,  до Т7 |
| 4. |  |
| 5. | Отримати від Т7 |
| 6. |  |
| 7. | Отримати  від Т5 |
| 8. |  |
| 9. | Отримати  від Т4 |
| 10. |  |
| 11. | Відправити  до T5 |
| 12. | Отримати  від Т5 |
| 13. | Обчислити |
| 14. | Отримати  від Т7 |
| 15. | Отримати  від Т5 |
| 16. | Отримати  від Т4 |
| 17. | Вивести *А* |

#### Розробка схеми взаємодії процесів

У програмі використовуються захищені входи для передачі даних з одного потоку до іншого. Кожен з потоків має такі захищені входи:

* DataZ (на рис. 3.2 позначено як DZ) – приймає вектор *Z*
* SortResult (на рис. 3.2 позначено як SR) – надсилає вектор результату сортування *S*
* DataT*i* (на рис. 3.4 позначено як DT*i*) – приймає дані від потоку з номером *i.*
* Result (на рис. 3.4 позначено як R) – надсилає вектор результату обчислення *А*

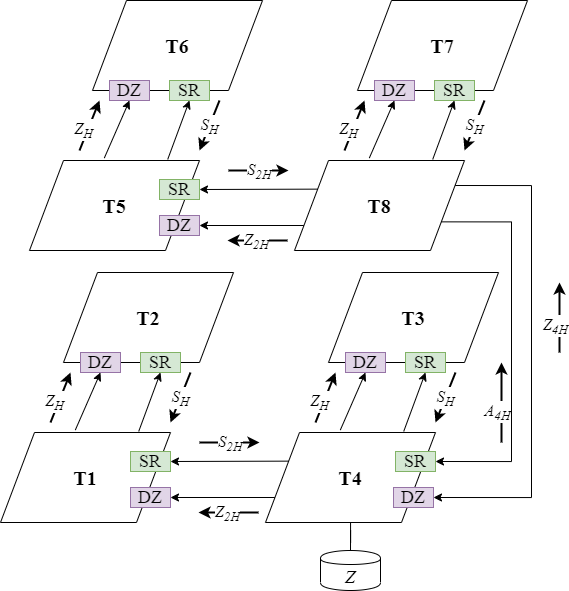


Рис. 3.2. Структурна схема взаємодії потоків під час підготовки до сортування та його виконання

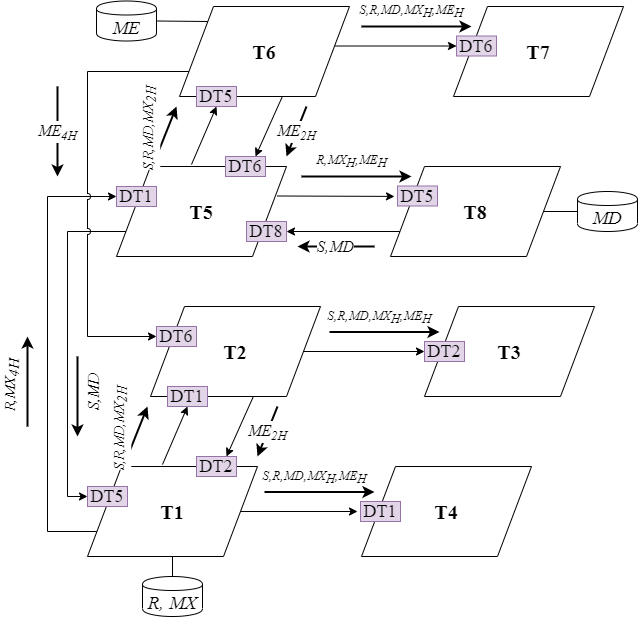


Рис. 3.3. Структурна схема взаємодії потоків під час обміну даними перед обчисленням виразу

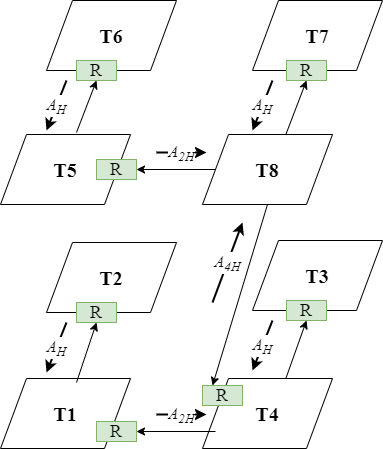


Рис. 3.4. Структурна схема взаємодії потоків під час передачі результатів після обчислення виразу

#### Розробка програми ПРГ2

Програма ПРГ2 згідно технічного завдання розроблена на мові програмування Ада. Для взаємодії процесів використовується механізм рандеву.

Програма складається з модулів *Main* та *Data*, які складається з:

* *Data*
  + Визначення констант *N, P, H.*
  + Визначення типів *Vector* та *Matrix*.
  + Процедур введення/виведення вектору та введення матриці.
  + Процедур копіювання визначеної частини вектору (або матриці) до іншого вектору (матриці)
  + Процедур сортування *Н* чисел вектору та сортування злиттям двох частин вектору.
  + Функції обчислення першого та останнього індексу для визначення частин довжиною *Н* векторів або матриць.
* *Main*
  + Опису типів потоків Т1 – Т8, в кожному з яких визначені захищені входи, які надсилають або отримують дані.

Лістинг розробленої програми наведено у додатку Д.

#### Тестування програми ПРГ2

В якості критеріїв порівняння двох реалізацій одного алгоритму обрано коефіцієнт прискорення  та коефіцієнт ефективності .

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



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



Для тестування використовувалась паралельна обчислювальна система з наступним апаратним забезпеченням:

* процесор: Intel Core i5-7200U, 4 ядра;
* оперативна пам’ять: DDR3 2133 МГц, 4 × 2048 Мб;
* Операційна система Linux 5.5.10-arch1-1;

Для виконання програми на заданій кількості ядер використовувався *Windows Task Manager*. Для вимірювання часу виконання використовувалася функція *Clock* мови *Ada*. Усі виміри часу повторювались тричі та обчислювалось середнє значення часу виконання для заданих *N* та *Р.*

Табл. 3.1 Час виконання програми

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* | *T1*,мс | *T2*,мс | *T3*,мс | *T4*,мс |
| 1000 | 8608 | 5955 | 5759 | 4474 |
| 1600 | 32405 | 18501 | 16103 | 14617 |
| 2400 | 131315 | 75872 | 69430 | 60730 |

Табл. 3.2 Коефіцієнт прискорення програми

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* |  |  |  |  |
| 1000 | 1 | 1,45 | 1,49 | 1,92 |
| 1600 | 1 | 1,75 | 2,01 | 2,22 |
| 2400 | 1 | 1,73 | 1,89 | 2,16 |

Табл. 3.3 Коефіцієнт ефективності програми

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* |  |  |  |  |
| 1000 | 100 | 72 | 50 | 48 |
| 1600 | 100 | 88 | 67 | 55 |
| 2400 | 100 | 87 | 63 | 54 |

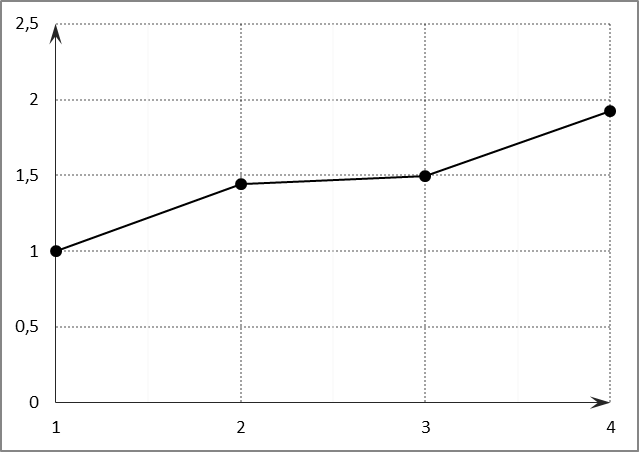


Рис. 3.5. Коефіцієнт прискорення програми при *N* = 1000

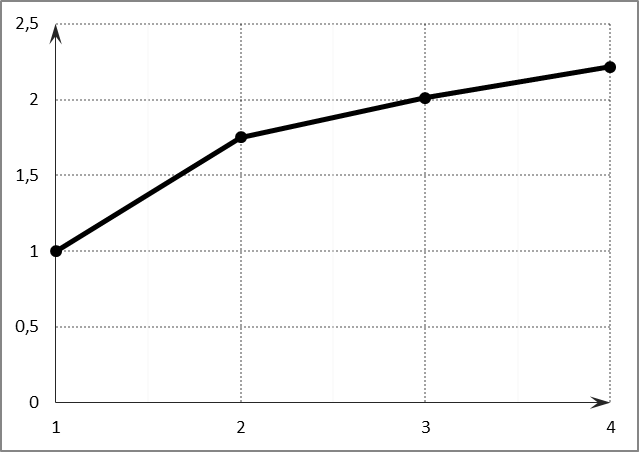


Рис. 3.6. Коефіцієнт прискорення програми при *N* = 1600

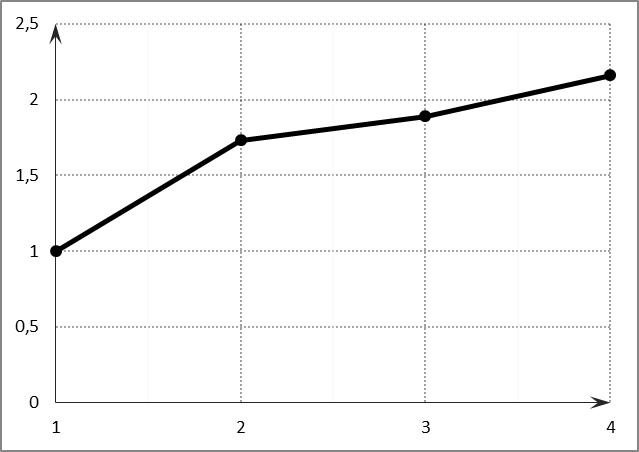


Рис. 3.7. Коефіцієнт прискорення програми при *N* = 2400

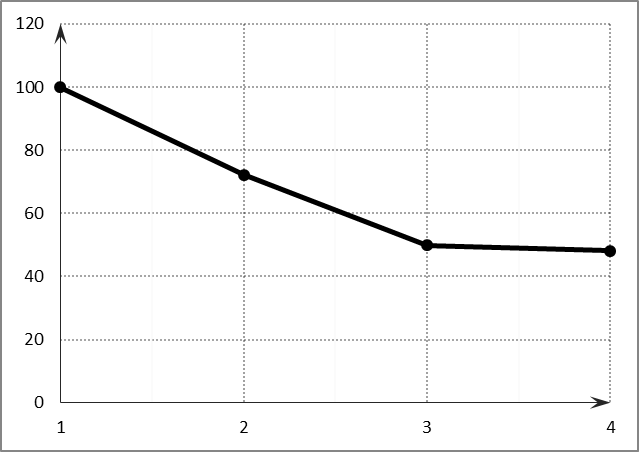


Рис. 3.8. Коефіцієнт ефективності програми при *N* = 1000

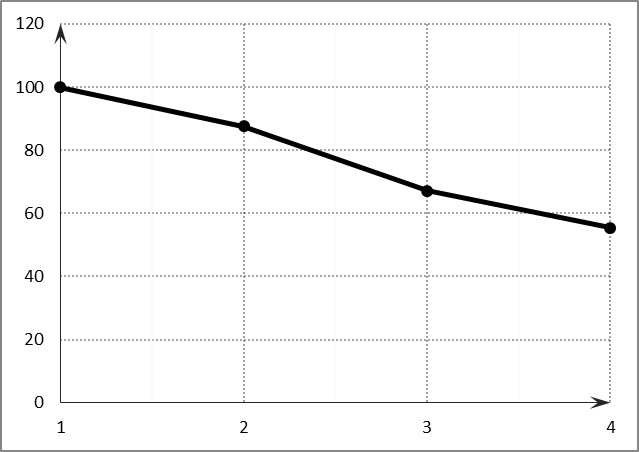


Рис. 3.9. Коефіцієнт ефективності програми при *N* = 1600

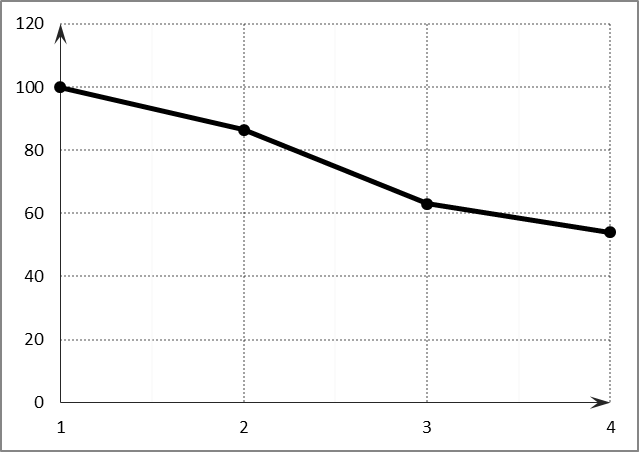


Рис. 3.10. Коефіцієнт ефективності програми при *N* = 2400

#### Висновки до розділу 3

Виконано розробку програми ПРГ2 з використанням мови Ада, зокрема захищених модулів. Тестування програми показало наступне:

* використання багатоядерної ПКС та програми ПРГ2 забезпечує скорочення часу обчислення заданої задачі. Значення  лежать в межах від 1,45 до 2,22. Значення  лежать в межах від 48% до 88% ;
* максимальне значення  забезпечує ПКС з *P* = 4 при *N* = 1600;
* мінімальне значення забезпечує ПКС з *P* = 2при *N* = 1000*;*
* максимальне значення  забезпечує ПКС з *P* = 2 при *N* = 1600;
* мінімальне значення забезпечує ПКС з *Р = 4* при *N* = 1000;

### ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ДО РОБОТИ

* У даному курсовому проекті виконано аналіз принципів роботи паралельних програм та засобів роботи з процесами у бібліотеці OpenMP.
* Також було розроблено програми на мові Ада для ПКС зі спільною та з локальною пам’яттю з використанням захищених модулів та механізма Рандеву відповдіно. Тестування обох цих програм, показало передбачувані результати, а саме, зростання прискорення виконання програми при збільшенні кількості процесорів (при незмінній розмірності векторів та матриць).
* Тестування показало, що обидві програми працюють з майже однаковою ефективністю, адже вони мають майже ідентичні графіки зміни  та , а різницю у 1-2 % у мінімальних та максимальних показниках можна вважати за похибку. Це може бути пов’язано з тим, що час, потрібний на очікування доступу до спільних ресурсів у ПРГ1 компенсується часом на пересилку повідомлень у ПРГ2.

### СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The OpenMP API specification for parallel programming [Електронне джерело]: openmp.org– Режим доступу:

<https://www.openmp.org/>

1. Ясько М.М. Навчальний посібник до вивчення курсу «Паралельні та розподілені обчислення» – РВВ ДНУ, 2010. – 76с.
2. Паралельне програмування з використанням технології OpenMp [Електронне джерело]: dn.khnu.km.ua – Режим доступу:

<http://dn.khnu.km.ua/dn/k_default.aspx?M=k1110&T=10_1&lng=1&st=0>

1. Директива OpenMP визначення паралельної області. Можливі опції [Електронне джерело]: um.co.ua – Режим доступу:

<http://um.co.ua/8/8-16/8-167503.html>

1. OpenMP | Directives [Електронне джерело]: docs.microsoft.com – Режим доступу:

<https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/2-directives?view=vs-2019>

1. ОСНОВНІ ПОНЯТТЯ І ПРИНЦИПИ РОБОТИ OPENMP [Електронне джерело]: sites.google.com/site/xdalasx13 – Режим доступу:

<https://sites.google.com/site/xdalasx13/home/p-ro_2>

### ДОДАТКИ

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Додаток А** | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Структурна схема ПКС зі СП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Додаток Б** | | | | | | | | | | | | | | |
|  |  |  |  | |  | | ІАЛЦ.467100.001 ПЗ | | | | | | | |
|  |  |  |  | |  | |
| Змн. | Арк. | № докум. | Підпис | | Дата | |
| Розроб. | | Молчанова В.С. |  | |  | | Розробка ПЗ ПКС. Алгоритм основної програми для ПКС зі СП | | Літ. | | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  | |  | |  | |  |  | 1 | 1 |
| Реценз. | |  |  | |  | | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | | |
| Н. Контр. | |  |  | |  | |
| Затверд. | | Стіренко С.Г. |  | |  | |
| **Додаток В** | | | | | | | | | | | | | | |
|  |  |  | |  | |  | | ІАЛЦ.467100.001 ПЗ | | | | | | |
|  |  |  | |  | |  | |
| Змн. | Арк. | № докум. | | Підпис | | Дата | |
| Розроб. | | Молчанова В.С. | |  | |  | | Розробка ПЗ ПКС. Алгоритм потока Т1 програми для ПКС зі СП | | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. | |  | |  | |  |  |  | 1 | 1 |
| Реценз. | |  | |  | |  | | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  | |  | |  | |
| Затверд. | | Стіренко С.Г. | |  | |  | |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т4 програми для ПКС зі СП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т6 програми для ПКС зі СП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т8 програми для ПКС зі СП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т*i* програми для ПКС зі СП, де | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Додаток Г** | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Структурна схема ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Додаток Д** | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм основної програми для ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Додаток Е** | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т1 програми для ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т2 програми для ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т*i* програми для ПКС з ЛП, де | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т4 програми для ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т5 програми для ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т6 програми для ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
|  |  |  |  |  | ІАЛЦ.467100.001 ПЗ | | | | | |
|  |  |  |  |  |
| Змн. | Арк. | № докум. | Підпис | Дата |
| Розроб. | | Молчанова В.С. |  |  | Розробка ПЗ ПКС. Алгоритм потока Т8 програми для ПКС з ЛП | Літ. | | | Арк. | Аркушів |
| Перевір. | | Корочкін О.В. |  |  |  |  |  | 1 | 1 |
| Реценз. | |  |  |  | НТУУ «КПІ», ФІОТ, ІВ-71 | | | | |
| Н. Контр. | |  |  |  |
| Затверд. | | Стіренко С.Г. |  |  |

#### Додаток Ж

**Лістинг програми ПРГ1 для ПКС з СП**

##### Файл main.adb

with Data; use Data;

with Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar, Ada.Float\_Text\_IO;

use Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar, Ada.Float\_Text\_IO;

procedure Main is

task type CalculateTask(taskNumber: Integer);

type CalculateTaskPointer is access CalculateTask;

type TasksArray is array(1..P) of CalculateTaskPointer;

type FlagArray is array (1..P) of Integer;

type indexes is range 1..8;

A, Z, R: Vector := new Vector\_t;

MX, MD, ME: Matrix := new Matrix\_t;

tArray: TasksArray;

ch : Character;

Res: Matrix := new Matrix\_t;

StartTime, FinishTime: Time;

DiffTime: Duration;

-----------------------------------------------------------

--------------DESCRIPTION OF PROTECTED MODULES-------------

-----------------------------------------------------------

protected ResourceModule is

procedure WriteS(V: in Vector);

procedure WriteR(V: in Vector);

procedure WriteMD(M: in Matrix);

function CopyS return Vector;

function CopyR return Vector;

function CopyMD return Matrix;

private

S: Vector;

R: Vector;

MD: Matrix;

end ResourceModule;

protected SynchronizeModule is

procedure SignalInput;

procedure SignalSort(index, val: Integer);

procedure SignalCalculate;

entry WaitInput;

entry WaitSortH\_2;

entry WaitSortH\_3;

entry WaitSortH\_5;

entry WaitSortH\_7;

entry WaitSort2H\_4;

entry WaitSort2H\_6;

entry WaitSort4H\_8;

entry WaitAllSort;

entry WaitCalculate;

private

flagInput: Integer := 0;

flagSort: FlagArray := (0, 0, 0, 0, 0, 0, 0,0);

flagCalculate: Integer := 0;

end SynchronizeModule;

-----------------------------------------------------------

----------------BODIES OF PROTECTED MODULES----------------

-----------------------------------------------------------

protected body ResourceModule is

protected body SynchronizeModule is

procedure SignalInput is

begin

flagInput := flagInput + 1;

end SignalInput;

procedure SignalSort(index, val: Integer) is

begin

flagSort(index) := val;

end SignalSort;

procedure SignalCalculate is

begin

flagCalculate := flagCalculate + 1;

end SignalCalculate;

entry WaitInput

when flagInput = 4 is

begin

null;

end WaitInput;

entry WaitSortH\_2

when flagSort(2) = 1 is

begin

null;

end WaitSortH\_2;

entry WaitSortH\_3

when flagSort(3) = 1 is

begin

null;

end WaitSortH\_3;

entry WaitSortH\_5

when flagSort(5) = 1 is

begin

null;

end WaitSortH\_5;

entry WaitSortH\_7

when flagSort(7) = 1 is

begin

null;

end WaitSortH\_7;

entry WaitSort2H\_4

when flagSort(4) = 2 is

begin

null;

end WaitSort2H\_4;

entry WaitSort2H\_6

when flagSort(6) = 2 is

begin

null;

end WaitSort2H\_6;

entry WaitSort4H\_8

when flagSort(8) = 4 is

begin

null;

end WaitSort4H\_8;

entry WaitAllSort

when flagSort(1) = 8 is

begin

null;

end WaitAllSort;

entry WaitCalculate

when flagCalculate = P is

begin

null;

end WaitCalculate;

end SynchronizeModule;

task body CalculateTask is

first, last: Integer;

Si: Vector;

Ri: Vector;

MDi: Matrix;

sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

--Input

if taskNumber = 1 then

InputVector(R);

InputMatrix(MX);

ResourceModule.WriteR(R);

SynchronizeModule.SignalInput;

elsif taskNumber = 4 then

InputVector(Z);

SynchronizeModule.SignalInput;

elsif taskNumber = 6 then

InputMatrix(ME);

SynchronizeModule.SignalInput;

elsif taskNumber = 8 then

InputMatrix(MD);

ResourceModule.WriteMD(MD);

SynchronizeModule.SignalInput;

end if;

SynchronizeModule.WaitInput;

--Sort

SortH(Z, first, last);

SynchronizeModule.SignalSort(taskNumber, 1);

if (taskNumber = 1) then

SynchronizeModule.WaitSortH\_2;

MergeSort(Z, first, GetLast(2));

SynchronizeModule.WaitSort2H\_4;

MergeSort(Z, first, GetLast(4));

SynchronizeModule.WaitSort4H\_8;

MergeSort(Z, 1, N);

ResourceModule.WriteS(Z);

SynchronizeModule.SignalSort(1, 8);

elsif (taskNumber = 4) then

SynchronizeModule.WaitSortH\_3;

MergeSort(Z, GetFirst(3), last);

SynchronizeModule.SignalSort(4, 2);

elsif (taskNumber = 6) then

SynchronizeModule.WaitSortH\_5;

MergeSort(Z, GetFirst(5), last);

SynchronizeModule.SignalSort(6, 2);

elsif (taskNumber = 8) then

SynchronizeModule.WaitSortH\_7;

MergeSort(Z, GetFirst(7), last);

SynchronizeModule.WaitSort2H\_6;

MergeSort(Z, GetFirst(5), last);

SynchronizeModule.SignalSort(8, 4);

end if;

SynchronizeModule.WaitAllSort;

Si := ResourceModule.CopyS;

Ri := ResourceModule.CopyR;

MDi := ResourceModule.CopyMD;

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MDi(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + Si(j)\*MX(j)(i) + Si(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

if taskNumber = 8 then

SynchronizeModule.SignalCalculate;

SynchronizeModule.WaitCalculate;

if N <= 8 then

OutputVector(A);

end if;

FinishTime := Clock;

DiffTime := FinishTime - StartTime;

Put("Time = ");

Put(Float(DiffTime), 1);

Put\_Line("");

else

SynchronizeModule.SignalCalculate;

end if;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end CalculateTask;

begin

Get(ch);

StartTime := Clock;

for i in 1..P loop

tArray(i) := new CalculateTask(i);

end loop;

end Main;

##### Файл data.ads

package Data is

N: Integer := 1000;

P: Integer := 8;

H: Integer := N / P;

type Vector\_t is array(1..N) of Integer;

type Vector is access Vector\_t;

type Matrix\_t is array(1..N) of Vector;

type Matrix is access Matrix\_t;

procedure InputVector(V: out Vector);

procedure InputMatrix(M: out Matrix);

procedure OutputVector(V : in Vector);

procedure MultiplyMatrixH(M, MH: Matrix; first, last: Integer; Res: out Matrix);

procedure SortH(V : in out Vector; first, last: Integer);

procedure MergeSort(V: in out Vector; first, last: Integer);

function GetFirst(taskNumber: Integer) return Integer;

function GetLast(taskNumber: Integer) return Integer;

end Data;

##### Файл data.adb

with Ada.INTEGER\_TEXT\_IO; use Ada.INTEGER\_TEXT\_IO;

with Ada.Text\_IO; use Ada.Text\_IO;

package body Data is

procedure InputVector(V: out Vector) is

begin

for i in 1 .. N loop

V(i) := 1;

end loop;

end InputVector;

procedure InputMatrix(M: out Matrix) is

begin

for i in 1 .. N loop

M(i) := new Vector\_t;

for j in 1 .. N loop

M(i)(j) := 1;

end loop;

end loop;

end InputMatrix;

procedure OutputVector(V : in Vector) is

begin

for i in 1..N loop

Ada.Integer\_Text\_IO.Put ( V(i) );

end loop;

Put\_Line("");

end OutputVector;

procedure MultiplyMatrixH(M, MH: Matrix; first, last: Integer; Res: out Matrix) is

sum: Integer;

begin

for i in 1..N loop

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + M(i)(k)\*MH(k)(j);

end loop;

Res(j)(i) := sum;

end loop;

end loop;

end MultiplyMatrixH;

procedure SortH(V : in out Vector; first, last: Integer) is

buf: Integer;

begin

for i in first..last loop

for j in i..last loop

if V(i) > V(j) then

buf := V(i);

V(i) := V(j);

V(j) := buf;

end if;

end loop;

end loop;

end SortH;

procedure MergeSort(V: in out Vector; first, last: Integer) is

res: Vector := new Vector\_t;

lEnd: Integer := last - (last-first+1)/2;

i1: Integer := first;

i2: Integer := lEnd+1;

i: integer := 1;

begin

while (i1 <= lEnd or i2 <= last) loop

if V(i1) < V(i2) then

res(i) := V(i1);

i1 := i1 + 1;

if (i1 > lend) then

while (i2 <= last) loop

i := i+1;

res(i) := V(i2);

i2 := i2 + 1;

end loop;

end if;

else

res(i) := V(i2);

i2 := i2 + 1;

if (i2> last) then

while (i1 <= lEnd) loop

i := i+1;

res(i) := V(i1);

i1 := i1 + 1;

end loop;

end if;

end if;

i := i+1;

end loop;

--Write result of sort to the input vector

i1 := 1;

for j in first..last loop

V(j) := res(i1);

i1 := i1+1;

end loop;

end MergeSort;

function GetFirst(taskNumber: Integer) return Integer is

begin

return (taskNumber - 1) \* H + 1;

end GetFirst;

function GetLast(taskNumber: Integer) return Integer is

begin

return taskNumber \* H;

end GetLast;

end Data;

#### Додаток З

**Лістинг програми ПРГ2 для ПКС з ЛП**

##### Файл main.adb

with Data; use Data;

with Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar, Ada.Float\_Text\_IO;

use Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar, Ada.Float\_Text\_IO;

procedure Main is

StartTime: Time;

ch : Character;

procedure RunTasks is

-----------------------------------------------------------

--------------------DESCRIPTION OF TASKS-------------------

-----------------------------------------------------------

--Note: Parameter name Z\_4H\_2\_3\_4 in task's entry means----

--------that it contains 4 parts of length H:--------------

--------for task itself and for tasks 2, 3 and 4-----------

-----------------------------------------------------------

task T1 is

entry DataZ(Z\_2H\_2: in Vector);

entry SortResult(S\_2H\_2: out Vector);

entry DataT2(ME\_2H\_4: in Matrix);

entry DataT5(S\_8H: in Vector; MD\_8H: in Matrix);

entry Result(A\_2H\_2: out Vector);

end T1;

task T2 is

entry DataZ(Z\_H: in Vector);

entry SortResult(S\_H: out Vector);

entry DataT6(ME\_4H\_1\_3\_4: in Matrix);

entry DataT1(S\_8H, R\_8H: in Vector; MD\_8H, MX\_2H\_3: in Matrix);

entry Result(A\_H: out Vector);

end T2;

task T3 is

entry DataZ(Z\_H: in Vector);

entry SortResult(S\_H: out Vector);

entry DataT2(S\_8H, R\_8H: in Vector; MD\_8H, MX\_H, ME\_H: in Matrix);

entry Result(A\_H: out Vector);

end T3;

task T4 is

entry SortResult(S\_4H\_1\_2\_3: out Vector);

entry DataT1(S\_8H, R\_8H: in Vector; MD\_8H, MX\_H, ME\_H: in Matrix);

entry Result(A\_4H\_1\_2\_3: out Vector);

end T4;

task T5 is

entry DataZ(Z\_2H\_6: in Vector);

entry SortResult(S\_2H\_6: out Vector);

entry DataT6(ME\_2H\_8: in Matrix);

entry DataT8(S\_8H: in Vector; MD\_8H: in Matrix);

entry DataT1(R\_8H: in Vector; MX\_4H\_6\_7\_8: in Matrix);

entry Result(A\_2H\_6: out Vector);

end T5;

task T6 is

entry DataZ(Z\_H: in Vector);

entry SortResult(S\_H: out Vector);

entry DataT5(S\_8H, R\_8H: in Vector; MD\_8H, MX\_2H\_7: in Matrix);

entry Result(A\_H: out Vector);

end T6;

task T7 is

entry DataZ(Z\_H: in Vector);

entry SortResult(S\_H: out Vector);

entry DataT6(S\_8H, R\_8H: in Vector; MD\_8H, MX\_H, ME\_H: in Matrix);

entry Result(A\_H: out Vector);

end T7;

task T8 is

entry DataZ(Z\_4H\_5\_6\_7: in Vector);

entry DataT5(R\_8H: in Vector; MX\_H, ME\_H: in Matrix);

end T8;

task body T1 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 1;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

InputVector(R);

InputMatrix(MX);

accept DataZ (Z\_2H\_2 : in Vector) do

CopyVectorPart(Z\_2H\_2, first, GetLast(2), Z);

end DataZ;

T2.DataZ(Z\_H => Z);

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

T2.SortResult(S\_H => S);

MergeSort(S, first, GetLast(2));

accept SortResult (S\_2H\_2 : out Vector) do

CopyVectorPart(S, first, GetLast(2), S\_2H\_2);

end SortResult;

for t in 1..2 loop

select

accept DataT5 (S\_8H : in Vector; MD\_8H : in Matrix) do

CopyVectorPart(S\_8H, 1, N, S);

CopyMatrixPart(MD\_8H, 1, N, MD);

end DataT5;

or

accept DataT2 (ME\_2H\_4 : in Matrix) do

CopyMatrixPart(ME\_2H\_4, first, last, ME);

CopyMatrixPart(ME\_2H\_4, GetFirst(4), GetLast(4), ME);

end DataT2;

end select;

end loop;

T5.DataT1(R, MX);

T2.DataT1(S\_8H => S,

R\_8H => R,

MD\_8H => MD,

MX\_2H\_3 => MX);

T4.DataT1(S\_8H => S,

R\_8H => R,

MD\_8H => MD,

MX\_H => MX,

ME\_H => ME);

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

T2.Result(A\_H => A);

accept Result (A\_2H\_2 : out Vector) do

CopyVectorPart(A, first, GetLast(2), A\_2H\_2);

end Result;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T1;

task body T2 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 2;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

accept DataZ(Z\_H : in Vector) do

CopyVectorPart(Z\_H, first, last, Z);

end DataZ;

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

accept SortResult (S\_H : out Vector) do

CopyVectorPart(S, first, last, S\_H);

end SortResult;

accept DataT6 (ME\_4H\_1\_3\_4 : in Matrix) do

CopyMatrixPart(ME\_4H\_1\_3\_4, GetFirst(1), GetLast(4), ME);

end DataT6;

T1.DataT2(ME\_2H\_4 => ME);

accept DataT1 (S\_8H : in Vector; R\_8H : in Vector; MD\_8H : in Matrix; MX\_2H\_3 : in Matrix) do

CopyVectorPart(S\_8H, 1, N, S);

CopyVectorPart(R\_8H, 1, N, R);

CopyMatrixPart(MD\_8H, 1, N, MD);

CopyMatrixPart(MX\_2H\_3, first, GetLast(3), MX);

end DataT1;

T3.DataT2(S\_8H => S,

R\_8H => R,

MD\_8H => MD,

MX\_H => MX,

ME\_H => ME);

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

accept Result (A\_H : out Vector) do

CopyVectorPart(A, first, last, A\_H);

end Result;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T2;

task body T3 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 3;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

accept DataZ(Z\_H : in Vector) do

CopyVectorPart(Z\_H, first, last, Z);

end DataZ;

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

accept SortResult (S\_H : out Vector) do

CopyVectorPart(S, first, last, S\_H);

end SortResult;

accept DataT2 (S\_8H : in Vector; R\_8H : in Vector; MD\_8H : in Matrix; MX\_H : in Matrix; ME\_H : in Matrix) do

CopyVectorPart(S\_8H, 1, N, S);

CopyVectorPart(R\_8H, 1, N, R);

CopyMatrixPart(MD\_8H, 1, N, MD);

CopyMatrixPart(MX\_H, first, last, MX);

CopyMatrixPart(ME\_H, first, last, ME);

end DataT2;

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

accept Result (A\_H : out Vector) do

CopyVectorPart(A, first, last, A\_H);

end Result;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T3;

task body T4 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 4;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

InputVector(Z);

T8.DataZ(Z\_4H\_5\_6\_7 => Z);

T1.DataZ(Z\_2H\_2 => Z);

T3.DataZ(Z\_H => Z);

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

T3.SortResult(S\_H => S);

MergeSort(S, GetFirst(3), last);

T1.SortResult(S\_2H\_2 => S);

MergeSort(S, GetFirst(1), last);

accept SortResult (S\_4H\_1\_2\_3 : out Vector) do

CopyVectorPart(S, GetFirst(1), last, S\_4H\_1\_2\_3);

end SortResult;

accept DataT1 (S\_8H : in Vector; R\_8H : in Vector; MD\_8H : in Matrix; MX\_H : in Matrix; ME\_H : in Matrix) do

CopyVectorPart(S\_8H, 1, N, S);

CopyVectorPart(R\_8H, 1, N, R);

CopyMatrixPart(MD\_8H, 1, N, MD);

CopyMatrixPart(MX\_H, first, last, MX);

CopyMatrixPart(ME\_H, first, last, ME);

end DataT1;

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

T3.Result(A\_H => A);

T1.Result(A\_2H\_2 => A);

accept Result (A\_4H\_1\_2\_3 : out Vector) do

CopyVectorPart(A, GetFirst(1), last, A\_4H\_1\_2\_3);

end Result;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T4;

task body T5 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 5;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

InputVector(R);

InputMatrix(MX);

accept DataZ (Z\_2H\_6 : in Vector) do

CopyVectorPart(Z\_2H\_6, first, GetLast(6), Z);

end DataZ;

T6.DataZ(Z\_H => Z);

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

T6.SortResult(S\_H => S);

MergeSort(S, first, GetLast(6));

accept SortResult (S\_2H\_6 : out Vector) do

CopyVectorPart(S, first, GetLast(6), S\_2H\_6);

end SortResult;

accept DataT8 (S\_8H : in Vector; MD\_8H : in Matrix) do

CopyVectorPart(S\_8H, 1, N, S);

CopyMatrixPart(MD\_8H, 1, N, MD);

end DataT8;

T1.DataT5(S\_8H => S,

MD\_8H => MD);

for t in 1..2 loop

select

accept DataT6 (ME\_2H\_8 : in Matrix) do

CopyMatrixPart(ME\_2H\_8, first, last, ME);

CopyMatrixPart(ME\_2H\_8, GetFirst(8), GetLast(8), ME);

end DataT6;

or

accept DataT1 (R\_8H : in Vector; MX\_4H\_6\_7\_8 : in Matrix) do

CopyVectorPart(R\_8H, 1, N, R);

CopyMatrixPart(MX\_4H\_6\_7\_8, first, GetLast(8), MX);

end DataT1;

end select;

end loop;

T6.DataT5(S\_8H => S,

R\_8H => R,

MD\_8H => MD,

MX\_2H\_7 => MX);

T8.DataT5(R\_8H => R,

MX\_H => MX,

ME\_H => ME);

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

T6.Result(A\_H => A);

accept Result (A\_2H\_6 : out Vector) do

CopyVectorPart(A, first, GetLast(6), A\_2H\_6);

end Result;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T5;

task body T6 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 6;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

InputMatrix(ME);

accept DataZ(Z\_H : in Vector) do

CopyVectorPart(Z\_H, first, last, Z);

end DataZ;

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

accept SortResult (S\_H : out Vector) do

CopyVectorPart(S, first, last, S\_H);

end SortResult;

T2.DataT6(ME\_4H\_1\_3\_4 => ME);

T5.DataT6(ME\_2H\_8 => ME);

accept DataT5 (S\_8H : in Vector; R\_8H : in Vector; MD\_8H : in Matrix; MX\_2H\_7 : in Matrix) do

CopyVectorPart(S\_8H, 1, N, S);

CopyVectorPart(R\_8H, 1, N, R);

CopyMatrixPart(MD\_8H, 1, N, MD);

CopyMatrixPart(MX\_2H\_7, first, GetLast(7), MX);

end DataT5;

T7.DataT6(S\_8H => S,

R\_8H => R,

MD\_8H => MD,

MX\_H => MX,

ME\_H => ME);

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

accept Result (A\_H : out Vector) do

CopyVectorPart(A, first, last, A\_H);

end Result;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T6;

task body T7 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 7;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

accept DataZ(Z\_H : in Vector) do

CopyVectorPart(Z\_H, first, last, Z);

end DataZ;

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

accept SortResult (S\_H : out Vector) do

CopyVectorPart(S, first, last, S\_H);

end SortResult;

accept DataT6 (S\_8H : in Vector; R\_8H : in Vector; MD\_8H : in Matrix; MX\_H : in Matrix; ME\_H : in Matrix) do

CopyVectorPart(S\_8H, 1, N, S);

CopyVectorPart(R\_8H, 1, N, R);

CopyMatrixPart(MD\_8H, 1, N, MD);

CopyMatrixPart(MX\_H, first, last, MX);

CopyMatrixPart(ME\_H, first, last, ME);

end DataT6;

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

accept Result (A\_H : out Vector) do

CopyVectorPart(A, first, last, A\_H);

end Result;

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T7;

task body T8 is

Z, S, R, A: Vector := new Vector\_t;

MD, MX, ME, Res: Matrix := new Matrix\_t;

taskNumber: Integer := 8;

first, last, sum: Integer;

begin

Put\_Line("Task " & Integer'Image(taskNumber) & " started...");

first := GetFirst(taskNumber);

last := GetLast(taskNumber);

InputVector(S);

InputMatrix(MD);

accept DataZ (Z\_4H\_5\_6\_7 : in Vector) do

CopyVectorPart(Z\_4H\_5\_6\_7, GetFirst(5), last, Z);

end DataZ;

T5.DataZ(Z\_2H\_6 => Z);

T7.DataZ(Z\_H => Z);

SortH(Z, first, last);

CopyVectorPart(Z, first, last, S);

T7.SortResult(S\_H => S);

MergeSort(S, GetFirst(7), last);

T5.SortResult(S\_2H\_6 => S);

MergeSort(S, GetFirst(5), last);

T4.SortResult(S\_4H\_1\_2\_3 => S);

MergeSort(S, GetFirst(1), last);

T5.DataT8(S\_8H => S,

MD\_8H => MD);

accept DataT5 (R\_8H : in Vector; MX\_H : in Matrix; ME\_H : in Matrix) do

CopyVectorPart(R\_8H, 1, N, R);

CopyMatrixPart(MX\_H, first, last, MX);

CopyMatrixPart(ME\_H, first, last, ME);

end DataT5;

for i in 1..N loop

if Res(i) = null then

Res(i) := new Vector\_t;

end if;

for j in first..last loop

sum := 0;

for k in 1..N loop

sum := sum + MD(i)(k)\*ME(k)(j);

end loop;

Res(i)(j) := sum;

end loop;

end loop;

for i in first..last loop

A(i) := 0;

for j in 1..N loop

A(i) := A(i) + S(j)\*MX(j)(i) + S(1)\*R(j)\*Res(j)(i);

end loop;

end loop;

T7.Result(A\_H => A);

T5.Result(A\_2H\_6 => A);

T4.Result(A\_4H\_1\_2\_3 => A);

if N <= 8 then

OutputVector(A);

end if;

Put("Time = ");

Put(Float(Clock - StartTime), 1);

Put\_Line("");

Put\_Line("Task " & Integer'Image(taskNumber) & " finished");

end T8;

begin

null;

end RunTasks;

begin

Get(ch);

StartTime := Clock;

RunTasks;

end Main;

##### Файл data.ads

package Data is

N: Integer := 1000;

P: Integer := 8;

H: Integer := N / P;

type Vector\_t is array(1..N) of Integer;

type Vector is access Vector\_t;

type Matrix\_t is array(1..N) of Vector;

type Matrix is access Matrix\_t;

procedure InputVector(V: out Vector);

procedure InputMatrix(M: out Matrix);

procedure OutputVector(V : in Vector);

procedure CopyVectorPart(source: in Vector; first, last: Integer; result: out Vector);

procedure CopyMatrixPart(source: in MAtrix; first, last: Integer; result: out Matrix);

procedure SortH(V : in out Vector; first, last: Integer);

procedure MergeSort(V: in out Vector; first, last: Integer);

function GetFirst(taskNumber: Integer) return Integer;

function GetLast(taskNumber: Integer) return Integer;

end Data;

##### Файл data.adb

with Ada.INTEGER\_TEXT\_IO; use Ada.INTEGER\_TEXT\_IO;

with Ada.Text\_IO; use Ada.Text\_IO;

package body Data is

procedure InputVector(V: out Vector) is

begin

for i in 1 .. N loop

V(i) := 1;

end loop;

end InputVector;

procedure InputMatrix(M: out Matrix) is

begin

for i in 1 .. N loop

M(i) := new Vector\_t;

for j in 1 .. N loop

M(i)(j) := 1;

end loop;

end loop;

end InputMatrix;

procedure OutputVector(V : in Vector) is

begin

for i in 1..N loop

Ada.Integer\_Text\_IO.Put ( V(i) );

end loop;

Put\_Line("");

end OutputVector;

procedure CopyVectorPart(source: in Vector; first, last: Integer; result: out Vector) is

begin

if result = null then

result := new Vector\_t;

end if;

for i in first..last loop

result(i) := source(i);

end loop;

end;

procedure CopyMatrixPart(source: in Matrix; first, last: Integer; result: out Matrix) is

begin

if result = null then

result := new Matrix\_t;

end if;

for i in 1..N loop

CopyVectorPart(source(i), first, last, result(i));

end loop;

end;

procedure SortH(V : in out Vector; first, last: Integer) is

buf: Integer;

begin

for i in first..last loop

for j in i..last loop

if V(i) > V(j) then

buf := V(i);

V(i) := V(j);

V(j) := buf;

end if;

end loop;

end loop;

end SortH;

procedure MergeSort(V: in out Vector; first, last: Integer) is

res: Vector := new Vector\_t;

lEnd: Integer := last - (last-first+1)/2;

i1: Integer := first;

i2: Integer := lEnd+1;

i: integer := 1;

begin

while (i1 <= lEnd or i2 <= last) loop

if V(i1) < V(i2) then

res(i) := V(i1);

i1 := i1 + 1;

if (i1 > lend) then

while (i2 <= last) loop

i := i+1;

res(i) := V(i2);

i2 := i2 + 1;

end loop;

end if;

else

res(i) := V(i2);

i2 := i2 + 1;

if (i2> last) then

while (i1 <= lEnd) loop

i := i+1;

res(i) := V(i1);

i1 := i1 + 1;

end loop;

end if;

end if;

i := i+1;

end loop;

--Write result of sort to the input vector

i1 := 1;

for j in first..last loop

V(j) := res(i1);

i1 := i1+1;

end loop;

end MergeSort;

function GetFirst(taskNumber: Integer) return Integer is

begin

return (taskNumber - 1) \* H + 1;

end GetFirst;

function GetLast(taskNumber: Integer) return Integer is

begin

return taskNumber \* H;

end GetLast;

end Data;