ALMATY TECHNOLOGICAL UNIVERSITY

Высокопроизводительные параллельные вычисления

Лекция №9
Тема: Типовые задачи обработки графов. Параллельный алгоритм Флойда, параллельный алгоритм Прима

Козбакова Айнур Холдасовна– PhD, профессор, кафедра «Информационные системы»
Факультет «Инжиниринг и информационных технологии» 01.09.2025 г.

В лекции рассматриваются различные типовые задачи, возникающие при обработке графов. Приводятся алгоритмы, применяемые для решения этих задач, и обсуждаются пути их распараллеливания. Дается теоретическая оценка эффективности рассматриваемых алгоритмов. Анализируются результаты вычислительных экспериментов

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 [[26]].

Пусть G есть *граф*

G=(V,R),

для которого набор вершин V_i, 0<=i<=n, задается множеством V, а *список* дуг *графа*

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

определяется множеством R. В общем случае дугам *графа* могут приписываться некоторые числовые характеристики (*веса*) w_j, 0<=j<=m (взвешенный *граф*). Пример взвешенного *графа* приведен на рис. 1.

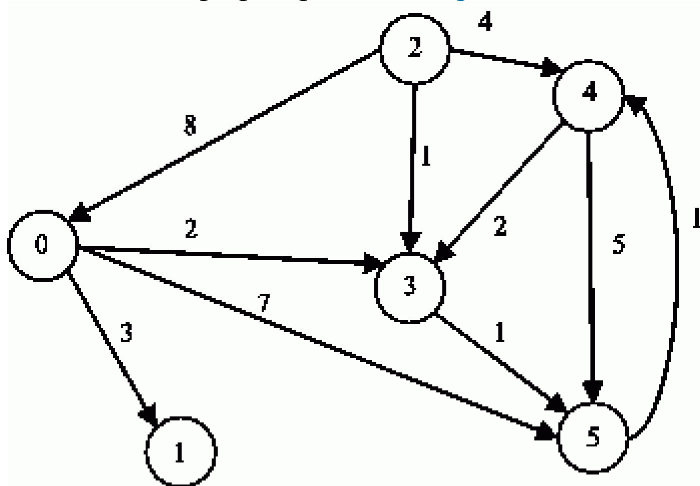

Рис. 1. Пример взвешенного ориентированного графа

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<=i, j<=n,

ненулевые значения элементов которой соответствуют дугам *графа*:

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

(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. 1, приведена на рис. 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. 2. Матрица смежности для графа с рис. 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 применяться матрицы смежности.

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, 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* (*the Floyd algorithm*) (см, например, [[26]]).

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 1. Общая схема *алгоритма Флойда*

```
// Алгоритм 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 [[26]].

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 значений (см. Алгоритм 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 применять индексы обновляемых в подзадачах элементов).

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. 3.

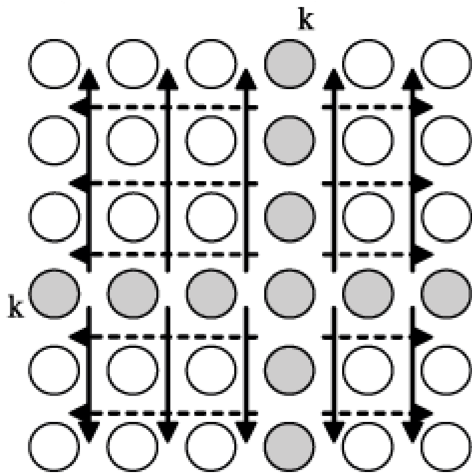

Рис. 3. Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

1.4. Масштабирование и распределение подзадач по процессорам

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач n^2 ($p \ll n^2$). 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.

Следует отметить, что при таком способе разбиения данных на каждой итерации *алгоритма Флойда* потребуется передавать между подзадачами только элементы одной из строк матрицы A. Для оптимального выполнения подобной коммуникационной операции топология сети должна обеспечивать эффективное представление структуры сети передачи данных в виде гиперкуба или полного *графа*.

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 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, имеет порядок сложности 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:

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

Следовательно, 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, такая операция может быть выполнена 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

$$T_p(comm) = n\lceil \log_2 p \rceil(\alpha + wn/\beta), \quad (2)$$

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:

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

где τ есть время выполнения операции выбора минимального значения.

Программная реализация

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

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

```
// Программа 1. — Алгоритм Флойда
int ProcRank;    // Ранг текущего процесса
int ProcNum;     // Количество процессов

// Функция вычисления минимума
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;
}

// Главная функция программы
int main(int argc, char* argv[]) {
  int *pMatrix;       // Матрица смежности
  int  Size;          // Размер матрицы смежности
  int *pProcRows;               // Строки матрицы смежности текущего процесса
  int  RowNum;                  // Число строк для текущего процесса

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

  // Инициализация данных
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);
```

```
  // Распределение данных между процессами
  DataDistribution(pMatrix, pProcRows, Size, RowNum);

  // Параллельный алгоритм Флойда
  ParallelFloyd(pProcRows, Size, RowNum);

  // Сбор результатов работы алгоритма
  ResultCollection(pMatrix, pProcRows, Size, RowNum);

  // Завершение вычислений процесса
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
  return 0;
}
```

1.

Функция Min вычисляет меньшее из двух целых чисел, учитывая применяемый способ задания несуществующих дуг в матрице смежности (в рассматриваемой реализации для этого используется значение -1).

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

Функция DataDistribution распределяет исходные данные между процессами. Каждый процесс получает горизонтальную полосу матрицы смежности.

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*.

```
// Параллельный алгоритм Флойда
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;

  for(int k = 0; k < Size; k++) {
    // Распределение k-й строки среди процессов
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Обновление элементов матрицы смежности
    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. Функция RowDistribution. Данная функция рассылает k -ю строку матрицы смежности всем процессам программы.

```
// Функция для рассылки строки всем процессам
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank;          // Ранг процесса, которому принадлежит k-я строка
  int ProcRowNum;           // Номер k-й строки в полосе матрицы

  // Нахождение ранга процесса – владельца k-й строки
  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)
    // Копирование строки в массив pRow
    copy(&pProcRows[ProcRowNum * Size], &pProcRows[(ProcRowNum + 1) *
      Size], pRow);

  // Распределение k-й строки между процессами
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}
```

2.

1.7. Результаты вычислительных экспериментов

Эксперименты проводились на вычислительном кластере Нижегородского университета на базе процессоров Intel Xeon 4 *EM64T*, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft *Compute Cluster* Server.

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 для величины τ было получено значение 7,14 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной

способности β соответственно 130 мкс и 53,29 Мбайт/с. Все вычисления производились над числовыми значениями типа int, размер которого на данной платформе равен 4 байта (следовательно, $w=4$).

Результаты вычислительных экспериментов приведены в таблице 1. Эксперименты выполнялись с использованием двух, четырех и восьми процессоров. Время указано в секундах.

Таблица 1. Результаты вычислительных экспериментов для параллельного *алгоритма Флойда*

Кол-во вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	8,037	4,152	1,936	2,067	3,888	0,941	8,544
2000	59,812	30,323	1,972	15,375	3,890	8,058	7,423
3000	197,111	99,264	1,986	50,232	3,924	25,643	7,687
4000	461,785	232,507	1,986	117,220	3,939	69,946	6,602
5000	884,622	443,747	1,994	224,441	3,941	128,078	6,907

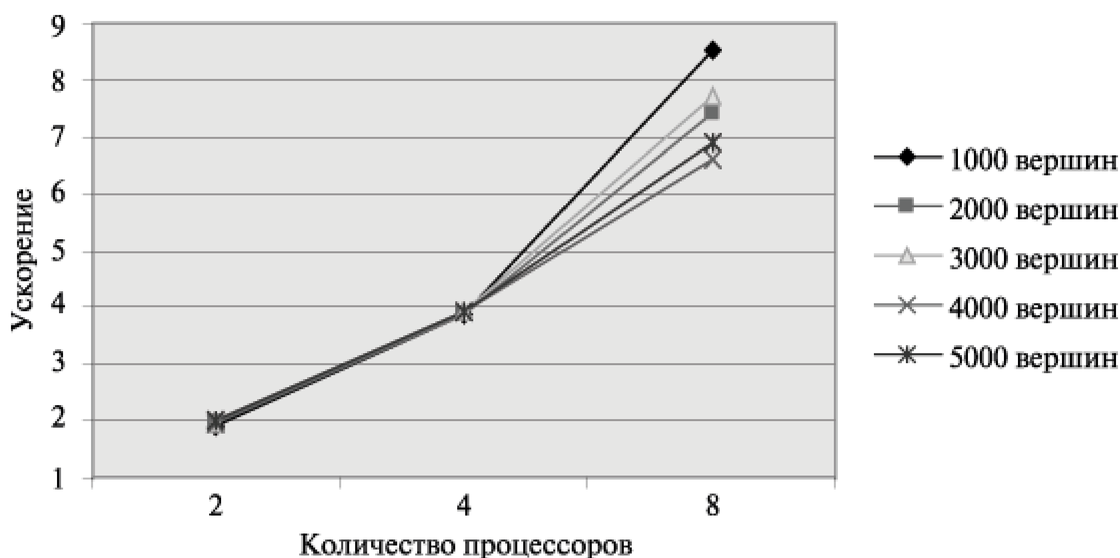

Рис. 4. Графики зависимости ускорения параллельного алгоритма Флойда от числа используемых процессоров при различном количестве вершин графа

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (3) приведено в таблице 2 и на рис. 5.

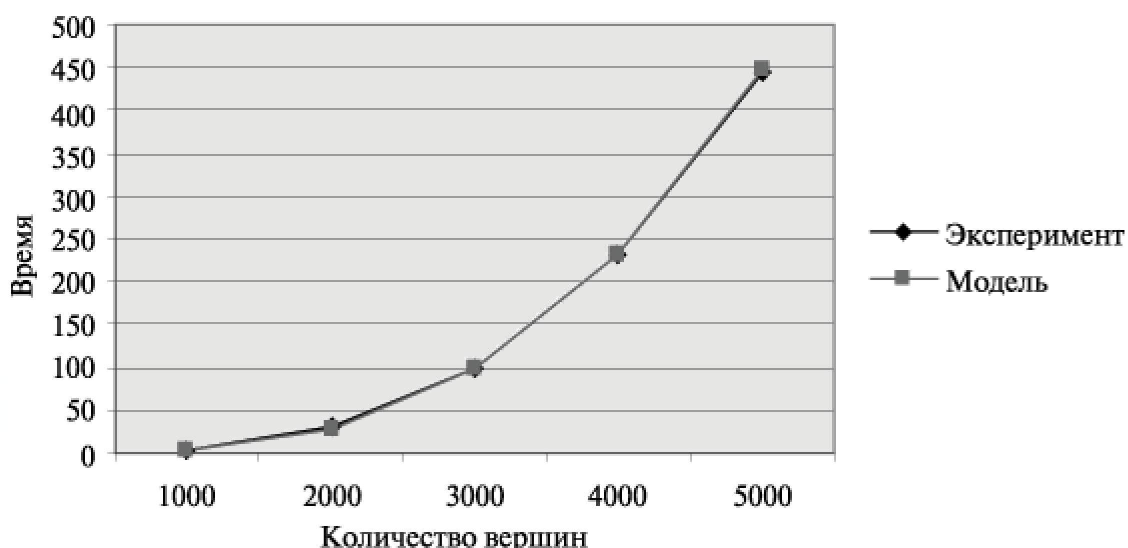

Рис. 5. Графики экспериментально установленного времени работы параллельного алгоритма Флойда и теоретической оценки в зависимости от количества вершин графа при использовании двух процессоров

Таблица 2. Сравнение экспериментального и теоретического времени работы *алгоритма Флойда*

Количество вершин	**Последовательный алгоритм**	**Параллельный алгоритм**					
		2 процессора		**4 процессора**		**8 процессоров**	
	T_1^*	T_2	T_2^*	T_4	T_4^*	T_8	T_8^*
1000	8,038	3,776	4,152	2,196	2,067	1,509	0,941
2000	59,812	29,123	30,323	15,405	15,375	8,826	8,058
3000	197,111	97,465	99,264	50,336	50,232	27,307	25,643
4000	461,785	230,220	232,507	117,701	117,220	62,306	69,946
5000	884,622	448,811	443,747	228,211	224,441	119,179	128,078

адача 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 минимально охватывающего дерева приведен на рис. 6.

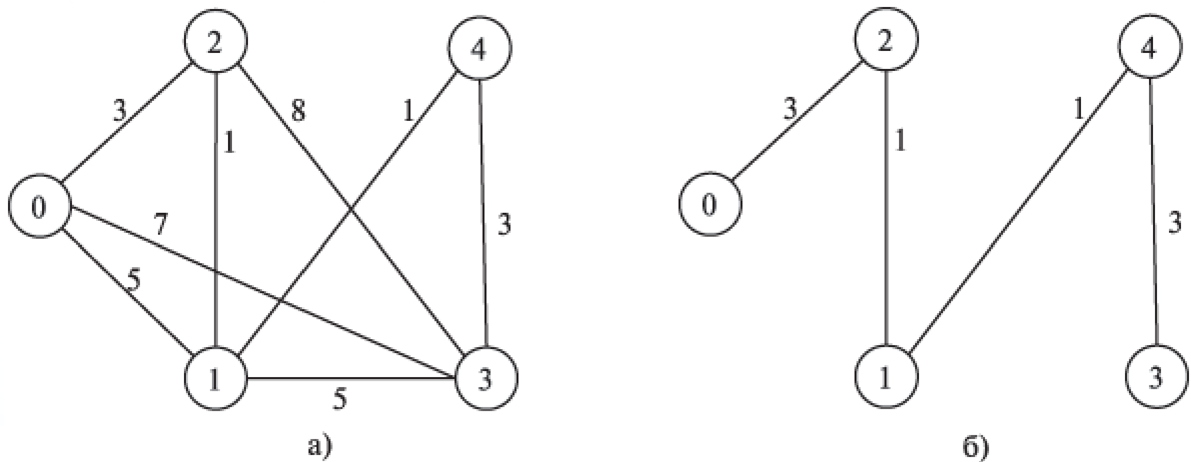

Рис. 6. Пример взвешенного неориентированного графа (а) и соответствующему ему минимально охватывающего дерева (б)

Дадим общее описание алгоритма решения поставленной задачи, известного под названием *метода Прима* (*the Prim method*); более полная информация может быть получена, например, в [[26]].

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, $1<=i<=n$, 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 равным ∞). 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 $V_T t : d_t, \ i \notin V_T$;
- 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* $T_1 \sim n^2$.

2.2. Разделение вычислений на независимые части

Оценим возможности параллельного выполнения рассмотренного алгоритма нахождения минимально 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, $1<=j<=p$, должен содержать:

- набор вершин
  $$V_j = \{\nu_{i_j+1}, \nu_{i_j+2}, \dots, \nu_{i_j+k}\},\ i_j = k\cdot(j-1), k = \lceil n/p \rceil;$$
- соответствующий этому набору блок из k величин
  $$\Delta_j = \{d_{i_j+1}, d_{i_j+2}, \dots, d_{i_j+k}\};$$
- вертикальную полосу матрицы смежности *графа* G из k соседних столбцов
  $$A_j = \{\alpha_{i_j+1}, \alpha_{i_j+2}, \dots, \alpha_{i_j+k}\}(\alpha_s \text{ есть } s\text{-й столбец матрицы } A);$$
- 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.

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.

2.4. Масштабирование и распределение подзадач по процессорам

По определению 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 оптимальной реализации

требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должна обеспечивать эффективное представление в виде гиперкуба или полного *графа*.

Анализ эффективности параллельных вычислений

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

$$S_p = \frac{n^2}{(n^2/p)} = p \quad \text{и} \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \quad (4)$$

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 возникают 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:

$$T_p(calc) = 2n\lceil n/p \rceil \cdot \tau \quad (5)$$

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

Операция сбора данных от всех процессоров на одном из процессоров может быть произведена 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 "Оценка коммуникационной трудоемкости параллельных алгоритмов"):

$$T_p^1(comm) = n(\alpha \log_2 p + 3w(p-1)/\beta), \quad (6)$$

где α – латентность сети передачи данных, β – пропускная способность сети, а w есть размер одного пересылаемого элемента данных в байтах (коэффициент 3 в выражении соответствует числу передаваемых значений между процессорами – длина минимальной дуги и номера двух 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:

$$T_p^2(comm) = n \log_2 p(\alpha + w/\beta) \quad (7)$$

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

$$T_p = 2n\lceil n/p \rceil \cdot \tau + n(\alpha \cdot \log_2 p + 3w(p-1)/\beta + log_2 p(\alpha + w/\beta)). \quad (8)$$

2.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного *алгоритма Прима* осуществлялись при тех же условиях, что и ранее выполненные (см. п. 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 4,76 нсек. Все вычисления производились над числовыми значениями типа int, размер которого на данной платформе равен 4 байта (следовательно, w=4).

Результаты вычислительных экспериментов даны в таблице 3. Эксперименты проводились с использованием двух, четырех и восьми процессоров. Время указано в секундах.

Таблица 3. Результаты вычислительных экспериментов для параллельного *алгоритма Прима*

Кол-во вершин	**Последовательный алгоритм**	**Параллельный алгоритм**					
		2 процессора		**4 процессора**		**8 процессоров**	
		Время	**Ускорение**	**Время**	**Ускорение**	**Время**	**Ускорение**
1000	0,044	0,248	0,176	0,932	0,047	1,574	0,028
2000	0,208	0,684	0,304	1,800	0,115	2,159	0,096
3000	0,485	1,403	0,346	2,214	0,219	3,195	0,152
4000	0,873	1,946	0,622	3,324	0,263	5,431	0,161
5000	1,432	2,665	0,736	2,933	0,488	4,119	0,348
6000	2,189	2,900	0,821	4,291	0,510	7,737	0,283
7000	3,042	3,236	0,940	6,327	0,481	8,825	0,345
8000	4,150	4,462	0,930	6,993	0,593	10,390	0,399
9000	5,622	5,834	0,964	7,475	0,752	10,764	0,522

10000	7,512	6,990	1,075	8,597	0,874	14,095	0,533

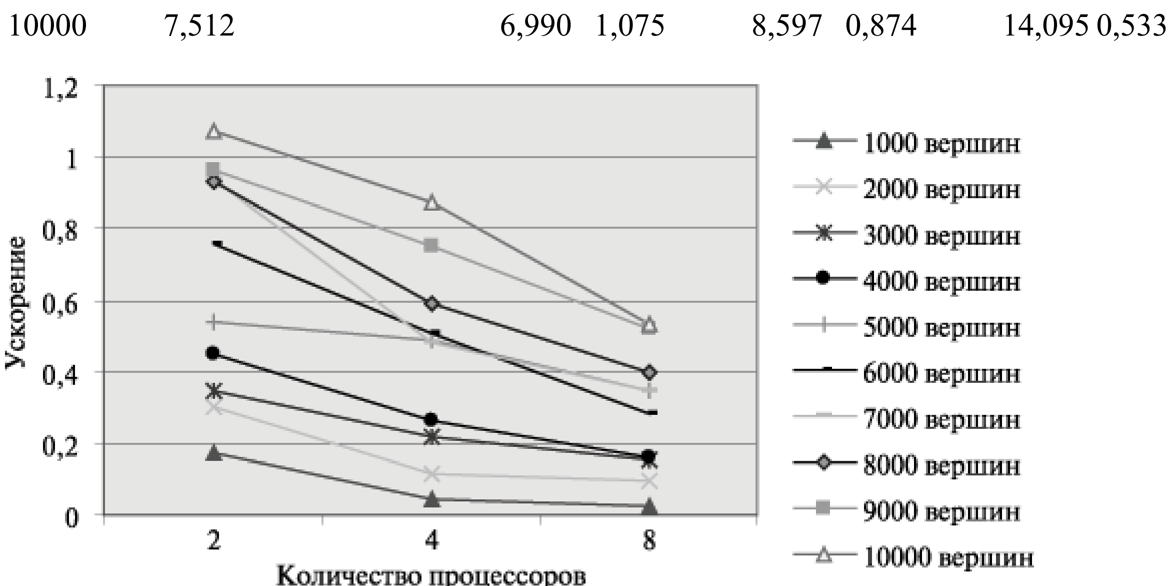

Рис. 7. Графики зависимости ускорения параллельного алгоритма Прима от числа используемых процессоров при различном количестве вершин в модели

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (8) приведено в таблице 4 и на рис. 8.

Таблица 4. Сравнение экспериментального и теоретического времени работы *алгоритма Прима*

Количество вершин	**Последовательный алгоритм**	**Параллельный алгоритм**					
		2 процессора		**4 процессора**		**8 процессоров**	
	T_1^*	T_2	T_2^*	T_4	T_4^*	T_8	T_8^*
1000	0,044	0,405	0,248	0,804	0,932	1,205	1,574
2000	0,208	0,820	0,684	1,613	1,800	2,412	2,159
3000	0,485	1,245	1,403	2,426	2,214	3,622	3,195
4000	0,873	1,679	1,946	3,245	3,324	4,834	5,431
5000	1,432	2,122	2,665	4,068	2,933	6,048	4,119
6000	2,189	2,575	2,900	4,896	4,291	7,265	7,737
7000	3,042	3,038	3,236	5,728	6,327	8,484	8,825
8000	4,150	3,510	4,462	6,566	6,993	9,705	10,390
9000	5,622	3,991	5,834	7,408	7,475	10,929	10,764

10000	7,512	4,482	6,990	8,255	8,597	12,155	14,095

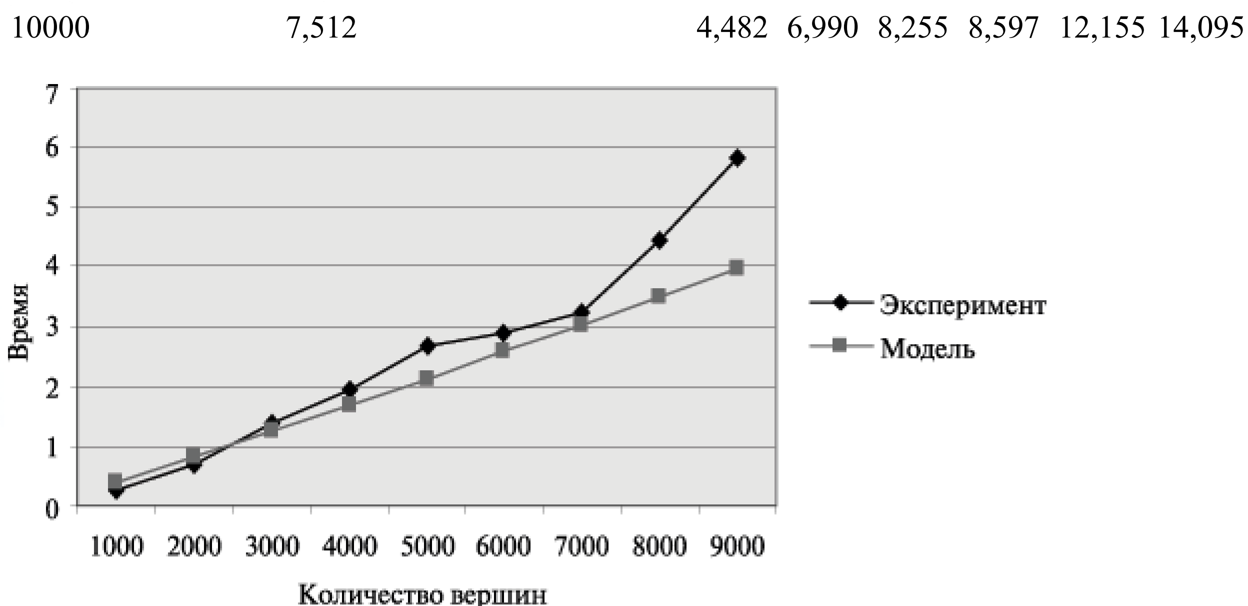

Рис. 8. Графики экспериментально установленного времени работы параллельного алгоритма Прима и теоретической оценки в зависимости от количества вершин в модели при использовании двух процессоров

Как можно заметить из табл. 4 и рис. 8, теоретические оценки определяют время выполнения *алгоритма Прима* с достаточно высокой погрешностью. Причина такого расхождения может состоять в том, что модель Хокни менее точна при оценке времени передачи сообщений с небольшим объемом передаваемых данных. Для уточнения получаемых оценок необходимым является использование других более точных моделей расчета трудоемкости коммуникационных операций – обсуждение этого вопроса проведено в "Оценка коммуникационной трудоемкости параллельных алгоритмов" .