11. Обработка графов

Математические модели в виде *графов* широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор *типовых алгоритмов обработки графов*. Рассмотрению вопросов теории графов, алгоритмов моделирования, анализу и решению задач на графах посвящено достаточно много различных изданий, в качестве возможного руководства по данной тематике может быть рекомендована работа Кормен и др. (1999).

Пусть ***G*** есть граф

$$G = (V, R),$$

для которого набор вершин v_i, $1 \le i \le n$, задается множеством ***V***, а список дуг графа

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

определяется множеством ***R***. В общем случае дугам графа могут приписываться некоторые числовые характеристики (*веса*) w_j, $1 \le j \le m$ (*взвешенный граф*). Пример взвешенного графа приведен на рис. 11.1.

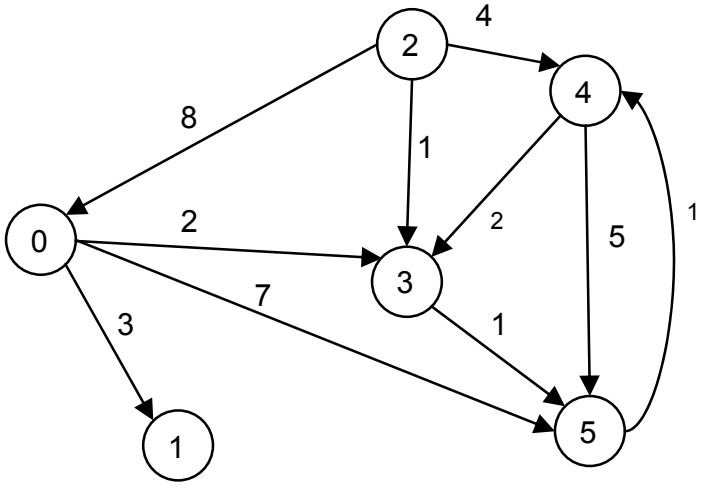

Рис. 11.1. Пример взвешенного ориентированного графа

Известны различные способы задания графов. При малом количестве дуг в графе (т.е. $m << n^2$) 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. $m \sim n^2$), может быть эффективно обеспечено при помощи *матрицы смежности*

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

ненулевые значения элементов которой соответствуют дугам графа

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

(для обозначения отсутствия ребра между вершинами в матрице смежности на соответствующей позиции используется знак бесконечности, при вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число). Так, например, матрица смежности, соответствующая графу на рис. 11.1, приведена на рис. 11.2.

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Рис. 11.2. Матрица смежности для графа из рис. 11.1

Как положительный момент такого способа представления графов можно отметить, что использование матрицы смежности позволяет применять при реализации вычислительных процедур анализа графов матричные алгоритмы обработки данных.

Далее мы рассмотрим способы параллельной реализации алгоритмов на графах на примере *задачи поиска кратчайших путей* между всеми парами пунктов назначения и *задачи выделения минимального охватывающего дерева* (*остова*) графа. Кроме того, мы рассмотрим *задачу оптимального разделения графов*, широко используемую для организации параллельных вычислений. Для представления графов при рассмотрении всех перечисленных задач будут использоваться матрицы смежности.

11.1. Задача поиска всех кратчайших путей

Исходной информацией для задачи является взвешенный граф $G = (V, R)$, содержащий ***n*** вершин ($|V| = n$), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать

ориентированным, т.е., если из вершины i есть ребро в вершину j, то из этого не следует наличие ребра из j в i. В случае, если вершины все же соединены взаимообратными ребрами, то веса, приписанные им, могут не совпадать. Рассмотрим задачу, в которой для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется *алгоритм Флойда* (*Floyd*) (см, например, Кормен и др. (1999)).

11.1.1. Последовательный алгоритм Флойда

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок n^3. В общем виде данный алгоритм может быть представлен следующим образом:

```
// Алгоритм 11.1
// Последовательный алгоритм Флойда
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      A[i,j] = min(A[i,j],A[i,k]+A[k,j]);
```

Алгоритм 11.1. Общая схема алгоритма Флойда

(реализация операции выбора минимального значения min должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A будет храниться требуемый результат - длины минимальных путей для каждой пары вершин исходного графа.

Дополнительная информация и доказательство правильности алгоритма Флойда могут быть получены, например, в книге Кормен и др. (1999).

11.1.2. Разделение вычислений на независимые части

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимальных значения (см. Алгоритм 11.1). Данная операция является достаточно простой и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A.

Покажем корректность такого способа организации параллелизма. Для этого нужно доказать, что операции обновления значений матрицы A на одной и той же итерации внешнего цикла k могут выполняться независимо. Иными словами, следует показать, что на итерации k не происходит изменения элементов A_{ik} и A_{kj} ни для одной пары индексов (i,j). Рассмотрим выражение, по которому происходит изменение элементов матрицы A:

$A_{ij} \leftarrow \min (A_{ij}, A_{ik} + A_{kj})$.

Для $i=k$ получим

$A_{kj} \leftarrow \min (A_{kj}, A_{kk} + A_{kj})$,

но тогда значение A_{kj} не изменится, т.к. A_{kk}=0.

Для $j=k$ выражение преобразуется к виду

$A_{ik} \leftarrow \min (A_{ik}, A_{ik} + A_{kk})$,

что также показывает неизменность значений A_{ik}. Как результат, необходимые условия для организации параллельных вычислений обеспечены и, тем самым, в качестве *базовой подзадачи* может быть использована операция обновления элементов матрицы A (для указания подзадач будем использовать индексы обновляемых в подзадачах элементов).

11.1.3. Выделение информационных зависимостей

Выполнение вычислений в подзадачах становится возможным только тогда, когда каждая подзадача (i,j) содержит необходимые для расчетов элементы A_{ij}, A_{ik}, A_{kj} матрицы A. Для исключения дублирования данных разместим в подзадаче (i,j) единственный элемент A_{ij}, тогда получение всех остальных необходимых значений может быть обеспечено только при помощи передачи данных. Таким

образом, каждый элемент A_{kj} строки k матрицы A должен быть передан всем подзадачам (k,j), $1 \le j \le n$, а каждый элемент A_{ik} столбца k матрицы A должен быть передан всем подзадачам (i,k), $1 \le i \le n$, - см. рис. 11.3.

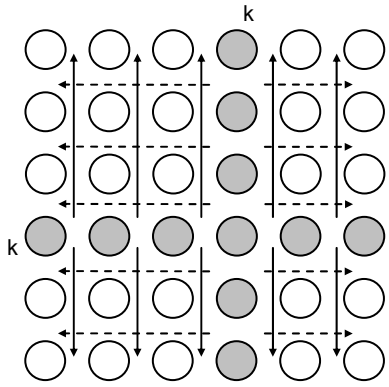

Рис. 11.3. Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

11.1.4. Масштабирование и распределение подзадач по процессорам

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач n^2 ($p << n^2$). Возможный способ укрупнения вычислений состоит в использовании *ленточной схемы* разбиения матрицы A – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк (*горизонтальное* разбиение) или столбцов (*вертикальное* разбиение) матрицы A. Эти два типа разбиения практически равноправны – учитывая дополнительный момент, что для алгоритмического языка С массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы A на горизонтальные полосы.

Следует отметить, при таком способе разбиения данных на каждой итерации алгоритма Флойда потребуется передавать между подзадачами только элементы одной из строк матрицы A. Для эффективного выполнения подобной коммуникационной операции топология сети должна представлять собой гиперкуб или полный граф.

11.1.5. Анализ эффективности параллельных вычислений

Выполним анализ эффективности параллельного алгоритма Флойда, обеспечивающего поиск всех кратчайших путей. Как и ранее, проведем этот анализ в два этапа. На первом этапе оценим порядок вычислительной сложности алгоритма, затем на втором этапе уточним полученные оценки и учтем трудоемкость выполнения коммуникационных операций.

Общая трудоемкость последовательного алгоритма, как уже отмечалось ранее, является равной n^3. Для параллельного алгоритма на отдельной итерации каждый процессор выполняет обновление элементов матрицы A. Всего в подзадачах n^2/p таких элементов, число итераций алгоритма равно n – таким образом, показатели ускорения и эффективности параллельного алгоритма Флойда имеют вид:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{и} \quad E_p = \frac{n^3}{p \cdot (n^3/p)} = 1. \qquad (11.1)$$

Таким образом, общий анализ сложности дает идеальные показатели эффективности параллельных вычислений. Для уточнения полученных соотношений введем в полученные выражения время выполнения базовой операции выбора минимального значения и учтем затраты на выполнение операций передачи данных между процессорами.

Коммуникационная операция, выполняемая на каждой итерации алгоритма Флойда, состоит в передаче одной из строк матрицы A всем процессорам вычислительной системы. Как уже показывалось ранее, выполнение такой операции может быть выполнено за $\lceil \log_2 p \rceil$ шагов. С учетом количества итераций алгоритма Флойда при использовании модели Хокни общая длительность выполнения коммуникационных операций может быть определена при помощи следующего выражения

$$T_p(comm) = n\lceil \log_2 p \rceil (\alpha + wn/\beta), \qquad (11.2)$$

где, как и ранее, α – латентность сети передачи данных, β – пропускная способность сети, а *w* есть размер элемента матрицы в байтах.

С учетом полученных соотношений общее время выполнения параллельного алгоритма Флойда может быть определено следующим образом:

$$T_p = n^2 \cdot \lceil n/p \rceil \cdot \tau + n \cdot \lceil \log_2(p) \rceil (\alpha + w \cdot n/\beta), \qquad (11.3)$$

где τ есть время выполнения операции выбора минимального значения.

11.1.6. Программная реализация

Представим возможный вариант параллельной реализации алгоритма Флойда. При этом описание отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимание общей схемы параллельных вычислений.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
int ProcRank;    // rank of current process
int ProcNum;     // number of processes

// Maximum evaluation function
int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

// Main function
int main(int argc, char* argv[]) {
  int *pMatrix;      // adjacency matrix
  int  Size;         // size of adjacency matrix
  int *pProcRows;    // rows of adjacency matrix
  int  RowNum;       // number of rows on current process

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  // Data initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  // Data distribituon to all processes
  DataDistribution(pMatrix, pProcRows, Size, RowNum);

  // The Floyd parallel algorithm
  ParallelFloyd(pProcRows, Size, RowNum);

  // Result collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);

  // Computation Termination
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
  return 0;
}
```

Функция *Min* вычисляет наименьшее среди двух целых чисел, учитывая используемый метод обозначения несуществующих дуг в матрице смежности (в рассматриваемой реализации используется значение -1).

Функция *ProcessInitialization* определяет исходные данные решаемой задачи (количество вершин графа), выделяет память для хранения данных и осуществляет ввод матрицы смежности (или формирует ее при помощи какого-либо датчика случайных числе) и распределяет данные между процессами.

Функция *DataDistribution* распределяет данные между процессами. Каждый процесс получает горизонтальную полосу исходной матрицы.

Функция *ResultCollection* осуществляет сбор со всех процессов горизонтальных полос результирующей матрицы кратчайших расстояний между любыми парами вершин графа.

Функция *ProcessTermination* выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

2. Функция ParallelFloyd. Данная функция осуществляет итеративное изменение матрицы смежности в соответствии с алгоритмом Флойда.

```
// The Floyd parallel algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;
  for(int k = 0; k < Size; k++) {
    // k-ой row distributin among processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Updating the adjacency matrix
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) &&
            (pRow     [j]            != -1)) {
          t1 = pProcRows[i * Size + j];
          t2 = pProcRows[i * Size + k] + pRow[j];
          pProcRows[i * Size + j] = Min(t1, t2);
        }
  }

  delete []pRow;
}
```

3. The function RowDistribution. Эта функция распределяет k-ю строку матрицы смежности среди процессов параллельной программы:

```
// The row distribution function
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank;
  int ProcRowNum;

  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {
    if(k < Ind + Num ) break;
    RestRows -= Num;
    Ind      += Num;
    Num       = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
     copy(&pProcRows[ProcRowNum * Size], &pProcRows[(ProcRowNum + 1) *
      Size], pRow);
```

```
   MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}
```

11.1.7. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма Флойда по поиску всех кратчайших путей проводились при условиях, указанных в п. 7.6.5, и состояли в следующем.

Для экспериментов использовался на вычислительный кластер на базе процессоров Intel XEON 4 EM64T 3000 Мгц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition.

Для оценки длительности τ базовой скалярной операции выбора минимального значения проводилось решение задачи поиска всех кратчайших путей при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение7,1 наносекунды. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 47 мкс и 53.29 Мбайт/с. Все вычисления производились над числовыми значениями типа int, т. е. величина w равна 4 байтам.

Результаты вычислительных экспериментов приведены в таблице 11.1. Эксперименты выполнялись с использованием двух, четырех и восьми процессоров. Время указано в секундах.

Таблица 11.1. Результаты вычислительных экспериментов для параллельного алгоритма Флойда

Количество вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	8,0370	4,1519	1,9357	2,0671	3,8880	0,9407	8,5439
2000	59,8119	30,3234	1,9725	15,3752	3,8901	8,0577	7,4229
3000	197,1114	99,2642	1,9857	50,2323	3,9240	25,6433	7,6867
4000	461,7849	232,5071	1,9861	117,2204	3,9395	69,9457	6,6021
5000	884,6221	443,7467	1,9935	224,4413	3,9414	128,0784	6,9069

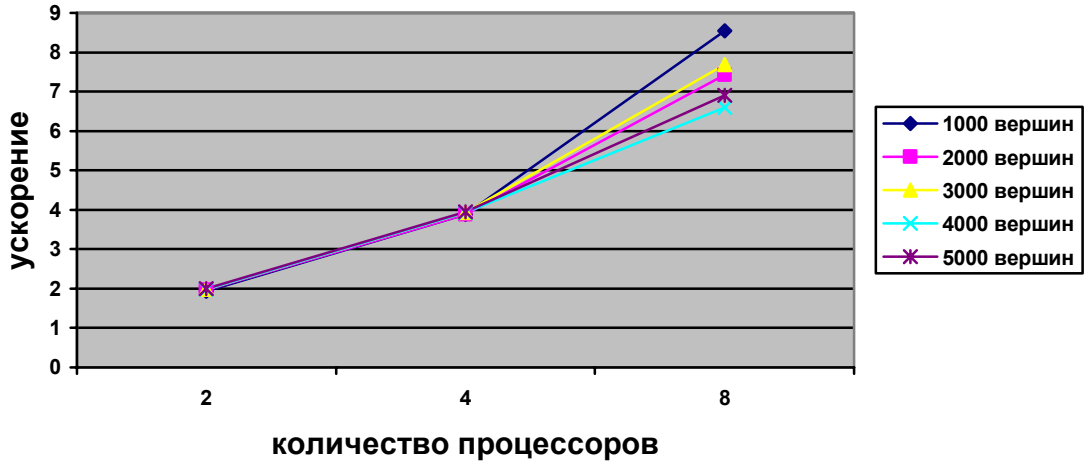

Рис. 11.4. Графики зависимости ускорения параллельного алгоритма Флойда от числа используемых процессоров при различном количестве вершин графа

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (11.3) приведено в таблице 11.2 и на рис. 11.5.

Таблица 11.2. Сравнение экспериментального и теоретического времени работы алгоритма Флойда

Количество вершин	Параллельный алгоритм		
	2 процессора	4 процессора	8 процессора

	T_2 (model)	T_2^*	T_4 (model)	T_4^*	T_8 (model)	T_8^*
1000	3,7757	4,1519	2,1960	2,0671	1,5090	0,9407
2000	29,1234	30,3234	15,4046	15,3752	8,8261	8,0577
3000	97,4645	99,2642	50,3361	50,2323	27,3066	25,6433
4000	230,2200	232,5071	117,7013	117,2204	62,3058	69,9457
5000	448,8112	443,7467	228,2108	224,4413	119,1790	128,0784

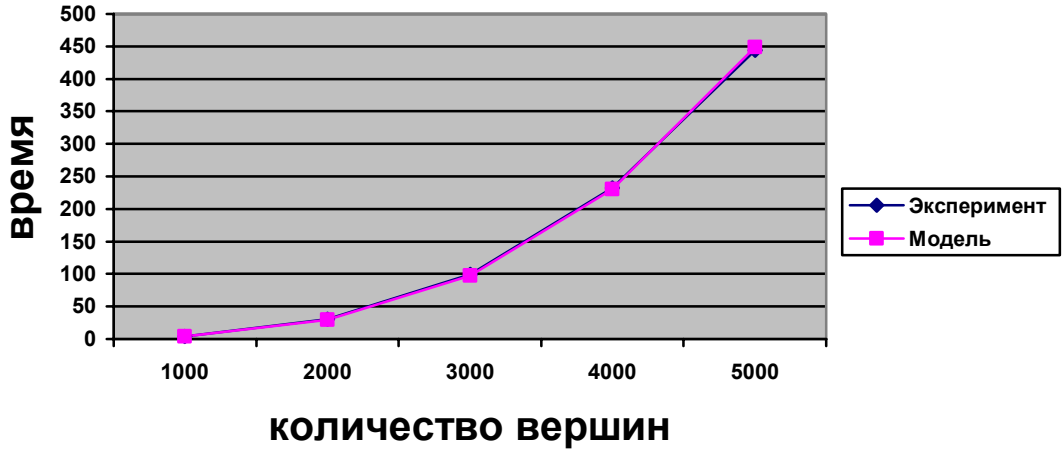

Рис. 11.5. Графики экспериментально установленного времени работы параллельного алгоритма Флойда и теоретической оценки в зависимости от количества вершин графа при использовании 2 процессоров

11.2. Задача нахождения минимального охватывающего дерева

Охватывающим деревом (или *остовом*) неориентированного графа ***G*** называется подграф ***T*** графа ***G***, который является деревом и содержит все вершины из ***G***. Определив вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, под *минимально охватывающим деревом (МОД)* ***T*** будем понимать охватывающее дерево минимального веса. Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием соединительных линий связи минимальной длины. Пример взвешенного неориентированного графа и соответствующего ему минимального охватывающего дерева приведен на рис. 11.6.

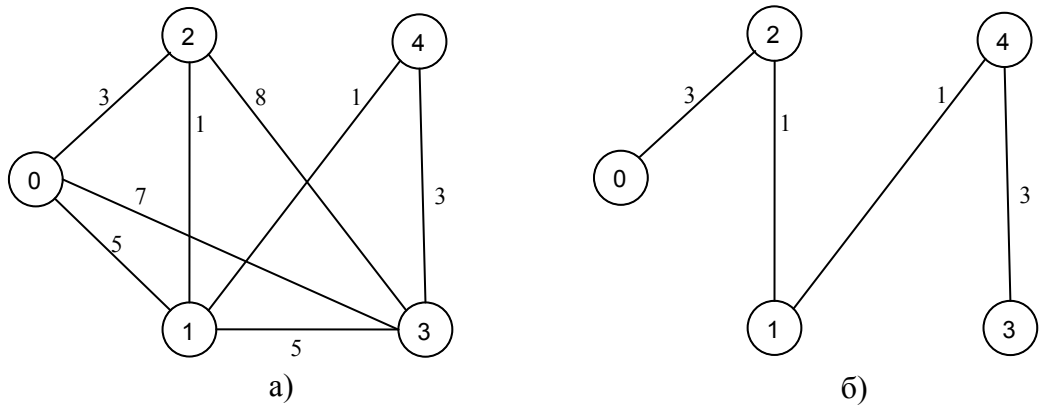

Рис. 11.6. Пример (а) взвешенного неориентированного графа и соответствующему ему (б) минимально охватывающего дерева

Дадим общее описание алгоритма решения поставленной задачи, известного под названием *метода Прима* (*Prim*), более полная информация может быть получена, например, в Кормен и др. (1999).

11.2.1. Последовательный алгоритм Прима

Алгоритм начинает работу с произвольной вершины графа, выбираемой 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 ∞). В начале работы алгоритма выбирается корневая вершина МОД s и полагается $V_T = \{s\}$, $d_s = 0$.

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

- определяются значения величин d_i для всех вершин, еще не включенных в состав МОД;
- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T $t : d_t = \min(d_i)$, $i \notin V_T$;
- вершина t включается в V_T.

После выполнения $n-1$ итерации метода МОД будет сформировано. Вес этого дерева может быть получен при помощи выражения

$$W_T = \sum_{i=1}^{n} d_i .$$

Трудоемкость нахождения МОД характеризуется квадратичной зависимостью от числа вершин графа $O(n^2)$.

11.2.2. Разделение вычислений на независимые части

Оценим возможности параллельного выполнения рассмотренного алгоритма нахождения минимального охватывающего дерева.

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.

Распределение данных между процессорами вычислительной системы должно обеспечивать независимость перечисленных операций алгоритма Прима. В частности, это может быть реализовано, если каждая вершина графа располагается на процессоре вместе со всей связанной с вершиной информацией. Соблюдение данного принципа приводит к тому, что при равномерной загрузке каждый процессор P_j, $1 \le j \le p$, должен содержать:

- набор вершин
$V_j = \{v_{i_j+1}, v_{i_j+2}, ..., v_{i_j+k}\}$, $i_j = k \cdot (j-1)$, $k = \lceil n/p \rceil$,
- соответствующий этому набору блок из k величин
$\Delta_j = \{d_{i_j+1}, d_{i_j+2}, ..., d_{i_j+k}\}$,
- вертикальную полосу матрицы смежности графа G из k соседних столбцов
$A_j = \{\alpha_{i_j+1}, \alpha_{i_j+2}, ..., \alpha_{i_j+k}\}$ (α_s есть s-ый столбец матрицы A),
- общую часть набора V_j и формируемого в процессе вычислений множества вершин V_T.

Как итог можем заключить, что *базовой подзадачей* в параллельном алгоритме Прима может служить процедура вычисления блока значений Δ_j для вершин V_j матрицы смежности A графа G.

11.2.3. Выделение информационных зависимостей

С учетом выбора базовых подзадач общая схема параллельного выполнения алгоритма Прима будет состоять в следующем:

- определяется 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, и выполнить сборку полученных значений на одном из процессоров;
- номер выбранной вершины для включения в охватывающее дерево передается всем процессорам;
- обновляются наборы величин d_i с учетом добавления новой вершины.

Таким образом, в ходе параллельных вычислений между процессорами выполняются два типа информационных взаимодействий - сбор данных от всех процессоров на одном из процессоров и передача сообщений от одного процессора всем процессорам вычислительной системы.

11.2.4. Масштабирование и распределение подзадач по процессорам

В соответствии с определением количество базовых подзадач всегда соответствует числу имеющихся процессоров и, тем самым, проблема масштабирования для параллельного алгоритма не возникает.

Распределение подзадач между процессорами должно учитывать характер выполняемых в алгоритме Прима коммуникационных операций. Для эффективной реализации требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должна иметь структуру гиперкуба или полного графа.

11.2.5. Анализ эффективности параллельных вычислений

Общий анализ сложности параллельного алгоритма Прима для нахождения минимального охватывающего дерева дает идеальные показатели эффективности параллельных вычислений.

$$S_p = \frac{n^2}{(n^2/p)} = p \quad \text{и} \quad E_p = \frac{n^2}{p\cdot(n^2/p)} = 1. \tag{11.4}$$

При этом следует отметить, что в ходе параллельных вычислений идеальная балансировка вычислительной нагрузки процессоров может быть нарушена. В зависимости от вида исходного графа G количество выбранных вершин в охватывающем дереве на разных процессорах может оказаться различным и распределение вычислений между процессорами станет неравномерным (вплоть до отсутствия вычислительной нагрузки на отдельных процессорах). Однако такие предельные ситуации нарушения балансировки в общем случае будут возникать достаточно редко, а организация динамического перераспределения вычислительной нагрузки между процессорами в ходе вычислений является интересной, но одновременно и очень сложной задачей.

Для уточнения полученных показателей эффективности параллельных вычислений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

При выполнении вычислений на отдельной итерации параллельного алгоритма Прима каждый процессор определяет номер ближайшей вершины из V_j до охватывающего дерева и осуществляет корректировку расстояний d_i после расширения МОД. Количество выполняемых операций в каждой из этих вычислительных процедур ограничивается сверху числом вершин, имеющихся на процессорах, т.е. величиной $\lceil n/p \rceil$. Как результат, с учетом общего количества итераций n время выполнения вычислительных операций параллельного алгоритма Прима может быть оценено при помощи соотношения:

$$T_p(calc) = 2n\lceil n/p \rceil \cdot \tau \tag{11.5}$$

(здесь, как и ранее, τ есть время выполнения одной элементарной скалярной операции).

Операция сбора данных от всех процессоров на одном из процессоров может быть выполнена за $\lceil \log_2 p \rceil$ итераций, при этом общая оценка длительности выполнения передачи данных определяется выражением (более подробное рассмотрение данной коммуникационной операции содержится в разделах 3 и 7):

$$T_p^1(comm) = n(\alpha \log_2 p + 3w(p-1)/\beta), \quad (11.6)$$

где α – латентность сети передачи данных, β – пропускная способность сети, а w есть размер одного пересылаемых элемента данных в байтах (коэффициент 3 в выражении соответствует числу передаваемых значений между процессорами – длина минимальной дуги и номера 2 вершин, которые соединяются этой дугой).

Коммуникационная операция передачи данных от одного процессора всем процессорам вычислительной системы также может быть выполнена за $\lceil \log_2 p \rceil$ итераций при общей оценке времени выполнения вида:

$$T_p^2(comm) = n \log_2 p(\alpha + w/\beta). \quad (11.7)$$

С учетом всех полученных соотношений, общее время выполнения параллельного алгоритма Прима составляет:

$$T_p = 2n\lceil n/p \rceil \cdot \tau + n(\alpha \cdot \log_2 p + 3w(p-1)/\beta + \log_2 p(\alpha + w/\beta)) \quad (11.8)$$

11.2.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма Прима проводились при тех же условиях, что и ранее выполненные эксперименты (см. п. 11.1.7).

Для оценки длительности τ базовой скалярной операции проводилось решение задачи нахождения минимального охватывающего дерева при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 4,8 наносекунды. Все вычисления производились над числовыми значениями типа int, т. е. величина w равна 4 байт.

Результаты вычислительных экспериментов приведены в таблице 11.3. Эксперименты проводились с использованием двух, четырех и шести процессоров. Время указано в секундах.

Таблица 11.3. Результаты вычислительных экспериментов по исследованию параллельного алгоритма Прима

Количество вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	0,0435	0,2476	0,1757	0,9320	0,0467	1,5735	0,0277
2000	0,2079	0,6837	0,3041	1,7999	0,1155	2,1591	0,0963
3000	0,4849	1,4034	0,3455	2,2136	0,2191	3,1953	0,1518
4000	0,8729	1,9455	0,6220	3,3237	0,2626	5,4309	0,1607
5000	1,4324	2,6647	0,7363	2,9331	0,4884	4,1189	0,3478
6000	2,1889	2,8999	0,8214	4,2911	0,5101	7,7373	0,2829
7000	3,0424	3,2364	1,0491	6,3273	0,4808	8,8255	0,3447
8000	4,1497	4,4621	1,2822	6,9931	0,5934	10,3898	0,3994
9000	5,6218	5,8340	1,2599	7,4747	0,7521	10,7636	0,5223
10000	7,5116	6,9902	1,2875	8,5968	0,8738	14,0951	0,5329

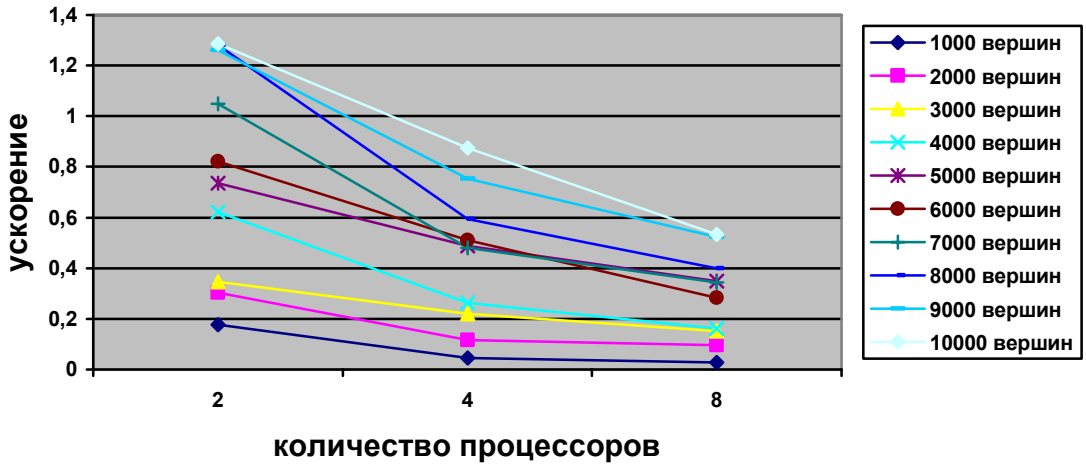

Рис. 11.7. Графики зависимости ускорения параллельного алгоритма Прима от числа используемых процессоров при различном количестве вершин в модели

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (11.3) приведено в таблице 11.4 и на рис. 11.8.

Таблица 11.4. Сравнение экспериментального и теоретического времени работы алгоритма Прима.

Количество вершин	Параллельный алгоритм					
	2 процессора		4 процессора		8 процессоров	
	T_2 (модель)	T_2^*	T_4 (модель)	T_4^*	T_8 (модель)	T_8^*
1000	0,4054	0,2476	0,8040	0,9320	1,2048	1,5735
2000	0,8203	0,6837	1,6128	1,7999	2,4120	2,1591
3000	1,2447	1,4034	2,4264	2,2136	3,6216	3,1953
4000	1,6786	1,9455	3,2447	3,3237	4,8335	5,4309
5000	2,1220	2,6647	4,0678	2,9331	6,0479	4,1189
6000	2,5750	2,8999	4,8957	4,2911	7,2646	7,7373
7000	3,0375	3,2364	5,7283	6,3273	8,4837	8,8255
8000	3,5095	4,4621	6,5656	6,9931	9,7052	10,3898
9000	3,9911	5,8340	7,4078	7,4747	10,9290	10,7636
10000	4,4821	6,9902	8,2546	8,5968	12,1552	14,0951

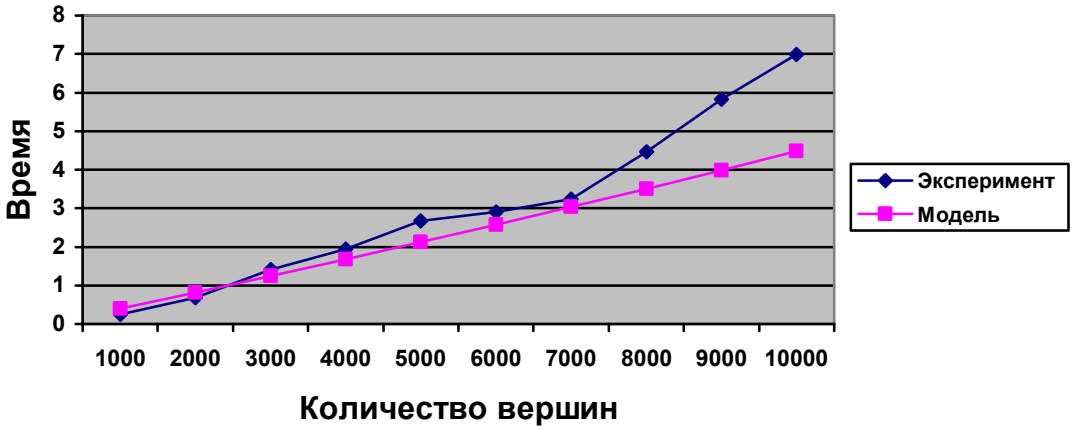

Рис. 11.8. Графики экспериментально установленного времени работы параллельного алгоритма Прима и теоретической оценки в зависимости от количества вершин в модели при использовании 2 процессоров

Как можно заметить по табл. 11.4 и рис. 11.8, теоретические оценки определяют время выполнения алгоритма Прима с достаточно высокой погрешностью. Причина такого расхождения состоит в том, что модель Хокни менее точна при оценке времени передачи сообщений с небольшим объемом передаваемых данных. В этом отношении, для уточнения получаемых оценок необходимым является использование других более точных моделей расчета трудоемкости коммуникационных операций – обсуждение этого вопроса проведено в разделе 3.

11.3. Задача оптимального разделения графов

Проблема оптимального разделения графов относится к числу часто возникающих задач при проведении различных научных исследований, использующих параллельные вычисления. В качестве примера можно привести задачи обработки данных, в которых области расчетов представляются в виде двухмерной или трехмерной сети. Вычисления в таких задачах сводятся, как правило, к выполнению тех или иных процедур обработки для каждого элемента (узла) сети. При этом в ходе вычислений между соседними элементами сети может происходить передача результатов обработки и т.п. Эффективное решение таких задач на многопроцессорных системах с распределенной памятью предполагает разделение сети между процессорами таким образом, чтобы каждому из процессоров выделялось примерно равное число элементов сети, а межпроцессорные коммуникации, необходимые для выполнения информационного обмена между соседними элементами, были минимальными. На рис. 11.9. показан пример нерегулярной сети, разделенной на 4 части (различные части разбиения сети выделены темным цветом различной интенсивности).

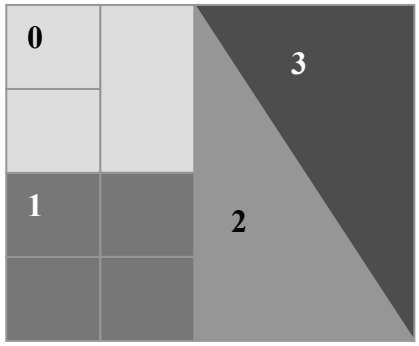

Рис. 11.9. Пример разделения нерегулярной сети

Очевидно, что такие задачи разделения сети между процессорами могут быть сведены к проблеме оптимального разделения графа. Данный подход целесообразен потому, что представление модели вычислений в виде графа позволяет легче решить вопросы хранения обрабатываемых данных и предоставляет возможность применения типовых алгоритмов обработки графов.

Для представления сети в виде графа каждому элементу сети можно поставить в соответствие вершину графа, а дуги графа использовать для отражения свойства близости элементов сети (например, определять дуги между вершинами графа тогда и только тогда, когда соответствующие элементы исходной сети являются соседними). При таком подходе, например, для сети на рис. 11.9, будет сформирован граф, приведенный на рис. 11.10.

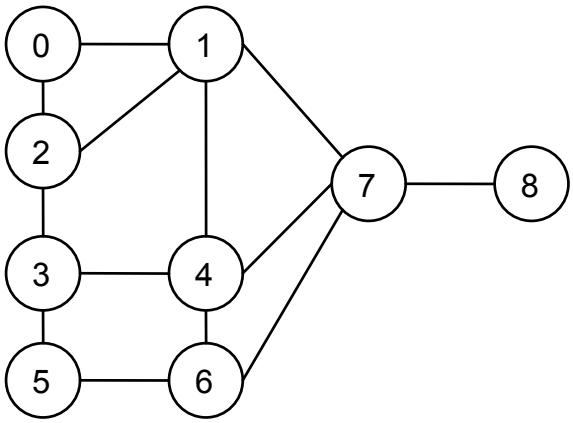

Рис. 11.10. Пример графа, моделирующего структуру сети на рис. 11.9

Дополнительная информация по проблеме разделения графов может быть получена, например, в Schloegel и др. (2000).

Задача оптимального разделения графов сама может являться предметом распараллеливания. Это бывает необходимо в тех случаях, когда вычислительной мощности и объема оперативной памяти обычных компьютеров недостаточно для эффективного решения задачи. Параллельные алгоритмы разделения графов рассматриваются во многих научных работах: Barnard (1995), Gilbert и др. (1987), Heath и др. (1995), Karypis и др. (1998, 1999), Raghavan (1995), Walshaw и др. (1999).

11.3.1. Постановка задачи оптимального разделения графов

Пусть дан взвешенный неориентированный граф $G=(V,E)$, каждой вершине $v \in V$ и каждому ребру $e \in E$ которого приписан вес. Задача оптимального разделения графа состоит в разбиении его вершин на непересекающиеся подмножества с максимально близкими суммарными весами вершин и минимальным суммарным весом ребер, проходящих между полученными подмножествами вершин.

Следует отметить возможную противоречивость указанных критериев разбиения графа – равновесность подмножеств вершин может не соответствовать минимальности весов граничных ребер и наоборот. В большинстве случаев необходимым является выбор того или иного компромиссного решения. Так, в случае невысокой доли коммуникаций может оказаться эффективным оптимизировать вес ребер только среди решений, обеспечивающих оптимальное разбиение множества вершин по весу.

Далее для простоты изложения учебного материала будем полагать веса вершин и ребер графа равными единице.

11.3.2. Метод рекурсивного деления пополам

Для решения задачи разбиения графа можно рекурсивно применить *метод бинарного деления*, при котором на первой итерации граф разделяется на две равные части, далее на втором шаге каждая из полученных частей также разбивается на две части и т.д. В данном подходе для разбиения графа на k частей необходимо $\log_2 k$ уровней рекурсии и выполнение $k-1$ деления пополам. В случае, когда требуемое количество разбиений k не является степенью двойки, каждое деление пополам необходимо осуществлять в соответствующем соотношении.

Поясним схему работы метода деления пополам на примере разделения графа на рис. 11.11 на 5 частей. Сначала граф следует разделить на 2 части в соотношении 2:3 (непрерывная линия), затем правую часть разбиения в отношении 1:3 (пунктирная линия), после этого осталось разделить 2 крайние подобласти слева и справа в отношении 1:1 (пунктир с точкой).

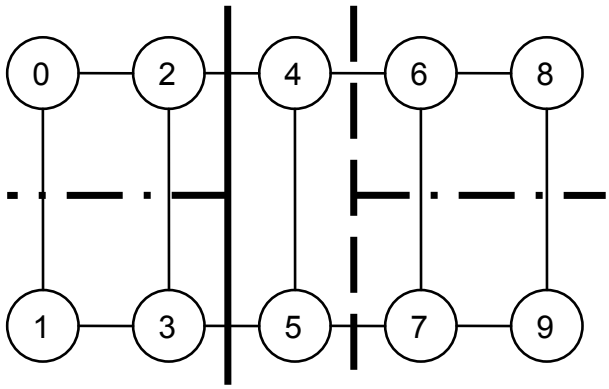

Рис. 11.11. Пример разбиения графа на 5 частей методом рекурсивного деления пополам

11.3.3. Геометрические методы

Геометрические методы (см., например, Berger и др. (1987), Gilbert и др. (1995), Heath и др. (1995), Miller и др. (1993), Nour-Omid и др. (1986), Patra и др. (1998), Pilkington и др. (1994), Raghavan (1993)) выполняют разбиение сетей, основываясь исключительно на координатной информации об узлах сети. Так как эти методы не принимают во внимание информацию о связности элементов сети, то они не могут явно привести к минимизации суммарного веса граничных ребер (в терминах графа, соответствующего сети). Для минимизации межпроцессорных коммуникаций геометрические методы оптимизируют некоторые вспомогательные показатели (например, длину границы между разделенными участками сети).

Обычно геометрические методы не требуют большого объема вычислений, однако качество их разбиения обычно уступает методам, принимающим во внимание связность элементов сети.

11.3.3.1. Покоординатное разбиение

Покоординатное разбиение (*coordinate nested dissection*)– это метод, основанный на рекурсивном делении пополам сети по наиболее длинной стороне. В качестве иллюстрации рис. 11.12 показан пример сети, при разделении которой именно такой способ разбиения дает существенно меньшее количество информационных связей между разделенными частями, по сравнению со случаем, когда сеть делится по меньшей (вертикальной) стороне.

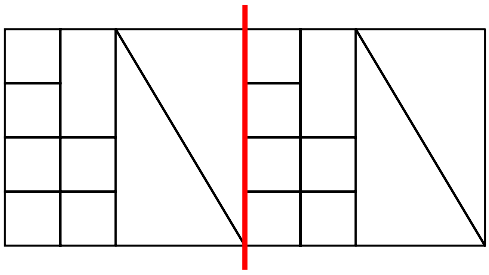

Рис. 11.12. Пример разделения сети графическим методом по наибольшей размерности (граница раздела показана жирной линией)

Общая схема выполнения метода состоит в следующем. Сначала вычисляются центры масс элементов сети. Полученные точки проектируются на ось, соответствующую наибольшей стороне разделяемой сети. Таким образом, мы получаем упорядоченный список всех элементов сети. Делением списка пополам (возможно, в нужной пропорции) мы получаем требуемую бисекцию. Аналогичным способом полученные фрагменты разбиения рекурсивно делятся на нужное число частей.

Метод координатного вложенного разбиения работает очень быстро и требует небольшого количества оперативной памяти. Однако, получаемое разбиение уступает по качеству более сложным и вычислительно-трудоемким методам. Кроме того, в случае сложной структуры сети алгоритм может получать разбиение с несвязанными подсетями.

11.3.3.2. Рекурсивный инерционный метод деления пополам

Предыдущая схема могла производить разбиение сети только по линии, перпендикулярной одной из координатных осей. Во многих случаях такое ограничение оказывается критичным для построения качественного разбиения. Достаточно повернуть сеть на рис. 11.12 под острым углом к координатным осям (см. рис. 11.13), чтобы убедиться в этом. Для минимизации границы между подсетями желательна возможность проведения линии разделения с любым требуемым углом поворота. Возможный способ определения угла поворота, используемый в *рекурсивном инерционном методе деления пополам* (*recursive inertial bisection*), состоит в использовании главной инерционной оси (см., например, Pothen, A. (1996)), считая элементы сети точечными массами. Линия бисекции, ортогональная полученной оси, как правило, дает границу наименьшей длины.

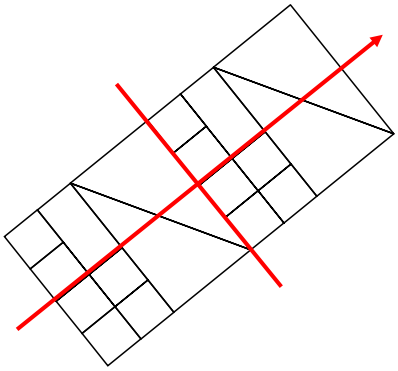

Рис. 11.13. Пример разделения сети методом рекурсивной инерционной бисекции. Стрелкой показана главная инерционная ось

11.3.3.3. Деление сети с использованием кривых Пеано

Одним из недостатков предыдущих графических методов является то, что при каждой бисекции эти методы учитывают только одну размерность. Таким образом, схемы, учитывающие больше размерностей, могут обеспечить лучшее разбиение.

Один из таких методов упорядочивает элементы в соответствии с позициями центров их масс вдоль кривых Пеано. Кривые Пеано – это кривые, полностью заполняющие фигуры больших размерностей (например, квадрат или куб). Применение таких кривых обеспечивает близость точек фигуры, соответствующих точкам, близким на кривой. После получения списка элементов сети, упорядоченного в соответствии с расположением на кривой, достаточно разделить список на необходимое число частей в соответствии с установленным порядком. Получаемый в результате такого подхода метод носит в литературе наименование *алгоритма деления сети с использованием кривых Пеано* (*space-filling curve technique*). Подробнее о методе можно прочитать в работах Ou и др. (1996), Patra и др. (1998), Pilkington и др. (1994).

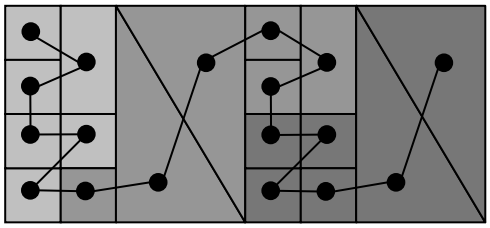

Рис. 11.14. Пример разделения сети на 3 части с использованием кривых Пеано

11.3.4. Комбинаторные методы

В отличие от геометрических методов, комбинаторные алгоритмы (см., например, George (1981), Schloegel и др. (2000)) обычно оперируют не с сетью, а с графом, построенным для этой сети. Соответственно, в отличие от геометрических схем комбинаторные методы не принимают во внимание информацию о близости расположения элементов сети друг относительно друга, руководствуясь только смежностью вершин графа. Комбинаторные методы обычно обеспечивают более сбалансированное

разбиение и меньшее информационное взаимодействие полученных подсетей. Однако комбинаторные методы имеют тенденцию работать существенно дольше, чем их геометрические аналоги.

11.3.4.1. Деление с учетом связности

С самых общих позиций понятно, что при разделении графа информационная зависимость между разделенными подграфами будет меньше, если соседние вершины (вершины, между которыми имеются дуги) будут находиться в одном подграфе. *Алгоритм деления графов с учетом связности* (*levelized nested dissection*) пытается достичь этого, последовательно добавляя к формируемому подграфу соседей. На каждой итерации алгоритма происходит разделение графа на 2 части. Таким образом, разделение графа на требуемое число частей достигается путем рекурсивного применения алгоритма.

Общая схема алгоритма может быть описана при помощи следующего набора правил.

```
1. Iteration = 0
2. Присвоение номера Iteration произвольной вершине графа
3. Присвоение ненумерованным соседям вершин с номером Iteration номера
   Iteration + 1
5. Iteration = Iteration + 1
6. Если еще есть неперенумерованные соседи, то переход на шаг 3
7. Разделение графа на 2 части в порядке нумерации
```

Алгоритм 11.2. Общая схема выполнения алгоритма деления графов с учетом связности

Для минимизации информационных зависимостей имеет смысл в качестве начальной выбирать граничную вершину. Поиск такой вершины можно осуществить методом, близким к рассмотренной схеме. Так, перенумеровав вершины графа в соответствии с алгоритмом 11.2 (начиная нумерацию из произвольной вершины), мы можем взять любую вершину с максимальным номером. Как нетрудно убедиться, она будет граничной.

Пример работы алгоритма приведен на рис. 11.15. Цифрами показаны номера, которые получили вершины в процессе разделения. Сплошной линией показана граница, разделяющая 2 подграфа. Также на рисунке показано лучшее решение (пунктирная линия). Очевидно, что полученное алгоритмом разбиение далеко от оптимального, так как в приведенном примере есть решение только с 3 пересеченными ребрами вместо 5.

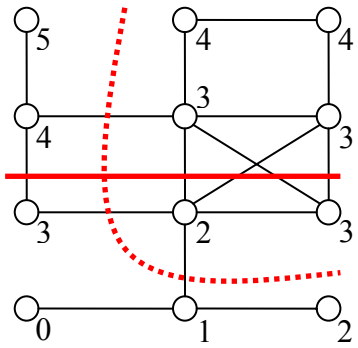

Рис. 11.15. Пример работы алгоритма деления графов с учетом связности

11.3.4.2. Алгоритм Кернигана-Лина

В *алгоритме Кернигана-Лина* (*Kernighan-Lin algorithm*) используется несколько иной подход для решения проблемы оптимального разбиения графа – при начале работы метода предполагается, что некоторое начальное разбиение графа уже существует, затем же имеющееся приближение улучшается в течение некоторого количества итераций. Используемый способ улучшения в алгоритме Кернигана-Лина состоит в обмене вершинами между подмножествами имеющегося разбиения графа (см. рис. 11.16). Для формирования требуемого количества частей графа может быть использована, как и ранее, рекурсивная процедура деления пополам.

Общая схема одной итерации алгоритма Кернигана-Лина может быть представлена следующим образом.

1. Формирование множества пар вершин для перестановки

   Из вершин, которые еще не были переставлены на данной итерации, формируются все возможные пары (в парах должны присутствовать по одной вершине из каждой части имеющегося разбиения графа).

2. Построение новых вариантов разбиения графа

   Каждая пара, подготовленная на шаге 1, поочередно используется для обмена вершин между частями имеющегося разбиения графа для получения множества новых вариантов деления.

3. Выбор лучшего варианта разбиения графа

   Для сформированного на шаге 2 множества новых делений графа выбирается лучший вариант. Данный способ фиксируется как новое текущее разбиение графа, а соответствующая выбранному варианту пара вершин отмечается как использованная на текущей итерации алгоритма.

4. Проверка использования всех вершин

   При наличии в графе вершин, еще неиспользованных при перестановках, выполнение итерации алгоритма снова продолжается с шага 1. Если же перебор вершин графа завершен, далее следует шаг 5.

5. Выбор наилучшего варианта разбиения графа

   Среди всех разбиений графа, полученных на шаге 3 проведенных итераций, выбирается (и фиксируется) наилучший вариант разбиения графа.

Алгоритм 11.3. Общая схема алгоритма Кернигана-Лина

Поясним дополнительно, что на шаге 2 итерации алгоритма перестановка вершин каждой очередной пары осуществляется для одного и того разбиения графа, выбранного до начала выполнения итерации или определенного на шаге 3. Общее количество выполняемых итераций, как правило, фиксируется заранее и является параметром алгоритма (за исключением случая остановки при отсутствии улучшения разбиения на очередной итерации).

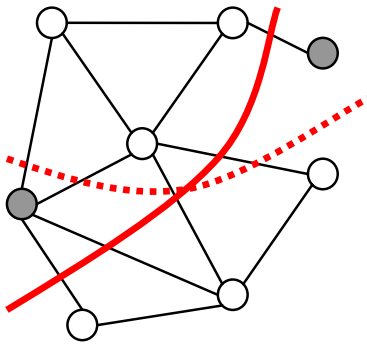

Рис. 11.16. Пример перестановки двух вершин (выделены серым) в методе Кернигана-Лина

11.3.5. Сравнение алгоритмов разбиения графов

Рассмотренные алгоритмы разбиения графов различаются точностью получаемых решений, временем выполнения и возможностями для распараллеливания (под точностью понимается величина близости получаемых при помощи алгоритмов решений к оптимальным вариантам разбиения графов). Выбор наиболее подходящего алгоритма в каждом конкретном случае является достаточно сложной и неочевидной задачей. Проведению такого выбора может содействовать сведенная воедино в табл. 11.5 (см. Schloegel и др. (2000)) общая характеристика ряда алгоритмов разделения графов, рассмотренных в данном разделе. Дополнительная информация по проблеме оптимального разбиения графов может быть получена, например, в Schloegel и др. (2000).

Таблица 11.5. Сравнительная таблица некоторых алгоритмов разделения графов

	Необходимость координатной информации	Точность	Время выполнения	Возможности для распарал-леливания

Покоординатное разбиение		да	●	●	●●●
Рекурсивный инерционный метод деления пополам		да	●●	●	●●●
Деление с учетом связности		нет	●●	●●	●●
Алгоритм Кернигана-Лина	1 итерация	нет	●●	●●	●
	10 итераций	нет	●●●◐	●●●	●●
	50 итераций	нет	●●●●	●●●◐	●●

Столбец "Необходимость координатной информации" отмечает использование алгоритмом координатной информации об элементах сети или вершинах графа.

Столбец "Точность" дает качественную характеристику величины приближения получаемых алгоритмом решений к оптимальных вариантам разбиения графов. Каждый дополнительный закрашенный кружок определяет примерно 10%-процентное улучшение точности приближения (соответственно, заштрихованный наполовину кружок означает 5%-процентное улучшение получаемого решения).

Столбец "Время выполнения" показывает относительное время, затрачиваемое различными алгоритмами разбиения. Каждый дополнительный закрашенный кружок соответствует увеличению времени разбиения примерно в 10 раз (заштрихованный наполовину кружок отвечает за увеличение времени разбиения примерно в 5 раз).

Столбец "Возможности для распараллеливания" характеризует свойства алгоритмов для параллельного выполнения. Алгоритм Кернигана-Лина при выполнении только одной итерации почти не поддается распараллеливанию. Этот же алгоритм при большем количестве итераций, а также метод деления с учетом связности, могут быть распараллелены со средней эффективностью. Алгоритм покоординатного разбиения и рекурсивный инерционный метод деления пополам обладают высокими показателями для распараллеливания.

11.4. Краткий обзор раздела

В разделе рассмотрены ряд алгоритмов для решения типовых задач обработки графов. Кроме того, в разделе приведен обзор методов разделения графа.

В подразделе 11.1 представлен *алгоритм Флойда* (*Floyd*) – дается общая вычислительная схема последовательного варианта метода, обсуждаются способы его распараллеливания, проводится анализ эффективности получаемых параллельных вычислений, рассматривается программная реализация метода и приводятся результаты вычислительных экспериментов. Используемый подход к распараллеливанию алгоритма Флойда состоит в разделении вершин графа между процессорами, а необходимое при этом информационное взаимодействие состоит в передаче одной строки матрицы смежности от одного процессора всем процессорам вычислительной системы на каждой итерации метода.

В подразделе 11.2 рассматривается *алгоритм Прима* (*Prim*) для решения задачи поиска минимального охватывающего дерева (остова) неориентированного взвешенного графа. Остовом графа называют связный подграф без циклов (дерево), содержащий все вершины исходного графа и ребра, имеющие минимальный суммарный вес. Для алгоритма дается общее описание его исходного последовательного варианта, определяются возможные способы его параллельного выполнения, определяются теоретические оценки ускорения и эффективности параллельных вычислений, рассматриваются результаты проведенных вычислительных экспериментов. Параллельный вариант алгоритма Прима, как и в предыдущем случае, основывается на разделении вершин графа между процессорами при несколько большем объеме информационных взаимодействий – на каждой итерации алгоритма необходимой является операция сбора данных на одном процессоре и последующая рассылка номера выбранной вершины графа всем процессорам вычислительной системы.

Рассматриваемая в подразделе 11.3 задача оптимального разделения графов является важной для многих научных исследований, использующих параллельные вычисления. Для примера в подразделе приведен общий способ перехода от двухмерной или трехмерной сети, моделирующей процесс вычислений, к соответствующему ей графу. Для решения задачи разбиения графов были рассмотрены *геометрические методы*, использующие при разделении сетей только координатную информацию об узлах сети, и *комбинаторные алгоритмы*, руководствующиеся смежностью вершин графа. К числу рассмотренных геометрических методов относятся *покоординатное разбиение* (*coordinate nested dissection*), *рекурсивный инерционный метод деления пополам* (*recursive inertial bisection*), *деление сети с использованием кривых Пеано* (*space-filling curve techniques*). К числу рассмотренных комбинаторных алгоритмов относятся *деление с учетом связности* (*levelized nested dissection*) и а*лгоритм Кернигана-*

Лина (*Kernighan-Lin algorithm*). Для сопоставления рассмотренных подходов приводится общая сравнительная характеристика алгоритмов по времени выполнения, точности получаемого решения, возможностей для распараллеливания и т.п.

11.5. Обзор литературы

Дополнительная информация по алгоритмам Флойда и Прима может быть получена, например, в Кормен и др. (1999).

Подробное рассмотрение вопросов, связанных с проблемой разделения графов, содержится в работах Schloegel и др. (2000), Berger и др. (1987), Gilbert и др. (1995), Heath и др. (1995), Miller и др. (1993), Nour-Omid и др. (1986), Patra и др. (1998), Pilkington и др. (1994), Raghavan (1993), George (1981).

Параллельные алгоритмы разделения графов рассматриваются в Barnard (1995), Gilbert и др. (1987), Heath и др. (1995), Karypis и др. (1998, 1999), Raghavan (1995), Walshaw и др. (1999).

11.6. Контрольные вопросы

1. Приведите определение графа. Какие основные способы используются для задания графов?
2. В чем состоит задача поиска всех кратчайших путей?
3. Приведите общую схему алгоритма Флойда. Какова трудоемкость алгоритма?
4. В чем состоит способ распараллеливания алгоритма Флойда?
5. В чем заключается задача нахождения минимального охватывающего дерева? Приведите пример использования задачи на практике.
6. Приведите общую схему алгоритма Прима. Какова трудоемкость алгоритма?
7. В чем состоит способ распараллеливания алгоритма Прима?
8. В чем отличие геометрических и комбинаторных методов разделения графа? Какие методы являются более предпочтительными? Почему?
9. Приведите описание метода покоординатного разбиения и алгоритма разделения с учетом связности. Какой из этих методов является более простым для реализации?

11.7. Задачи и упражнения

1. Используя приведенный программный код, выполните реализацию параллельного алгоритма Флойда. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом параметров используемой вычислительной системы. Сравните полученные оценки с экспериментальными данными.
2. Выполните реализацию параллельного алгоритма Прима. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом параметров используемой вычислительной системы. Сравните полученные оценки с экспериментальными данными.
3. Разработайте программную реализацию алгоритма Кернигана – Лина. Дайте оценку возможности распараллеливания этого алгоритма.