УДК 004.4

С.Д. Погорілий, д. т. н., проф.,
І.А. Висоцький, студент
Київський національний університет ім. Тараса Шевченка, Україна
sdp77@i.ua, ivan.vysotskyy@gmail.com

Використання паралельних обчислень для побудови безпровідних сенсорних мереж

Запропоновано використання паралельної реалізації обчислення мінімального остовного дерева на прикладі алгоритму Борувки для побудови надійних безпровідних сенсорних мереж. Виконано модифікацію алгоритму для паралельних систем та його формалізацію з використанням математичного апарату систем алгоритмічних алгебр В.М. Глушкова. Використано концепцію розпаралелювання за даними для архітектур з розподіленою пам'яттю. Проведено трансформацію алгоритму, отримано паралельну схему. Реалізовано паралельний алгоритм з використанням мови програмування С++ та технології MPI. Проведено кластерні експериментальні обчислення.

Ключові слова: сенсорні безпровідні мережі, мінімальне остовне дерево, алгоритм Борувки, паралельні обчислення, кластерні обчислення, розподілена пам'ять, системи алгоритмічних алгебр В. М. Глушкова, MPI.

Вступ

Останні кілька років інформатизація суспільства стає все більш масштабною. Дослідники використовують обчислювальні машини для розв'язання багатьох задач. Однак, деякі задачі вимагають надпотужних комп'ютерів. Прикладом такої задачі можуть бути дії над ґрафами, коли кількість вершин стає дуже великою.

Математичні моделі у вигляді ґрафів широко використовуються при моделюванні різноманітних явищ, процесів і систем. Таким чином, за допомогою алгоритмів на ґрафах може бути вирішена велика кількість як теоретичних, так і прикладних задач. Окремим класом таких задач виділяють пошук мінімального вкривного дерева ґрафа.

Метою роботи є оптимізація алгоритму Борувки за рахунок створення паралельної версії та визначення його робочих характеристик. Паралельні версії отримуються завдяки формальним перетворенням алгоритму з використанням математичного апарату систем алгоритмічних алгебр В.М.Глушкова[1,2]. Такий підхід дозволяє полегшити створення програмного продукту та оптимізувати його за вибраними характеристиками. Особливо такий підхід зручний при створенні складних програм та неочевидних перетвореннях алгоритмів.

Для оцінки ефективності отриманих схем створено програмну реалізацію з використанням технології MPI[3,4]. Критерієм ефективності вибрано час виконання програми в залежності від кількості паралельних гілок та розмірності задачі.

Постановка задачі

Алгоритми знаходження мінімального вкривного дерева дозволяють задачу мінімізації сумарної вартості зв'язків між елементами безпровідних однорангових мереж[5,6,7], яку можна представити у вигляді зваженого ґрафа.

Споживання електроенергії вузлами мережі та її запас визначають час роботи, або час життя мережі. Часом життя вважається відрізок часу, що триває від ініціалізації мережі, до виходу з неї першого вузла. Враховуючи те, що мережа є одноранговою, необхідно забезпечити наступні умови:

– максимізацію часу життя;
– живучість;
– автоматичну конфігурацію.

Серед підходів до вирішення цих проблем в мережах, розміри яких значно перевищують радіус дії передавача окремого вузла, можна виділити підхід, що ґрунтується на використанні алгоритмів побудови мінімального вкривного дерева ґрафа. Цей підхід дозволяє розробити стратегію комунікацій в мережі, на основі представлення мережі вкривним деревом та його періодичному оновленні, що дозволяє реалізувати ефективне балансування навантаження і таким чином досягти збільшення часу життя статичної безпровідної мережі. Енергетично вигідне дерево важливо мати для розповсюдження широкомовних сигналів, таких як періодичне оновлення інформації, підтвердження передачі даних, тому подібне. Для розв'язання проблем розповсюдження широкомовної інформації застосовується теорія ґрафів.

Розглянемо модель подібної мережі, кожен вузол якої має обмежену максимальну потужність

випромінювання, тоді ми отримаємо матрицю потужностей *Pij*, що необхідна для стійкого зв'язку між вузлами *i* та *j*. Вся мережа може бути представлена як зважений ґраф, що містить в собі ребра, відстань між вершинами яких менша деякої максимальної відстані, визначеної максимальною потужністю передавача. Існує два підходи до максимізації часу життя такої мережі, мінімізація сумарної випромінюваної потужності та мінімізація максимальної випромінюваної потужності. Останній є більш ефективним і фактично означає, що зв'язок буде встановлено з найближчим вузлом. Застосування такого підходу зводить задачу до пошуку мінімального вкривного дерева. Модель подібної мережі, представлена ґрафом G(V,E), зображена на рис. 1.

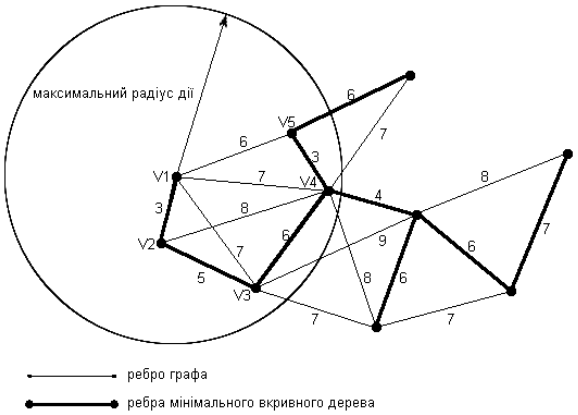


Рисунок 1 – Модель безпровідної сенсорної мережі

Вузли мережі V1, V2, V3,... являють собою вузли ґрафа G(V,E), усі можливі зв'язки між ними представлені набором зважених ребер, позначених відрізками Числа позначають вартість ребер. Вагова функція, що визначає вартості ребер у цій моделі, повинна враховувати параметри вузлів, такі як потужність випромінювання (фактично, відстань до найближчого вузла) та енергоресурс. Найефективнішим комунікаційним деревом в такому випадку буде набір ребер, що утворюють мінімальне вкривне дерево цього ґрафа. Розширенням даної моделі є модель адаптивної мережі, в якій структура вузлів-ретрансляторів з часом змінюється відповідно до ресурсів. Задача автоматичної конфігурації мережі вимагає періодичного оновлення мінімального вкривного дерева за максимально короткий час, так як час роботи окремого автономного елемента може бути порядку годин, і особливо актуальною ця проблема стає при обробці ґрафів великих розмінностей, коли час обрахунку мінімального вкривного дерева може бути сумірним із часом автономної роботи вузлів мережі.

Алгоритм Борувки

Алгоритм Борувки[8] - алгоритм побудови мінімального вкривного дерева зваженого неорієнтованого ґрафа. Побудова починається з дерева, що містить сукупність вершин і порожню множину ребер. В процесі роботи алгоритму множина ребер наповнюється, вершини ґрафа зливаються в компоненти зв'язності, поки не буде побудовано остовне дерево ґрафа.

Послідовність виконання алгоритму

1. Нехай спочатку *T* - порожня множина ребер (ґраф являє собою ліс, до якого кожна вершина входить як окреме дерево).
2. Поки T не є деревом:
   1) для кожної компоненти зв'язності (тобто, дерева в кістяковому лісі) у підґрафі з ребрами *T*, знайдемо найдешевше ребро, що пов'язує цю компоненту з будь-якою іншою компонентою зв'язності. Вважаємо, що вага ребер різна, або якось додатково впорядкована так, що завжди можна знайти єдине ребро з мінімальною вагою;
   2) додамо знайдені ребра до множини *T*.
3. Отримана множина ребер *T* є мінімальним кістяковим деревом початкового ґрафа.

Модифікація алгоритму Борувки для паралельних систем

Для розпаралелювання алгоритму кроки мають бути максимально зручними для паралельного незалежного виконання. Тому крок 2.1 зі схеми, наведеної вище, розділений на 2 кроки (2.1 та 2.2), які піддаються легкому розпаралелюванню за даними і незалежному виконанню в багатьох процесах одночасно. Отже, схема алгоритму набуває такого вигляду:

1. нехай спочатку *T* - порожня множина ребер (ґраф являє собою ліс, до якого кожна вершина входить як окреме дерево).
2. Поки T не є деревом:
   1) Для кожної вершини виконання пошуку найлегшого з усіх ребер, що сполучають цю вершину з вершиною іншого дерева;
   2) Для кожного дерева, існуючого на поточному кроці циклу, вибір найлегших з ребер, утворених в 2.1, тих, що інцидентні вершинам цього дерева;
   3) Додаємо знайдені в 2.2 ребра до множини *T*.
3. Отримана множина ребер *T* є мінімальним вкривним деревом початкового ґрафа.

Модель даних

Виконання алгоритму Борувки вимагає зберігання неорієнтованого ґрафа, кількість вершин якого *N*, зі зваженими ребрами. Для предста-

влення ґрафа використовується така абстрактна модель[9] даних:

– кожна вершина ґрафа має унікальне ім'я у вигляді номера (далі номер вершини);

– кожне дерево аналогічно має свій номер (далі корінь дерева вершини);

– кожному номеру дерева відповідає множина номерів вершин, які належать до дерева, і навпаки кожній вершині відповідає номер дерева, до якого вона належить;

– визначено окремий тип даних – ребро. Ребро *i-j* містить два числа *i* та *j* – номера вершин, яким інцидентне це ребро. Кожне ребро має вагу;

– множина *T*, яка в процесі виконання алгоритму наповнюється ребрами мінімального вкривного дерева.

Вхідними даними є кількість вершин *N* та інформація про ваги всіх можливих ребер у ґрафі. Це можуть бути координати точки на площині (тоді вагою ребра можна вважати відстань між точками) або ж матриця $N \times N$, кожен елемент якої позначає вагу ребра.

Результатом виконання алгоритму є заповнена множина *T*.

Формалізація алгоритму з використанням САА

Для розпаралелювання алгоритму виконано його формалізацію з використанням апарату систем алгоритмічних алгебр В.М.Глушкова (САА).

В схемах використовуються позначення оператора присвоєння ":=" та інкременту "++". Вираз "a:=b" означає, що змінній a присвоюється значення b. Вираз "d++" – збільшення значення змінної d на одиницю. Також в схемах алгоритму використовуються такі позначення:

величини:

N – Кількість вершин у вхідному ґрафі;

Weights – дані про ваги всіх можливих ребер на ґрафі;

edgesAmount – кількість ребер у мінімальному вкривному дереві;

trees – масив множин номерів вершин, що встановлює відповідність «номер дерева – множина номерів вершин». Тобто *trees*[i] – це множина номерів вершин, які мають корінь дерева i. $trees_i.size$ - кількість елементів у цій множині. $trees_{ij}$ – j-ий елемент множини $trees_i$;

T – множина ребер мінімального вкривного дерева;

tempEdges[0…N-1] – одновимірний масив для зберігання тимчасових ребер;

cheapestEdges[0…N-1] – двовимірний масив для зберігання результуючих ребер на поточній ітерації алгоритму;

vertexIndexes[0…N-1] – масив з номерами дерев, що встановлює відповідність «номер вершини – номер дерева». Вершина з номером *i* має корінь дерева *vertexIndexes*[i];

функції та процедури:

W(<номера вершин *i* та *j*> або <ребро *e*>*)* – повертає вагу ребра *i-j* або *e*;

Min(<номер вершини *i*>) – повертає такий номер вершини *j*, що ребро *i-j* – найменше з усіх ребер, таких, що корні дерев вершин *i* та *j* різні;

CheapestEdgeToTree(<номер дерева>, <масив ребер>) – повертає ребро, яке є найдешевшим з усіх ребер з масиву <масив ребер>, таких, що інцидентні вершинам дерева з номером <номер дерева>;

UpdateData(<масив ребер >) – включає ребра з масиву <масив ребер > у ґраф. Ребра копіюються в множину *T*, змінюється інформація про корні дерев вершин відповідно до створених новими ребрами дерев. Тобто фактично оновлюються масиви *trees* і *vertexIndexes*;

OutTree(<множина ребер T>*)* – виводить результат роботи алгоритму в файл;

умови:

α – істина, якщо *edgesAmount<N-1*;

μ – істина, якщо *i<N*;

оператори:

FindTempEdges =

$$= (i := 0) * \{ \underset{\mu}{} tempEdges_i := Min(i) * (i++) \};$$

FindCheapestEdges =

$$= (i := 0) * \{ \underset{\mu}{} cheapestEdges_i :=$$

$$:= CheapestEdgeToTree(i, tempEdges) *$$

$$* (i++) \};$$

UpdateData = UpdateData(cheapestEdges);

OutTree = OutTree(T);

Виконання всього алгоритму можна записати так:

$$Init() * \{ \underset{\alpha}{} (i := 0) * \{ \underset{\mu}{} tempEdges_i := Min(i) * (i++) \} *$$

$$* (i := 0) * \{ \underset{\mu}{} cheapestEdges_i :=$$

$$:= CheapestEdgeToTree(i, tempEdges, vertexIndexes) *$$

$$* (i++) \} * UpdateData(cheapestEdges) \} * OutTree(T).$$

Ця схема відображає всі етапи знаходження мінімального вкривного дерева. Згрупувавши в оператори кожен етап, отримаємо загальний вигляд схеми:

$$Init * \{ \underset{\alpha}{} FindTempEdges * FindCheapestEdges * \quad (1)$$

$$* UpdateData \} * OutTree(T).$$

Для детального опису операторів введемо додаткові позначення, наведені нижче.

Величини:

weight – довжина ребра;

weightArray[0…N-1] – масив для зберігання ваги ребра. Вага ребра, інцидентного вершині з номером *i* записується в комірку $weightArray_i$;

умови:

γ – істина, якщо $j<N$;

β – істина, якщо вершини з номерами *i,j* мають однаковий корінь дерева;

ω – істина, якщо $trees_i.size > 0$;

ρ – істина, якщо $weightArray_i > W(i,j)$;

ρ_1 – істина, якщо $weightArray_i > W(tempEdges_i)$;

λ – істина, якщо $vertexIndexes_j > vertexIndexes_i$.

оператори та функції:

InitWeightArray =

$$= (i := 0) * \underset{\mu}{\{} weightArray_i := \infty\} * (i++);$$

InitEdgesArray =

$$= (i := 0) * \underset{\mu}{\{} weightArray_i := \varnothing\} * (i++);$$

UpdateWeightIJ = $weightArray_i := W(i,j)$;

UpdateWeight = $weightArray_i := W(tempEdges_i)$;

MemTempEdge = $tempEdges_i := (i\text{-}j)$;

MemEdge = $cheapestEdges_i := (i\text{-}j)$;

NameVertexes(<множина>,<j>) =

$$= (i := 0) * \underset{i<_{множина.size}}{\{} vertexIndexes_{множина_i} := j * (i++)\};$$

DelPlural(<множина>) = множина:= $\varnothing$;

UnitePlurals(<множина1>< множина2>) =
(множина1:=множина1 $\cup$ множина2)*
* *DelPlural*(множина1);

UniteTrees(<edge i-j>) =

$$= \underset{i<j}{(} UnitePlurals(trees_i, trees_j) *$$

$$* NameVertexes(trees_i, j)$$

$$\vee UnitePlurals(trees_j, trees_i) *$$

$$* NameVertexes(trees_j, i));$$

<u>*FindTempEdges*</u> =

$$= InitWeightArray * (i := 0) * \underset{\mu}{\{} (j := 0) *$$

$$* \underset{\gamma\ \rho\wedge\bar{\beta}}{\{} (UpdateWeightIJ * MemTempEdge \vee E) *$$

$$* (j++)\} * (i++)\};$$

<u>*FindCheapestEdges*</u> =

$$= InitWeightArray * (i := 0) * \underset{\mu\ \omega}{\{} ((j := 0) *$$

$$* \underset{\gamma\ \rho_1}{\{} (UpdateWeight * MemEdge \vee E) *$$

$$* (j++)\} \vee E) * (i++)\};$$

<u>*UpdateData*</u> =

$$= (i := 0) * \underset{\mu}{\{} UniteTrees(cheapestEdges_i) * (i++)\} *$$

$$* InitEdgesArray.$$

Схема послідовного алгоритму в розгорнутому вигляді

$$Boruvka = Init * InitWeightArray * (i := 0) *$$

$$* \underset{\mu}{\{} (j := 0) * \underset{\gamma\ \rho\wedge\bar{\beta}}{\{} (UpdateWeightIJ *$$

$$* MemTempEdge \vee E) * (j++)\} * (i++)\} *$$

$$* InitWeightArray * (i := 0) * \underset{\mu}{\{} (j := 0) *$$

$$* \underset{\gamma\ \omega\wedge\rho_1}{\{} (UpdateWeight * MemEdge \vee E) *$$

$$* (j++)\} * (i++)\} * (i := 0) *$$

$$* \underset{\mu}{\{} UniteTrees(cheapestEdges_i) * (i++)\} *$$

$$* InitEdgesArray * OutTree(T).$$

Оцінка часу виконання послідовних операторів.

Оператор *FindTempEdges* являє собою два вкладених цикли від 0 до N, в яких виконується 2 перевірки і кілька операцій, що виконуються сталий час. Отже, даний оператор має складність $O(N^2)$, тобто час виконання A_1N^2, де A_1 – коефіцієнт, що приблизно дорівнює часу перевірки умови $\rho \wedge \bar{\beta}$.

Розглянемо частоту виконання умови $\rho \wedge \bar{\beta}$. Частота істинності умови ρ залежить від вхідних даних і в середньому константна f_ρ. З кожною ітерацією основного циклу програми кількість дерев у графі зменшується як мінімум вдвічі, отже, умова $\bar{\beta}$ за час прогонки циклів в середньому є істиною $2N/\log_2 N$ разів. Тобто додатковий час виконання операторів при істинності $\rho \wedge \bar{\beta}$ дорівнює $2f_\rho A_{12} N/\log_2 N$. Ріст цієї функції слабший за A_1N^2. Отже, час виконання оператора на великих графах можна записати як

$$t_{1serial} = A_1 \cdot N^2. \qquad (2)$$

Оператор *FindCheapestEdges* – два вкладених цикли, перший з яких виконує *N* разів перевірку умови ω. Коли ω – істина виконується вкладений цикл з N ітерацій, кожна з яких виконується в середньому константний час.

Частота істинності ω обчислюється з тих самих міркувань, що й $\bar{\beta}$ і теж в середньому пропорційна $N/\log_2 N$. Отже, складність виконання оператора $O(N^2/\log N)$. Час можна записати з деяким коефіцієнтом A_2:

$$t_{2serial} = \frac{A_2 \cdot N^2}{\log N}. \qquad (3)$$

Складність алгоритму можна оцінити як $O(N^2 \log N)$. Для великих графів можна записати час виконання алгоритму як суму (2) і (3), помножену на кількість ітерацій головного циклу:

$$t_{serial} = (t_{1serial} + t_{2serial}) \cdot \log N = \\ = N^2 \cdot (A_1 + A_2 / \log N) \cdot \log N \approx A \cdot N^2 \log N. \qquad (4)$$

Як бачимо, така реалізація не забезпечує складності $O(N\log N)$, яку можуть забезпечити найкращі реалізації алгоритму Борувки, проте така схема зручна для розпаралелювання і дає змогу мінімізувати затрати на синхронізацію процесів.

Принцип розподілу даних в паралельній версії алгоритму.

Для ефективного розпаралелювання загальної схеми (1) потрібно мінімізувати різницю часу обробки даних різними процесами. Для цього потрібно розбити дані між процесами на рівні частини.

Кожен процес обробляє тільки відведену йому частину даних.

В програмі множина вершин графа розбита на приблизно рівні частини, і кожен процес обробляє певну частину множини.

Для розбиття множини між процесами обчислюється початковий і кінцевий індекси, відповідно $\text{beginIndex}_{\text{rank}}$ та $\text{endIndex}_{\text{rank}}$ включно. Для обчислення цих індексів використовуються такі співвідношення:

$$beginIndex_{rank} = rank \cdot \left\lfloor \frac{N}{size} \right\rfloor + k_1;$$

$$k_1 = \begin{cases} rank, & rank < N(\bmod size); \\ N(\bmod size), & rank \geq N(\bmod size); \end{cases}$$

$$beginIndex_{rank} = (rank+1) \cdot \left\lfloor \frac{N}{size} \right\rfloor + k_2;$$

$$k_2 = \begin{cases} rank, & rank < N(\bmod size); \\ N(\bmod size) - 1, & rank \geq N(\bmod size). \end{cases}$$

Розпаралелювання алгоритму за допомогою САА

Для розподілення даних за зазначеним вище принципом необхідно ввести додаткові позначення для розбиття циклів та розподілу обчислень між процесами за допомогою асинхронної диз'юнкції.

Умова μ для N = 10 *(0≤i<10)* виглядатиме для двох процесів як $\mu = \mu_0 \vee \mu_1$, де μ_0*=(0≤i<5),* μ_1*=(5≤i<10)*. Також необхідно ввести позначення для реалізації обміну даними між процесами.

Розпаралелювання програми виконано за рахунок розпаралелювання циклів операторів *FindTempEdges* та *FindCheapestEdges*.

Додаткові позначення:

величини:

rank – процес, в якому виконується екземпляр програми, набуває значень [*0..size-1*];

size – загальна кількість процесів;

$beginIndex_{rank}$ – нижня межа діапазону даних, що обробляються процесом з номером *rank*;

$endIndex_{rank}$ – верхня межа діапазону даних, що обробляються процесом з номером *rank*.

умови:

μ_{rank} = *(*$beginIndex_{rank} \leq i \leq endIndex_{rank}$*)*, при цьому $\mu = \bigvee_{rank=0}^{size-1} \mu_{rank}$.

фільтри:

φ_r = *(rank == r)* якщо *r == all*, то наступні дії виконуються всіма процесами одночасно.

функції та процедури:

Exchange(array) – обмін даними масиву array між процесами. Кожен процес надсилає всим іншим частину масиву в діапазоні індексів [$beginIndex_{rank}$ .. $endIndex_{rank}$]. При цьому всі процеси зупиняють обчислення й чекають, поки не буде закінчено обмін повідомленнями. Таким чином відбувається синхронізація процесів;

SendToAll(data) – надсилання даних *data* всім процесам;

Rcv(data, rank) – отримання даних *data* з процесу *rank*.

Паралельні версії операторів

Читання файлу та ініціалізація даних виконується нульовим процесом як в послідовній версії, потім всі ініціалізовані дані надсилаються всім іншим процесам. Вивід в файл результату виконується тільки нульовим процесом

$$Init = \underline{\varphi_0} * Init * SendToAll(N, Weights) \dot{\vee} \underline{\varphi_{all}} * \\ * Rcv(N, Weights, 0);$$

$$OutTree = \underline{\varphi_0} * OutTree(T).$$

В процесі з номером *rank* виконання оператора *FindTempEdges* виглядає так:

$FindTempEdges_{rank}$ =

$$= (i := beginIndex_{rank}) * InitWeightArray * \\ * \{(j := 0) * \{ \underset{\mu}{(} \underset{\gamma\ \rho \wedge \overline{\beta}}{)} UpdateWeightIJ * \\ * MemTempEdge \vee E) * (j++)\} * (i++)\} * \\ * Exchange(tempEdges).$$

Скориставшись властивістю асинхронної диз'юнкції, запишемо виконання цього оператора в процесах *0…size-1*:

FindTempEdges =

$$= \underline{\varphi_0} * FindTempEdges_0 \dot{\vee} \underline{\varphi_1} * FindTempEdges_1 \dot{\vee} ... \\ ... \dot{\vee} \underline{\varphi_{size-1}} * FindTempEdges_{size-1} = \\ = \dot{\bigvee_{rank=0}^{size-1}} \underline{\varphi_{rank}} * FindTempEdges_{rank} = \\ = \dot{\bigvee_{rank=0}^{size-1}} \underline{\varphi_{rank}} * (i := beginIndex_{rank}) * \\ * InitWeightArray * \underset{\mu_{renk}}{\{} (j := 0) * \\ * \{ \underset{\gamma\ \rho \wedge \overline{\beta}}{(} UpdateWeightIJ * MemTempEdge \vee E) * \\ * (j++)\} * (i++)\}.$$

Аналогічно

$$\textbf{\textit{FindCheapestEdges}} =$$

$$= \bigvee_{rank=0}^{size-1} \underline{\varphi_{rank}} * InitWeightArray * (i := beginIndex_{rank}) *$$

$$* \underset{\mu_{rank}}{\{} (j := 0) * \{ \underset{\gamma\ \omega \wedge \rho_1}{(} UpdateWeight * MemEdge \vee E) *$$

$$* (j++)\} * (i++)\}.$$

Оператор *UpdateData* виконується послідовно й однаково на всіх процесах одночасно. Цей оператор виконує цикл, результат кожної наступної ітерації якого залежить від результату попередньої ітерації. Отже, при розпаралелюванні процесам необхідно було б обмінюватись даними при кожній ітерації циклу, що значно сповільнює виконання програми. Отже, розпаралелювання цього оператора недоцільне.

Одна з паралельних версій алгоритму може бути записаною так:

$$\begin{aligned} &Boruvka = \underline{\varphi_0} * Init * SendToAll(N,W) \vee \underline{\varphi_{all}} * \\ &* Rcv(N,W,0) * \{ \underset{\alpha}{\bigvee_{rank=0}^{size-1}} \underline{\varphi_{rank}} * FindTempEdges_{rank} * \\ &* \underline{\varphi_{all}} * Exchange(tempEdges) * \\ &* \bigvee_{rank=0}^{size-1} \underline{\varphi_{rank}} * FindCheapestEdges_{rank} * \\ &* \underline{\varphi_{all}} * Exchange(cheapestEdges) * \\ &* \underline{\varphi_{all}} * UpdateData\} * \underline{\varphi_0} * OutTree(T). \end{aligned} \quad (5)$$

Оцінка часу виконання паралельних операторів

Час виконання паралельних операторів складається з часу виконання паралельних складових окремо на кожному процесі та часу синхронізації.

Час виконання паралельних складових обчислюється аналогічно до послідовних. Якщо ступінь паралелізму (кількість процесів) дорівнює *size*, то час $t_{parallel}$ паралельного виконання алгоритму на ідеальній паралельній системі дорівнює

$$t_{parallel} = \frac{t_{serial}}{size}.$$

На реальних паралельних системах існують ще витрати на синхронізацію процесів. Позначимо його $t_d(N, size)$. Отже, час роботи паралельного алгоритму

$$t_{parallel} = \frac{A \cdot N^2 \log N}{size} + t_d(N, size).$$

Для оцінки t_d застосовано модель Хокні[10], згідно з якою комунікаційні часові витрати описуються формулою

$$t_d = \alpha + m/\beta,$$

де α – латентність мережі (час, необхідний на підготовку до передачі повідомлення), m – розмір повідомлення, β – пропускна здатність мережі.

В даній схемі за весь час виконання алгоритму передача повідомлень здійснюється в кожній ітерації алгоритму, тобто $\log_2 N$ разів. Процеси передають дані по черзі, отже, латентність мережі треба помножити на кількість процесів. Розмір повідомлення $m=CN$, де C – розмір одного елементу масиву даних. Отже, можна записати:

$$t_d = \log_2 N(size \cdot \alpha + C \cdot N/\beta);$$

$$t_{parallel} = \frac{A \cdot N^2 \log N}{size} + \log N(size \cdot \alpha + C \cdot N/\beta). \quad (6)$$

Для заданої кількості вершин N функція часу виконання (6) алгоритму має екстремум по параметру *size* в точці

$$size_{opt} = \sqrt{\frac{A}{\alpha}} \cdot N. \quad (7)$$

Тобто для кожного розміру графа N можна існує така оптимальна кількість паралельних задач $size_{opt}$, для якої час виконання алгоритму буде найменшим. Підставивши (7) в (6) отримано оптимальний час виконання алгоритму пропорційний $N\log N$:

$$t_{opt} = 2\sqrt{\alpha A} N \log N + \frac{CN}{\beta}.$$

В паралельній схемі (5) функція Exchange, яка виконує синхронізацію, викликається двічі за одну ітерацію і це збільшує час синхронізації.

При великому ступені паралелізму, можливо, доцільно виконувати синхронізацію процесів тільки в одному місці, а один з двох розпаралелених операторів залишити послідовним. З таких міркувань можна навести ще одну паралельну схему:

$$\begin{aligned} &Boruvka = \underline{\varphi_0} * Init * SendToAll(N,W) \vee \underline{\varphi_{all}} * \\ &* Rcv(N,W,0) * \{ \underset{\alpha}{\bigvee_{rank=0}^{size-1}} \underline{\varphi_{rank}} * FindTempEdges_{rank} * \\ &* \underline{\varphi_{all}} * Exchange(tempEdges) * \\ &* \underline{\varphi_{all}} * FindCheapestEdges * \\ &* \underline{\varphi_{all}} * Exchange(cheapestEdges) * \\ &* \underline{\varphi_{all}} * UpdateData\} * \underline{\varphi_0} * OutTree(T). \end{aligned} \quad (8)$$

Результати обчислювальних експериментів

Було написано програми за двома наведеними схемами розбиття алгоритму на паралельні задачі і за допомогою кожної з схем обчислювались мінімальні вкривні дерева для випадкових графів.

В таблицях 1 та 2 наведено виміри часу виконання алгоритму від кількості ядер та розміру графа відповідно для схем (5) та (8).

Таблиця 1. Експериментальні результати часу обчислень для схеми (5) з подвійною синхронізацією(сек.)

Кількість ядер / Розмір графа	1	2	4	6	8	12	16	20
1000	0,059	0,042	0,016	0,047	0,063	0,078	0,099	0,140
2000	0,229	0,151	0,078	0,104	0,099	0,114	0,104	0,172
3000	0,547	0,313	0,167	0,214	0,163	0,177	0,239	0,271
4000	0,97	0,52	0,29	0,38	0,27	0,29	0,54	0,37
5000	1,50	0,78	0,41	0,53	0,39	0,43	0,54	0,53
6000	2,08	1,08	0,61	0,65	0,53	0,62	0,76	0,50
7000	2,99	1,54	0,80	0,96	0,72	0,73	0,95	0,53
8000	3,21	2,00	1,08	1,07	0,89	0,69	0,87	1,04
9000	4,81	2,49	1,31	1,28	1,11	0,94	1,00	0,97
10000	5,75	2,91	1,50	1,47	1,29	1,08	1,07	1,05
20000	23,9	12,6	6,9	4,76	4,03	3,10	2,59	2,68
50000	157	84	43	30	24	15,7	13,2	11,0
100000	659	345	178	119	84	61,5	55,3	39,2
150000	1542	806	422	278	209	144	136	88
200000	2894	1502	806	518	395	270	231	165

Таблиця 2. Експериментальні результати часу обчислень для схеми (8) з одинарною синхронізацією(сек)

Кількість ядер / Розмір графа	1	2	4	6	8	12	16	20
1000	0,059	0,037	0,031	0,031	0,041	0,063	0,078	0,110
2000	0,229	0,146	0,078	0,110	0,094	0,099	0,120	0,156
3000	0,547	0,297	0,183	0,224	0,151	0,229	0,151	0,193
4000	0,97	0,53	0,28	0,34	0,23	0,32	0,40	0,53
5000	1,50	0,76	0,39	0,44	0,39	0,52	0,64	0,62
6000	2,08	1,09	0,58	0,70	0,48	0,56	0,69	0,61
7000	2,99	1,57	0,82	0,87	0,62	0,58	0,89	0,94
8000	3,21	1,82	0,97	1,04	0,84	0,71	1,09	0,92
9000	4,81	2,47	1,27	1,39	1,08	0,89	0,95	1,13
10000	5,75	2,97	1,48	1,50	1,03	0,97	0,87	1,09
20000	23,9	12,4	6,21	4,73	3,93	3,03	2,68	2,15
50000	157,2	84,7	41,7	28,8	23,3	15,7	12,4	10,2
100000	659	344	171	118	88	63	52	39
150000	1542	801	402	275	208	142	121	88
200000	2894	1492	757	515	386	271	217	164

Час вимірювався в програмі за допомогою функції *clock()* модуля *ctime*. У виміряний час не входить час читання графа з файлу та вивід результату. Було виміряно тільки час обчислення та збереження в оперативну пам'ять мінімального вкривного дерева.

Кластерні обчислення проведено для графів з кількістю вершин від 1000 до 200 000. Було проведено обчислення на 3 випадкових графах для кожної кількості вершин, а результат взято як середнє арифметичне з 3 результатів.

Для обчислень використано кластер Інформаційно-обчислювального центру Київського національного університету ім. Тараса Шевченка.

Виміри проведено на кластері розміром 1…5 вузлів. Кожен вузол кластера має 4-ядерний процесор, обчислення проведено на кількості ядер 2…20. На кожному ядрі запускався 1 процес.

Критерій ефективності виконання алгоритму – залежність часу виконання та прискорення порівняно з виконанням на алгоритму в одному процесі від кількості вершин графа та кількості

паралельних задач (ядер задіяних процесорів).

Для кожної кластерної системи коефіцієнт A є сталим і залежить від швидкодії процесорів. Латентність мережі α буде різною двох випадків:

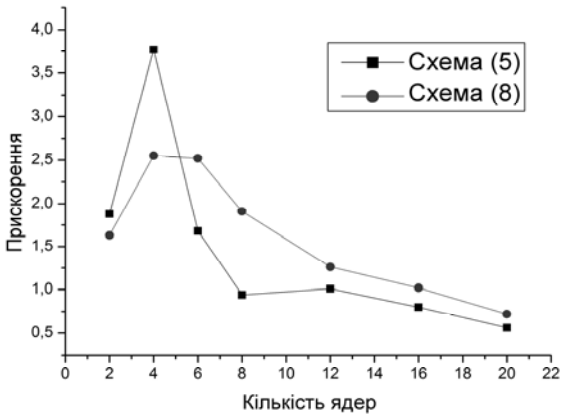
– всередині одного процесора (в таблицях до 4 ядер);
– між комп'ютерами.

В першому випадку (до 4 ядер) латентність мережі α нехтувано мала. В другому латентність мережі може складати десятки мілісекунд і синхронізація вносить суттєвий внесок в час виконання алгоритму.

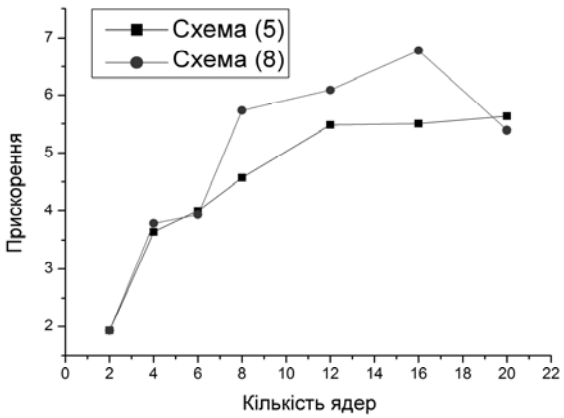
Час обчислення мінімального вкривного дерева на кластері з 20 ядрами – порядку хвилини для кількості сенсорів понад 50 тис.. Цей час на порядок менше автономної роботи сенсорних мереж.

Для порівняння схем наведемо графіки прискорення виконання алгоритму (в порівнянні з виконанням на одному ядрі) від кількості вершин.

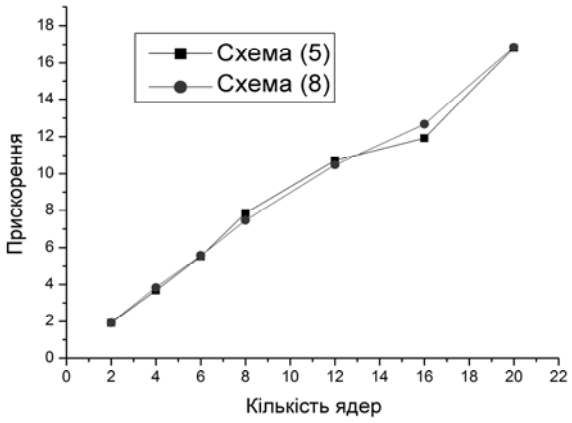
Для розрахунку малих ґрафів (порядку 1000 вершин, рис 3а) в межах одного процесора схема (5) з подвійною синхронізацією показала кращий результат ніж схема (8). Це можна пояснити тим, що в схемі (5) паралельні гілки менше навантажені, а передача даних в межах одного процесора виконується дуже швидко.



а



б



в

Рисунок 3 а – Залежність прискорення від кількості ядер для 1000 вершин, б – Залежність прискорення від кількості ядер для 10 000 вершин, в – Залежність прискорення від кількості ядер для 100 000 вершин

Для розрахунку ґрафів розміром порядку кількох тис. вершин на великій кількості ядер схема (8) з одинарною синхронізацією показала в середньому дещо кращий результат (рис 3б).

Для розрахунку ґрафів від 20 тис. вершин і більше однаково підходять обидві схеми. Експеримент не показав суттєвої різниці їх ефективності для великих ґрафів. В загальному обидві наведені схеми (5 і 8) добре підходять для розрахунку мінімального вкривного дерева на паралельних архітектурах.

Висновки

1. В статті запропоновано підхід до вирішення задачі побудови надійних безпровідних сенсорних мереж на основі паралельної реалізації алгоритму Борувки. Виконано формалізацію алгоритму Борувки з використанням математичного апарату САА та отримано послідовну схему. Для паралельного виконання алгоритму виконано його модифікацію.

2. За допомогою еквівалентних перетворень послідовної схеми отримано дві паралельні схеми алгоритму. Одна з них розрахована на максимальне розпаралелювання задач і зменшення навантаження на паралельні гілки. Друга ґрунтується на мінімізації часу синхронізації гілок, але передбачає дещо більше навантаження на кожну з них.

3. Проведено аналіз схем та обчислено складність модифікованого алгоритму, проведено оцінку часу його виконання на паралельних архітектурах в залежності від розміру вхідних даних та ступеню паралелізму. На основі аналізу алгоритму отримано формулу для визначення оптимального ступеню паралелізму для будь-якого розміру вхідного ґрафа.

4. Отримані схеми було використано як теоретичну базу та створено програми для обчислень

на кластерах. Проведено експериментальні обчислення. Аналіз результатів обчислень показує, що паралельні програми можна використовувати для швидкої автоматичної конфігурації сенсорних мереж. Це забезпечить їх стабільну роботу.

Список використаної літератури

1. Погорілий С.Д. Оптимізація алгоритмів маршрутизації з використанням систем алгоритмічних алгебр / С.Д. Погорілий, Д.М. Калита // УсиМ. – 2000. – № 4. – С. 20 – 30.
2. Погорілий С.Д. Програмне конструювання: підручник / С.Д. Погорілий; за ред. О.В. Третяка. – 2-е вид. – К.: Видавничо-поліграфічний центр «Київський університет», 2005. – 483 с.
3. Форум: Message Passing Interface Forum. – Режим доступу: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
4. Богачев. К.Ю. Основы параллельного программирования / К.Ю. Богачев. – М.: БИНОМ. Лаборатория знаний, 2003. – 342 с.
5. Sohraby K. Wireless sensor networks: technology, protocols, and applications / K. Sohraby, D. Minoli, T. Znati. – John Wiley and Sons, 2007. – 307 p.
6. Culler D. Guest Editors' Introduction: Overview of Sensor Networks / D. Culler, D. Estrin, M. Srivastava // Computer. – 2004. – Vol. 37, № 8. – P. 41-49.
7. WiseNET: An Ultralow-Power Wireless Sensor Network Solution / C.C. Enz, A. El-Hoiydi, J.-D. Decotignie et al. // Computer. – 2004. – Vol.37. – № 8. – P. 62-70.
8. Седжвик Р. Фундаментальные алгоритмы на С++: в ч. Ч. 5. Алгоритмы на ґрафах / Р. Седжвик ; пер с англ. – СПб.: ООО «ДиаСофтЮП», 2002. – 496 с.
9. Седжвик Р. Фундаментальные алгоритмы на С++. Анализ. Структуры данных. Сортировка. Поиск / Р. Седжвик ; пер с англ. – К: Издательство «ДиаСофт», 2001. – 688 с.
10. Hockney R. The communication challenge for MPP: Intel Paragon and Meiko CS-2 / R. Hockney // Parallel Computing. – 1994. – Vol. 20. – № 3. – P. 389 – 398.

Надійшла до редколегії 29.03.2012

С.Д. ПОГОРЕЛЫЙ, И.А. ВЫСОЦКИЙ
Киевский национальный университет им. Тараса Шевченка

ИСПОЛЬЗОВАНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ ДЛЯ ПОСТРОЕНИЯ БЕСПРОВОДНЫХ СЕНСОРНЫХ СЕТЕЙ

Предложено использование параллельной реализации вычисления минимального остовного дерева на примере алгоритма Борувки для построения надёжных беспроводных сетей. Выполнено модификацию алгоритма для параллельных систем и его формализацию с использованием математического аппарата систем алгоритмических алгебр В.М. Глушкова. Использована концепция распараллеливания по данным для архитектур с разделенной памятью. Проведено преобразование алгоритма, получена параллельная схема. Реализован параллельный алгоритм с использованием языка программирования С++ и технологии MPI. Проведены кластерные экспериментальные вычисления.

Ключевые слова: сенсорные беспроводные сети, минимальное остовное дерево, алгоритм Борувки, параллельные вычисления, кластерные вычисления, раздельная память, системы алгоритмических алгебр В. М. Глушкова, MPI.

S.D. POHORILYY, I.A. VYSOTSKYI
Taras Shevchenko National University of Kyiv

WIRELESS SENSOR NETWORKS CONSTRUCTION USING PARALLEL COMPUTING

The use of the parallel implementation of computing the minimum spanning tree on the example of Boruvka's algorithm proposed to build wireless sensor networks. The modification of the algorithm for parallel systems and its formalization using the mathematical apparatus of systems of algorithmic algebras by V.M.Glushkov performed. Used the concept of data parallelization for the systems with separate memory. A parallel scheme by dint of algorithm conversation obtained. Parallel algorithm is implemented using the C++ programming language and MPI technology. The experimental cluster computing performed.

Keywords: wireless sensor networks, minimum spanning tree, Boruvka's algorithm, parallel computings, cluster computing, separate memory ,systems of algorithmic algebras by V.M. Glushkov, MPI.