Лекция: Методы параллельных вычислений

Методы параллельных вычислений

- Матричные вычисления
- Сортировка
- Обработка графов
- Оптимизация

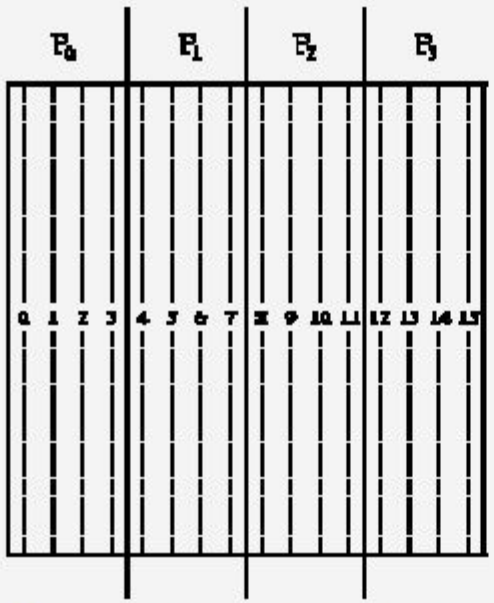
Способы распределения матриц…

Возможные способы распределения данных (частей матриц) по процессорам

- Ленточная (*striped*) или блочная (*checkerboard*) схема распределения
- Последовательный (*block*) или циклический (*cyclic*) способ группирования

Матричные вычисления

Способы распределения матриц…

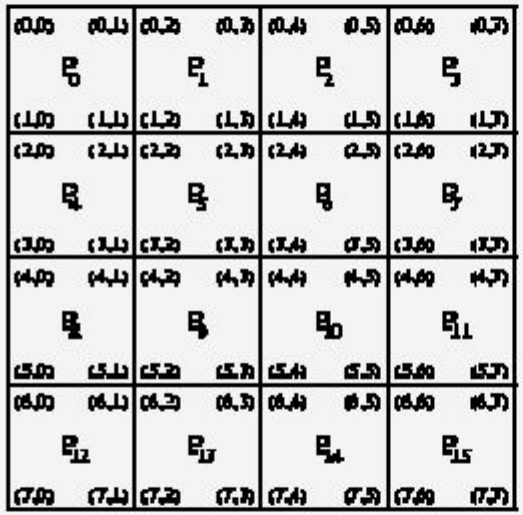


Распределение матрицы по столбцам

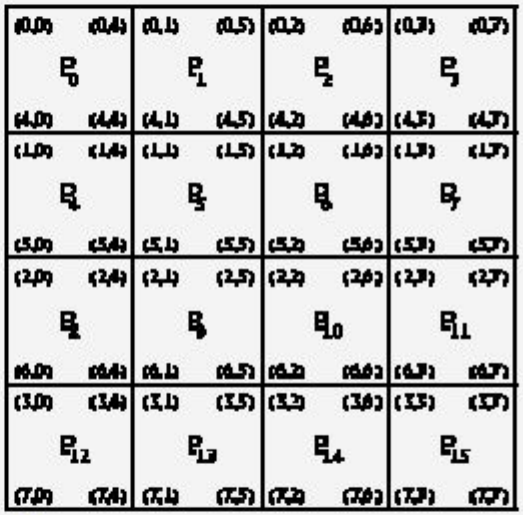
Распределение матрицы по строкам

Ленточная схема распределения матрицы по процессорам

Способы распределения матриц…



Блочное-последовательное распределение



Блочное-циклическое распределение

Блочная схема распределения матрицы по процессорам

Матричные вычисления

Задача перемножения матриц: *ленточная схема*

- Для матрица ***A*** используется горизонтальное разбиение, для матрицы ***B*** – вертикальное разбиение
- Размер полосы (количество строк или столбцов) - $n/\sqrt{p}$
- На процессор для вычислений передается по одной полосе матриц ***A*** и ***B***; при распределении данных каждая полоса матрицы ***A*** должна пересечься с каждой полосой матрицы ***B*** – всего ***p*** вариантов;
- Каждый процессор вычисляет блок результирующей матрицы ***C*** размером $n/\sqrt{p}$ на $n/\sqrt{p}$

Задача перемножения матриц: *ленточная схема*

Оценка эффективности

- Начальное распределение данных – ***p*** операций передачи объемом по $2n^2 / \sqrt{p}$
- Объем вычислений на каждом процессоре - $2n^3 / p$
- Завершающая передача блоков результирующей матрицы ***C*** - ***p*** операций передачи объемом по n^2 / p

Задача перемножения матриц: *блочный подход*

Операцию матричного умножения матриц ***A*** и ***B*** в блочном виде можно представить следующим образом:

$$\begin{pmatrix} A_{11}A_{12}...A_{1k} \\ \cdots \\ A_{k1}A_{k2}...A_{kk} \end{pmatrix} \times \begin{pmatrix} B_{11}B_{12}...B_{1k} \\ \cdots \\ B_{k1}B_{k2}...B_{kk} \end{pmatrix} = \begin{pmatrix} C_{11}C_{12}...C_{1k} \\ \cdots \\ C_{k1}C_{k2}...C_{kk} \end{pmatrix}$$

где каждый блок $\boldsymbol{C_{ij}}$ матрицы ***C*** определяется в соответствии с выражением

$$C_{ij} = \sum_{l=1}^{k} A_{il} B_{lj}$$

Задача перемножения матриц: *блочный подход*

- Размер блоков - $(n/\sqrt{p})x(n/\sqrt{p})$
- На процессор для вычислений передается по одному блоку матриц ***A*** и ***B***; при распределении данных блоки матрицы ***A*** встречаются однократно;
- В ходе вычислений блоки матриц передаются между процессорами (всего $\sqrt{p}$ итераций); схема коммуникационного взаимодействия соответствует топологии типа *двумерной прямоугольной решетки-тора*

Задача перемножения матриц: *алгоритм Фокса*

Пусть $p=k^2$, $n=mk$ и процессоры образуют логическую прямоугольную решетку размером $k \times k$ (обозначим через p_{ij} процессор, располагаемый на пересечении i строки и j столбца решетки).

Основные правила метода, известного как *алгоритм Фокса*, состоят в следующем:

- каждый из процессоров решетки отвечает за вычисление одного блока матрицы C;
- в ходе вычислений на каждом из процессоров p_{ij} располагается четыре матричных блока:
  - блок C_{ij} матрицы C, вычисляемый процессором;
  - блок A_{ij} матрицы A, размещенный в процессоре перед началом вычислений;
  - блоки A'_{ij}, B'_{ij} матриц A и B, получаемые процессором в ходе выполнения вычислений.

Матричные вычисления

Задача перемножения матриц: *алгоритм Фокса*

Выполнение метода включает следующие действия:

• **этап инициализации**, на котором на каждый процессор p_{ij} передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех процессорах;

• **этап вычислений**, на каждой итерации l, $1 \le l \le k$, которого выполняется:

- для каждой строки i, $1 \le i \le k$, процессорной решетки блок A_{ij} процессора p_{ij} пересылается на все процессоры той же строки i; индекс j, определяющий положение процессора p_{ij} в строке, вычисляется по соотношению

$$j = (i + l - 1) \bmod k + 1$$

(`mod` есть операция получения остатка от целого деления);

– полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждого процессора p_{ij} перемножаются и прибавляются к блоку C_{ij}

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij}$$

– блоки B'_{ij} каждого процессора p_{ij} пересылаются процессорам p_{ij}, являющимися соседями сверху в столбцах процессорной решетки (блоки процессоров из первой строки решетки пересылаются процессорам последней строки решетки).

Задача перемножения матриц: *алгоритм Фокса*

Последовательность действий в алгоритме Фокса ***(p=2×2):***

1 итерация

1)

		1	2
1	A'_{ij}	A_{11}	A_{11}
	B'_{ij}	B_{11}	B_{12}
	C_{ij}	0	0
2	A'_{ij}	A_{22}	A_{22}
	B'_{ij}	B_{21}	B_{22}
	C_{ij}	0	0

2)

	1	2
A'_{ij}	A_{11}	A_{11}
B'_{ij}	B_{11}	B_{12}
C_{ij}	$A_{11}B_{11}$	$A_{11}B_{12}$
A'_{ij}	A_{22}	A_{22}
B'_{ij}	B_{21}	B_{22}
C_{ij}	$A_{22}B_{21}$	$A_{22}B_{22}$

3)

	1	2
A'_{ij}	A_{11}	A_{11}
B'_{ij}	B_{21}	B_{22}
C_{ij}	$A_{11}B_{11}$	$A_{11}B_{12}$
A'_{ij}	A_{22}	A_{22}
B'_{ij}	B_{11}	B_{12}
C_{ij}	$A_{22}B_{21}$	$A_{22}B_{22}$

Матричные вычисления

Задача перемножения матриц: *алгоритм Фокса*

Последовательность действий в алгоритме Фокса ***(p=2×2):***

2 итерация

1)

		1	2
1	A'_{ij}	A_{12}	A_{12}
	B'_{ij}	B_{21}	B_{22}
	C_{ij}	$A_{11}B_{11}$	$A_{11}B_{12}$
2	A'_{ij}	A_{21}	A_{21}
	B'_{ij}	B_{11}	B_{12}
	C_{ij}	$A_{22}B_{21}$	$A_{22}B_{22}$

2)

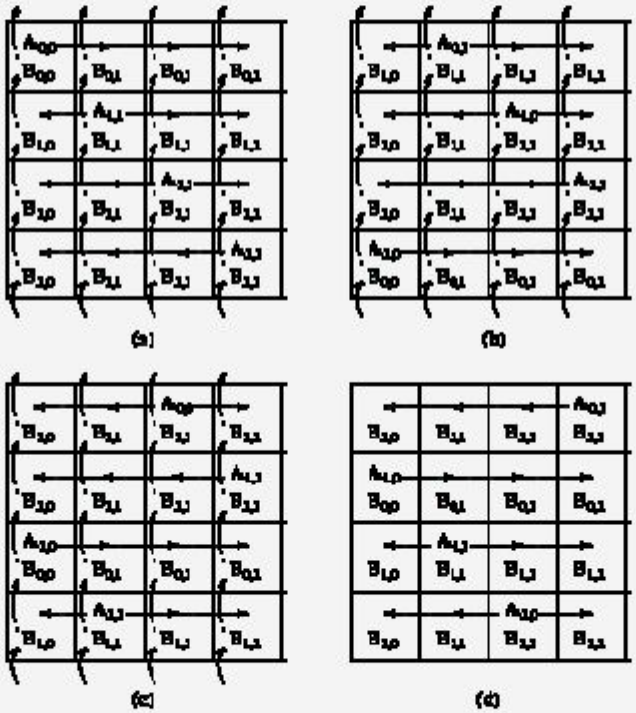
	1	2
A'_{ij}	A_{12}	A_{12}
B'_{ij}	B_{21}	B_{22}
C_{ij}	$A_{11}B_{11} + A_{12}B_{21}$	$A_{11}B_{12} + A_{12}B_{22}$
A'_{ij}	A_{21}	A_{21}
B'_{ij}	B_{11}	B_{12}
C_{ij}	$A_{22}B_{21} + A_{21}B_{11}$	$A_{22}B_{22} + A_{21}B_{12}$

3)

	1	2
A'_{ij}	A_{12}	A_{12}
B'_{ij}	B_{21}	B_{22}
C_{ij}	$A_{11}B_{11} + A_{12}B_{21}$	$A_{11}B_{12} + A_{12}B_{22}$
A'_{ij}	A_{21}	A_{21}
B'_{ij}	B_{11}	B_{12}
C_{ij}	$A_{22}B_{21} + A_{21}B_{11}$	$A_{22}B_{22} + A_{21}B_{12}$

Задача перемножения матриц: *алгоритм Фокса*

Последовательность пересылок матричных блоков в алгоритме Фокса ***(p=4×4):***



Задача перемножения матриц: *алгоритм Фокса*

Анализ эффективности для топологии гиперкуб:

- Трудоемкость рассылки блоков матрицы ***A*** по строкам процессорной решетки доминирует над временем передач блоков матрицы ***B***, (сдвиг на позицию по вертикали)
- Максимальная длительность рассылки по строкам решетки при реальной топологии сети в виде гиперкуба – $\log\sqrt{p}$, т.е., длительность рассылки блоков на одной итерации $(t_н + t_к n^2 / p)\log\sqrt{p}$
- Количество итераций алгоритма – $\sqrt{p}$
- Общее количество арифметических операций для каждого процессора – $\boldsymbol{n^3/p}$

$$T_p = (n^3 / p) + (t_н + t_к n^2 / p)\sqrt{p}\log\sqrt{p}$$

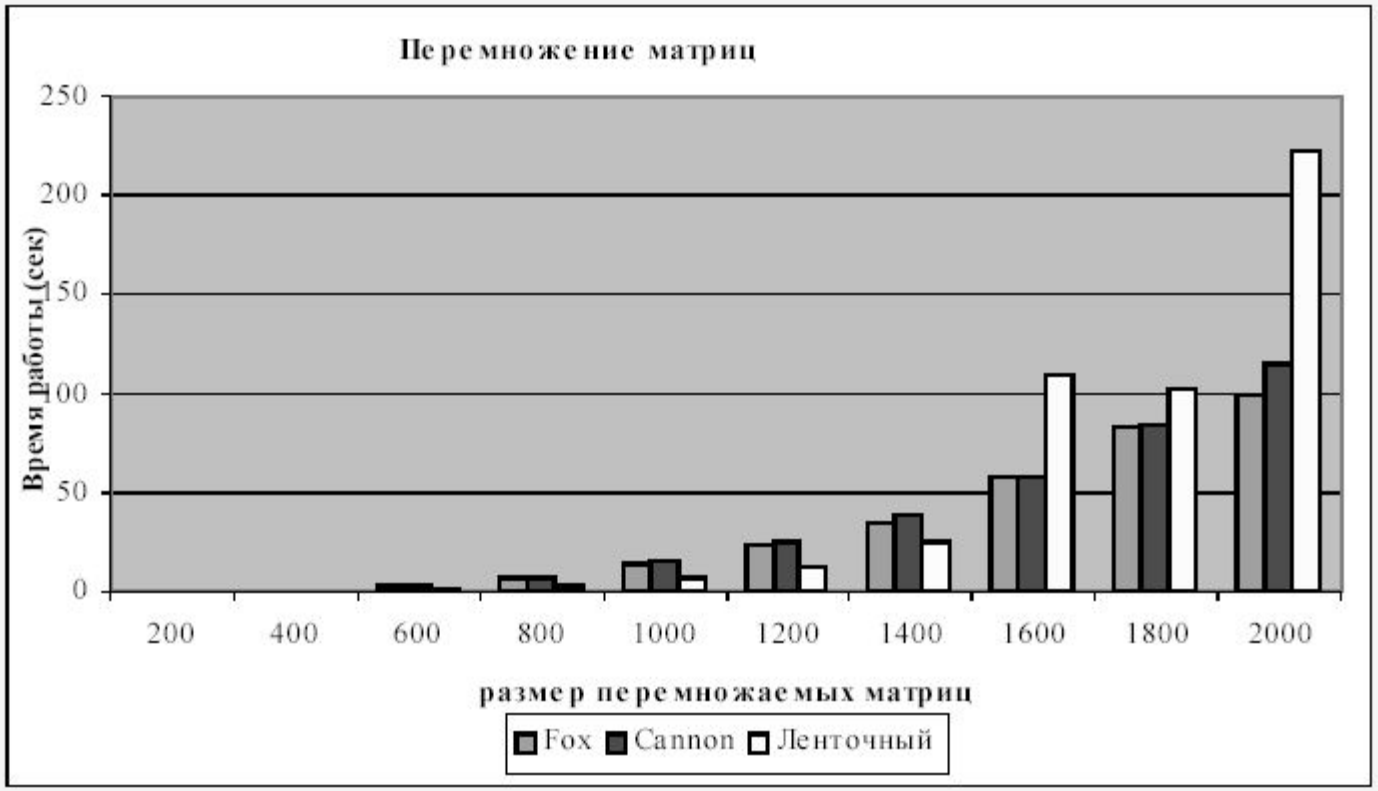
Задача перемножения матриц: *алгоритм Кеннона*

- Начальное распределение блоков - A_{ij}, B_{ij} на p_{ij}
- Начальное перераспределение блоков
  - A_{ij}, смещается влево по строке по кольцу на i позиций
  - B_{ij}, смещается вверх по столбцу по кольцу на j позиций

  (после такого перераспределения для блоков каждого процессора может быть выполнена операция умножения)
- После очередной итерации перемножения блоков, блоки A_{ij} смещаются влево (блоки B_{ij} - вверх) по кольцу на 1 позицию

Оценка времени выполнения алгоритма для гиперкуба: $T_p = (n^3 / p) + 2(t_н + t_к n^2 / p)\sqrt{p}$

Задача перемножения матриц



Транспонирование матриц

Блочное разбиение, топология - прямоугольная решетка

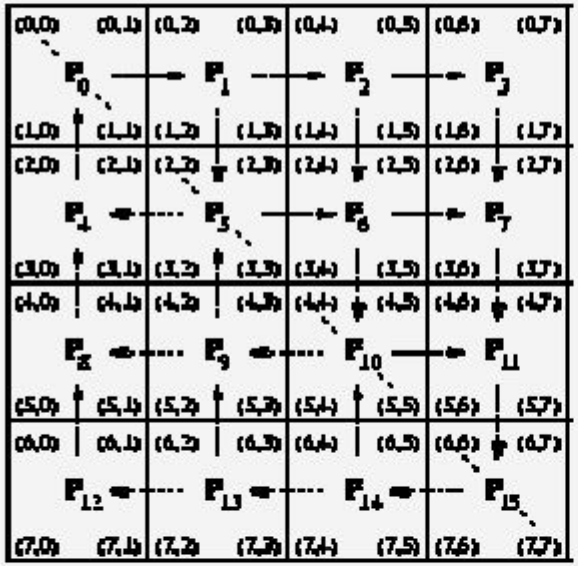
- Транспонирование выполняется за два этапа
  1. Пересылка блоков – блоки A_{ij} ($i>j$) обмениваются с блоками A_{ji}; при передаче A_{ij} пересылается вверх по столбцу до главной диагонали, затем вправо по строке до целевого процессора; для блока A_{ji} пересылка выполняется сверху вниз и затем справа налево
  2. Транспонирование блоков на каждом процессоре
- Оценка времени выполнения передачи
  - Пересылка блоков – $2(t_н + t_к n^2 / p)\sqrt{p}$
  - Транспонирование блоков – $n^2/(2p)$

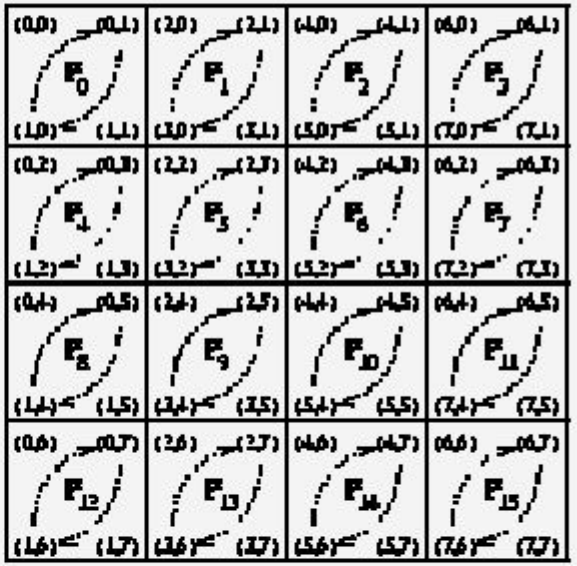
$$T_p = (n^2 / 2p) + 2(t_н + t_к n^2 / p)\sqrt{p}$$

Матричные вычисления

Транспонирование матриц

Блочное разбиение, топология - прямоугольная решетка





Транспонирование матриц: *рекурсивный алгоритм*

Блочное разбиение, топология – гиперкуб

- На первой итерации матрица A на четыре блока верхнего уровня; блоки A_{10} и A_{01} меняются друг с другом
- Далее подобные действия повторяются для каждого блока верхнего уровня параллельно до получения отдельных процессорных блоков

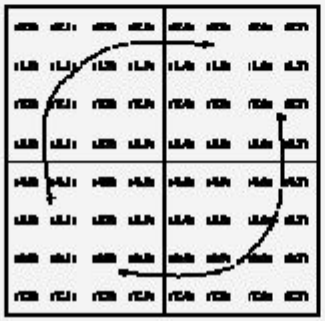
Оценка времени выполнения для метода передачи сообщений

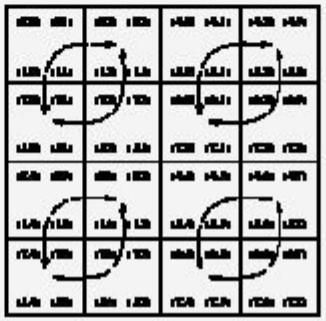
$$T_p = (n^2 / 2p) + (t_н + t_к n^2 / p) \log \sqrt{p}$$

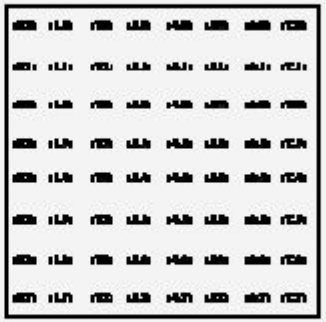
Матричные вычисления

Транспонирование матриц: *рекурсивный алгоритм*

Блочное разбиение, топология – гиперкуб (n=8)



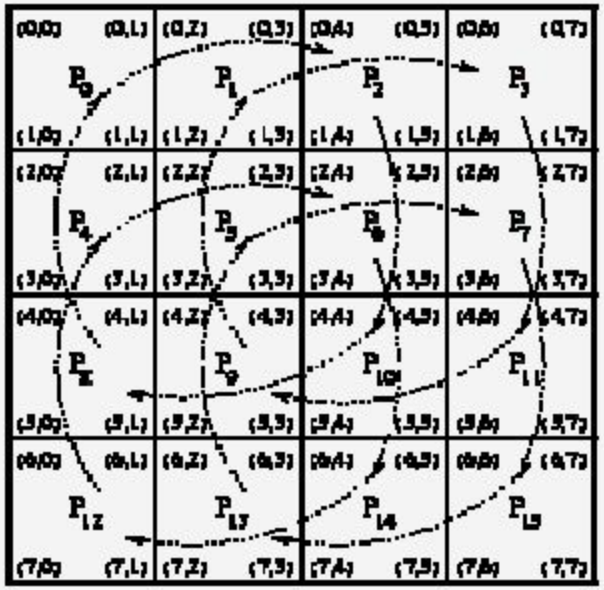




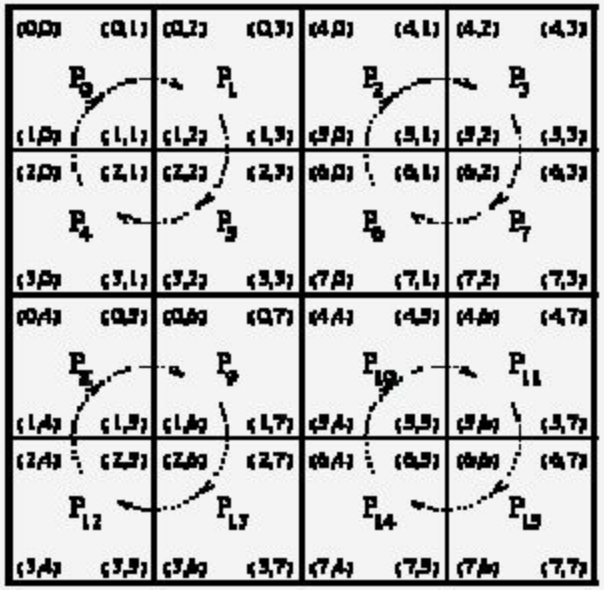
Матричные вычисления

Транспонирование матриц: *рекурсивный алгоритм*

Блочное разбиение, топология – гиперкуб (p=16)



(a)

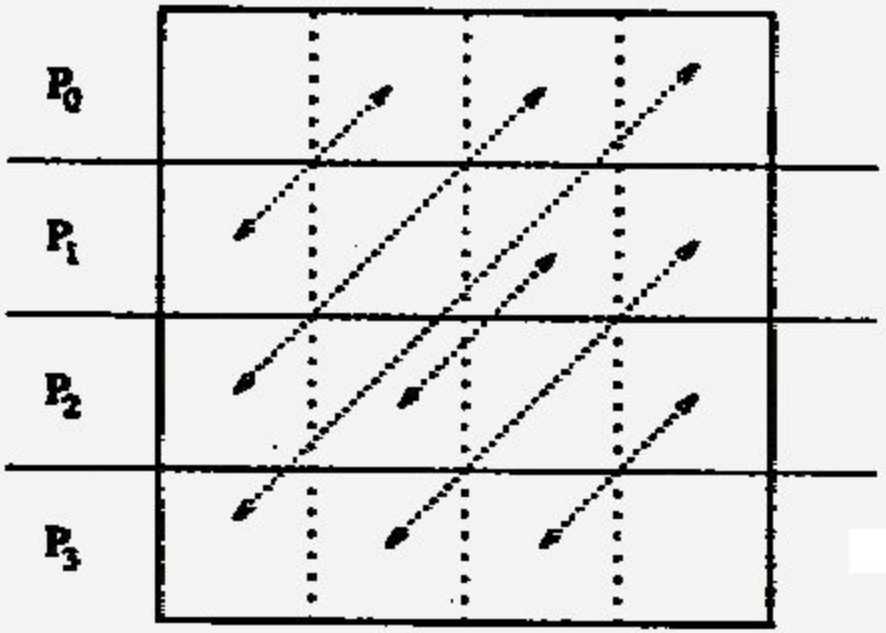


(b)

Транспонирование матриц

Ленточное разбиение

Структура передач данных имеет характер *обобщенной операции рассылки "от всех процессоров всем процессорам"*

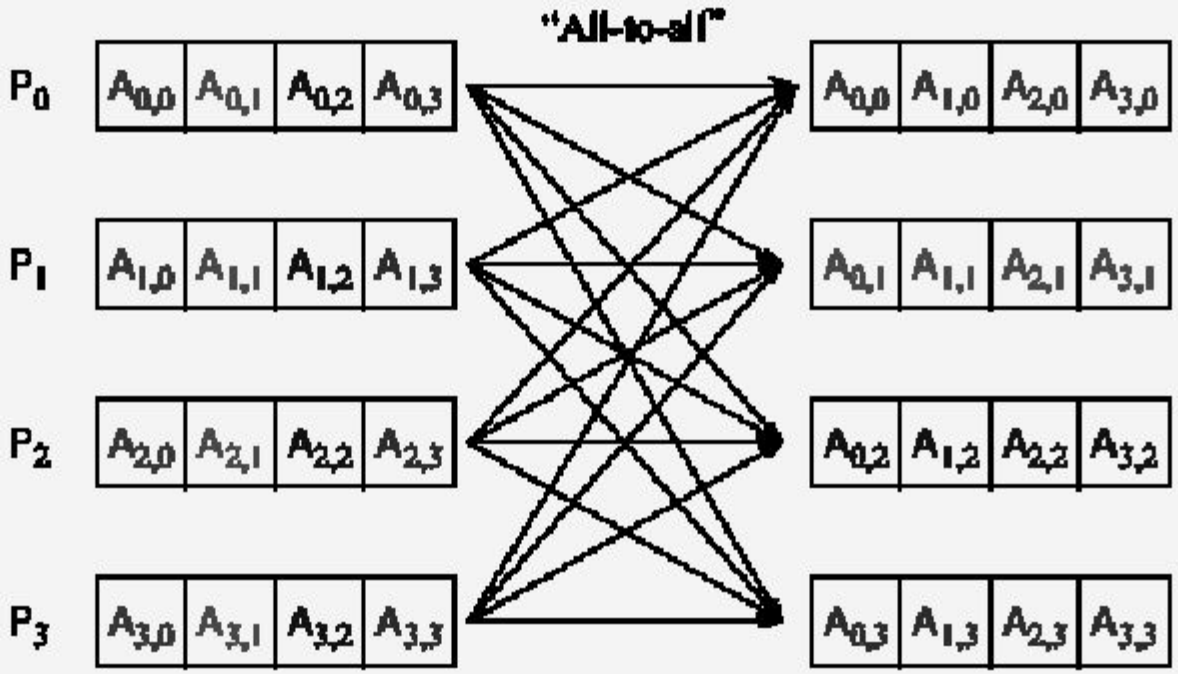


Время выполнения для гиперкуба при методе передачи пакетов

$$T_p = (n^2 / 2p) + t_н(p-1) + t_к n^2 / p + 0.5 t_с p \log p$$

Транспонирование матриц

Ленточное разбиение



Сортировка…

Сортировка является одной из типовых проблем обработки данных, и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, ..., a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a_1', a_2', ..., a_n') : a_1' \le a_2' \le ... \le a_n'\}$$

(здесь и далее все пояснения для краткости будут даваться только на примере упорядочивания данных по возрастанию)

Сортировка…

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T_1 \sim n^2$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$$T_1 \sim n \log_2 n$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из ***n*** значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Сортировка…

Ускорение сортировки может быть обеспечено при использовании нескольких ***(p, p>1)*** процессоров.

• Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются между собой.

• Результирующий (упорядоченный) набор, как правило, также разделен между процессорами; при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

Сортировка

Параллельное обобщение базовой операции...

- многие методы основаны на применении одной и той же базовой операции "*сравнить и переставить*" (*compare-exchange*), состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановки этих значений, если их порядок не соответствует условиям сортировки

```
// операция "сравнить и переставить"
if ( a[i] > a[j] ) {
   temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```

Сортировка

Параллельное обобщение базовой операции...

- целенаправленное применение операции "сравнить и переставить" позволяет упорядочить данные; в способах выбора пар значений для сравнения собственно и проявляется различие алгоритмов сортировки. Так, например, в *пузырьковой сортировке* осуществляется последовательное сравнение всех соседних элементов; в результате прохода по упорядочиваемому набору данных в последнем (верхнем) элементе оказывается максимальное значение ("всплывание пузырька"); далее для продолжения сортировки этот уже упорядоченный элемент отбрасывается и действия алгоритма повторяются

```
// пузырьковая сортировка
for ( i=1; i<n; i++ )
  for ( j=0; j<n-i; j++ )
    <сравнить и переставить (a[j],a[j+1])>
```

Сортировка

Параллельное обобщение базовой операции…

Для параллельного обобщения выделенной базовой операции сортировки рассмотрим первоначально ситуацию, когда количество процессоров совпадает с числом сортируемых значений (т.е. $p=n$).

Тогда сравнение значений a_i и a_j, располагаемых, например, на процессорах P_i и P_j, можно организовать следующим образом:

- выполнить взаимообмен имеющихся на процессорах P_i и P_j значений (с сохранением на этих процессорах исходных элементов);
- сравнить на каждом процессоре P_i и P_j получившиеся одинаковые пары значений (a_i,a_j); результаты сравнения используются для разделения данных между процессорами – на одном процессоре (например, P_i) остается меньший элемент, другой процессор (т.е. P_j) запоминает для дальней обработки большее значение пары

$$a'_i=min(a_i, a_j),\ a'_j=max(a_i, a_j)$$

Сортировка

Параллельное обобщение базовой операции...

Для случая $p<n$ каждый процессор содержит *блок* сортируемых данных из n/p элементов (данные блоки обычно упорядочиваются перед началом параллельной сортировки).

Далее взаимодействие пары процессоров P_i и P_j для упорядочения содержимого блоков A_i и A_j может состоять в следующем:

- выполнить взаимообмен блоков между процессорами P_i и P_j;
- объединить блоки A_i и A_j на каждом процессоре в один отсортированный блок двойного размера (при исходной упорядоченности блоков A_i и A_j процедура их объединения сводится к быстрой операции слияния упорядоченных наборов данных);
- разделить полученный двойной блок на две равные части и оставить одну из этих частей (например, с меньшими значениями данных) на процессоре P_i, а другую часть (с большими значениями соответственно) – на процессоре P_j

$$[A_i \cup A_j]_{copm} = A_i^{'} \cup A_j^{'} : \forall a_i^{'} \in A_i^{'}, \forall a_j^{'} \in A_j^{'} \Rightarrow a_i^{'} \le a_j^{'}$$

Параллельное обобщение базовой операции

Рассмотренная процедура обычно именуется в литературе как операция "*сравнить и разделить*" (*compare-split*).

Следует отметить, что сформированные в результате такой процедуры блоки на процессорах P_i и P_j совпадают по размеру с исходными блоками A_i и A_j и все значения, расположенные на процессоре P_i, являются меньшими значений на процессоре P_j .

Пузырьковая сортировка…

- *Алгоритм пузырьковой сортировки*

```
// пузырьковая сортировка
for ( i=1; i<n; i++ )
  for ( j=0; j<n-i; j++ )
    <сравнить и переставить (a[j],a[j+1])>
```

в прямом виде сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно.

- Для параллельного применения используется модификация алгоритма, известная как метод *чет-нечетной перестановки* (*odd-even transposition*)
  - на всех нечетных итерациях сравниваются пары

    $(a_1, a_2), (a_3, a_4)$,…(a_{n-1}, a_n), (при четном n),
  - на четных итерациях обрабатываются элементы

    $(a_2, a_3), (a_4, a_5)$,…(a_{n-2}, a_{n-1}),.

После n-кратного повторения подобных итераций сортировки исходный набор данных оказывается упорядоченным.

Сортировка

Пузырьковая сортировка…

Пример использования метода *чет-нечетной перестановки*

№ и тип итерации	Процессоры			
	1	2	3	4
Исходные данные	2 3	3 8	5 6	1 4
1 нечет (1,2),(3.4)	2 3	3 8	5 6	1 4
	2 3	3 8	1 4	5 6
2 чет (2,3)	2 3	3 8	1 4	5 6
	2 3	1 3	4 8	5 6
3 нечет (1,2),(3.4)	2 3	1 3	4 8	5 6
	1 2	3 3	4 5	6 8
4 чет (2,3)	1 2	3 3	4 5	6 8
	1 2	3 3	4 5	6 8

Пузырьковая сортировка…

Коммуникационная сложность алгоритма $T_{пд} \sim n$

Вычислительная трудоемкость алгоритма определяется выражением

$$T_p = (n/p)\log(n/p) + 2n$$

где

- первая часть соотношения учитывает сложность первоначальной сортировки блоков,
- вторая величина задает общее количество операций для слияния блоков в ходе исполнения операций "сравнить и разделить" (слияние двух блоков требует ***2(n/p)*** операций, всего выполняется ***p*** итераций).

Пузырьковая сортировка…

С учетом данной оценки показатели эффективности параллельного алгоритма имеют вид:

$$S_p = \frac{n \log n}{(n/p)\log(n/p) + 2n} \qquad E_p = \frac{n \log n}{p[(n/p)\log(n/p) + 2n]}$$

Анализ выражений показывает, что если количество процессоров совпадает с числом сортируемых данных (т.е. ***p=n***), эффективность использования процессоров падает с ростом ***n***; получение асимптотически ненулевого значения показателя может быть обеспечено при количестве процессоров, пропорциональных величине ***log n***.

Сортировка Шелла…

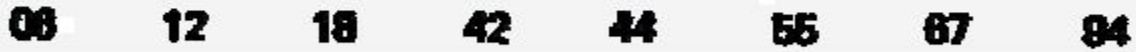
Общая идея метода состоит в сравнении на начальных стадиях сортировки пар значений, располагаемых достаточно далеко друг от друга в упорядочиваемом наборе данных. Такая модификация метода сортировки позволяет быстро переставлять далекие неупорядоченные пары значений (сортировка таких пар обычно требует большого количества перестановок, если используется сравнение только соседних элементов).

Сортировка Шелла…

4-сорировка

2-сортировка

1-сортировка



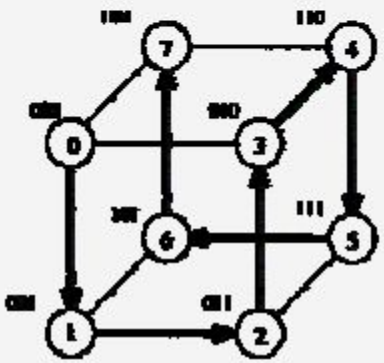
Сортировка Шелла…

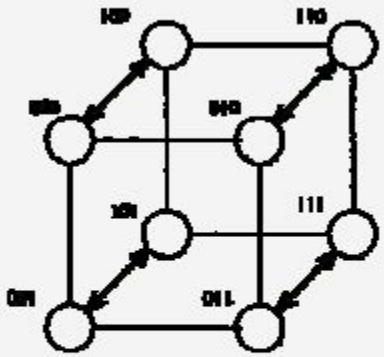
Для организации параллельных вычислений для многопроцессорной системы с топологией в виде ***N***-мерного гиперкуба (т.е. $p=2^N$) может быть предложен следующий параллельный аналог метода Шелла.

Выполнение сортировки может быть разделено на **два этапа** (для установления порядка расположения блоков сортируемого набора данных по процессорам используется логическая линейная топология сети)

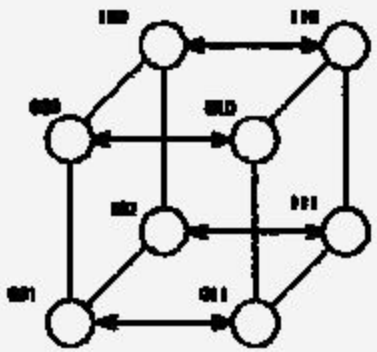
- На **первом этапе** (***N***-итераций) осуществляется взаимодействие процессоров, являющихся соседними в структуре гиперкуба (но эти процессоры могут оказаться далекими при линейной нумерации)
- **Второй этап** состоит в реализации обычных итераций параллельного алгоритма чет-нечетной перестановки. Итерации данного этапа выполняются до прекращения фактического изменения сортируемого набора и, тем самым, общее количество таких итераций может быть различным - от 2 до ***p***.

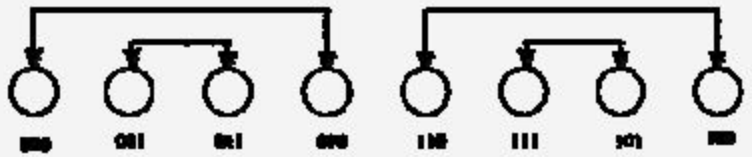
Сортировка Шелла…

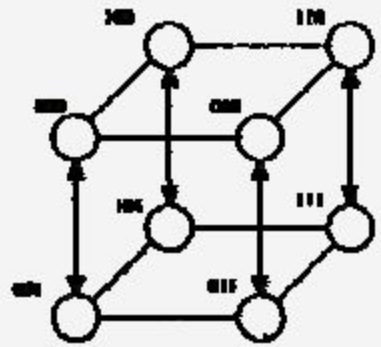




Сортировка Шелла…







Сортировка Шелла

Трудоемкость параллельного варианта алгоритма Шелла определяется выражением

$$T_p = (n/p)\log(n/p) + (2n/p)\log p + L(2n/p)$$

где вторая и третья части соотношения фиксируют вычислительную сложность первого и второго этапов сортировки соответственно.

L – число итераций чет-нечетной перестановки

Как можно заметить, эффективность данного параллельного способа сортировки оказывается лучше показателей обычного алгоритма чет-нечетной перестановки при $L<p$.

Быстрая сортировка…

Алгоритм быстрой сортировки основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности (для любой пары блоков существует блок, все значения которого будут меньше значений другого блока).

На *первой итерации* метода осуществляется деление исходного набора данных на первые две части – для организации такого деления выбирается некоторый *ведущий элемент* и все значения набора, меньшие ведущего элемента, переносятся в первый формируемый блок, все остальные значения образуют второй блок набора.

На *второй итерации* сортировки описанные правила применяются последовательно для обоих сформированных блоков и т.д. После выполнения ***log n*** итераций исходный массив данных оказывается упорядоченным.

Быстрая сортировка…

Быстрая сортировка…

Эффективность быстрой сортировки в значительной степени определяется успешностью выбора ведущих элементов при формировании блоков.

В худшем случае трудоемкость метода имеет тот же порядок сложности, что и пузырьковая сортировка (т.е. $T_1 \sim n^2$).

При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма совпадает с быстродействием наиболее эффективных способов сортировки ($T_1 \sim n \log n$).

В среднем случае количество операций, выполняемых алгоритмом быстрой сортировки, определяется выражением

$$T_1 \sim 1.4\, n \log n$$

Быстрая сортировка…

Параллельное обобщение алгоритма быстрой сортировки для вычислительной системы с топологией в виде ***N*** -мерного гиперкуба (результирующее расположение блоков будет соответствовать нумерации процессоров гиперкуба).

Первая итерация параллельного метода:

- выбрать каким-либо образом ведущий элемент и разослать его по всем процессорам системы;
- разделить на каждом процессоре имеющийся блок данных на две части с использованием полученного ведущего элемента;
- образовать пары процессоров, для которых битовое представление номеров отличается только в позиции ***N***, и осуществить взаимообмен данными между этими процессорами.

После выполнения итерации на процессорах, для которых в битовом представлении номера бит позиции ***N*** равен 0, должны оказаться части блоков со значениями, меньшими ведущего элемента; процессоры с номерами, в которых бит ***N*** равен 1, должны собрать все значения данных, превышающие ведущий элемент.

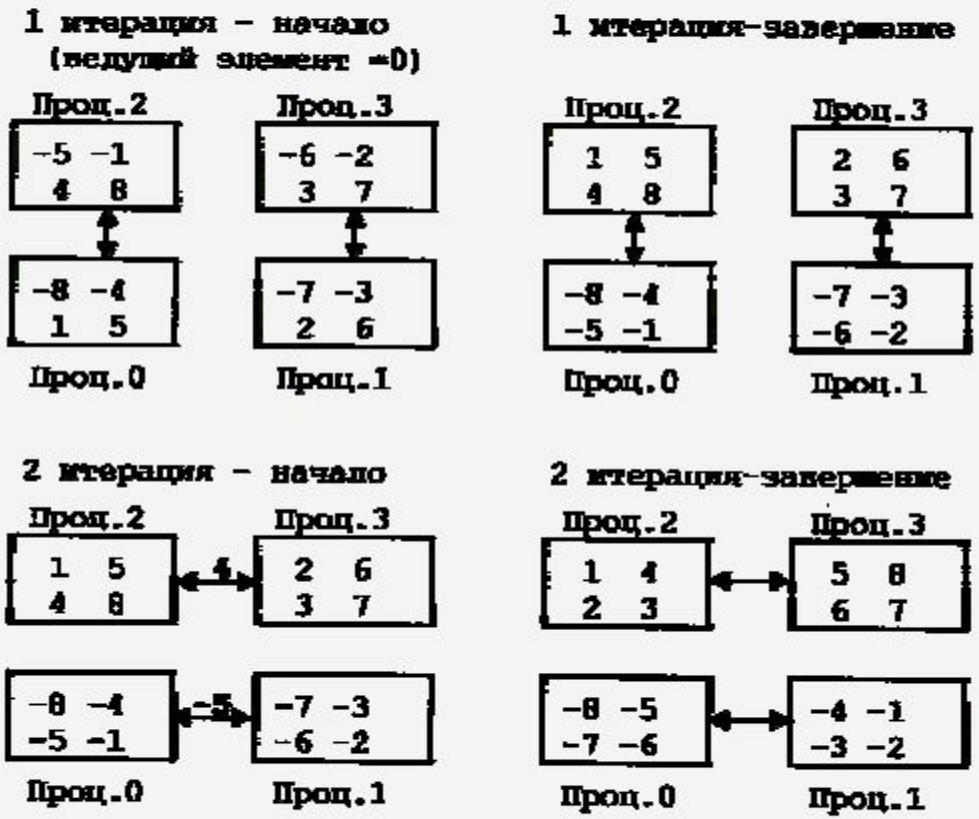
Быстрая сортировка…

В результате выполнения первой итерации сортировки исходный набор оказывается разделенным на две части, одна из которых (со значениями меньшими, чем значение ведущего элемента) располагается на процессорах, в битовом представлении номеров которых бит ***N*** равен 0.

Таких процессоров всего ***p/2*** и, таким образом, исходный ***N*** - мерный гиперкуб также оказывается разделенным на два гиперкуба размерности ***N-1***. К этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура.

После ***N*** -кратного повторения подобных итераций для завершения сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре вычислительной системы.

Быстрая сортировка…



Быстрая сортировка…

Длительность выполняемых операций передачи данных определяется операцией рассылки ведущего элемента на каждой итерации сортировки - общее количество межпроцессорных обменов для этой операции на ***N***-мерном гиперкубе может быть ограничено оценкой

$$\sum_{i=1}^{N} i = N(N+1)/2 \sim \log^2 p$$

и взаимообменом частей блоков между соседними парами процессоров – общее количество таких передач совпадает с количеством итераций сортировки, т.е. равно ***log p***, объем передаваемых данных не превышает удвоенного объема процессорного блока, т.е. ограничен величиной ***2n/p***.

Быстрая сортировка

Вычислительная трудоемкость метода обуславливается сложностью локальной сортировки процессорных блоков, временем выбора ведущих элементов и сложностью разделения блоков, что в целом может быть выражено при помощи соотношения

$$T_p = (n/p)\log(n/p) + \log p + \log(n/p)\log p$$

(при построении данной оценки предполагалось, что для выбора значения ведущего элемента при упорядоченности процессорных блоков данных достаточно одной операции).

Обработка графов…

Пусть $G=(V,R)$ есть граф, для которого набор вершин v_i, $1 \le i \le n$, задается множеством V, а список дуг графа

$$r_j = (v_{s_j}, v_{t_j}) \quad 1 \le j \le k$$

определяется множеством R. В общем случае дугам графа могут приписываться некоторые числовые характеристики w_j, $1 \le j \le k$, , (*взвешенный граф*).

Обработка графов…

Способы задания графов.

• При малом количестве дуг в графе (т.е. $k<<n^2$) целесообразно использовать для определения графов списки, перечисляющие имеющиеся в графах дуги.

• Представление достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т.е. $k \sim n^2$), может быть эффективно обеспечено при помощи *матрицы инцидентности*

$$A = (a_{ij}) \qquad 1 \le i, j \le n$$

ненулевые значения элементов которой соответствуют дугам графа

$$a_{ij} = \begin{cases} w(v_i, v_j), & \text{если} \quad (v_i, v_j) \in R, \\ 0, & \text{если} \quad i = j, \\ \infty, & \text{иначе.} \end{cases}$$

Обработка графов

Нахождение минимально охватывающего дерева…

Охватывающим деревом (или *остовом*) неориентированного графа ***G*** называется подграф ***T*** графа ***G***, который является деревом и содержит все вершины из ***G***. Определим вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, тогда под *минимально охватывающим деревом* (*МОД*) ***G*** будем понимать охватывающее дерево минимального веса.

Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием наименьшего количества соединительных линий связи.

Обработка графов

Нахождение минимально охватывающего дерева…

Метод Прима (Prim).

Алгоритм начинает работу с произвольной вершины графа, выбираемого в качестве корня дерева, и в ходе последовательно выполняемых итераций расширяет конструируемое дерево до МОД. Пусть V_T есть множество вершин, уже включенных алгоритмом в МОД, а величины d_i, $1 \le i \le n$, характеризуют дуги минимальной длины от вершин, еще не включенных в дерево, до множества V_T, т.е.

$$\forall i \notin V_T \Rightarrow d_i = \min\{w(i,u) : u \in V_T, (i,u) \in R\}$$

(если для какой либо вершины $i \notin V_T$ не существует ни одной дуги в V_T, значение d_i устанавливается в ∞). При начале работе алгоритма выбирается корневая вершина МОД s и полагается

$$V_T=\{s\},\ d_s=0.$$

Обработка графов

Нахождение минимально охватывающего дерева…

Действия, выполняемые на каждой итерации алгоритма Прима, состоят в следующем:

- определяются значения величин d_i для всех вершин, еще не включенные в состав МОД;

- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T;

$$t : d_t = \min d_i, i \notin V_T$$

- включение выбранной вершины в V_T.

После выполнения $n-1$ итераций метода МОД будет сформировано; вес этого дерева может быть получен при помощи выражения; трудоемкость нахождения МОД характеризуется квадратичной зависимостью от числа вершин графа

$$T_1 \sim n^2$$

Нахождение минимально охватывающего дерева…

Разработка параллельного варианта метода.

• Итерации метода должны выполняться последовательно и, тем самым, не могут быть распараллелены.

• С другой стороны, выполняемые на каждой итерации алгоритма действия являются независимыми и могут реализовываться одновременно. Так, например, определение величин d_i может осуществляться для каждой вершины графа в отдельности, нахождение дуги минимального веса может быть реализовано по каскадной схеме и т.д.

Обработка графов

Нахождение минимально охватывающего дерева…

Параллельное выполнение вычислений может быть обеспечено, например, когда каждый процессор P_j, $1 \le i \le p$, будет содержать

- набор вершин $V_j = \{v_{i_j+1}, v_{i_j+2}, \ldots, v_{i_j+k}\}$, $i_j=k(j-1)$, $k=n/p$,
- соответствующий этому набору блок из k величин d_i, $1 \le i \le n$,
- вертикальную полосу матрицы инцидентности графа G из k соседних столбцов,
- общую часть набора V_j и формируемого в процессе вычислений множества вершин V_T.

Обработка графов

Нахождение минимально охватывающего дерева…

Итерация **параллельного варианта алгоритма Прима**:

- определяются значения d_i для всех вершин, еще не включенные в состав МОД; данные вычисления выполняются независимо на каждом процессоре; трудоемкость операции ограничивается сверху величиной n/p (на первой итерации алгоритма необходим перебор всех вершин, что требует вычислений порядка n^2/p);
- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T; для выбора такой вершины необходимо осуществить поиск минимума в наборах величин d_i, имеющихся на каждом из процессоров (количество параллельных операций n/p), и выполнить сборку полученных значений (длительность такой операции передачи данных на гиперкубе, например, пропорциональна величине $\log p$);
- рассылка номера выбранной вершины для включения в охватывающее дерево всем процессорам (для гиперкуба сложность этой операции также определяется величиной $\log p$).

Обработка графов

Нахождение минимально охватывающего дерева

Получение МОД обеспечивается при выполнении ***n*** итераций алгоритма Прима; как результат, общая трудоемкость метода определяется соотношением

$$T_p = 2n^2 / p + 2n\log p$$

Как результат, показатели эффективности параллельного алгоритма имеют вид

$$S_p = \frac{n^2}{2n^2 / p + 2n\log p} \qquad E_p = \frac{n^2}{p[2n^2 / p + 2n\log p]}$$

Анализ выражений показывает, что достижение асимптотически ненулевого значения показателя E_p становится возможным при количестве процессоров, пропорциональном величине ***n/log*** **n**.

Обработка графов

Поиск кратчайших путей…

Задача поиска кратчайших путей на графе состоит в нахождении путей минимального веса от некоторой заданной вершины до всех имеющихся вершин графа.

Постановка подобной проблемы имеет важное практическое значение в различных приложениях, когда веса дуг означают время, стоимость, расстояние, затраты и т.п.

Обработка графов

Поиск кратчайших путей…

Возможный способ решения поставленной задачи, известный как *алгоритм Дейкстры*, практически совпадает с методом Прима. Различие состоит лишь в интерпретации и в правиле оценки вспомогательных величин d_i, $1 \le i \le n$. В алгоритме Дейкстры эти величины означают суммарный вес пути от начальной вершины до всех остальных вершин графа. Как результат, после выбора очередной вершины t графа для включения в множество выбранных вершин V_T, значения величин d_i, $1 \le i \le n$, пересчитываются в соответствии с новым правилом

$$\forall i \notin V_T \Rightarrow d_i = \min\{d_i, d_t + w(t,i)\}$$

Поиск кратчайших путей…

С учетом измененного правила пересчета величин d_i, $1 \le i \le n$, схема параллельного выполнения алгоритма Дейкстры может быть сформирована по аналогии с параллельным вариантом метода Прима.

Конкретизация такой схемы и определение показателей эффективности метода для разных топологий вычислительной системы - темы самостоятельных заданий.

Поиск кратчайших путей

Показатели эффективности алгоритмов Прима и Дейкстры для разных топологий

Топология	К-во процессоров для асимптотот. ненулевой эффективности	Трудоемкость при таком количестве процессоров
Кольцо	$\sqrt{n}$	$n^{1.5}$
Решетка	$n^{0.66}$	$n^{1.33}$
Гиперкуб	$n/\log n$	$n\log n$