УДК 681.3.06 DOI: 10.18372/2073-4751.70.16847

Корочкін О.В., к.т.н.,
orcid.org/0000-0002-6569-5849,
Русанова О.В., к.т.н.,
orcid.org/0000-0003-0145-3012,
Демчик В.І.

ЕФЕКТИВНІСТЬ ДРІБНОЗЕРНИСТОГО ПАРАЛЕЛІЗМУ В СУЧАСНИХ КОМП'ЮТЕРНИХ СИСТЕМАХ

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

avcora@gmail.com

Вступ

Програмування для багатоядерних комп'ютерних систем (БКС) пов'язано з використанням концепції потоків, в якій програма представляється як набір паралельних легких процесів. Як правило, потоки реалізують обчислення в рамках середнього або крупнозернистого паралелізму. При цьому виникає складна проблема організації взаємодії потоків, пов'язана з вирішенням задач взаємного виключення і синхронізації, для вирішення яких в БКС використовуються спеціальні конструкції: семафори, атомік змінні, м'ютекси, події, критичні секції, монітори.

Зі зростанням кількості ядер в БКС та відповідно кількості потоків в програмі з'являється можливість зниження розміру зернистості паралелізму, на основі якого можна знизити складність задачі взаємодії потоків, спростити (автоматизувати) процес її вирішення і як наслідок – зменшити час виконання програми.

Мета

Метою даної роботі є дослідження можливості покращення часу виконання паралельних програм в БКС за оптимального рахунок оптимального поєднання середнє та дрібнозернистого паралелізму, а також застосування сучасних засобів їх реалізації.

Основна частина

При виборі підходу до проектування програмного забезпечення для БКС одним із перших вирішується питання гранулярності. Гранулярність - це міра відношення обсягу обчислень, виконаних в паралельній задачі, до обсягу комунікацій (для обміну повідомленнями). Ступінь гранулярності варіюється від дрібнозернистої до крупнозернистої.

Крупнозернистий паралелізм (coarse grained): кожне паралельне обчислення досить незалежне від інших, причому потрібен відносно рідкісний обмін інформацією між окремими обчисленнями. Одиницями розпаралелювання є великі частини програми, що включають тисячі команд. Цей рівень паралелізму відрізняється відносною простотою реалізації та забезпечується операційною системою.

Дрібнозернистий паралелізм (fine grained): кожне паралельне обчислення досить мале і елементарне, складається з десятків команд. Зазвичай розпаралеленими одиницями є елементи виразів або окремі ітерації циклу, що мають невеликі залежності між даними. Сам термін «дрібнозернистий паралелізм» говорить про простоту і швидкість будь-якої обчислювальні дії. Характерна особливість дрібнозернистого паралелізму полягає в приблизному рівність інтенсивності обчислень і обміну даними. Цей рівень паралелізму часто використовується розпаралелюючим (векторизуючим) компілятором.

Ефективне паралельне виконання вимагає майстерного балансу між ступенем гранулярності програм і величиною комунікаційної затримки, що виникає між різними гранулами. Зокрема, якщо комунікаційна затримка мінімальна, то найкращу продуктивність обіцяє дрібнозернисте розбиття програми. Це той випадок, коли діє

паралелізм даних. Якщо комунікаційна затримка велика, краще крупнозернисте розбиття програм [1].

Сучасні мови та бібліотеки паралельного програмування наряду з засобами створення потоків та організації їх взаємодії містять інструменти і для підтримки дрібнозернистого паралелізму.

Засоби реалізації дрібнозернистого паралелізму реалізовано в бібліотеці OpenMP та мовах Java і C# [1-2].

Бібліотека OpenMP. Дрібнозернистий паралелізм представлений спеціальною директивою препроцесору #pragma omp parallel for, яка відноситься до директив розділення роботи (work-sharing directive). Такі директиви застосовуються не для паралельного виконання коду, а для логічного розподілу групи потоків, щоб реалізувати вказані конструкції керуючої логіки. Директива #pragma omp for повідомляє, що при виконанні циклу в паралельному режимі ітерації циклу повинні бути розподілені між групою потоків. Виконання наступного програмного коду в чотирипроцесорній системі відбувалося би так, як показано на рис. 1:

```
int size=100;
  #pragma omp parallel for
   for(int i = 0; i < size; ++i)
    Calculations();
  ShowResults();
```

Такий розподіл використовується за замовчуванням та називається статичним плануванням паралелізму (static scheduling).

В OpenMP передбачені і інші, більш гнучкі види планування, такі як динамічне планування (dynamic scheduling), планування в період виконання (runtime scheduling) і кероване планування (guided scheduling). Для задання одного з цих режимів використовується спеціальний розділ, формат якого виглядає наступним чином: schedule(алгоритм планування[,число ітерацій]). Використовується даний механізм у випадку коли різні ітерації виконують різні обсяги роботи, та визначає чи матиме змогу потік, який вже завершив свою ітерацію, прийняти на себе частину роботи іншого потоку.

При написанні більш складних паралельних програм однією з ключових задач стає вирішення задачі синхронізації потоків. В OpenMP реалізована неявна бар'єрна синхронізація в кінці кожного блоку #pragma omp parallel for. Ключовою задачею, яка потребує вирішення при написанні паралельної програми, є задача взаємного виключення, яка дозволяє уникнути ситуації, коли велика кількість потоків буде блокуватися внаслідок звернення до однієї і тієї ж області пам'яті, що містить наприклад деяку змінну (спільний ресурс), яка використовується в усіх потоках. При великій кількості потоків, які створюються під час застосування дрібнозернистого паралелізму, ця проблема є особливо гострою.

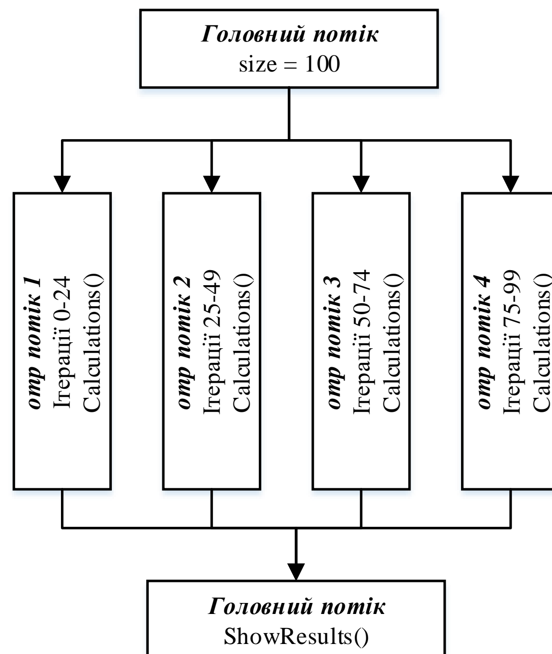

Рис. 1. Розподіл ітерацій циклу по паралельних потоках засобами OpenMP

В OpenMP передбачена можливість застосування спеціальних неблокуючих засобів вирішення завдання взаємного виключення. Це прагма #pragma omp atomic, а також модифікатори прагми #pragma omp parallel for: shared, private, firstprivate, lastprivate, reduction.

Мова Java. Розробникам пропонується надзвичайно гнучкі та потужні

засоби реалізації дрібнозернистого паралелізму, в основу яких покладено так механізм «Fork-Join».

За фактичну реалізацію цього механізму в Java обрано рекурсивний алгоритм, якого полягає в наступному.

Здійснюється перевірка можливості розподілу дій даного потоку на дві менші задачі;

1. Якщо перевірка успішна, то виконується розподіл (Fork), шляхом створення нових потоків для кожної нової задачі. В кожному новому потоку алгоритм починається заново. Потік, який виконав розподіл блокується до тих пір, поки обидва створені ним потоки не закінчать свою роботу, а після виконує остаточний збір результату;

2. Якщо перевірка не успішна (досягнуто межі так званого «зерна паралелізму»), то виконуються задані обчислення в цьому потоці, по закінченню яких відбувається з'єднання з породившим потоком (Join).

Таку реалізацію містять два класи: RecursiveAction та RecursiveTask. Головна різниця між ними полягає в тому, що коли необхідно порахувати якесь конкретне числове значення певної великої функції (наприклад суму елементів вектора) то зручно використовувати RecursiveTask, а у випадку загальних операцій, результатом роботи яких не є конкретне число, краще використати RecursiveAction. Наслідуючи дані класи та описавши власне перевантаження методу compute розробник може налаштувати роботу під вирішення конкретної задачі, що є надзвичайно ефективним.

Використання класу RecursiveAction для реалізації дрібнозернистого паралелізму в мові Java:

```
class ParallelFor extends
 RecursiveAction {
private int from, to;volatile final int
TASK_LEN = 25;
public ParallelFor(int from, int to)
{this.from = from

this.to = to;
}
protected void compute() {
int len = to - from;
if (len < TASK_LEN)
work(from, to);
else
int mid = (from + to) >>> 1;
ForkJoinTask<Void> parallelFor1 =
new ParallelFor(from, mid).fork();
ForkJoinTask<Void> parallelFor2 =
new ParallelFor(mid, to).fork();
parallelFor1.join();
parallelFor2.join();
}
}
}
```

Змінна TASK_LEN визначає глибину розбиття, або, інакше кажучи, обсяг роботи, при досягненні якого потік приступить до його виконання, а не подальшого розпаралелювання. Запуск на виконання відбувається за допомогою методу invoke(), викликаного на екземплярі класу *ForkJoinPool:*

```
ForkJoinPool pool = new
 ForkJoinPool(); pool.invoke(new
ParallelFor(from, to));
```

Якщо описати метод work() наступним чином:

```
public void work(int from, int to){
 for (int i = from; i < to; i++)
}
 Calculations();} },
```

та задати початкові значення змінних TASK_LEN = 25, from = 0, to = 100, то ми ємо аналогічне до розглянутого в попередньому розділі паралельне виконання циклу із 100 ітерацій, в якому кожен із робочих потоків візьме на себе по 25 ітерацій.

Проте, на відміну від OpenMP, структура утворених потоків та їх ієрархія управлінняя буде деревовидною, такою як наведена на рис. 2.

Утворена структура є складнішою за ту, яку можна було спостерігати в OpenMP. Проте, як показують наведені в наступному розділі результати дослідження, такий підхід до організації дрібнозернистого паралелізму виявився надзвичайно ефективним.

Мова C#. В мові C# також з'явилась можливість реалізації дрібнозернистого паралелізму. Весь необхідний для цього функціонал вмістив в себе статичний клас System.Threading.Tasks.Parallel, а саме три його основні методи Parallel.For(), Parallel.ForEach(), Parallel.Invoke() та різні їх перевантаження. Parallel.For та Parallel.ForEach забезпечують паралельне виконання циклів for та foreach відповідно.

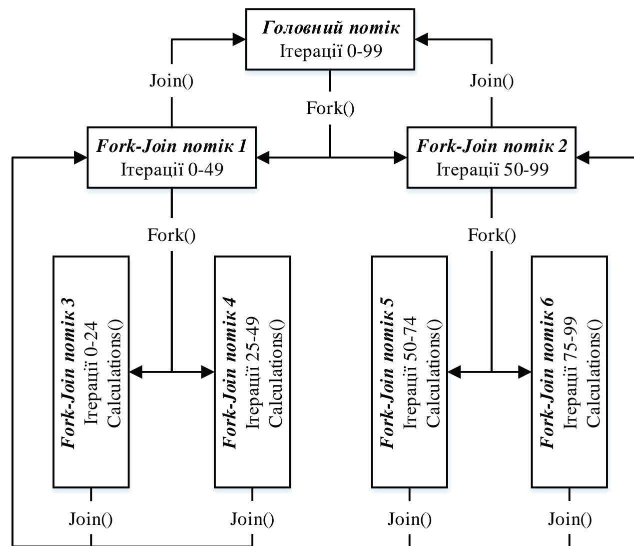

Рис. 2. Fork-Join розбиття циклу засобами мови Java

В основу кожного з наведених вище методів покладено механізм, аналогічний тому, який застосовується в мові Java. Але також його значно спрощено шляхом застосування механізму делегування та використанням анонімних методів. Від розробника не вимагається написання вручну всієї рекурсивної частини та регулювання зерна паралелізму, за це відповідає середовище виконання. Тому на практиці реалізація дрібнозернистого паралелізму стає надзвичайно простою та майже не відрізняється від класичного однопоточного підходу.

Наступний код забезпечує аналогічне до попередніх розпаралелювання циклу зі ста ітерацій.

```
Parallel.For(0, 100, i => {
 Calculation(); });
```

Хоч розбиття в мові C# і відбувається за тим самим механізмом що і в мові Java, проте, шляхом спрощення, яке забезпечує віртуальне середовище виконання CLR, отримана в результаті структура потоків та ієрархія їх управління буде аналогічною до тієї, яка малася в бібліотеці OpenMP

Присутні в даному класі перевантаження методів націлені на максимальну параметризацію паралелізму, в залежності від конкретної задачі, що реалізується.

Запропонований в роботі підхід базується на використанні в паралельної програмі двох видів паралелізму: середньозернистого і дрібнозернистого. При цьому паралельна програма включає набір традиційних потоків по кількості ядер БКС. Кожен з цих потоків додатково реалізує внутрішній (дрібнозернистий) паралелізм шляхом створення підпотоків за допомогою відповідних засобів типу Fork-Join. Початкова кількість традиційних потоків може зменшуватися з метою забезпечення дрібнозернистого паралелізму вільними процесорними ресурсами

Тестування програм проводилось на КС, оснащеною шестиядерним процесором AMD Phenom II на базі оновленої архітектури K10 та модулем ОЗУ об'ємом 4 ГБ типу DDR3.

Програмне забезпечення: операційна система Windows 7, бібліотека OpenMP 3.1, віртульна машина JVM 1.8, .NET Framework 4.7.

В табл. 1 наведені отримані результати тестування паралельних програм для операції множення матриць для різних значень N (розмірність матриць). Представлений час виконання програм, що побудовано:

- з використанням середньозернистого паралелізму через механізм потоків;
- з використанням дрібнозернистого паралелізму через описані в попередньому розділі механізми типа fork-join;
- з використанням змішаного паралелізму, де кожен потік додатково використовує паралельну обробку через fork-join конструкції.

Нижче наведено графіки, які демонструють залежність коефіцієнтів прискорення (Кп), для всіх трьох рівнів паралелізму, від значень N.

Окрім того, додатково було проведено окреме, більш детальне тестування дрібнозернистого паралелізму, з метою виявлення шляхів підвищення його ефективності.

На цьому етапі тестувалися ті програми із розробленого пакету, які були написані з використанням лише дрібнозернистого паралелізму. Нижче наведено графіки, які демонструють залежність коефіцієнтів прискорення (Кп), для всіх трьох рівнів паралелізму, від значень N.

Окрім того, додатково було проведено окреме, більш детальне тестування дрібнозернистого паралелізму, з метою виявлення шляхів підвищення його ефективності.

На цьому етапі тестувалися ті програми із розробленого пакету, які були написані з використанням лише дрібнозернистого паралелізму.

Проводилися багаторазові заміри часу виконання операції множення двох матриць розмірністю 1500*1500 елементів.

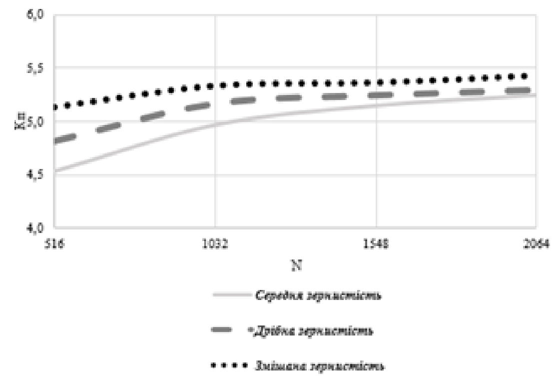

Рис. 3. Графік залежності Кп від N. Бібліотека OpenMP

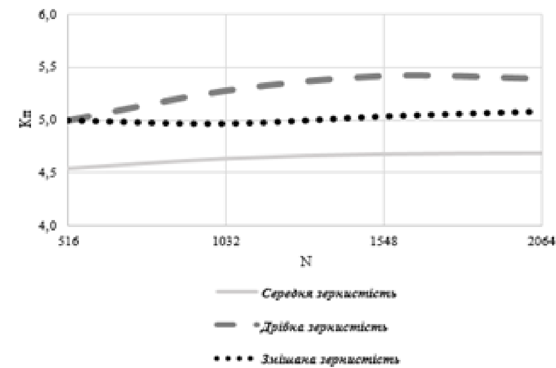

Рис. 4. Графік залежності Кп від N. Мова Java

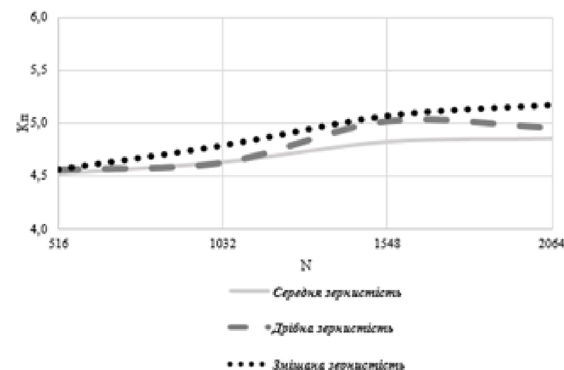

Рис. 5. Графік залежності Кп від N. Мова C#

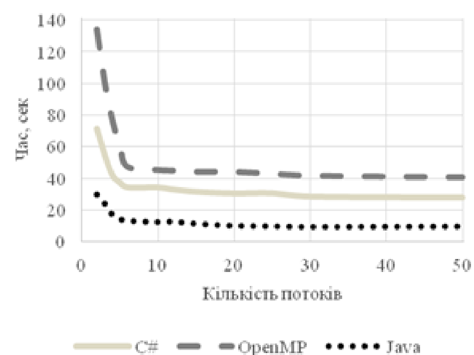

Рис. 6. Графік залежності часу роботи дрібнозернистого паралелізму від кілько-сті потоків.

Перевірялася залежність часу виконання від кількості програмно-реалізованих потоків. Нижче наведено графік, який демонструє виявлену залежність часу роботи від кількості потоків (P). Детальні результати наведено в табл. 2.

Таблиця 1. Результати тестування паралелізму різного ступеню зернистості

N	Час виконання (сек.)								
	Середня зернистість			Дрібна зернистість			Змішана зернистість		
	OpenMP	Java	C#	OpenMP	Java	C#	OpenMP	Java	C#
516	1,7	0,2	1,2	1,6	0,2	1,1	1,5	0,2	1,1
1032	13,3	3,3	9,8	12,8	2,9	9,8	12,4	3,1	9,5
1548	46,7	14,0	35,6	45,9	12,1	34,1	44,9	13,0	33,8
2064	111,7	38,5	79,8	110,8	33,5	78,1	108,1	35,5	74,8

Таблиця 2. Результати тестування дрібнозернистого паралелізму

Кількість потоків	Час виконання (сек.)		
	C#	OpenMP	Java
2	71,309	133,944	29,67
3	55,1	101,41	24,12
4	42,099	77,393	16,894
5	37,202	60,942	14,167
6	34,222	47,532	13,101
10	34,113	45,551	12,24
12	32,841	45,117	12,485
15	31,405	44,37	11,195
20	30,404	44,33	9,899
25	30,511	43,101	9,636
30	28,329	41,9	9,211
50	27,683	40,88	9,5

Висновки

Результати тестування показали ефективність БКС при реалізації рішенні розглянутої математичної задачі засобами мов Java і C# та бібліотеки OpenMP. При цьому можливе скорочення часу виконання програм при застосуванні паралелізму будь-якого ступеню зернистості (значення Кп лежать в межах 4,5-5,5). Найкращий результат по часу виконання програми отримано для мови Java.

Середньозернистий паралелізм показав достатню ефективність, але мав найгірший результат. При цьому прискорення, досягнуте засобами OpenMP постійно зростає зі збільшенням обсягу оброблюваних даних, а засобами Java та C# залишається приблизно на одному і тому ж рівні.

Застосування дрібнозернистого паралелізму також виявилося ефективним, в цьому випадку коефіцієнт прискорення стабільно зростає зі збільшенням обсягу оброблюваних даних. Найбільш ефективним цей тип паралелізму виявився в мові Java, завдяки потужному механізму підтримки fork-join.

Окрім того, проведене додаткове тестування дрібнозернистого паралелізму виявило спадаючу експоненціальну залежність між кількістю потоків, які дозволено створити програмі, та часом її роботи. Як було вказано в розділі 3, головною причиною цього є те, що чим більша кількість потоків, тим більше від них звернень до спільних ресурсів, що призводить до значних простоїв внаслідок вирішення задачі взаємного виключення.

Оптимальна кількість потоків дрібнозернистого паралелізму, незалежно від того, якими засобами його було організовано, лежить в межах від 5 до 20.

Запропонований в роботі підхід, який ґрунтується на змішаному паралелізму, показав свою ефективність і дозволив

збільшити Кп при використанні в мові C# та бібліотеці OpenMP. При цьому спостерігається зростання Кп при збільшенні обсягу оброблюваних даних, що є одним з найважливіших аргументів доцільності застосування даного підходу в БКС

Можна припустити, що ефективність використання змішаного паралелізму буде збільшуватися при зростанні кількості ядер в БКС, де :

- з'являться додаткові процесорні ресурси для його реалізації;
- можна зменшити розмір дрібної зернистості;
- можна знайти оптимальне співвідношення між кількістю потоків і підпотоків. закладений в бібліотеці, аналогічно тому, як це зроблено в моделі Fork-Join.

Крім того, ефективність реалізації змішаного паралелізму в OpenMP можна покращити шляхом більш ефективної реалізації потужного механізму управління підпотоками, що закладений в бібліотеці, аналогічно тому, як це зроблено в моделі Fork-Join.

Таким чином, можна стверджувати що застосування комбінованого паралелізму в більшості випадків є ефективним підходом до реалізації об'ємних паралельних обчислень на багатоядерних комп'ютерних системах.

Література

1. *Doug L.* A Java Fork/Join Framework. In Proceedings of the ACM 2000 conference on Java Grande (JAVA '00). – 2020. – P. 36-43.

2. *Ponge J.* Fork and Join: Java Can Excel at Painless Parallel Programming Too! [Електронний ресурс]. – Режим доступу: http://www.oracle.com/technetwork/articles/java/fork-join-422606.html

Корочкін О.В., Русанова О.В., Демчик В.І.

ЕФЕКТИВНІСТЬ ДРІБНОЗЕРНИСТОГО ПАРАЛЕЛІЗМУ В СУЧАСНИХ КОМП'ЮТЕРНИХ СИСТЕМАХ

При виборі підходу до проектування програмного забезпечення для багатоядерних компютерних систем одним із перших вирішується питання гранулярності. Ступінь гранулярності варіюється від дрібнозернистої до крупнозернистої. Крупнозернистий паралелізм (coarse grained): кожне паралельне обчислення досить незалежне від інших, причому потрібен відносно рідкісний обмін інформацією між окремими обчисленнями. Дрібнозернистий паралелізм (fine grained): кожне паралельне обчислення досить мале і елементарне, складається з десятків команд. Ефективне паралельне виконання вимагає майстерного балансу між ступенем гранулярності програм і величиною комунікаційної затримки, що виникає між різними гранулами. Зокрема, якщо комунікаційна затримка мінімальна, то найкращу продуктивність обіцяє дрібнозернисте розбиття програми. Це той випадок, коли діє паралелізм даних. Якщо комунікаційна затримка велика, краще крупнозернисте розбиття програм Сучасні мови та бібліотеки паралельного програмування наряду з засобами створення потоків та організації їх взаємодії містять інструменти і для підтримки дрібнозернистого паралелізму. В работі наведени результати дослідження вікористання дрібнозернистго паралелізму з використанням засобів різних мов і бібліотек паралельного програмування. Показан що його використання в оптимальному поєднанні з іншими відами паралелізму надає можливість покращити ефективність багатоядерних компютерних систем.

Ключові слова: *багатоядерні комп'ютерні системи, дрібнозернистий паралелізм.*

Korochkin O.V., Rusanova O.V., Demchyk V.I.

EFFICIENCY OF SMALL GRAIN PARALLELISM IN MODERN COMPUTER SYSTEMS

When choosing an approach to designing software for multi-core computer systems, one of the first things to solve is the issue of granularity. The degree of granularity varies from fine-grained to coarse-grained. Coarse-grained parallelism: each parallel computation is quite independent of the others, and relatively rare exchange of information between individual computations is required. Fine-grained parallelism: each parallel calculation is quite small and elementary, consisting of dozens of commands. Effective parallel execution requires a masterful balance between the degree of granularity of programs and the amount of communication delay that occurs between different granules. In particular, if the communication delay is minimal, the best performance is promised by a fine-grained division of the program. This is the case when data parallelism works. If the communication delay is large, a coarse-grained division of programs is better. Modern parallel programming languages and libraries, along with tools for creating streams and organizing their interaction, also contain tools for supporting fine-grained parallelism. The paper presents the results of research into the use of fine-grained parallelism with the use of tools of different parallel programming languages and libraries. It is shown that its use in an optimal combination with other types of parallelism provides an opportunity to improve the efficiency of multi-core computer systems.

Keywords*: multi-core computer systems, fine-grained parallelism.*