ГЛАВА 10
ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ НА ГРАФАХ

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 можно выделить 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 можно рекомендовать работу [17].

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. 10.1.

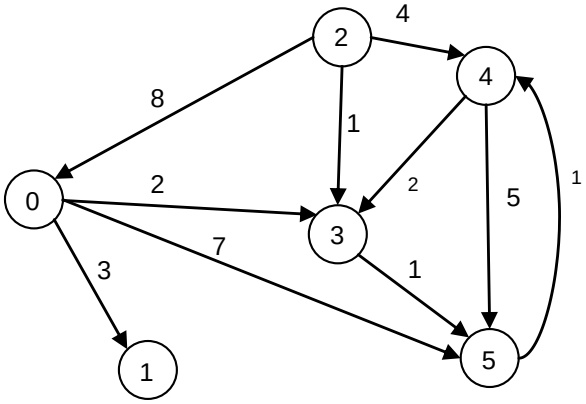

Рис. 10.1. Пример взвешенного ориентированного графа

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. Такие списки получил название *списков примыканий*. Список примыканий графа $G = (V, R)$ с числом вершин n записывается в виде одномерного массива длины n, каждый элемент которого представляет собой ссылку на список. Такой список приписан каждой вершине графа, и он содержит по одному элементу на каждую вершину графа, соседнюю с данной. Каждое звено списка хранит номер соседней вершины и вес ребра. Например, список примыканий, соответствующий графу, изображенному на рис. 10.1, приведен на рис. 10.2.

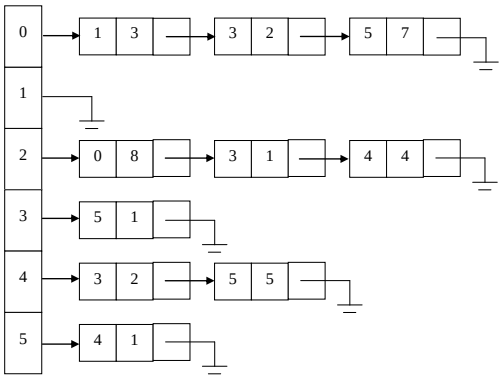

Рис. 10.2. Список примыканий для графа из рис. 10.1

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 этот знак может быть заменен, например, на любое отрицательное число). Так, например, матрица смежности, соответствующая графу на рис. 10.1, приведена на рис. 10.3.

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

Рис. 10.3. Матрица смежности для графа из рис. 10.1

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, подлежащих обработке, при рассмотрении всех перечисленных задач будут использоваться матрицы смежности. Для представления минимального охватывающего дерева, полученного в результате выполнения алгоритма Прима, будет использоваться список примыканий.

10.1. 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 заданных расстояниях между населенными пунктами и другие подобного рода задачи.

В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется *алгоритм Флойда* (*Floyd*) (см, например, [17]).

10.1.1. 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:

```
// Программа 10.1
// 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]);
```

(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 работе [17].

10.1.2. 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 минимального значения (см. алгоритм 10.1). 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).

10.1.3. Масштабирование и распределение подзадач по процессорам

Как правило, число доступных вычислительных элементов 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 C массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы A на горизонтальные полосы.

10.1.4. Анализ эффективности параллельных вычислений

Как и ранее, при анализе эффективности параллельного алгоритма Флойда будем предполагать, что время выполнения складывается из времени вычислений (которые могут быть выполнены вычислительными элементами параллельно) и времени, необходимого на загрузку необходимых данных из оперативной памяти в кэш. Доступ к памяти осуществляется строго последовательно.

Для выполнения алгоритма Флойда над графом, содержащим n вершин, требуется выполнение n итераций алгоритма, на каждой из которых параллельно выполняется обновление всех элементов матрицы смежности. Значит, время выполнения вычислений составляет:

$$T_p\ \ calc\ = n \cdot \frac{n^2}{p} \cdot \tau, \tag{10.1}$$

где τ есть время выполнения операции выбора минимального значения.

Если количество вершин в графе настолько велико, что описывающая граф матрица смежности не может быть полностью помещена в кэш, то на каждой итерации внешнего цикла k выполняется вытеснение «первых» элементов матрицы – для того, чтобы записать на их место значения, располагаемые в конце матрицы. Для перехода к выполнению следующей итерации необходимо снова считать значения из начала матрицы. Таким образом, происходит повторное считывание всех элементов матрицы из оперативной памяти в кэш, и затраты на доступ к памяти составляют:

$$T_p\ \ mem\ = n \cdot \frac{64n^2}{\beta}, \tag{10.2}$$

где β есть пропускная способность канала доступа к оперативной памяти

Если, как и в предыдущих главах, учесть латентность памяти, то время доступа к памяти можно получить по следующей формуле:

$$T_p\ \ mem\ = n^3 \cdot \left(\alpha + \frac{64}{\beta} \right). \tag{10.3}$$

При получении итоговой оценки времени выполнения параллельного алгоритма Флойда необходимо также учитывать затраты на организацию и закрытие параллельных секций:

$$T_p = \frac{n^3}{p} \cdot \tau + n^3 \cdot \left(\alpha + \frac{64}{\beta} \right) + n\delta, \tag{10.4}$$

где δ есть накладные расходы на организацию параллельности на каждой итерации алгоритма.

Построенная модель является моделью на худший случай. Для более точной оценки необходимо учесть частоту кэш промахов γ (см. п. 6.5.4):

$$T_p = \frac{n^3}{p} \cdot \tau + \gamma \cdot n^3 \cdot \left(\alpha + \frac{64}{\beta} \right) + n\delta, \tag{10.5}$$

10.1.5. Программная реализация

Представим возможный вариант программы, выполняющей параллельный алгоритм Флойда поиска всех кратчайших путей.

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

```
// Программа 10.2
// Параллельный алгоритм Флойда
void main(int argc, char* argv[]) {
  double *pMatrix;     // Матрица смежности
  int Size;            // Количество вершин

  // Инициализация данных
  ProcessInitialization(pMatrix, Size);

  // Выполнение параллельного алгоритма Флойда
  ParallelFloyd(pMatrix, Size);

  // Завершение вычислений
  ProcessTermination(pMatrix);
}
```

2. Функция ProcessInitialization. Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества вершин в графе, выделения памяти для хранения матрицы смежности и для заполнения этой матрицы значениями. Начальные значения элементов матрицы смежности задаются в функции *RandomDataInitialization.*

```
// Функция выделения памяти и инициализации данных
void ProcessInitialization(double *&pMatrix,
  int& Size) {
  do {
    printf("Введите количество вершин: ");
    scanf("%d", &Size);
    if(Size <= 2)
      printf("Количество должно быть больше 2 \n");
  } while(Size <= 2);

  printf("Количество вершин в графе %d \n", Size);

  // Выделение памяти для матрицы смежности
  pMatrix = new double[Size * Size];
```

```
  // Инициализация данных
  RandomDataInitialization(pMatrix, Size);
}
```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

3. Функция ParallelFloyd. Данная функция выполняет параллельный алгоритм Флойда поиска кратчайших путей для всех пар вершин.

```
// Функция для параллельного алгоритма Флойда
void ParallelFloyd(double *pMatrix, int Size) {
  double t1, t2;
  for(int k = 0; k < Size; k++)
#pragma omp parallel for private (t1, t2)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] +
            pMatrix[k * Size + j];
          pMatrix[i * Size + j] = Min(t1, t2);
      }
}
```

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

```
double Min(double A, double B) {
  double 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;
}
```

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

10.1.6. Результаты вычислительных экспериментов

Эксперименты проводились в тех же условиях, что и в предыдущих главах. Для снижения сложности построения теоретических оценок времени выполнения алгоритмов при компиляции и построении программ для проведения вычислительных экспериментов функция оптимизации кода компилятором была отключена (результаты оценки влияния компиляторной оптимизации на эффективность программного кода приведены в п. 7.2.4).

Результаты вычислительных экспериментов приведены в табл. 10.1. Времена выполнения алгоритмов указаны в секундах.

Таблица 10.1.

Результаты вычислительных экспериментов для параллельного алгоритма Флойда (при использовании двух и четырех вычислительных ядер)

Количество вершин в графе	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
100	0,0549	0,0259	2,1227	0,0133	4,1305
200	0,4389	0,2004	2,1898	0,1024	4,2870
300	1,4739	0,6708	2,1971	0,3416	4,3151
400	3,4846	1,5809	2,2042	0,8069	4,3187
500	6,8002	3,0822	2,2063	1,5710	4,3285
600	11,7399	5,3205	2,2065	2,7112	4,3301
700	18,6608	8,4392	2,2112	4,2986	4,3411
800	27,9942	12,5910	2,2234	6,3962	4,3767
900	39,9018	17,9152	2,2273	9,0951	4,3872
1000	54,7534	24,5916	2,2265	12,4759	4,3887

В табл. 10.2, 10.3 и на рис. 10.4, 10.5 представлены результаты сравнения времени выполнения параллельного алгоритма Флойда с использованием двух потоков со временем, полученным при помощи моделей (10.4) и (10.5).

Чтобы оценить время одной операции выбора минимального значения τ, измерим время выполнения последовательного алгоритма Флойда при малых объемах данных, таких, чтобы матрица смежности, описывающая граф, могла быть полностью помещена в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицу

случайными значениями – выполнение этого действия гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, т. к. нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение τ, равное 55,178 нс.

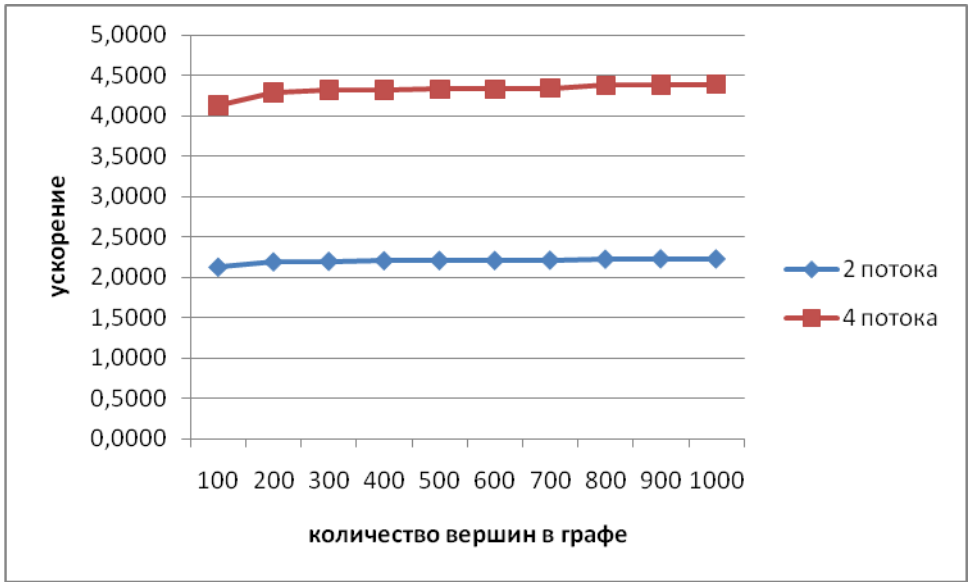

Рис. 10.4. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма Флойда

Оценки времени латентности α и величины пропускной способности канала доступа к оперативной памяти β проводилась в п. 6.5.4 и определены для используемого вычислительного узла как α = 8,31 нс. и β = 12,44 Гб/с. Величина накладных расходов δ на параллельность была оценена в главе 6 и составляет 0,25·мкс. Частота кэш-промахов для двух потоков оказалась равной 0,0083, а для четырех потоков эта величина была оценена как 0,0090.

Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах сортируемого массива составляет не более 1%.

Таблица 10.2.

Сравнение экспериментального и теоретического времени выполнения параллельного метода Флойда с использованием двух потоков

Размер массива	T_p	T_p^* (*calc*) (модель)	Модель 10.4 – оценка сверху		Модель 10.5 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
100	0,0259	0,0276	0,0131	0,0407	0,0001	0,0277
200	0,2004	0,2208	0,1048	0,3256	0,0009	0,2216
300	0,6708	0,7450	0,3537	1,0987	0,0029	0,7479
400	1,5809	1,7658	0,8385	2,6043	0,0070	1,7728
500	3,0822	3,4488	1,6377	5,0864	0,0136	3,4623
600	5,3205	5,9594	2,8299	8,7893	0,0235	5,9829
700	8,4392	9,4632	4,4938	13,9570	0,0373	9,5005
800	12,5910	14,1258	6,7079	20,8337	0,0557	14,1814
900	17,9152	20,1126	9,5509	29,6635	0,0793	20,1919
1000	24,5916	27,5893	13,1014	40,6906	0,1087	27,6980

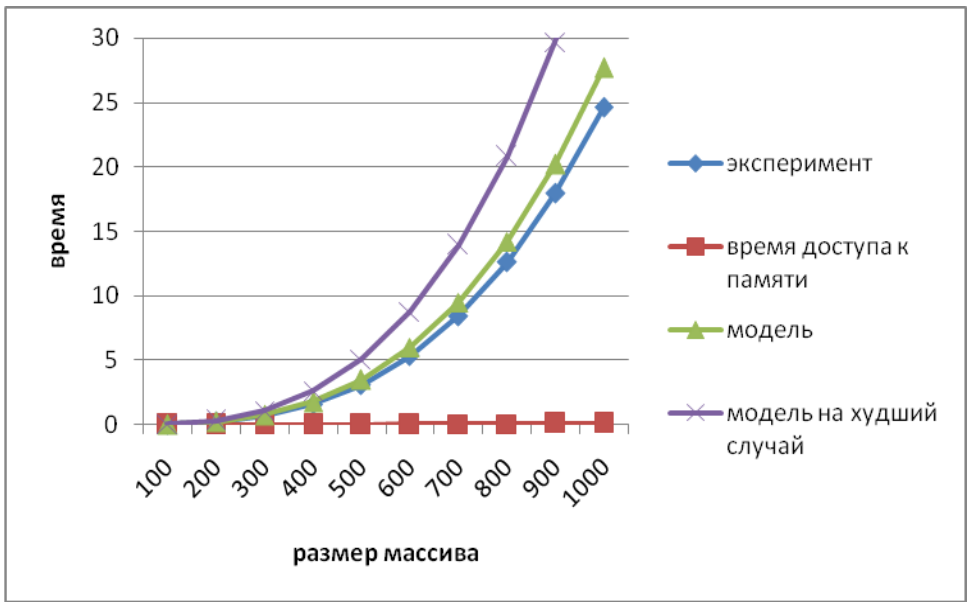

Рис. 10.5. График зависимости экспериментального и теоретического времен выполнения параллельного алгоритма Флойда от объема исходных данных при использовании двух потоков

Таблица 10.3.

Сравнение экспериментального и теоретического времен выполнения параллельного метода Флойда с использованием четырех потоков

Размер массива	T_p	T_p^* (*calc*) (модель)	Модель 10.4 – оценка сверху		Модель 10.5 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
100	0,0133	0,0138	0,0131	0,0269	0,0001	0,0139
200	0,1024	0,1104	0,1048	0,2152	0,0009	0,1113
300	0,3416	0,3725	0,3537	0,7263	0,0032	0,3757
400	0,8069	0,8829	0,8385	1,7214	0,0075	0,8905
500	1,5710	1,7244	1,6377	3,3621	0,0147	1,7392
600	2,7112	2,9798	2,8299	5,8097	0,0255	3,0052
700	4,2986	4,7317	4,4938	9,2255	0,0404	4,7721
800	6,3962	7,0630	6,7079	13,7709	0,0604	7,1234
900	9,0951	10,0564	9,5509	19,6073	0,0860	10,1424
1000	12,4759	13,7948	13,1014	26,8961	0,1179	13,9127

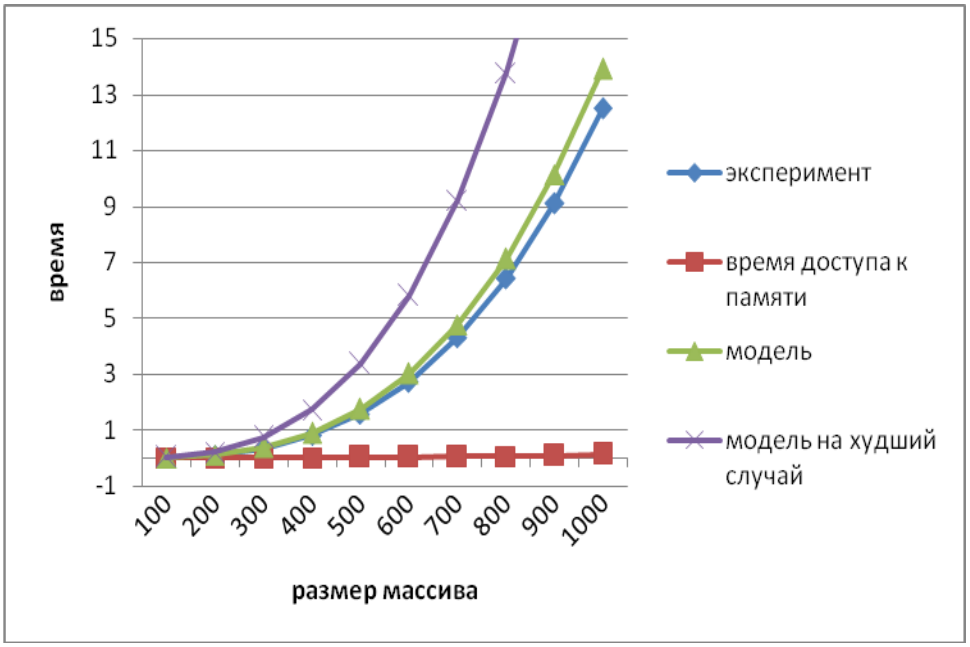

Рис. 10.6. График зависимости экспериментального и теоретического времен выполнения параллельного алгоритма Флойда от объема исходных данных при использовании четырех потоков

10.2. 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. 10.7.

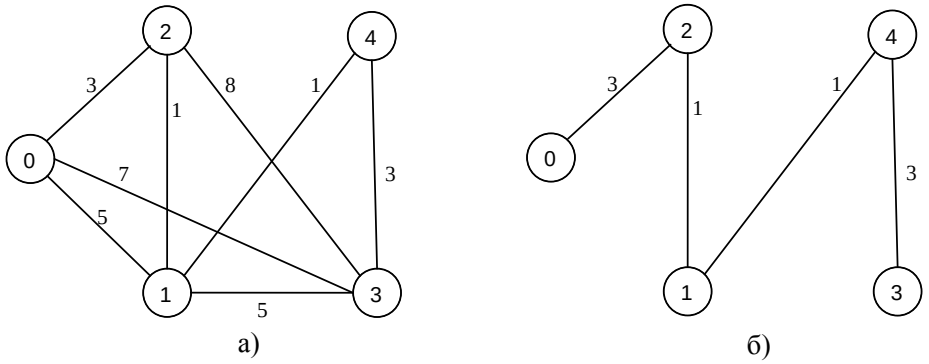

Рис. 10.7. Пример (а) взвешенного неориентированного графа и соответствующему ему (б) минимально охватывающего дерева

Дадим общее описание алгоритма решения поставленной задачи, известного под названием *метода Прима* (*Prim*), более полная информация может быть получена, например, в [17].

10.2.1. 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 $O(n^2)$.

10.2.2. Программная реализация последовательного алгоритма Прима

Представим возможный вариант программы, выполняющей последовательный алгоритм Прима построения минимального охватывающего дерева.

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

```
// Программа 10.3
// Последовательный алгоритм Прима
void main(int argc, char* argv[]) {
  double *pMatrix;               // Матрица смежности
  TTreeNode** pMinSpanningTree;  // Мин. охв. дерево
  int  Size;                     // Количество вершин
```

```
  // Инициализация данных
  ProcessInitialization(pMatrix, pMinSpanningTree,
    Size);

  // Выполнение последовательного алгоритма Прима
  SerialPrim(pMatrix, pMinSpanningTree, Size);

  // Завершение вычислений
  ProcessTermination(pMatrix, pMinSpanningTree);
}
```

Поясним использование структур данных. Как уже отмечалось выше, для хранения минимального охватывающего дерева будет использоваться список примыканий, который представляет собой массив, число элементов в котором совпадает с числом вершин в графе; каждый *i*-й элемент массива есть указатель на звено списка, описывающее вершину графа, соседнюю с вершиной *i*. Для описания вершины графа предназначена структура *TTreeNode*:

```
struct TTreeNode {
  int NodeNum;      // Номер вершины
  double Distance;  // Вес дуги
  TTreeNode* pNext; // Следующее звено списка
} ;
```

2. Функция ProcessInitialization. Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества вершин в графе, выделения памяти для хранения матрицы смежности и для заполнения этой матрицы значениями. Начальные значения элементов матрицы смежности задаются в функции *RandomDataInitialization.*

```
// Функция выделения памяти и инициализации данных
void ProcessInitialization (double *&pMatrix,
  TTreeNode** &pMinSpanningTree, int& Size) {
  do {
    printf("Введите количество вершин: ");
    scanf("%d", &Size);
    if(Size <= 2)
      printf("Количество должно быть больше 2\n");
  } while(Size <= 2);

  printf("Количество вершин в графе\n", Size);

  // Выделение памяти для матрицы смежности
```

```
  pMatrix = new double[Size * Size];
  // Выделение памяти для охватывающего дерева
  pMinSpanningTree = new TTreeNode* [Size];

  // Инициализация данных
  RandomDataInitialization(pMatrix, pMinSpanningTree,
    Size);
}
```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

3. Функция SerialPrim. Данная функция выполняет последовательный алгоритм Прима построения минимального охватывающего дерева.

```
// Функция для последовательного алгоритма Прима
void SerialPrim(double *pMatrix,
  TTreeNode** pMinSpanningTree, int Size) {
  // Номер последней вершины, добавленной в дерево
  int LastAdded;
  // Номер вершины, ближайщей до дерева
  TGraphNode NearestNode;
  // Список вершин, еще не добавленных в дерево
  TGraphNode **NotInMinSpanningTree =
    new TGraphNode* [Size-1];

  // Начальная вершина имеет номер 0
  LastAdded = 0;
  // Формирование списка вершин графа
  for (int i=0; i<Size-1; i++) {
    NotInMinSpanningTree[i] = new TGraphNode;
    NotInMinSpanningTree[i]->NodeNum = i+1;
    NotInMinSpanningTree[i]->Distance = -1.0f;
    NotInMinSpanningTree[i]->ParentNodeNum = -1;
  }

  // Итерации алгоритма Прима
  for (int Iter=1; Iter<Size; Iter++) {
    // Корректировка расстояний
    for (int i=0; i<Size-1; i++) {
      if (NotInMinSpanningTree[i] != NULL) {
        double t1 = NotInMinSpanningTree[i]->Distance;
        double t2 =
```

```
          pMatrix[(NotInMinSpanningTree[i]->NodeNum)*
          Size+LastAdded];
        if (((t1<0) && (t2>0)) || ((t1>0) && (t2>0) &&
         (t1>t2))) {
          NotInMinSpanningTree[i]->Distance = t2;
          NotInMinSpanningTree[i]->ParentNodeNum =
            LastAdded;
        }
      }
    }

    // Выбор ближайщей вершины
    NearestNode.NodeNum = -1;
    NearestNode.Distance = MAX_VALUE;
    for (int i=0; i<Size-1; i++) {
      if (NotInMinSpanningTree[i] != NULL) {
        double t1 = NotInMinSpanningTree[i]->Distance;
        double t2 = NearestNode.Distance;
        if ((t1>0) && (t1<t2)) {
          NearestNode.Distance = t1;
          NearestNode.NodeNum =
            NotInMinSpanningTree[i]->NodeNum;
        }
      }
    }

    // Добавление ближайщей вершины в дерево
    pMinSpanningTree[NearestNode.NodeNum] =
      new TTreeNode;
    pMinSpanningTree[NearestNode.NodeNum]->NodeNum =
      NotInMinSpanningTree[NearestNode.NodeNum-1]->
      ParentNodeNum;
    pMinSpanningTree[NearestNode.NodeNum]->Distance =
      NearestNode.Distance;
    pMinSpanningTree[NearestNode.NodeNum]->pNext =
      NULL;

    int Parent =
      NotInMinSpanningTree[NearestNode.NodeNum-1]->
        ParentNodeNum;
    if (pMinSpanningTree[Parent] != NULL) {
      TTreeNode *tmp = new TTreeNode;
      tmp->pNext = pMinSpanningTree[Parent]->pNext;
```

```
      tmp->Distance = NearestNode.Distance;
      tmp->NodeNum = NearestNode.NodeNum;
      pMinSpanningTree[Parent]->pNext = tmp;
    }
    else {
      pMinSpanningTree[Parent] = new TTreeNode;
      pMinSpanningTree[Parent]->pNext = NULL;
      pMinSpanningTree[Parent]->Distance =
        NearestNode.Distance;
      pMinSpanningTree[Parent]->NodeNum =
        NearestNode.NodeNum;
    }

    LastAdded = NearestNode.NodeNum;
    delete
      NotInMinSpanningTree[NearestNode.NodeNum-1];
    NotInMinSpanningTree[NearestNode.NodeNum-1] =
      NULL;
  }

  delete [] NotInMinSpanningTree;
}
```

Структура данных *TGraphNode* предназначена для описания вершины, не включенной в состав МОД. Каждый экземпляр этой структуры содержит номер вершины, текущее расстояние до МОД и номер вершины в МОД, смежной с ней:

```
struct TGraphNode {
  int NodeNum;       // Номер вершины
  double Distance;   // Расстояние до дерева
  int ParentNodeNum; // Номер вершины в дереве
};
```

10.2.3. 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 осу-

ществляться для каждой вершины графа в отдельности, нахождение дуги минимального веса может быть реализовано по каскадной схеме и т. д.

Распределение данных между вычислительными элементами должно обеспечивать независимость перечисленных операций алгоритма Прима.

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 вычислительных элементов, и выполнить редукцию полученных значений;
- номер выбранной вершины для включения в охватывающее дерево передается всем вычислительным элементам;
- обновляются наборы величин d_i с учетом добавления новой вершины.

10.2.4. Анализ эффективности параллельных вычислений

Как и ранее, при анализе эффективности параллельного алгоритма Прима будем предполагать, что время выполнения складывается из времени вычислений (которые могут быть выполнены вычислительными элементами параллельно) и времени, необходимого на загрузку необходимых данных из оперативной памяти в кэш. Доступ к памяти осуществляется строго последовательно.

Для выполнения алгоритма Прима над графом, содержащим n вершин, требуется выполнение n итераций алгоритма, на каждой из которых параллельно выполняется пересчет расстояний от вершин, не входящих в состав МОД, до МОД, и выбор вершины, расстояние от которой минимально. Следовательно, на каждой i-й итерации алгоритма выполняется $(n - i)$ операций выбора минимума для пересчета расстояний и $(n - i)$ операций сравнения для выбора ближайшей вершины (кроме того, после выбора потоками «локальных» ближайших вершин необходимо выполнить редукцию полученных значений для получения «глобальной» ближайшей вершины). Таким образом, для оценки времени выполнения вычислений может быть использовано соотношение:

$$T_p\ calc = \left(2 \cdot \frac{\sum_{i=0}^{n-1} n-i}{p} + \log_2 p \right) \cdot \tau = \left(2 \cdot \frac{n^2/2}{p} + \log_2 p \right) \cdot \tau = \left(\frac{n^2}{p} + \log_2 p \right) \cdot \tau, \quad (10.6)$$

где τ есть время выполнения операции выбора минимального значения.

Если количество вершин в исходном графе настолько велико, что описывающая граф матрица смежности не может быть полностью помещена в кэш, то на каждой итерации внешнего цикла происходит считывание из оперативной памяти в кэш вычислительных элементов необходимых значений. Так, для пересчета расстояний от вершин, не включенных в состав МОД, до МОД, необходимо знать расстояние от этих вершин до вершины, включенной в состав МОД на последнем шаге. Следовательно, на i-й итерации алгоритма Прима необходимо загрузить из оперативной памяти $(n - i)$ значений. Каждое значение – это число с расширенной точностью типа double, то есть занимает 8 байт. Следует напомнить, что данные из оперативной памяти считываются строками по 64 байта. Таким образом, затраты на доступ к памяти составляют:

$$T_p \ \ mem \ = \sum_{i=0}^{n-1} \frac{64 \cdot \ n-i}{\beta} = 64 \cdot \frac{n^2 / 2}{\beta} \qquad (10.7)$$

Также следует учесть латентность оперативной памяти α:

$$T_p \ \ mem \ = (n^2/2) \cdot \left(\alpha + \frac{64}{\beta} \right) \qquad (10.8)$$

Необходимо отметить, что при выборе вершины, находящейся на наименьшем расстоянии от минимального охватывающего дерева, и при добавлении новой вершины в состав МОД, используются структуры данных сравнительно небольшого размера, которые сохраняются в кэш при переходе от одной итерации к другой.

При получении итоговой оценки времени выполнения параллельного алгоритма Прима необходимо также учитывать затраты на организацию и закрытие параллельных секций. На каждой итерации алгоритма создаются две параллельные секции: первая отвечает за параллельное выполнение пересчета расстояний, а вторая – за выбор ближайшей вершины:

$$T_p = \left(\frac{n^2}{p} + \log_2 p \right) \cdot \tau + (n^2 / 2) \cdot \left(\alpha + \frac{64}{\beta} \right) + 2n\delta \text{,,} \qquad (10.9)$$

где δ есть накладные расходы на организацию параллельности на каждой итерации алгоритма.

Для улучшения точности модели необходимо учесть частоту кэш-промахов γ (см. п. 6.5.4):

$$T_p = \left(\frac{n^2}{p} + \log_2 p \right) \cdot \tau + \gamma (n^2 / 2) \cdot \left(\alpha + \frac{64}{\beta} \right) + 2n\delta \text{,} \qquad (10.10)$$

10.2.5. Программная реализация параллельного алгоритма Прима

Представим возможный вариант программы, выполняющей параллельный алгоритм Прима построения минимального охватывающего дерева.

Для распараллеливания этапа пересчета расстояний от вершин, не включенных в состав минимального охватывающего дерева, до МОД, достаточно распределить итерации цикла, просматривающего все элементы массива *NotInSpanningTree*, между потоками параллельной программы при помощи директивы *parallel for*. Однако, поскольку некоторые элементы этого массива могут быть пустыми и, следовательно, не подлежат обработке, то для лучшей балансировки вычислительной нагрузки вычислительных элементов следует использовать динамическое планирование.

Для выбора вершины, ближайшей к уже построенной части МОД, применим следующий подход. В каждом потоке заведем локальную переменную *ThreadNearestNode* для хранения «локальной» ближайшей вершины. Выбор «локальных» ближайших вершин выполняется потоками параллельно. Далее выполняется редукция полученных значений при помощи механизма критических секций.

```
// Программа 10.4
// Функция для параллельного алгоритма Прима
void ParallelPrim(double *pMatrix,
  TTreeNode** pMinSpanningTree,  int Size) {
  // Номер последней вершины, добавленной в дерево
  int LastAdded;
  // Номер вершины, ближайщей до дерева
  TGraphNode NearestNode;
  // Список вершин, еще не добавленных в дерево
  TGraphNode **NotInMinSpanningTree =
    new TGraphNode* [Size-1];

  // Начальная вершина имеет номер 0
  LastAdded = 0;
  // Формирование списка вершин графа
  for (int i=0; i<Size-1; i++)
  {
    NotInMinSpanningTree[i] = new TGraphNode;
    NotInMinSpanningTree[i]->NodeNum = i+1;
    NotInMinSpanningTree[i]->Distance = -1.0f;
    NotInMinSpanningTree[i]->ParentNodeNum = -1;
  }
```

```
  // Итерации алгоритма Прима
  for (int Iter=1; Iter<Size; Iter++)
  {
    // Корректировка расстояний
#pragma omp parallel for schedule (dynamic,1)
    for (int i=0; i<Size-1; i++) {
      if (NotInMinSpanningTree[i] != NULL) {
        double t1 = NotInMinSpanningTree[i]->Distance;
        double t2 =
          pMatrix[(NotInMinSpanningTree[i]->NodeNum)*
          Size+LastAdded];
        if (((t1<0) && (t2>0)) || ((t1>0) && (t2>0) &&
         (t1>t2))) {
          NotInMinSpanningTree[i]->Distance = t2;
          NotInMinSpanningTree[i]->ParentNodeNum =
           LastAdded;
        }
      }
    }

    // Выбор ближайщей вершины
    NearestNode.NodeNum = -1;
    NearestNode.Distance = MAX_VALUE;
#pragma omp parallel
    {
      TGraphNode ThreadNearestNode;
      ThreadNearestNode.NodeNum = -1;
      ThreadNearestNode.Distance = MAX_VALUE;
#pragma omp for schedule (dynamic,1)
      for (int i=0; i<Size-1; i++) {
        if (NotInMinSpanningTree[i] != NULL) {
          double t1=NotInMinSpanningTree[i]->Distance;
          double t2=ThreadNearestNode.Distance;
          if ((t1>0) && (t1<t2)) {
            ThreadNearestNode.Distance = t1;
            ThreadNearestNode.NodeNum =
              NotInMinSpanningTree[i]->NodeNum;
          }
        }
      } // for
#pragma omp critical
      {
        if (ThreadNearestNode.Distance <
```

```
          NearestNode.Distance) {
          NearestNode.Distance =
            ThreadNearestNode.Distance;
          NearestNode.NodeNum =
            ThreadNearestNode.NodeNum;

        }
      } // pragma omp critical
    } // pragma omp parallel

    // Добавление ближайщей вершины в дерево
    pMinSpanningTree[NearestNode.NodeNum] =
      new  TTreeNode;
    pMinSpanningTree[NearestNode.NodeNum]->NodeNum =
      NotInMinSpanningTree[NearestNode.NodeNum-1]->
      ParentNodeNum;
    pMinSpanningTree[NearestNode.NodeNum]->Distance =
     NearestNode.Distance;
    pMinSpanningTree[NearestNode.NodeNum]->pNext =
      NULL;

    int Parent =
      NotInMinSpanningTree[NearestNode.NodeNum-1]->
      ParentNodeNum;
    if (pMinSpanningTree[Parent] != NULL) {
      TTreeNode *tmp = new TTreeNode;
      tmp->pNext = pMinSpanningTree[Parent]->pNext;
      tmp->Distance = NearestNode.Distance;
      tmp->NodeNum = NearestNode.NodeNum;
      pMinSpanningTree[Parent]->pNext = tmp;
    }
    else {
      pMinSpanningTree[Parent] = new TTreeNode;
      pMinSpanningTree[Parent]->pNext = NULL;
      pMinSpanningTree[Parent]->Distance =
        NearestNode.Distance;
      pMinSpanningTree[Parent]->NodeNum =
        NearestNode.NodeNum;
    }

    LastAdded = NearestNode.NodeNum;
    delete
      NotInMinSpanningTree[NearestNode.NodeNum-1];
```

```
    NotInMinSpanningTree[NearestNode.NodeNum-1] =
      NULL;
  }
  delete [] NotInMinSpanningTree;
}
```

10.2.6. Результаты вычислительных экспериментов

Результаты вычислительных экспериментов приведены в табл. 10.4. Времена выполнения алгоритмов указаны в секундах

В табл. 10.5, 10.6 и на рис. 10.9, 10.10 представлены результаты сравнения времени выполнения параллельного метода Прима с использованием двух и четырех потоков со временем, полученным при помощи моделей (10.9) и (10.10).

Чтобы оценить время одной операции выбора минимального значения τ, воспользуемся уже известным подходом. Измерим время выполнения последовательного алгоритма Прима при малых объемах данных, таких, чтобы матрица смежности, описывающая граф, могла быть полностью помещена в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений, заполним матрицу случайными значениями. Выполнение этого действия гарантирует предварительное перемещение данных в кэш..

Таблица 10.4.

Результаты вычислительных экспериментов для параллельного алгоритма Прима (при использовании двух и четырех вычислительных ядер)

Количество вершин в графе	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
100	0,0005	0,0009	0,5351	0,0009	0,5333
200	0,0017	0,0024	0,7209	0,0020	0,8487
300	0,0038	0,0038	0,9942	0,0037	1,0148
400	0,0068	0,0059	1,1392	0,0048	1,3983
500	0,0109	0,0088	1,2367	0,0070	1,5501
600	0,0167	0,0120	1,3934	0,0089	1,8820
700	0,0236	0,0160	1,4696	0,0114	2,0725
800	0,0332	0,0208	1,5956	0,0144	2,3017
900	0,0467	0,0256	1,8257	0,0177	2,6315
1000	0,0616	0,0328	1,8802	0,0209	2,9435

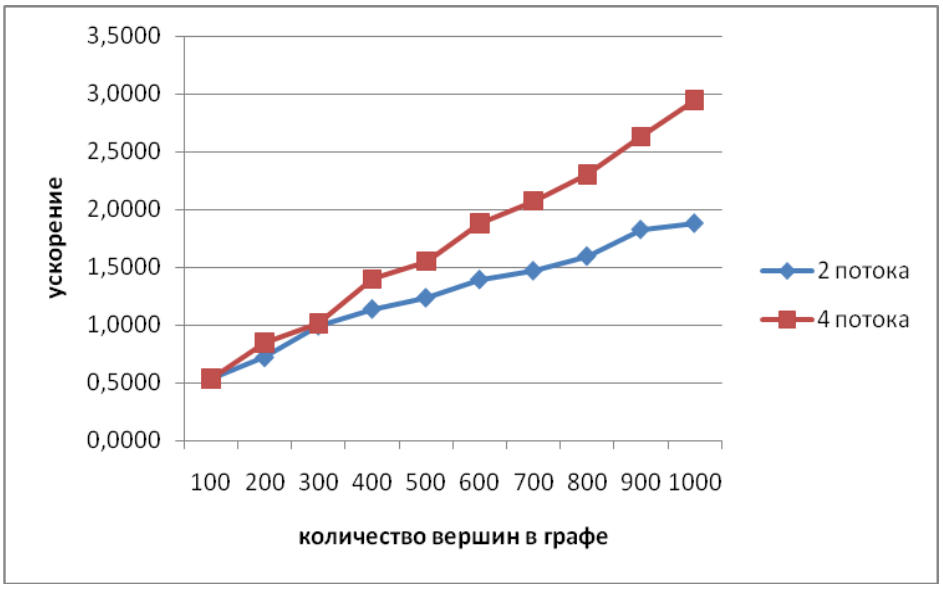

Рис. 10.8. Зависимость ускорения от количества исходных данных при выполнении параллельного метода Прима

Далее при решении задачи все время будет тратиться непосредственно на вычисления, т. к. нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение τ, равное 47,5 нс. Как и ранее, латентность α и пропускная способность канала доступа к оперативной памяти β являются равными α = 8,31 нс. и β = 12,44 Гб/с.

Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,5656, а для четырех потоков эта величина была оценена как 0,6614. Как отмечалось в главе 6, время δ, необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

Таблица 10.5.

Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Прима с использованием двух потоков

Размер массива	T_p	T_p^* (*calc*) (модель)	Модель 10.9 – оценка сверху		Модель 10.10 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*

100	0,0009	0,0003	0,0001	0,0004	0,0000	0,0003
200	0,0024	0,0011	0,0003	0,0013	0,0001	0,0012
300	0,0038	0,0023	0,0006	0,0029	0,0003	0,0026
400	0,0059	0,0040	0,0010	0,0050	0,0006	0,0046
500	0,0088	0,0062	0,0016	0,0078	0,0009	0,0071
600	0,0120	0,0089	0,0024	0,0112	0,0013	0,0102
700	0,0160	0,0120	0,0032	0,0152	0,0018	0,0138
800	0,0208	0,0156	0,0042	0,0198	0,0024	0,0180
900	0,0256	0,0197	0,0053	0,0250	0,0030	0,0227
1000	0,0328	0,0243	0,0066	0,0308	0,0037	0,0280

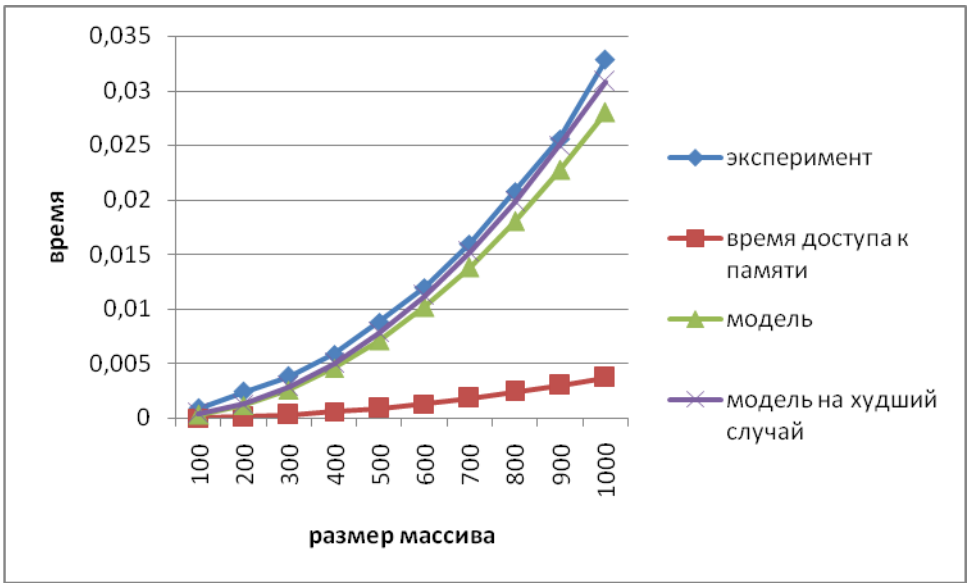

Рис. 10.9. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Прима от объема исходных данных при использовании двух потоков

Таблица 10.5.

Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Прима с использованием четырех потоков

Размер массива	T_p	T_p^* (*calc*) (модель)	Модель 10.9 – оценка сверху		Модель 10.10 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*

100	0,0009	0,0002	0,0001	0,0002	0,0000	0,0002
200	0,0020	0,0006	0,0003	0,0008	0,0002	0,0007
300	0,0037	0,0012	0,0006	0,0018	0,0004	0,0016
400	0,0048	0,0021	0,0010	0,0031	0,0007	0,0028
500	0,0070	0,0032	0,0016	0,0049	0,0011	0,0043
600	0,0089	0,0046	0,0024	0,0069	0,0016	0,0061
700	0,0114	0,0062	0,0032	0,0094	0,0021	0,0083
800	0,0144	0,0080	0,0042	0,0122	0,0028	0,0108
900	0,0177	0,0101	0,0053	0,0154	0,0035	0,0136
1000	0,0209	0,0124	0,0066	0,0189	0,0043	0,0167

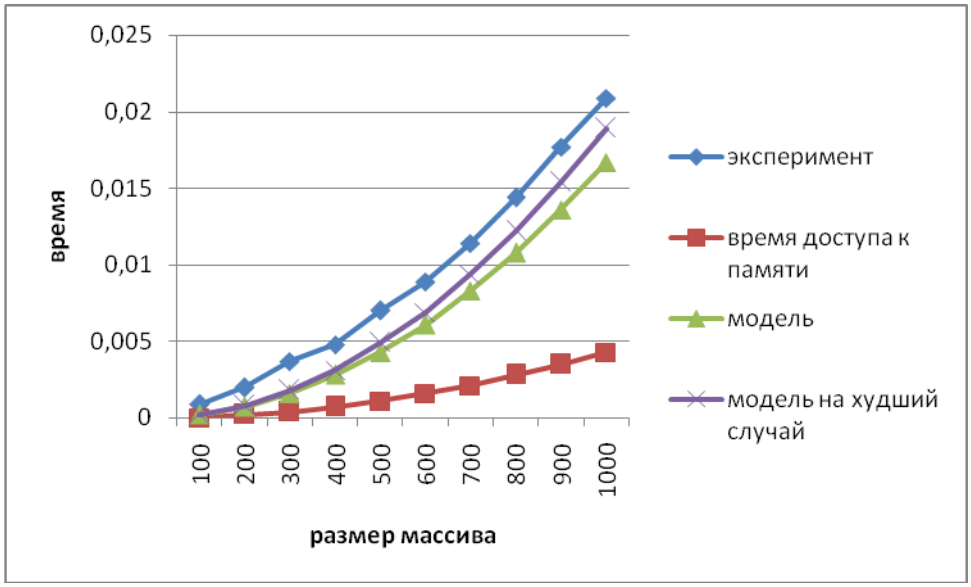

Рис. 10.10. График зависимости экспериментального и теоретического времен выполнения параллельного алгоритма Прима от объема исходных данных при использовании четырех потоков

10.3. 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 или вычислительных системах с общей памятью, в которых вычислительные элементы имеют раздельный кэш, предполагает разделение сети между вычислительными элементами таким образом, чтобы каждому из вычислительных элементов выделялось примерно равное число элементов сети, а межпроцессорные коммуникации, необходимые для выполнения информационного обмена между соседними элементами, были минимальными. На рис. 10.11. показан пример нерегулярной сети, разделенной на 4 части (различные части разбиения сети выделены темным цветом различной интенсивности).

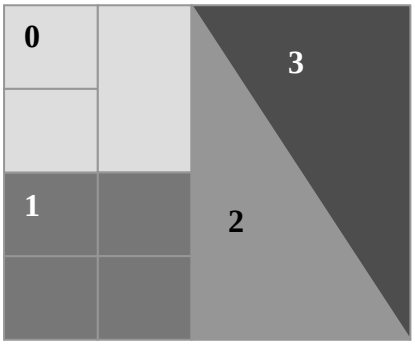

Рис. 10.11. Пример разделения нерегулярной сети

Очевидно, что такие задачи разделения сети между процессорами могут быть сведены к проблеме оптимального разделения графа. Данный подход целесообразен еще и 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. 10.11, будет сформирован граф, приведенный на рис. 10.12.

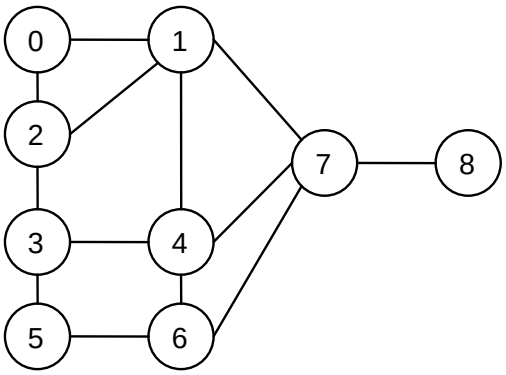

Рис. 10.12. Пример графа, моделирующего структуру сети на рис. 10.11

Дополнительная информация по проблеме разделения графов может быть получена, например, в [90].

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 – см., например, [41–42,59,69–70,87,98].

10.3.1. 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.

10.3.2. 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. 10.13 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).

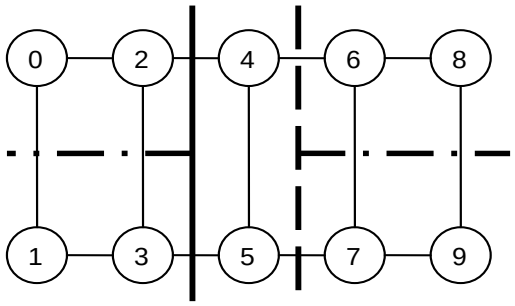

Рис. 10.13. Пример разбиения графа на 5 частей методом рекурсивного деления пополам

10.3.3. Геометрические методы

Геометрические методы (см., например, [43,58,65,73,75,77,81,86]) 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.

1. Покоординатное разбиение. *Покоординатное разбиение* (*coordinate nested dissection*)– это метод, основанный на рекурсивном делении пополам сети по наиболее длинной стороне. В качестве иллюстрации на рис. 10.13 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.

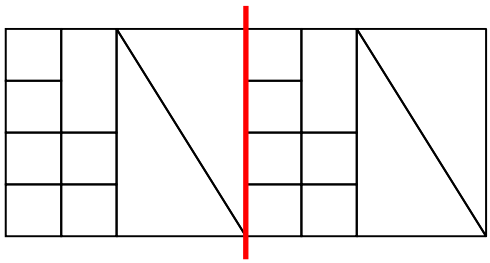

Рис. 10.14. Пример разделения сети графическим методом по наибольшей размерности (граница раздела показана жирной линией)

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.

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. 10.14 под острым углом к координатным осям (см. рис. 10.15), 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, [84]), считая элементы сети точечными массами. Линия бисекции, ортогональная полученной оси, как правило, дает границу наименьшей длины.

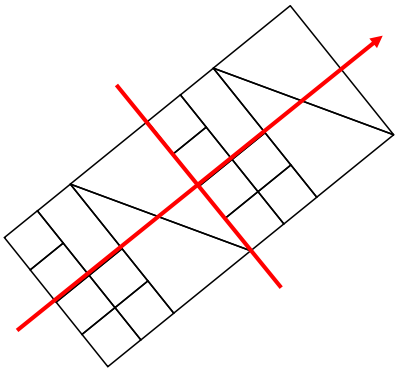

Рис. 10.15. Пример разделения сети методом рекурсивной инерционной бисекции (стрелкой показана главная инерционная ось)

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 [76–77,81].

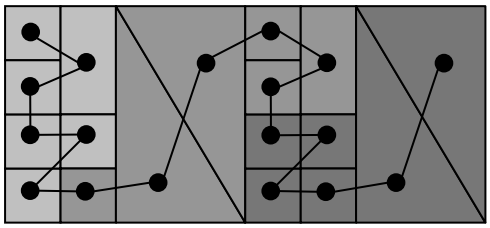

Рис. 10.16. Пример разделения сети на 3 части с использованием кривых Пеано

10.3.4. Комбинаторные методы

В отличие от геометрических методов, комбинаторные алгоритмы (см., например, [57,90]) 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.

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 две 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 10.1. 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 10.2 (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. 10.17. 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, т. к. в приведенном примере есть решение только с тремя пересеченными ребрами вместо пяти.

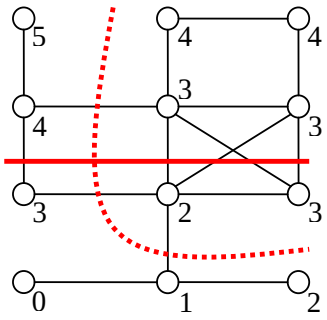

Рис. 10.17. Пример работы алгоритма деления графов с учетом связности

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 уже 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. 10.18). 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 10.2. Общая схема алгоритма Кернигана–Лина

Поясним дополнительно, что на шаге 2 итерации алгоритма перестановка вершин каждой очередной пары осуществляется для одного и того же 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).

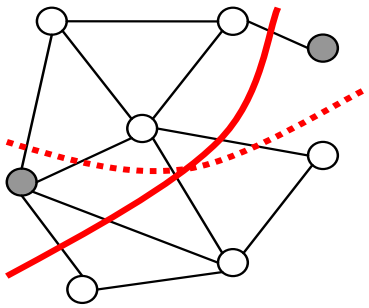

Рис. 10.18. Пример перестановки двух вершин (выделены серым) в методе Кернигана–Лина

10.3.5. 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. 10.7 (см. [90]) общая характеристика ряда алгоритмов разделения графов, рассмотренных в данном разделе. Дополнительная информация по проблеме оптимального разбиения графов может быть получена, например, в [90].

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).

Таблица 10.6.

Сравнительная таблица некоторых алгоритмов разделения графов

		Необходимость координатной информации	Точость	Время выполнения	Возможности для распараллеливания
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 «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.

10.4. Краткий обзор главы

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

В 10.1 представлен *алгоритм Флойда* (*Floyd*) для решения задачи поиска путей минимальной длины между каждой парой вершин графа. Для алгоритма 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.

В 10.2 рассматрен *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 параллельных вычислений, рассматриваются результаты проведенных вычислительных экспериментов.

Рассматриваемая в 10.3 задача оптимального разделения графов является важной для многих научных исследований, использующих параллельные вычисления. Для примера в разделе приведен общий способ перехода от двухмерной или трехмерной сети, моделирующей процесс вычислений, к соответствующему ей графу. Для решения задачи разбиения графов 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 комбинатор-

ных 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.

10.5. Обзор литературы

Дополнительная информация по алгоритмам Флойда и Прима может быть получена, например, в [17].

Подробное рассмотрение вопросов, связанных с проблемой разделения графов, содержится в работах [43,57–58,65,73,75,77,81,86,90].

Параллельные алгоритмы разделения графов рассматриваются в [41,58,65,69–70,87,98].

10.6. 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?

10.7. Задачи и упражнения

1. Используя приведенный программный код, выполните реализацию параллельного алгоритма Флойда. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом пара-

метров 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.