Підвищення швидкодії алгоритму “проштовхування передпотоку”

Погорілий С.Д., Кузьмін О.В.
Київський національний університет імені Тараса Шевченка
sdp@univ.kiev.ua, dozz@ukr.net

Abstract

Pogoriliy S., Kuzmin O. Speedup "push-relabel" algorithm. *"Push-relabel" sequential algorithms are formalized, with regular schemes built. A method for paralleling is suggested and parallel regular schemes of algorithms are formalized and realized on cluster architectures.*

Вступ

Існує багато задач, які вирішуються за допомогою методів теорії графів, наприклад, задачі визначення максимального потоку в мережі. У завданні про максимальний потік стоїть задача знайти максимальну швидкість пересилання продукту від джерела до стоку, при якій не порушуватимуться обмеження пропускної спроможності. Це одне з простих завдань, що виникають у транспортних мережах. Більш того, основні методи можна застосовувати для рішення задачі на транспортних, комунікаційних, електричних мережах, при моделюванні різних процесів фізики і хімії, в деяких операціях над матрицями, для розв’язання споріднених задач теорії графів.

Основною вимогою до будь-яких алгоритмів є час виконання. Особливо це актуально для задач великої розмірності.

Одним зі способів зменшення часу виконання є розпаралелювання алгоритму або його частин з метою подальшої реалізації на кластерній архітектурі. Перетворити алгоритм на паралельний можна за допомогою досвіду, інтуїції або апарату еквівалентних перетворювань.

В роботі розглянуто алгоритм “проштовхування передпотоку” виконано його перетворення та оптимізацію за допомогою формалізованих схем алгоритмів Систем Алгоритмічних Алгебр (САА), та аналіз часу виконання на різних платформах.

Алгоритм “проштовхування передпотоку”

В алгоритмі виконуються дві основні операції: проштовхування надлишку потоку від вершини до однієї з сусідніх з нею і підйом вершини. Застосування цих операцій залежить від висот вершин, яким ми зараз дамо точніші визначення.

Нехай G=(V, E) – транспортна мережа з джерелом s і стоком t, а f – деякий передпотік в G. Функція h:V→N є функцією висоти, якщо h(s)=|V|, h(t)=0 і h(u)≤h(v)+1 для будь-якого залишкового ребра (u, v) є Ef.

Процедура Push працює таким чином. Передбачається, що вершина u має позитивний надлишок e(u) і залишкова пропускна спроможність ребра (u, v) позитивна. Тоді можна збільшити потік із u в v, якщо h(u)=h(v)+1, на величину df(u, v)=min(e(u), cf(u, v)), при цьому надлишок e(u) не стає негативним і не буде перевищена пропускна спроможність c(u, v). Таким чином, якщо функція f була передпотоком перед викликом процедури Push, вона залишиться передпотоком і після її виконання.

Основна операція Relabel(u) застосовується, якщо вершина u переповнена і h(u)≤h(v) для всіх ребер (u, v) є Ef. Іншими словами, переповнену вершину u можна піддати підйому, якщо всі вершини v, для яких є залишкова пропускна спроможність від u до v, розташовані не нижче u, так що проштовхнути потік з u не можна. (Ні джерело s, ні стік t не можуть бути переповнені; отже, ні s, ні t не можна піддавати підйому).

Операції проштовхування надлишку потоку і підйом вершини здійснюється доки не залишиться жодної переповненої вершини, після цього передпотік виявляється максимальним потоком [1].

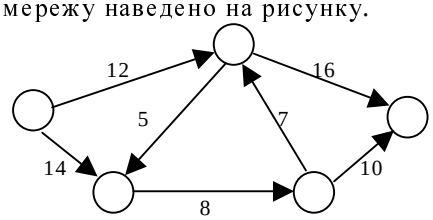
Алгоритм можна представити в такому вигляді:

```
while (e(v) ≠0: v є V)
{
 for (u=0;u<|V|;u++)
{
```

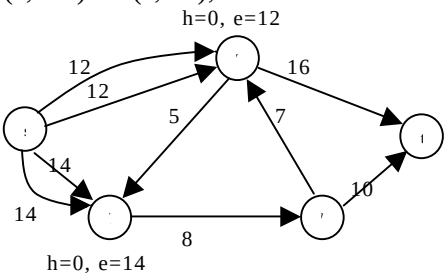
```
        for (v=0; v<|V|; v++)
        {
        if (e(u) ≠ 0 ∧ h(u)≤h(v))
                relabel (u);

        if(e(u)≠0 ∧ cf(u, v)>0 ∧ h(u)==h(v)+1)
                push(u,v);
        }
}
}
```

Приклад роботи алгоритму "проштовхування передпотоку"

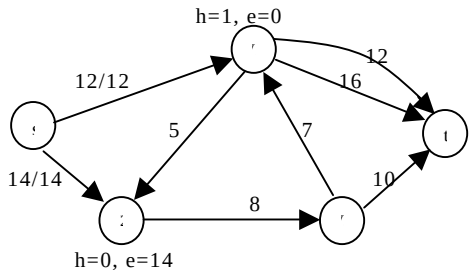
Нижче наведено приклад роботи алгоритму для мережі з 5 вузлів. Граф, що задає мережу наведено на рисунку.



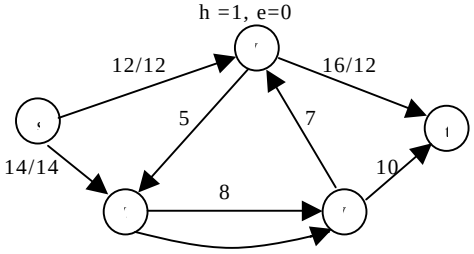
f(s, V1) = c(s,V1);
f(s, V2) = c(s,V2);



h(V1) = min(h(s),h(V2), h(t))+1=1;
d = min(c(V1,t), e(V1)) = 12;
f(V1, t) = f(V1, t) + d = 12;

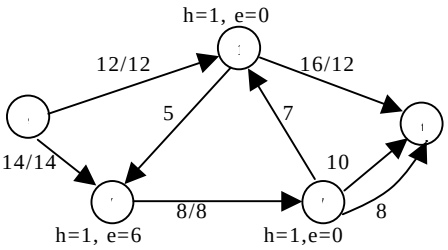


h(V2) = min(h(s), h(V3)) + 1 = 1;
d = min(c(V2, V3), e(V2)) = 8;
f(V2, V3) = f(V2, V3) + d = 8;

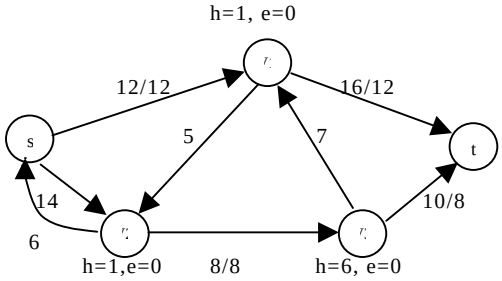


h=1, e=6 8 h=0 e=8

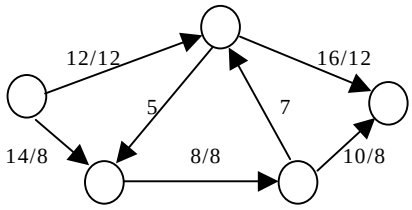
h(V3) = min(h(V1), h(t)) + 1 = 1;
d = min(c(V1,t), e(V3)) = 8;
f(V3, t) = f(V3, t) + d = 8;



h(V2) = min(h(s)) + 1 = 6;
d = min(c(V2,s), e(V2)) = 6;
f(V2, s) = f(V2, s) + = – 8;



В результаті отримали:



Представлення алгоритму "проштовхування передпотоку" у вигляді САА-схеми

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 та ін.).

Розглянемо деякі операції САА:

Ліве множення $(A \cdot \alpha)$ умови α на оператор A являє собою нову умову β таку, що $\beta(m) = (A \cdot \alpha)(m) =^{def} = \alpha(A(m))$.

*Композиція A*B* – послідовне виконання операторів A і B.

α – диз'юнкція $_{\alpha}(A \vee B)$ -- аналог мовою C – if (α) A else B;

α – ітерація $_{\alpha}\{A\}$ – while (α) A;

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 алгоритму.

Згідно теореми 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 і оптимізації вже не з алгоритмом, а з його САА-схемою[2].

Визначивши певну САА можна представити алгоритм "проштовхування передпотоку" у вигляді схеми на даній САА.

Множина операторів

• P:

$$(f(u,v)=\begin{cases} c(u,v), & u=s \\ -c(u,v), & v=s; h(u)=\begin{cases} |V|, & u=s \\ 0, & else \end{cases} \\ 0, & else \end{cases});$$

//ініціалізація передпотока

• RELABEL:

h(u)=1+min{h(v):(u,v)∈ E_f}; //операція підйому

• PUSH: (df = min(e(u), cf(u, v));

f(u, v) = f(u, v) + df;

f(v, u) = – f(u, v);

e(u) = e(u) – df;

e(v) = e(v) + df;); //операція проштовхування

• B: (flag=(e(u)!=0), u∈ V); //перевірка на наявність переповнених вершин

Множина умов:

• *j:* 0 < u < |V|; // u менше розміру матриці суміжностей графа

• *i:* 0 ≤ v ≤ |V|; // v менше розміру матриці суміжностей графа

• μ*:*e(u) ≠ 0; // u-не порожня вершина

• *p:* e(u) ≠ 0 ∧ cf(u,v)>0 ∧ h(u)==h(v)+1;

• β*:* flag ==1; //в графі ще є не порожні вершиниж

Алгоритм "проштовхування передпотоку" =

$$P^*_{\beta}\{_j\{_{\mu}(RELABEL,E)^*_i\{_p(PUSH,E)\}\}^*B\} \quad (1)$$

Це САА-схема послідовного алгоритму "проштовхування передпотоку" знаходження максимальний потік в транспортній мережі.

Розпаралелювання алгоритму "проштовхування передпотоку"

При роботі з великими об'ємами даних, наприклад при використанні матриць великої розмірності для підвищення ефективності роботи програми більш вигідним є розпаралелювання за даними. В такому випадку більш значного підвищення ефективності роботи програми можна досягти якщо розподілити роботу між паралельними процесами не за рахунок виконання одним процесом певної операції над одними і тими ж даними, а за рахунок виконання окремим паралельним процесом повної сукупності потрібних дій над своєю частиною даних.

Розподілити дані між різними паралельними процесами можна наступним чином. Нехай кожен процес обробляє окрему частину матриці суміжностей. З точки зору алгоритму це означає, що окремий процес працює зі своєю частиною вершин графа.

Розглянемо на прикладі алгоритму "проштовхування передпотоку" розподілення даних між різними паралельними процесами.

САА-схеми можна перетворити на САА-М схему, що отримані розширенням стандартних САА шляхом введення трьох додаткових операцій, орієнтованих на формалізацію паралельних обчислень.

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

$\underline{\alpha}$(m)= E(m), якщо α(m)=1, //E – тотожній оператор (E(m)=m)

N(m), якщо α(m)=0; //N – невизначений оператор

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

Асинхронна диз'юнкція. A||B – паралельне виконання A і B.

Отримаємо паралельний алгоритм, в якому використовуються описана концепція, за допомогою апарату еквівалентних перетворень. Для цього застосуємо формулу $_{\alpha}(A,E)=\underline{\alpha}\,A$:

$$P^*_{\beta}\{_j\{\underline{\mu}\,RELABEL^*_i\{\underline{p}PUSH\}\}^*B\} \quad (2)$$

Умовно позначимо:

$j=j_1 \wedge j_2 \wedge \ldots \wedge j_p$, (1)

j_i=(ll<u<rl), де ll=i*|V|/p, rl=ll+|V|/p, p- кількість процесів.

Величини ll і rl визначають межі зміни номерів вершин графа, з якими працює окремий процес.

Тоді (2) буде мати вигляд:

$$P^*_{\beta}\{_{j_1 \wedge j_2 \wedge ... \wedge j_p}\{\underline{\mu}RELABEL^*_i\{\underline{p}PUSH\}\}^* B\} \quad (3)$$

Позначимо $Z=\underline{\mu}RELABEL\ ^*_i\{\ \underline{p}PUSH\}$

Якщо умова циклу – кон'юнкція, то (3) можна представити в вигляді паралельної формули:

$$P^*_{\beta}\{\#_{j_1}\{Z\}\|_{j_2}\{Z\}\|...\|_{j_i}\{Z\}\|...\|_{j_p}\{Z\}\#^*B\} \quad (4)$$

В цій схемі вказівники ll та rl локальні для кожної гілки виконання. Символами "#...#" позначається частина алгоритму, оператори якої повинні виконуватися паралельно. А символами "||" розділяються оператори, що належать до паралельних гілок.

Цикл $_{j_i}\{Z\}$, що виконується процесом в такому випадку матиме наступний вигляд.

```
for (u=ll;u<rl;u++)
{
        if (e(u) ≠ 0)
                relabel (u);
        for (v=0; v<size; v++)
        {
        if(e(u)≠0 ∧ cf(u, v)>0 ∧ h(u)==h(v)+1)
                push(u,v);
        }
}
```

Паралельна програма, що використовує MPI

MPI – це стандарт на програмний інструментарій для забезпечення зв'язку між гілками паралельної програми.

Для MPI прийняте писати програму, що містить код всіх гілок відразу. MPI-завантажувачем запускається вказана кількість екземплярів програми. Кожен екземпляр визначає свій порядковий номер в запущеному колективі, і залежно від цього номера і розміру колективу виконує ту або іншу вітку алгоритму. Така модель паралелізму називається *Single program/Multiple data* (SPMD), і є окремим випадком моделі *Multiple instruction/Multiple data* (MIMD) [3].

Кожна гілка має власний простір даних, повністю ізольований від інших гілок. З цин пов'язана проблема обміну даних.

Кожна гілка, яка працює зі своєю частиною графа повинна знати і може змінювати інші частини, які належать іншим паралельним гілкам і до яких вона не має доступ. Тому після кожної ітерації потрібно виконувати обмін даними між всіма паралельними гілками. Отже для систем з розподіленою пам'яттю формула (4) повинна включати і обмін даними.

Позначимо E – оператор обміну даними що були зміненими в певній гілці. Тоді (4) буде мати вигляд:

$$P^*_{\beta}\{\#_{j_1}\{Z^*_{\chi}\{E\}\}\|_{j_2}\{Z^*_{\chi}\{E\}\}\|...\|_{j_p}\{Z^*_{\chi}\{E\}\}\#^*B\},$$

де χ=1,2…p.

Перепозначимо $M = Z^*_{\chi}\{E\}$, і остаточно отримаємо: о

$$P^*_{\beta}\{\#_{j_1}\{M\}\|_{j_2}\{M\}\|...\|_{j_p}\{M\}\#^*B\} \quad (5).$$

Схема (5) являє собою САА-М "проштовхування передпотоку", для систем із розподіленою пам'яттю.

У такий спосіб обчислювальні затрати для розв'язання задачі розподіляються між усіма процесами. Але в цьому випадку з'являються додаткові накладні витрати, що пов'язані з обміном даними між процесами при кожній ітерації циклу задачі.

Розпаралелювання алгоритму "проштовхування передпотоку" за допомогою MPI відбувається за наступною схемою:

1. Визначається кількість паралельних гілок.
2. Кожна гілка виконує свою частину коду.
3. Після кожної ітерації гілки обмінюються даними.
4. Після завершення ітерацій не обхідно вивести результат і завершую програму.

Розпаралелювання за допомогою MPI було реалізовано на обчислювальному кластері Інформаційно-Обчислювального центру Київського національного університету імені Тараса Шевченка та Microsoft кластері Київського національного університету імені Тараса Шевченка.

Залежність часу виконання алгоритму від розмірності матриці та кількості паралельних гілок зображені на рис. 1 для обчислювального кластера Інформаційно-Обчислювального центру Київського національного університету імені Тараса Шевченка, рис. 2. для Microsoft кластері Київського національного університету імені Тараса Шевченка.

Як бачимо з графіків, від збільшення кількості паралельних гілок час роботи програми не зменшується. Це пояснюється тим, що при використанні MPI потрібен додатковий час для обміну даними між процесами, та їх синхронізацією при кожній ітерації. Відмінність в графіках обумовлена різницею реалізації MPI для Windows та Linux платформ.

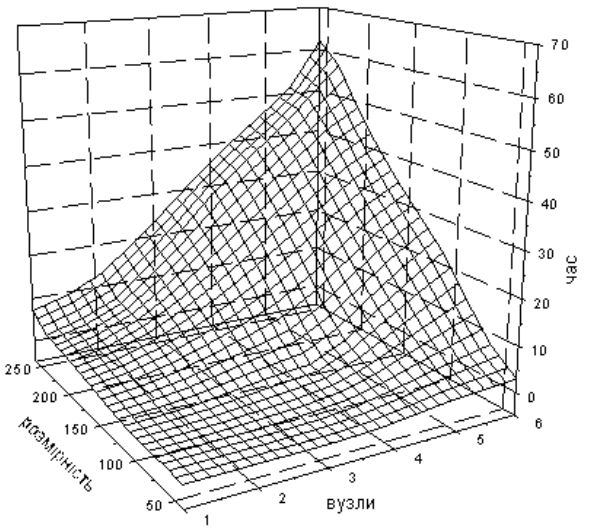


Рис. 1. Результати роботи програми, що використовує MPI на обчислювальному кластері Інформаційно-Обчислювального центру Київського національного університету імені Тараса Шевченка

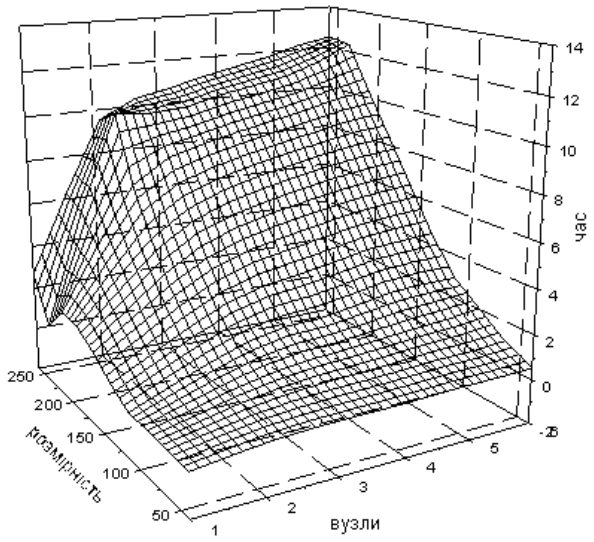


Рис. 2. Результати роботи програми, що використовує MPI на Microsoft кластері Київського національного університету імені Тараса Шевченка

Паралельна програма, що використовує потоки

Потік – це частина програми, що виконується паралельно. Оскільки потік – це частина програми, відповідно кожен потік має доступ до ресурсів пам'яті, що належать даній програмі. На перший погляд може здатися, що все просто, але в даному 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 в протилежному випадку. В бібліотечному файлі *pthread.h* описані об'єкти синхронізації типу *mutex* і функції для роботи з ними [4].

Оскільки паралельні потоки працюють зі спільними даними, то необхідно ввести додаткові оператори синхронізації:

L – оператор, що блокує доступ до простору спільних змінних для усіх паралельних гілок окрім поточної. Якщо ж даний ресурс вже заблоковано, то оператор очікує його звільнення.

U – оператор, що знімає блокування простору спільних змінних. Тоді формула (4) буде мати вигляд:

$$P^*_{\beta}\{\#_{j_1}\{L^*Z^*U\}\|_{j_2}\{L^*Z^*U\}\|..\|_{j_p}\{L^*Z^*U\}\#^*B\}$$

Перепозначимо $N = Z^*_{\chi}\{E\}$, і остаточно отримаємо:

$$P^*_{\beta}\{\#_{j_1}\{N\}\|_{j_2}\{N\}\|...\|_{j_p}\{N\}\#^*B\} \quad (6).$$

Розпаралелювання алгоритму "проштовхування передпотоку" на рівні паралельних потоків відбувається за наступною схемою:

1. Декларація потрібних змінних і матриці в динамічній пам'яті основної програми.
2. Створення необхідної кількості паралельних потоків.
3. В кожному потоці виконуються необхідні дії над даними, якщо доступ до них не заблоковано, в іншому випадку очікування моменту розблокування даних.
4. В основній програмі очікування завершення роботи всіх потоків.
5. Виведення результатів обчислень і завершення виконання основної програми.

Дані, щодо розпаралелювання за даною схемою зображені на рис.3.

Програма, що використовує потоки працює зі спільною пам'яттю, а отже, на відміну від MPI, не потребує додаткового часу для обміну даними після кожної ітерації. Тому при збільшенні потоків час виконання суттєво зменшується.

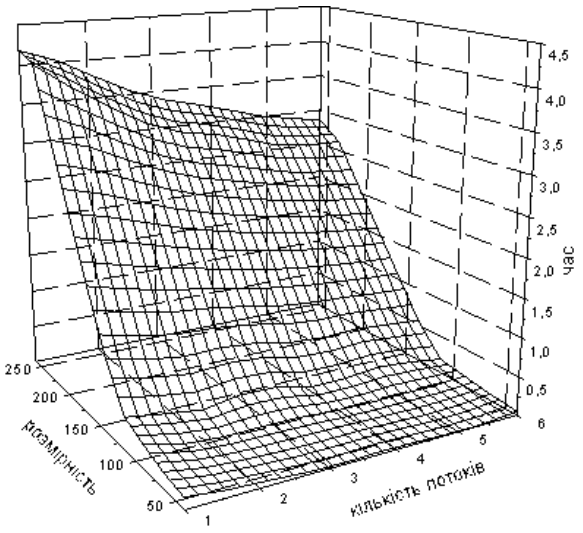


Рис.3. Залежність часу роботи алгоритму від кількості паралельних потоків

При розпаралелюванні на двох паралельних потоках середнє відносне зменшення часу роботи алгоритму складає 18.6%.

При розпаралелюванні на восьми паралельних потоків середнє відносне зменшення часу роботи алгоритму складає 48.8%.

Всновки

В роботі проведено аналіз алгоритму “проштовхування передпотоку” знаходження максимального потоку шляхом дослідження САА-схем алгоритмів. Побудовано схему послідовного алгоритму і отримано схему паралельного алгоритму. При розпаралелюванні використовувалась концепція паралелізму за даними. Реалізовано паралельний алгоритм “проштовхування передпотоку” на рівні паралельних потоків та MPI. Проведено аналіз результатів оптимізації алгоритму в залежності від способу реалізації та кількості паралельних потоків.

Отримані результати дають привід сподіватися на подальше підвищення ефективності роботи алгоритму за допомогою перетворення САА-схеми та знаходження більш ефективних шляхів розподілення даних між різними процесами.

Перелік посилань

1. Кормен Т., Лейзерсон Ч., Ривест Р. «Агоритмы, построение и анализ». М.:МЦНМО, 2000. – 1296 с.
2. Многоуровневое структурное проектирование программ: Теоретические основы, инструментарий / Е.Л. Ющенко, Г.Е. Цейтлин, В.П. Грицай, Т.К. Терзян. – М.: Финансы и статистика, 1989. – 208 с.
3. www.cluster.kiev.ua
4. К.Ю.Богачёв «Основы паралельного программирования» М.: БИНОМ. Лабаратория знаний, 2003. – 342 с.
5. http://www.csa.ru/~il/mpi_tutor/. (И.Евсеев «MPI для начинающих» Учебное пособие+примеры).
6. http://parallel.ru
7. Погорілий С.Д. Автоматизація наукових досліджень. Основоположні математичні відомості. Програмне забезпечення. За редакцією академіка АПН України Третяка О.В. К.: ВПЦ “Київський Університет”, 2002. - 290с.
8. Роберт Сейджвик «Фундаментальные алгоритмы на С++» часть 5 Алгоритмы на графах. СПб: ООО «ДиаСофтЮП», 2002. – 496.
9. http://www.citforum.ru/programming/c_unix/index.shtml. (Андрей Богатырев «Хрестоматия по программированию на Си в UNIX»).
10. Погорілий С.Д. Програмне конструювання. За редакцією академіка АПН України Третяка О.В. К.: ВПЦ "Київський університет", 2005. – 440 с.