УДК 681.3

**Ю.В. БОЙКО, С.Д. ПОГОРІЛИЙ, І.Ю. ШКУЛІПА**

# ДОСЛІДЖЕННЯ ПАРАЛЕЛЬНИХ СХЕМ АЛГОРИТМУ ПРИМА

***Abstract:*** *Prim's minimal spanning tree algorithm finding is considered. Its formalization in terms of Glushkov's modified systems of algorithmic algebras (SAA-M) was made. A number of schemes of parallel algorithm were obtained. Some methods of experimental implementation of achieved schemes were proposed with using of different parallel programming paradigms. Experimental searching performance gain for different schemes was carried out by using cluster computation.*

***Key words:*** *minimal spanning tree, Prim's algorithm, modified systems of algorithmic algebras, formalization, SAA-M scheme, paralleling, process, thread, parallelism, Java technology.*

***Анотація:*** *Розглянуто алгоритм Прима знаходження мінімального покривного дерева графа. Виконано його формалізацію у термінах модифікованих систем алгоритмічних алгебр В.М. Глушкова (САА-М). Отримано низку САА-М схем паралельної версії алгоритму. Запропоновано підходи до реалізації отриманих схем з використанням різних парадигм паралельного програмування. Виконано експериментальне дослідження приросту швидкодії для різних схем при проведенні кластерних обчислень.*

***Ключові слова:*** *мінімальне покривне дерево, алгоритм Прима, модифіковані системи алгоритмічних алгебр, формалізація, САА-М схема, розпаралелювання, процес, потік, паралелізм, Java - технологія.*

***Аннотация:*** *Рассмотрен алгоритм Прима нахождения каркаса минимального веса графа. Выполнена его формализация в терминах модифицированных систем алгоритмических алгебр В.М. Глушкова (САА-М). Получено несколько САА-М схем параллельной версии алгоритма. Предложены методы реализации полученных схем с использованием разных парадигм параллельного программирования. Выполнено экспериментальное исследование прироста быстродействия различных схем для разных систем с использованием кластерных вычислений.*

***Ключевые слова:*** *каркас минимального веса, алгоритм Прима, модифицированные системы алгоритмических алгебр, формализация, САА-М схема, распараллеливание, процесс, поток, параллелизм, Java – технология.*

## 1. Вступ

Математичні моделі у вигляді графів широко застосовуються при моделюванні різноманітних явищ, процесів і систем. Багато теоретичних і реальних прикладних завдань можуть бути вирішені за допомогою тих або інших процедур аналізу графових моделей. Серед цих процедур може бути виділений певний набір типових алгоритмів обробки графів.

Покривним деревом неорієнтованого графа [1] називається підграф вихідного графа, що є деревом і містить усі вершини даного графа. Визначивши вагу підграфа для зваженого графа як суму ваг ребер, що входять у підграф, під мінімальним покривним деревом (МПД) будемо розуміти покривне дерево мінімальної ваги. Змістовна інтерпретація завдання знаходження МПД може складатися, наприклад, у практичному прикладі побудови локальної мережі персональних комп'ютерів із прокладанням найменшої кількості сполучних ліній зв'язку.

Алгоритм Прима є найбільш ефективним алгоритмом знаходження мінімального покривного дерева для насичених графів. На відміну, наприклад, від алгоритму Крускала [1], час роботи якого залежить від кількості ребер графа, час роботи алгоритму Прима залежить лише від кількості вершин [1].

Метою роботи є отримання низки паралельних САА-М схем алгоритму Прима та їх експериментальна реалізація з метою дослідження підвищення швидкодії паралельного алгоритму.

## 2. Алгоритм Прима

Алгоритм Прима полягає в поступовій побудові дерева, додаванням одного ребра за крок: побудову починаємо з однієї вершини і розглядаємо її як дерево, що складається з однієї вершини, потім

додаємо до нього N-1 вершину. При цьому кожен раз вибираємо мінімальне ребро, що з'єднує вершину, яка вже включена до дерева, з вершиною, що ще не міститься в МПД. Для реалізації методу необхідні дві величини множинного типу: $V$ та $U$. Спочатку значенням $V$ є всі вершини графа, а $U$ порожнє. Якщо ребро $(i, j)$ задовольняє умові для включення до дерева, то номери вершин $i$ й $j$ вилучаються з $V$ і додаються в $U$. Більш докладно алгоритм Прима описано у [1].

Якщо граф представлено у вигляді матриці суміжностей, то метод, за своєю суттю, представляє собою пошук мінімального елемента матриці в кожному її стовпчику та додавання його на відповідне місце в результуючій матриці, що представляє МПД.

Розглянемо неорієнтовний граф $G(V,U)$ з навантаженими ребрами і введемо такі позначення:

- $V$ – множина вершин графа;
- $U$ – множина вершин результуючого дерева;
- Size=Card($V$) – кількість вершин графа.

Реалізацію виконано мовою програмування С++. Граф представлено у вигляді класу Graph з полями Matrix – матриця суміжностей та Size – кількість вершин, а також методом PrimTree(), що знаходить і повертає МПД графа, та деякими додатковими функціями.

Протокольна частина опису класу мовою С++ схематично має такий вигляд:

```
template <class Type>
class Graph
{
private :
// матриця суміжностей графа
  Type* Matrix;
// кількість вершин графа
  int Size;
public :
// конструктор за замовчанням, створює граф розміром 0
  Graph(void);
// конструктор з одним параметром – кількістю вершин графа, створює граф з
// відповідною кількістю вершин без ребер
  Graph(int );
// конструктор з матрицею та кількістю вершин, створює граф відповідного розміру
// з вказаною матрицею суміжностей
  Graph(Type* ,int );
// деструктор
  ~Graph();
// конструктор копіювання
  Graph<Type> &operator =(const Graph<Type> &RParam)
  {
    int Length= RParam.Size*RParam.Size;
    Size= RParam.Size;
    for(int i=0;i<Length;i++)
    {
      Matrix[i]=RParam.Matrix[i];
    }
    return *this;
  }
// Повернення (i,j)-го елемента матриці
  Type GetElement(int, int);
// Встановлення (i,j)-го елемента матриці
```

```
  Type SetElement(int, int);
// Обрахунок та повернення результуючого покривного дерева за методом Прима
  Graph<Type> PrimTree();
};
```

Основний цикл, що знаходить мінімальне покривне дерево графа за методом Прима виглядає таким чином:

```
while (!V.empty())
{
  min= (unsigned short)(-1);
  l= 0;
  t= 0;
  for (int i=1;i<=Size;i++)
  {
   if (U.find(i)==U.end())
   {
     for (int j=1;j<=Size;j++)
     {
      if ((!(U.find(j)==U.end()))&&(GetElement(i,j)<min)
        &&(GetElement(i,j)!=0) )
      {
        min= GetElement(i,j);
        l= i;
        t= j;
      };
     };
     U.insert(l);
     V.erase(l);
     TREE.SetElement(l,t,GetElement(l,t));
   }
  }
}
```

### 3. Формування паралельної схеми алгоритму Прима

**Модифіковані Системи Алгоритмічних Алгебр.** В кібернетиці до числа найбільш фундаментальних відноситься поняття зворотного зв'язку, на якому базується керування функціонуванням складних систем різної природи. Зворотний зв'язок втілений у концепції абстрактної моделі керування, що базується на абстрактній моделі однопроцесорної ЕОМ В.М. Глушкова. Для формалізованого представлення алгоритмів функціонування абстрактної моделі ЕОМ В.М. Глушков запропонував математичний апарат систем алгоритмічних (мікропрограмних) алгебр (САА) [2].

Фіксована САА являє собою двоосновну алгебраїчну систему, основами якої є множина операторів і множина умов. Операції САА поділяються на логічні і операторні. До логічних відносяться узагальнені булеві операції і операція лівого множення оператора на умову (призначена для прогнозування обчислювального процесу), а до операторних – основні конструкції структурного програмування (послідовне виконання, циклічне виконання тощо.

*Теорема Глушкова.* Для довільного алгоритму існує (в загальному випадку не єдина) САА, в якій цей алгоритм може бути представлений регулярною схемою. Існує конструктивна процедура регуляризації довільного алгоритма.

Розглянута вище модель абстрактного представлення алгоритмів, очевидно, допускає узагальнення на багатопроцесорний випадок. З абстрактними моделями багатопроцесорних

обчислювальних систем асоційовані так звані модифіковані САА (САА-М), що отримані розширенням сигнатури САА шляхом введення трьох додаткових операцій, орієнтованих на формалізацію паралельних обчислень.

*Фільтрація.* $\underline{\alpha}$ – унарна операція, що залежить від умови $\alpha$ і породжує оператори-фільтри, такі що:

$\underline{\alpha}(m) = E(m)$, якщо $\alpha(m) = 1$, $//E$ – тотожній оператор $(E(m) = m)$, де $m$ – це певний оператор $N(m)$, якщо $\alpha(m) = 0$, $//N$ – невизначений оператор.

Фільтри забезпечують гнучке управління обчислювальним процесом. При цьому обчислювальні гілки з істинними фільтруючими умовами $\alpha$ продовжують працювати, а інші обриваються.

*Синхронна диз'юнкція.* AVB – бінарна операція, що полягає в синхронному застосуванні операторів A і B.

(AVB)(m)= m0, де $m0 = A(m) = B(m)$;

$A(m)$, коли $A(m) \neq \omega$ і $B(m) = \omega$

$B(m)$, коли $B(m) \neq \omega$ і $A(m) = \omega$

$\omega$ в інших випадках,

де $\omega$ – стан невизначеності.

*Асинхронна диз'юнкція.* $A \overset{\bullet}{\vee} B$ – паралельне виконання A і B.

**Формалізація алгоритму Прима.** Отже, згідно з теоремою Глушкова, кожен алгоритм має певне представлення у вигляді САА схеми. Тобто, якщо визначити основи САА для конкретного алгоритму, можна представити цей алгоритм у вигляді схеми і проводити подальші трансформації й оптимізації вже не з алгоритмом, а з його САА схемою.

Визначивши певну множину операторів та умов, можна представити алгоритм Прима у вигляді САА схеми.

Введемо множину операторів:

- M1: // змінній min присвоюється максимальне значення типу int для знаходження мінімального ребра;
- I1: // ітераційній змінній $i$ присвоюється значення 1;
- J1: // ітераційній змінній $j$ присвоюється значення 1;
- J3 : // ітераційній змінній $j$ присвоюється значення $i+1$;
- I2 : // інкрементне збільшення значення $i$;
- J2: // інкрементне збільшення значення $j$;
- M2: min=GetElement(i,j); // змінній min присвоюється значення $(i, j)$-го елемента матриці суміжностей графа;
- L0, T0, I0: // обнуляються змінні l, $t$, $i$;

▪ Vi: V.insert(i); // номер вершини і додається до множини $V$ ;

▪ L1: // змінній l присвоюється значення змінної $i$ ;

▪ T1: // змінній $t$ присвоюється значення змінної $j$ ;

▪ Ul: U.insert(l) // номер вершини l додається до множини $U$ ;

▪ Ut: U.insert(t) // номер вершини $t$ додається до множини $U$ ;

▪ Vl: V.erase(l) // номер вершини l видаляється з множини $V$ ;

▪ Vt: V.erase(t) // номер вершини $t$ видаляється з множини $V$ ;

▪ TS: TREE.SetElement(l,t,min) // ребро, що з'єднує вершини з номерами l і $t$ і вагою min, додається до МПД;

▪ Tree; // створення матриці результуючого дерева, заповненої 0;

Множина предикатів:

▪ αi: // $i$ менше або дорівнює розміру матриці суміжностей графа;

▪ αj: // $j$ менше або дорівнює розміру матриці суміжностей графа;

▪ αi1: // і менше розміру матриці суміжностей графа;

▪ ω: $i$ не знаходиться у множині $U$ ;

▪ υ: $V$ – не пуста множина;

▪ ξ : $j$ не знаходиться у множині $U$ ;

▪ α: // $(i, j)$ – й елемент матриці суміжностей менше min;

▪ β: // $(i, j)$ – й елемент матриці суміжностей, не рівний нулю;

Тоді послідовна САА схема алгоритма Прима виглядатиме таким чином:

*П0*= M1*I1*$_{αi}${Vi*I2}*Tree*
I1*$_{αi1}${J3* $_{αj}${$_{α\wedge β}$(M2*L1*T1)*J2}*I2}
*Ul*Ut*Vl*Vt*TS*
$_{υ}${M1*L0*T0*I1*
$_{αi}$ {$_{ω}$(J1*$_{αj}${$_{(ξ\wedge α\wedge β)}$(M2*L1*T1)*J2}*Ul*Ut*TS)*I2}}

**Формування паралельної схеми** САА схем перетворити її на САА-М схему паралельної версії алгоритму Прима, тобто **алгоритму Прима.** Отже, маючи САА схему алгоритму Прима, можна шляхом формальних трансформацій трансформувати послідовний алгоритм у паралельний [2].

У наведеній нижче схемі використано властивість розпаралелювання циклу і введено три операції асинхронної диз'юнкції.

*Пπ1= M1*I1*[$_{αi}${Vi}$\dot{\vee}$ {I2}]*Tree*
I1*[$_{αi1}${J3}$\dot{\vee}$ $_{αi1}${I2}$\dot{\vee}$ $_{αi1}${ $_{αj}${$_{α\wedge β}$(M2*L1*T1)*J2}}}]
*Ul*Ut*Vl*Vt*TS*
$_{υ}${M1*L0*T0*I1*
$_{αi}$ {$_{ω}$(J1*$_{αj}${$_{(ξ\wedge α\wedge β)}$(M2*L1*T1)*J2}*Ul*Ut*TS)*I2}}*

Наступну схему можна отримати лише за рахунок використання властивості розпаралелювання циклу.

*Пπ2= M1*I1*$_{αi}${Vi*I2}*Tree*
I1*$_{αi1}${J3*$_{αj}${$_{α\wedge β}$(M2*L1*T1)*J2}*I2}*

$$*Ul*Ut*Vl*Vt*TS*$$
$$_{\upsilon}\{M1*L0*T0\}\dot{\vee}$$
$$_{\upsilon}\{I1*_{\alpha i}\{_{\omega}(J1*_{\alpha j}\{_{(\xi\wedge\alpha\wedge\beta)}(M2*L1*T1)*J2\}*Ul*Ut*TS)*I2\}\}$$

Остання схема *Пπ2* є менш ефективною для використання у практичних цілях через те, що часи виконання операторів циклів, які виконуються паралельно, суттєво відрізняються, тобто потрібна синхронізація. При більш глибокому аналізі, з конкретизацією позначених операторів, виявляється, що використання синхронізації також дасть дуже несуттєві результати зменшення часу роботи програми.

Третя схема отримана лише за рахунок введення операції асинхронної диз'юнкції. Часи виконання паралельних операторів приблизно одного порядку, тому їх паралельне виконання гіпотетично може дати більш суттєві результати зменшення часу роботи алгоритму.

$$\Pi\pi3= M1*I1*_{\alpha i}\{Vi*I2\}*Tree*$$
$$I1*_{\alpha i1}\{J3*_{\alpha j}\{_{\alpha\wedge\beta}(M2\dot{\vee} L1\dot{\vee} T1)*J2\}*I2\}$$
$$*[Ul\dot{\vee} Ut\dot{\vee} Vl\dot{\vee} Vt\dot{\vee} TS]*$$
$$_{\upsilon}\{M1*L0*T0*I1*$$
$$_{\alpha i}\{_{\omega}(J1*_{\alpha j}\{_{(\xi\wedge\alpha\wedge\beta)}(M2\dot{\vee} L1\dot{\vee} T1)*J2\}*Ul\dot{\vee} Ut\dot{\vee} TS)*I2\}\}$$

**Розпаралелювання алгоритму Прима за даними.** Експериментальна реалізація всіх вищерозглянутих схем не дає суттєвого зменшення часу роботи паралельного алгоритму у порівнянні з послідовним. Найбільш цікавою з точки зору практичної реалізації є наступна схема, в якій використана та властивість алгоритму Прима, що в основному циклі, що знаходить МПД, відсутній зв'язок за даними. Тобто можна розбити матрицю суміжностей на $\kappa$ частин та знаходити мінімальні елементи стовпчиків паралельно, в загальному випадку асинхронно.

Увівши позначення

- BeginThreads; // створення паралельних гілок;
- EndThreads; // знищення паралельних гілок;
- <u>IS: i=Start;</u> // присвоюємо ітераційній змінній і значення Start для подальшого використання в циклі;
- <u>αe: i<= End;</u> // і менше або дорівнює End,

де змінні Start і End позначають початковий і кінцевий індекси частини матриці суміжностей, що обробляється окремою паралельною гілкою алгоритма.

Тоді отримаємо наступну схему алгоритма Прима.

$$\Pi\pi4= M1*I1*_{\alpha i}\{Vi*I2\}*Tree*\ BeginThreads*$$
$$_{\upsilon}\{M1*L0*T0*\ (\pi A_1\dot{\vee}\ \pi A_2\dot{\vee}\ \ldots\ \pi A_M)*$$
$$*EndThreads,$$

де оператори

$$\pi A_i=IS_{\alpha e}\ \{_{\omega}(J1*_{\alpha j}\{_{(\xi\wedge\alpha\wedge\beta)}(M2*L1*T1)*J2\}*Ul*Ut*TS)*I2\}\},$$

а $M$ – кількість паралельних гілок, що виконуються програмою.

Реалізація даної схеми дає найбільш цікаві результати, які будуть наведені нижче з урахуванням системного рівня розпаралелювання та вибраної технології.

**4. Експериментальна реалізація паралельної схеми алгоритму Прима**

**Архітектура обчислювального кластера.** Найбільш доступний спосіб підвищення обчислювальних ресурсів до суперкомп'ютерного рівня надають кластерні технології. У найпростішому випадку за допомогою мережевих комунікаційних пристроїв поєднуються однотипні комп'ютери і до їх програмного забезпечення додається комунікаційна бібліотека типу MPI (*Message Passing Interface*). Це дозволяє використати набір комп'ютерів як єдиний обчислювальний ресурс для запуску паралельних програм. Однак навряд чи таку систему можна назвати повноцінним кластером. Виявляється, що для ефективної експлуатації такої системи, необхідна деяка диспетчерська система, яка б розподіляла завдання користувачів по обчислювальних вузлах і блокувала запуск інших завдань на зайнятих вузлах. Існує досить велика кількість таких систем, як комерційних, так і безкоштовно розповсюджуваних. Причому більшість із них не вимагає ідентичності обчислювальних вузлів.

Обчислювальний кластер Інформаційно-обчислювального центру Київського національного університету імені Тараса Шевченка є частиною проекту "Комп'ютерна мережа Київського університету" і створений для вирішення прикладних задач, що потребують великих витрат машинного часу та оперують великими об'ємами інформації [7].

Кластер Київського національного університету імені Тараса Шевченка належить до гетерогенних кластерів типу MOSIX. Система складається із 13 двопроцесорних вузлів на базі Intel® Pentium-III 933МГц та 1ГГц і Intel® Xeon 2.4ГГц. У ролі службової мережі використовується Gigabit Ethernet. На вузлах встановлено операційну систему Linux на основі поставки RedHat 7.1 з Kernel - 2.4.21 і openMosix -1. Кожен вузол кластера має однакову структуру каталогів та містить однаковий набір програмного забезпечення, інстальованого в одні й ті ж каталоги. Основною моделлю програмування є MPI, підтримується також PVM (Parallel Virtual Machine) [8].

Для експериментальної реалізації отриманих у попередньому розділі схем використано декілька парадигм паралельного програмування:

- процесо-орієнтований паралелізм;
- потоко-орієнтований паралелізм;
- застосування Java-технології;
- застосування зовнішніх бібліотек для моделювання спільної пам'яті на розподілених архітектурах.

**Моделювання спільної пам'яті.** У даний час використовується така класифікація паралельних комп'ютерів за архітектурою підсистеми оперативної пам'яті: системи зі спільною пам'яттю, у яких є одна велика віртуальна пам'ять і всі процесори мають однаковий доступ до даних і команд, що зберігаються в цій пам'яті; системи з розподіленою пам'яттю, у яких кожен процесор має свою локальну оперативну пам'ять і до цієї пам'яті в інших процесорів немає доступу.

Коли застосовується паралельна архітектура з розподіленою пам'яттю, потрібно забезпечити можливість обміну даними між задачами, що працюють на різних машинах. Один з

методів – це використання технології MPI, яка надає механізм взаємодії паралельних гілок всередині паралельного застосування незалежно від машинної архітектури. На комп'ютері з розподіленою пам'яттю програма перемноження матриць, наприклад, повинна створити копії матриць, що перемножуються, на кожному процесорі, що технічно здійснюється передачею на ці процесори повідомлення, що містить необхідні дані. У випадку системи зі спільною пам'яттю досить лише один раз задати відповідну структуру даних і розмістити її в оперативній пам'яті.

Найпростіший спосіб створити багатопроцесорний обчислювальний комплекс із спільною пам'яттю – взяти кілька процесорів, з'єднати їх із загальною шиною та поєднати цю шину з оперативною пам'яттю. Цей простий спосіб є не дуже вдалим, оскільки між процесорами виникає боротьба за доступ до шини, і якщо один процесор приймає команду або передає дані, всі інші процесори змушені будуть перейти в режим очікування. Це приводить до того, що починаючи з деякого числа процесорів, швидкодія такої системи перестане збільшуватися при додаванні нового процесора.

Поліпшити картину може застосування кеш-пам'яті для зберігання команд. При наявності локальної, тобто приналежної певному процесору кеш-пам'яті, необхідна йому команда з великою ймовірністю буде перебувати в кеш-пам'яті. У результаті цього зменшується кількість звернень до шини, і швидкодія системи зростає. Разом з тим виникає нова проблема – проблема кеш-когерентності. Ця проблема полягає в тому, що, наприклад, двом процесорам для виконання різних операцій знадобилося значення $V$ , це значення буде зберігатися у вигляді двох копій у кеш-пам'яті обох процесорів. Один із процесорів може змінити це значення в результаті виконання своєї команди, і воно буде передано в оперативну пам'ять комп'ютера. Але в кеш-пам'яті другого процесора усе ще зберігається старе значення. Отже, необхідно забезпечити своєчасне відновлення даних у кеш-пам'яті всіх процесорів комп'ютера.

Є й інші реалізації спільної пам'яті. Це, наприклад, спільна пам'ять із дискретними модулями пам'яті. Фізична пам'ять складається з декількох модулів, хоча віртуальний адресний простір залишається загальним. Замість загальної шини у цьому випадку використовується перемикач, який направляє запити від процесора до пам'яті. Такий перемикач може одночасно обробляти кілька запитів до пам'яті, тому, якщо всі процесори звертаються до різних модулів пам'яті, швидкодія зростає.

Якщо використовуються паралельні системи за розподіленою пам'яттю, то є можливість програмного моделювання спільної пам'яті.

Існує досить велика кількість бібліотек, що дозволяють створити програмну модель спільної пам'яті. Майже у всіх них в основі лежать дві основні ідеї – створення віртуального масиву пам'яті, до якого рівноправно і рівноймовірно матимуть доступ усі паралельні процеси, та створення спільної пам'яті за рахунок одночасного виконання декількох процесів на різних вузлах паралельної обчислювальної системи й обміну повідомленнями між цими процесами на основі клієнт-серверної взаємодії. Такі бібліотеки також мають інструментарій для створення процесів чи потоків за рахунок системних викликів, але прив'язаних до спільного простору пам'яті.

У процесі роботи було використано дві бібліотеки для моделювання спільної пам'яті: Quarks та ARCH [12], але результатів роботи програм, що використовують ці бібліотеки, не вдалося

отримати, оскільки вони є структурно орієнтованими і тому не були інстальовані на кластері університету.

**Паралельна програма, що використовує процесо-орієнтовану парадигму.** Поняття процесу вперше з’явилося в операційній системі UNIX [3]. Під процесом слід розуміти окрему програму чи застосування, яке виконується в даний момент часу і має власну пам’ять, певний пріоритет виконання, права доступу до апаратних ресурсів тощо.

Ідея розпаралелювання обрахунків на рівні паралельних процесів полягає у розподіленні певних дій між різними паралельними програмами з однаковим пріоритетом виконання.

Створення паралельних процесів і передача їм певної сукупності дискретних даних є нескладною задачею. Проблеми виникають при передачі даних з процеса, що виконує обрахунки до основного процеса, який обробляє результати. Полягають вони саме в тому, що кожен процес має власну пам’ять, яка унеможливлює роботу в ситуації, коли, наприклад, кожен процес обраховує певну окрему частину результуючої матриці.

Існують два шляхи вирішення проблеми:

▪ організація черги обміну даними між процесами;

▪ використання спільної[*] пам’яті.

Перший спосіб є найменш зручним для передачі великих об’ємів інформації. Його незручність полягає в тому, що потрібно слідкувати за тим, щоб основний процес коректно зчитував дані і не завершив свою роботу до того, як черга стане порожньою. До того ж передача через чергу великої кількості даних може відбуватися досить значний час, що суттєво знижує ефективність роботи програми.

Таких проблем повністю позбавлений другий спосіб. Ідея створення спільної пам’яті полягає у тому, щоб створити для всіх процесів спільну пам’ять, яка може використовуватися всіма процесами паралельно, подібно до звичайної динамічної пам’яті послідовної програми. В цій пам’яті потрібно розмістити вихідні і, головне, результуючі дані, зробивши їх таким чином доступними будь-якому процесу [6].

Розпаралелювання алгоритму Прима на рівні паралельних процесів відбувається за наступною схемою:

▪ створюємо спільну пам'ять і приєднуємо її до всіх процесів;

▪ розміщуємо у спільній пам’яті вихідну і результуючу матриці;

▪ клонуємо основний процес потрібну кількість разів (створюємо процес-нащадок, ідентичний основному);

▪ викликаємо у процесі-нащадку функцію, що має виконати даний процес, і передаємо до неї всі необхідні дані, після чого завершуємо роботу процесу-нащадка (батьківський процес продовжує роботу);

▪ в основному процесі чекаємо завершення всіх створених процесів-нащадків;

---

[*] Термін «спільна пам'ять» в даному випадку слід розуміти не як апаратно-реалізовану спільну пам'ять для багатьох процесорів, а як адресний простір батьківського процесу, що приєднується до всіх процесів-нащадків.

▪ виводимо результати обчислень (результуюча матриця).

Результати по підвищенню ефективності роботи програми, створеної за схемою *Пπ4*, зображено на рис. 1.

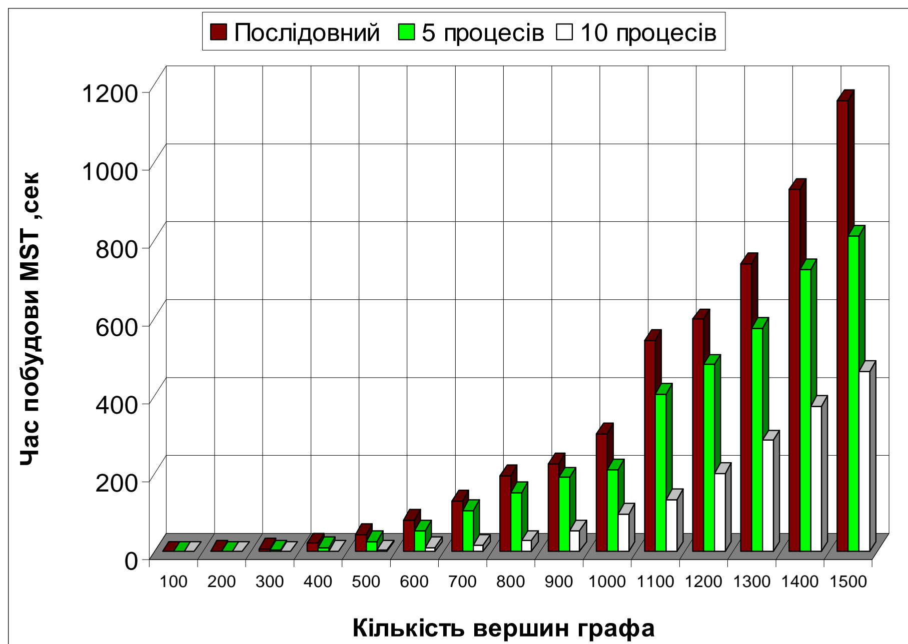


Рис. 1. Залежність часу роботи алгоритму Прима від кількості паралельних процесів

При розпаралелюванні на п'ять паралельних процесів середнє відносне зменшення часу роботи алгоритму Прима складає **26,5 %**.

При розпаралелюванні на десять паралельних процесів середнє відносне зменшення часу роботи алгоритму Прима складає **75,6 %**.

**Паралельна програма, що використовує потоко-орієнтовану парадигму.** Поняття потоку суттєво відмінне від поняття процесу [10]. Якщо процес – це окрема програма, що має свої ресурси, то потік – це частина процесу (програми), що виконується паралельно в межах ресурсів процесу. Оскільки потік – це частина програми, він має доступ до ресурсів пам'яті, що належать програмі. Тобто підхід, заснований на потоко-орієнтованій парадигмі, позбавлений проблем передачі даних, що виникають при роботі з процесами, але в цьому випадку виникають нові проблеми. По-перше, в будь-якому випадку синхронізація є необхідною складовою при роботі з паралельними потоками (необхідно, щоб головний потік програми не завершив свою роботу до того, як завершаться всі інші потоки).

Друга проблема – це виникнення колізій, коли декілька паралельних потоків намагаються отримати доступ до одного глобального ресурсу. В такому випадку необхідно використовувати так звані семафори для синхронізації роботи паралельних потоків. Семафор є певним аналогом булевої змінної, що приймає значення 1, коли ресурс не зайнятий іншим потоком, і 0 – у протилежному випадку. В бібліотечному файлі *pthread.h* описані об'єкти синхронізації типу *mutex* і функції для роботи з ними [3].

Розпаралелювання алгоритму Прима на рівні паралельних потоків відбувається за такою схемою:

▪ декларуємо потрібні змінні і матриці в динамічній пам’яті основної програми;

▪ виділяємо пам’ять для потоків;

▪ створюємо необхідну кількість паралельних потоків і передаємо їм функцію, яку вони повинні виконати;

▪ в основній програмі чекаємо завершення роботи всіх потоків;

▪ виводимо результати обчислень і завершуємо виконання основної програми.

Результати по підвищенню ефективності роботи програми, яка відповідає схемі *Пπ4*, зображено на рис. 2.

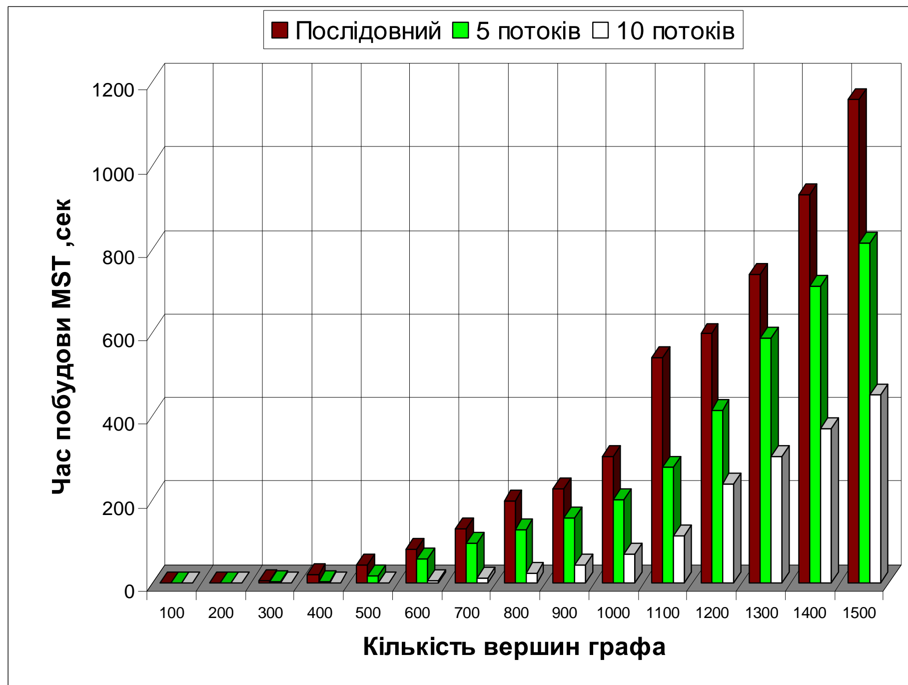


Рис. 2. Залежність часу роботи алгоритму Прима від кількості паралельних потоків

При розпаралелюванні на п’ять паралельних потоків середнє відносне зменшення часу роботи алгоритму Прима складає **35,8%.**

При розпаралелюванні на десять паралельних потоків середнє відносне зменшення часу роботи алгоритму Прима складає **77,5 %.**

**Паралельна програма на основі Java-технології.** Програма, що використовує Java, за принципом роботи майже нічим не відрізняється від програми, що використовує потоки, як описано в попередньому пункті. Єдина відмінність у тому, що вона написана мовою Java і працює на віртуальній Java-машині. Слід зазначити, що технологія Java має вбудовану підтримку паралельних потоків на рівні мови, а також досить непогану підтримку регулярних виразів та операцій порівняння, що дуже добре підходить саме для вибраного алгоритму (більш докладно про Java-технологію – у [10–14]).Тому результати дослідження виявилися досить цікавими.

Результати по підвищенню ефективності роботи програми (схема *Пπ4*), написаної на основі Java, представлено на рис. 3.

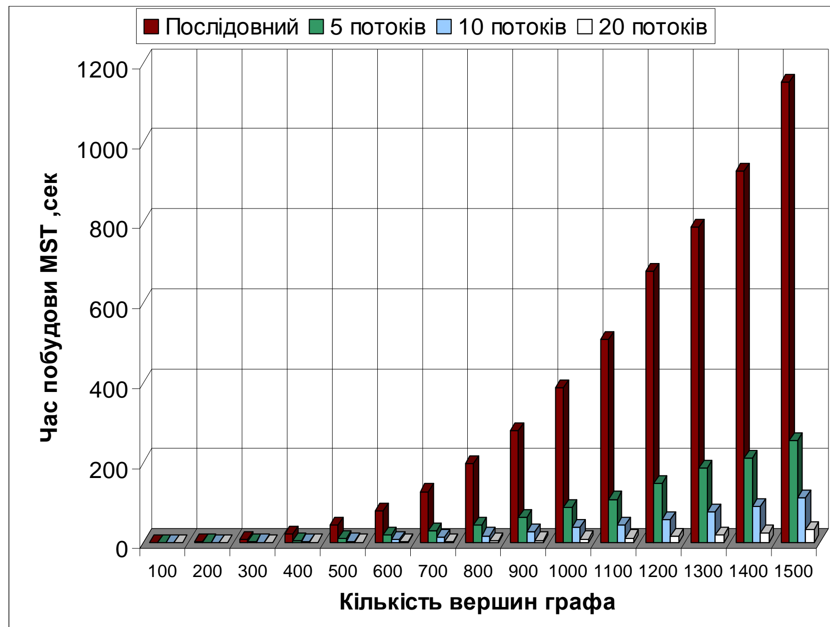


Рис. 3. Залежність часу роботи алгоритму Прима від кількості паралельних потоків. Java версія

При розпаралелюванні на п'ять паралельних потоків час роботи алгоритму у порівнянні з послідовним зменшується в середньому у **4** рази.

При розпаралелюванні на десять паралельних потоків час роботи алгоритму у порівнянні з послідовним зменшується в середньому у **10** разів.

При розпаралелюванні на двадцять паралельних потоків час роботи алгоритму у порівнянні з послідовним зменшується в середньому у **19** разів.

## 5. Порівняльний аналіз реалізацій алгоритма Прима

Для порівняння ефективності роботи різних реалізацій алгоритму Прима було вибрано паралельні програми, що використовують потоки у версії С++ та Java. Результати порівняння наведено на рис. 4.

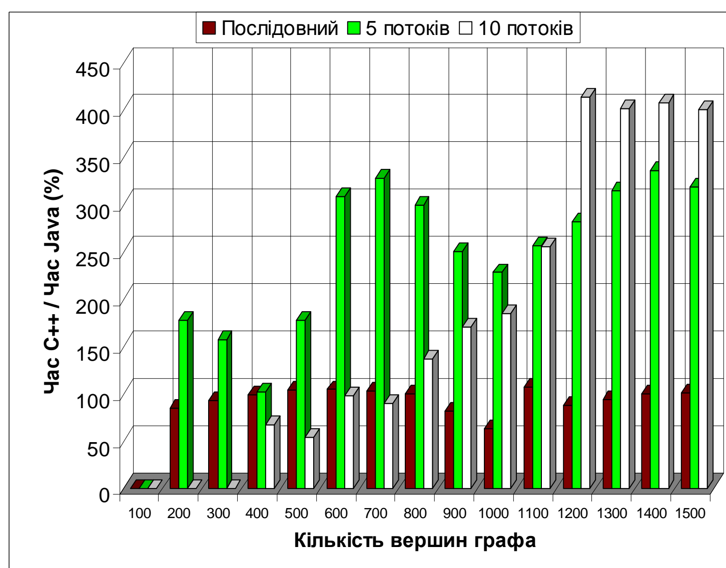


Рис. 4. Відносне зменшення часу роботи алгоритму Прима від кількості паралельних потоків та методу реалізації (С++/Java) у відсотках

При розпаралелюванні на п'ять паралельних потоків Java дає середній виграш 135,75% у порівнянні з С++ реалізацією.

При розпаралелюванні на десять паралельних потоків Java дає середній виграш 78,53% у порівнянні з С++ реалізацією.

Переваги Java-технології пояснюються більш ефективною моделлю спільної пам'яті, яка реалізована у віртуальній Java-машині, що дає суттєвий виграш у продуктивності обчислень.

**6. Висновки**

▪ Основною задачею оптимізації алгоритму Прима є підвищення продуктивності при зростанні розмірності вихідного графа за рахунок створення паралельних версій.

▪ Підвищення продуктивності алгоритму можна досягти шляхом використання різних класичних парадигм паралельного програмування (процеси, потоки, MPI тощо). Але програмна реалізація паралельних версій на кластері зводить переваги для графів великої розмірності практично нанівець за рахунок особливостей архітектури кластера.

▪ Застосування парадигми, що базується на технології Java, дозволяє досягти значного підвищення продуктивності алгоритму Прима при значному збільшенні кількості потоків.

▪ Для вибраної реалізації алгоритму Прима (граф представлено у вигляді матриці суміжностей) застосування технології Java є суттєво більш ефективним, ніж класичні підходи з реалізацією мовою С++.

**СПИСОК ЛІТЕРАТУРИ**


1. Седжвик Р. Фундаментальные алгоритмы на С++. – Ч. 5: Алгоритмы на графах. – Киев: «DiaSoft», 2002. – 484 с.
2. Ющенко Е.Л., Цейллин Г.Е., Грицай В.П., Терзян Т.К. Многоуровневое структурное проектирование программ. – М., 1989. – 254 с.
3. Богачёв К.Ю. Основы параллельного программирования. – М.:«Бином», 2003. – 289 с.
4. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. – Ростов-на-Дону: ЮГИНФО РГУ, 2003. – 207 с.
5. Богатырев А. Хрестоматия по программированию на Си в UNIX. http://citforum.univ.kiev.ua/programming/c_unix/index.shtml.
6. Boyko Y.V., Vystoropsky O.O., Nychyporuk T.V., Sudakov O.O. Kiyv National Taras Shevchenko University high-performance computing cluster // Third international young scientists' conference on Applied Physics. – 2003. – June 18–20. – P. 180–181.
7. Судаков О.О., Бойко Ю.В., Третяк О.В., Короткова Т.П. Оптимізація продуктивності обчислювального кластера на базі розподілених слабозв'язаних компонентів // Математичні машини і системи. – 2004. – № 4. – С. 57–65.
8. Судаков О.О., Бойко Ю.В. GRID-ресурси інформаційно-обчислювального центру Київського національного університету імені Тараса Шевченка // Проблеми програмування. – 2006. – № 2–3. – С.165–169.
9. Погорілий С.Д. Програмне конструювання: Підручник / Під ред. академіка АПН України Третяка О.В. – К.: ВПЦ, Київський університет, 2005. – 438 с.
10. Погорілий С.Д., Камардіна О.О., Кордаш Ю.С. Про підвищення швидкодії алгоритмів формування мінімального вкриваючого дерева // Математичні машини и системи. – 2005. – № 4. – С. 30–38.
11. http://citforum.ru.
12. http://unicc.univ.kiev.ua.
13. http://parallel.ru.
14. http://java.sun.com.
15. http://www.javaworld.com.
16. http://www.javaportal.ru.
17. http://www.code.net.
18. http://www.javable.com.