

СУПЕРКОМПЬЮТЕРНЫЙ КОНСОРЦИУМ УНИВЕРСИТЕТОВ РОССИИ

Проект

Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения



Московский государственный университет им. М.В. Ломоносова



Нижегородский государственный университет им. Н.И. Лобачевского

- Национальный исследовательский университет -

Учебный курс

ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ МНОГОПРОЦЕССОРНЫХ МНОГОЯДЕРНЫХ СИСТЕМ

Лекция 10.

Параллельные алгоритмы на графах

Гергель В.П., профессор, д.т.н.

Содержание



- Обработка графов
- Задача поиска всех кратчайших путей
- Задача нахождения минимального охватывающего дерева
- Задача оптимального разделения графов



- Математические модели в виде *графов* широко используются при моделировании разнообразных явлений, процессов и систем.
- Многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей, среди которых может быть выделен определенный набор *типовых алгоритмов обработки графов*.

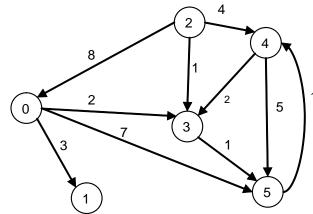


• Пусть G есть граф G = (V, R), для которого набор вершин v_i , $1 \le i \le n$, задается множеством V, а список дуг графа

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

определяется множеством R.

- В общем случае дугам графа могут приписываться некоторые числовые характеристики (seca) w_j , $1 \le j \le m$ (sseemehhb"i $cpa\phi$).
- Пример взвешенного графа представлен на рисунке

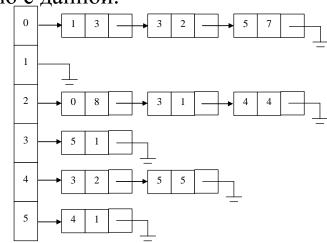






• Способы задания графов

- С помощью списка, перечисляющего имеющиеся в графах дуги *списка примыканий*:
 - Целесообразен при малом количестве дуг в графе (т.е. $m << n^2$).
 - Для графа G=(V, R) с числом вершин n записывается в виде одномерного массива длины n, каждый элемент которого представляет собой ссылку на список.
 - Приписан каждой вершине графа и содержит по одному элементу на каждую вершину графа, соседнюю с данной.
 - Каждое звено хранит номер соседней вершины и вес ребра.
 - Для приведенного примера список примыкания выглядит следующим образом.







- Задание с помощью матрицы смежности
 - Эффективно для достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т.е. $m \sim n^2$).
 - Позволяет применять матричные алгоритмы обработки данных.
 - Представляет собой $A = (a_{ij}), 1 \le i, j \le n,$ где ненулевые значения элементов соответствуют дугам графа

$$a_{ij} = egin{cases} w(v_i, v_j), \ ec \pi u \ (v_i, v_j) \in R, \ 0, \ ec \pi u \ i = j, \ \infty, \ u$$
наче.

- При вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число. $\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \end{pmatrix}$
- Для приведенного примера матрица смежности выглядит следующим образом.

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





- Далее рассмотрим способы параллельной реализации алгоритмов на графах на примере задачи поиска кратчайших путей между всеми парами пунктов назначения, задачи выделения минимального охватывающего дерева (остова) графа и задачи оптимального разделения графов, широко используемой 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.



- Исходная информация взвешенный граф G = (V, R) , содержащий n вершин (|V| = n), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать ориентированным.
- Рассмотрим задачу, в которой для имеющегося графа *G* требуется найти минимальные длины путей между каждой парой вершин графа.
- В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.
- В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется алгоритм Флойда (Floyd)ю





• Последовательный алгоритм Флойда

- Сложность алгоритма имеет порядок n^3 .
- В общем виде данный алгоритм может быть представлен следующим образом:

```
// Алгоритм 10.1

// Serial Floyd algorithm

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]);
```

- Заметим, что:

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





• 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.
 - 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.
 - 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_{ki} 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_{ki} 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 $6a3080\tilde{u}$ nod3adauu может быть использована операция обновления элементов матрицы A.



- 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.



• Анализ эффективности параллельных вычислений

- Будем предполагать, что время выполнения складывается из времени вычислений 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 *п* вершин, требуется выполнение *п* итераций алгоритма, на каждой из которых параллельно выполняется обновление всех элементов матрицы смежности.
- Значит, время выполнения вычислений составляет:

$$T_p \blacktriangleleft alc = n \cdot \frac{n^2}{p} \cdot \tau$$

где т есть время выполнения операции выбора минимального значения.





- Затраты на доступ к памяти составляют:

$$T_p$$
 (nem) $= n \cdot \frac{64n^2}{\beta}$

где β есть пропускная способность канала доступа к оперативной памяти

 Если учесть латентность памяти, то время доступа к памяти можно получить по следующей формуле:

$$T_p$$
 (nem) $= n^3 \cdot \left(\alpha + \frac{64}{\beta}\right)$



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

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

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:

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



• Программная реализация

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

1. Главная функция программы.





2. Функция ProcessInitialization.

```
// Function for memory allocation and data initialization
void ProcessInitialization(double *&pMatrix, int& Size) {
  do {
    printf("Enter the number of vertices: ");
    scanf("%d", &Size);
    if(Size \le 2)
      printf("The number of vertices should be greater than
two\n");
  } while(Size <= 2);</pre>
  printf("Using graph with %d vertices\n", Size);
  // Allocate memory for the adjacency matrix
  pMatrix = new double[Size * Size];
  // Data initalization
  RandomDataInitialization(pMatrix, Size);
```





3. Функция ParallelFloyd.

```
// Function for parallel Floyd algorithm's execution
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.

```
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;
}</pre>
```

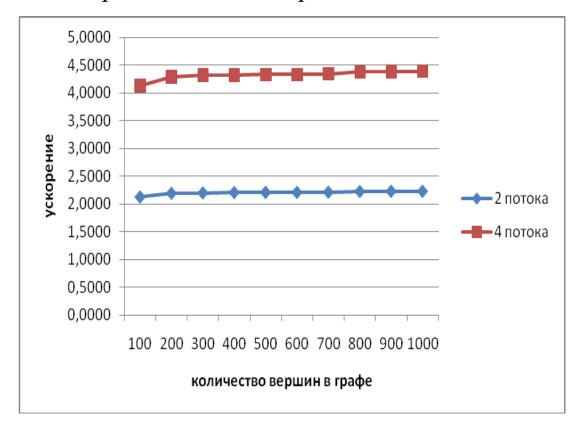


• Результаты вычислительных экспериментов

- Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008.
- Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.
- Для снижения сложности построения теоретических оценок времени выполнения алгоритмов при компиляции и построении программ для проведения вычислительных экспериментов функция оптимизации кода компилятором была отключенаю



 Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма Флойда





- 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.
- Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение τ , равное 55,178 нс.
- Оценки времени латентности α и величины пропускной способности канала доступа к оперативной памяти β проводилась в лекции 6 и определены для используемого вычислительного узла как $\alpha = 8,31$ нс. и $\beta = 12,44$ Гб/с.
- Величина накладных расходов δ на параллельность была оценена в лекции 6 и составляет 0,25 мкс.
- Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0083, а для четырех потоков значение этой величины была оценена как 0,0090.





 График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Флойда от объема исходных данных при использовании двух потоков

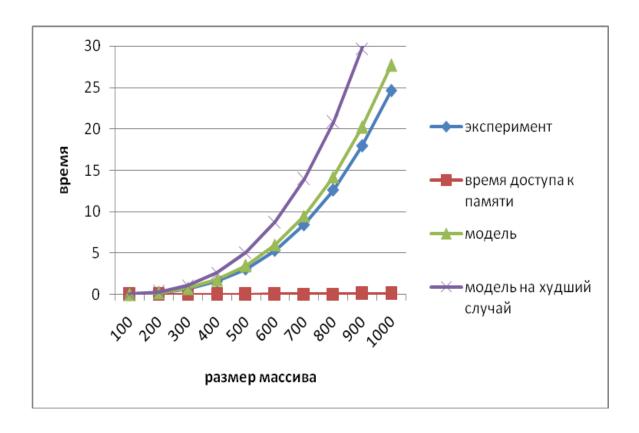
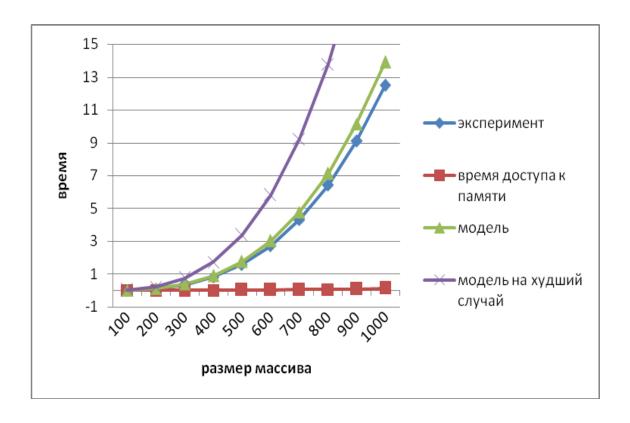






 График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Флойда от объема исходных данных при использовании четырех потоков



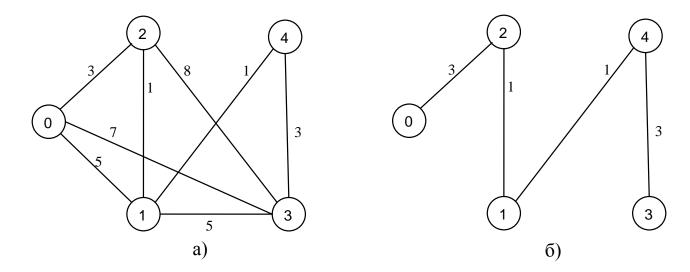




- Охватывающим деревом (остовом) неориентированного графа G называется подграф T графа G, который является деревом и содержит все вершины из G.
- Определив вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, под *минимально* охватывающим деревом (МОД) Т будем понимать охватывающее 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:



• Рассмотрим общее описание алгоритма решения поставленной задачи, известного под названием *метода* Прима (Prim)ю





• 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.

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

- 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 в и полагается $V_T = \{s\}$, $d_s = 0$.





- Действия, выполняемые на каждой итерации алгоритма Прима, состоят в следующем:
 - определяются значения величин 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 n

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

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





- Программная реализация последовательного алгоритма Прима
 - 1. Главная функция программы.

```
// Программа 10.2
// Serial Prim algorithm
void main(int argc, char* argv[]) {
  double *pMatrix;
                              // Adjacency matrix
 TTreeNode** pMinSpanningTree; // Minimum spanning tree
  int Size:
                                 // Size of adjacency matrix
  // Process initialization
  ProcessInitialization(pMatrix, pMinSpanningTree, Size);
 // Serial Prim algorithm
  SerialPrim(pMatrix, pMinSpanningTree, Size);
  // Process termination
  ProcessTermination(pMatrix, pMinSpanningTree);
```



- Для хранения минимального охватывающего дерева используется список примыканий, который представляет из себя массив, число элементов в котором совпадает с числом вершин в графе;
- каждый i-ый элемент массива есть указатель на звено списка, описывающее вершину графа, соседнюю с вершиной i.
- Для описания вершины графа предназначена структура *TTreeNode*:



2. Функция ProcessInitialization.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double *&pMatrix, TTreeNode** &pMinSpanningTree,
  int& Size) {
  do {
    printf("Enter the number of vertices: ");
    scanf("%d", &Size);
    if(Size \le 2)
      printf("The number of vertices should be greater than two\n");
  } while(Size <= 2);</pre>
  printf("Using graph with %d vertices\n", Size);
  // Allocate memory for the adjacency matrix
  pMatrix = new double[Size * Size];
  // Allocate memory for the spanning tree
  pMinSpanningTree = new TTreeNode* [Size];
  // Data initalization
  RandomDataInitialization(pMatrix, pMinSpanningTree, Size);
```



3. Функция SerialPrim.

```
// Function for serial Prim algorithm's execution
void SerialPrim(double *pMatrix, TTreeNode** pMinSpanningTree, int Size) {
  // Number of vertex that was added to the spanning tree on previous step
  int LastAdded;
  // Number of vertex which is nearest to the spanning tree
  TGraphNode NearestNode;
  // List of vertices, that haven't been added to minimum spanning tree
  TGraphNode **NotInMinSpanningTree = new TGraphNode* [Size-1];
  // Vertex with 0 number is the root of minimum spanning tree
  LastAdded = 0:
  // The other vertexes are not added to spanning tree
  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;
// Prim's algorithm iterations
```



```
for (int Iter=1; Iter<Size; Iter++) {</pre>
    // Recalculation of the distances
    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;
    // Choose the nearest vertex
    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;
```





```
// Add the nearest vertex to adjacency list,
   // which describes minimum spanning tree
  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:



• Разделение вычислений на независимые части

- 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.
 - 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 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.



• Анализ эффективности параллельных вычислений

- Будем предполагать, что время выполнения складывается из времени вычислений 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 *п* вершин, требуется выполнение *п* 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.
- На каждой i-ой итерации алгоритма выполняется (n-i) операций выбора минимума для пересчета расстояний и (n-i) операций сравнения для выбора ближайшей вершины.
- Таким образом, для оценки времени выполнения вычислений может быть использовано соотношение:

$$T_{p} \operatorname{Calc} = \left(2 \cdot \frac{\sum_{i=0}^{n-1} \operatorname{C} - i}{p} + \log_{2} p\right) \cdot \tau = \left(2 \cdot \frac{n^{2}/2}{p} + \log_{2} p\right) \cdot \tau = \left(\frac{n^{2}}{p} + \log_{2} p\right) \cdot \tau$$

где τ есть время выполнения операции выбора минимального значения.





- На каждой итерации внешнего цикла происходит считывание из оперативной памяти в кэш вычислительных элементов необходимых значений.
- 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*-ой итерации алгоритма Прима необходимо загрузить из оперативной памяти (*n-i*) значений. Каждое значение занимает 8 байт. Данные из оперативной памяти считываются строками по 64 байта.
- Таким образом, затраты на доступ к памяти составляют:

$$T_{p}$$
 (nem) = $\sum_{i=0}^{n-1} \frac{64 \cdot (1-i)}{\beta} = 64 \cdot \frac{n^{2}/2}{\beta}$

- Учитывая латентность оперативной памяти α :

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





- При выборе вершины, находящейся на наименьшем расстоянии от МОД, и при добавлении новой вершины в состав МОД, используются структуры данных сравнительно небольшого размера, которые сохраняются в кэш при переходе от одной итерации алгоритма Прима к другой.
- На каждой итерации алгоритма создается две параллельные секции:
 - первая отвечает за параллельное выполнение пересчета расстояний, а вторая за выбор ближайшей вершины:

$$T_{p} = \left(\frac{n^{2}}{p} + \log_{2} p\right) \cdot \tau + (n^{2}/2) \cdot \left(\alpha + \frac{64}{\beta}\right) + 2n\delta$$

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

— Для улучшения точности модели необходимо учесть частоту кэш промахов γ $T_p = \left(\frac{n^2}{n} + \log_2 p\right) \cdot \tau + \gamma (n^2/2) \cdot \left(\alpha + \frac{64}{8}\right) + 2n\delta$





• Программная реализация параллельного алгоритма Прима

- Для распараллеливания этапа пересчета расстояний от вершин, не включенных в состав МОД, до МОД, достаточно распределить итерации цикла, просматривающего все элементы массива *NotInSpanningTree*, между потоками параллельной программы при помощи директивы *parallel for*.
- 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.





```
// Function for parallel Prim algorithm's execution
void ParallelPrim(double *pMatrix, TTreeNode** pMinSpanningTree, int Size)
  // Number of vertex that was added to the spanning tree on previous step
  int LastAdded;
  // Number of vertex which is nearest to the spanning tree
  TGraphNode NearestNode;
  // List of vertices, that haven't been added to minimum spanning tree
  TGraphNode **NotInMinSpanningTree = new TGraphNode* [Size-1];
  // Vertex with 0 number is the root of minimum spanning tree
  LastAdded = 0:
  // The other vertexes are not added to spanning tree
  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;
```



```
// Prim's algorithm iterations
  for (int Iter=1; Iter<Size; Iter++)</pre>
    // Recalculation of the distances
#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;
    // Choose the nearest vertex
    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) {</pre>
          NearestNode.Distance = ThreadNearestNode.Distance;
          NearestNode.NodeNum = ThreadNearestNode.NodeNum;
      } // pragma omp critical
 // pragma omp parallel
```



```
// Add the nearest vertex to adjacency list, which describes minimum spanning tree
  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;
```



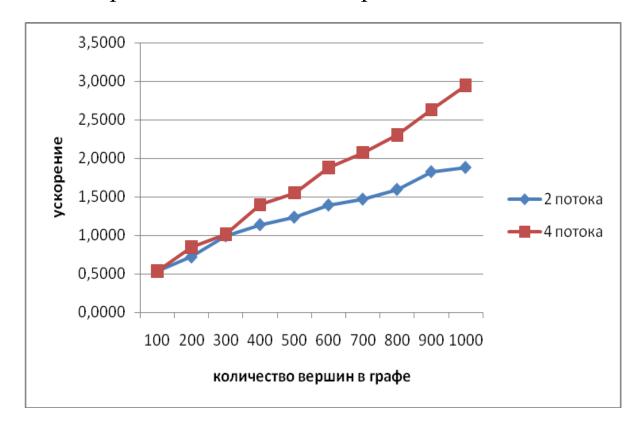


• Результаты вычислительных экспериментов

- Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008.
- Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.



 Зависимость ускорения от количества исходных данных при выполнении параллельного метода Прима





- Измерим время выполнения последовательного алгоритма Прима при малых объемах данных, таких, чтобы матрица смежности, описывающая граф, могла быть полностью помещена в кэш вычислительного элемента.
- 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.
- Латентность α и пропускная способность канала доступа к оперативной памяти β являются равными $\alpha = 8,31$ нс и $\beta = 12,44$ Гб/с.
- Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,5656, а для четырех потоков значение этой величины была оценена как 0,6614.
- Как отмечалось в лекции 6, время δ, необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

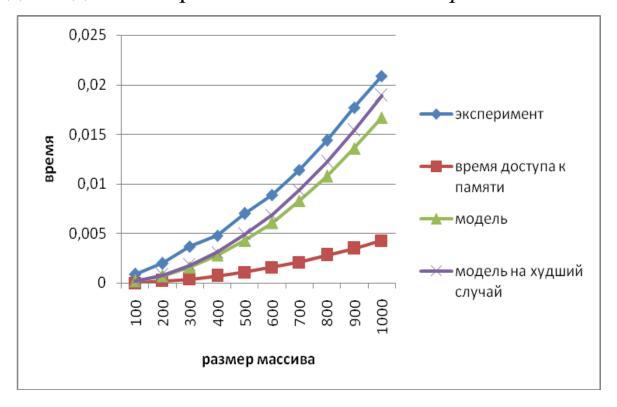


 График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Прима от объема исходных данных при использовании двух потоков





 График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Прима от объема исходных данных при использовании четырех потоков

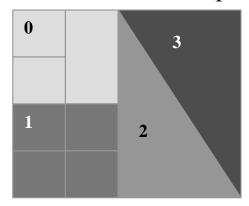




- Проблема оптимального разделения графов относится к числу часто возникающих задач при проведении различных научных исследований, использующих параллельные вычисления.
- В качестве примера можно привести задачи обработки данных, в которых области расчетов представляются в виде двухмерной или трехмерной сети.
 - Вычисления в таких задачах сводятся, как правило, к выполнению тех или иных процедур обработки для каждого элемента (узла) сети.
 - 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 были минимальными.



• Пример нерегулярной сети, разделенной на 4 части (различные части разбиения сети выделены темным цветом различной интенсивности)

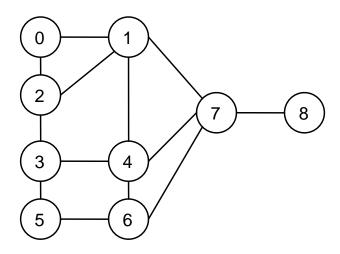


- Такие задачи разделения сети между процессорами могут быть сведены к проблеме оптимального разделения графа.
- Представление модели вычислений в виде графа позволяет легче решить вопросы хранения обрабатываемых данных и предоставляет возможность применения типовых алгоритмов обработки графов.
- Для представления сети в виде графа каждому элементу сети можно поставить в соответствие вершину графа, а дуги графа использовать для отражения свойства близости элементов сети.





• Для приведенного примера сети будет сформирован граф:



• Задача оптимального разделения графов может являться предметом распараллеливания.



• 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.
- Отметим 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.
- Для простоты будем полагать веса вершин и ребер графа равными единице.



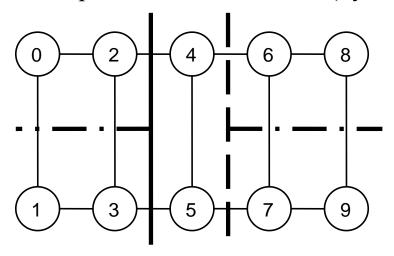


• Метод рекурсивного деления пополам

- 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.
- 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 приведенном примере разделения графа 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).





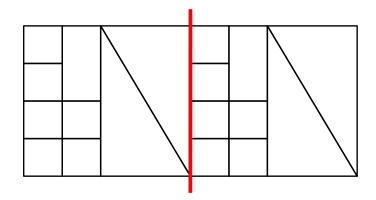
• Геометрические методы

- Геометрические методы 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.
- 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 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.
 - Пример разделения сети графическим методом по наибольшей размерности (граница раздела показана жирной линией)





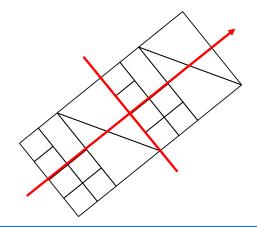
- Общая схема выполнения метода состоит в следующем:
 - Вычисляются центры масс элементов сети.
 - Полученные точки проектируются на ось, соответствующую наибольшей стороне разделяемой сети. Таким образом, мы получаем упорядоченный список всех элементов сети.
 - Делением списка пополам (возможно, в нужной 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.
- Получаемое разбиение уступает по качеству более сложным и вычислительно-трудоемким методам.
- 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.
- 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, считая элементы сети точечными массами.
 - Пример разделения сети методом рекурсивной инерционной бисекции (стрелкой показана главная инерционная ось)





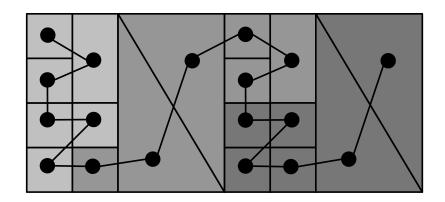


3. Деление сети с использованием кривых Пеано.

- Одним из недостатков предыдущих графических методов является то, что при каждой бисекции эти методы учитывают только одну размерность.
- Схемы, учитывающие больше размерностей, могут обеспечить лучшее разбиение. Один из таких методов упорядочивает элементы в соответствии с позициями центров их масс вдоль кривых Пеано.
- 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)*.



• Пример разделения сети на 3 части с использованием кривых Пеано





• Комбинаторные методы

- В отличие от геометрических методов, комбинаторные алгоритмы обычно оперируют не с сетью, а с графом, построенным для этой сети.
- Не принимают во внимание информацию о близости расположения элементов сети друг относительно друга, руководствуясь только смежностью вершин графа.
- Обычно обеспечивают более сбалансированное разбиение и меньшее информационное взаимодействие полученных подсетей.
- Однако комбинаторные методы имеют тенденцию работать существенно дольше, чем их геометрические аналоги.



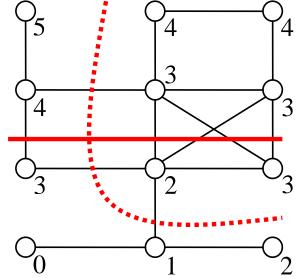
1. Деление с учетом связности.

- При разделении графа информационная зависимость между разделенными подграфами будет меньше, если соседние вершины будут находиться в одном подграфе.
- Алгоритм деления графов с учетом связности (levelized nested dissection) пытается достичь этого, последовательно добавляя к формируемому подграфу соседей.
- 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





- Для минимизации информационных зависимостей имеет смысл в качестве начальной выбирать граничную вершину. Перенумеровав вершины графа в соответствии с алгоритмом деления графов с учетом связности, мы можем взять любую вершину с максимальным номером она будет граничной.
- Пример работы алгоритма приведен на рисунке.
- Цифрами показаны номера, которые получили вершины в процессе разделения.
- Сплошной линией показана граница, разделяющая 2 подграфа.
- Решение далеко от идеального







2. Алгоритм Кернигана-Лина.

- В алгоритме Кернигана-Лина (Kernighan-Lin algorithm) при начале работы метода предполагается, что некоторое начальное разбиение графа уже существует, затем же имеющееся приближение улучшается в течение некоторого количества итераций.
- Используемый способ улучшения в алгоритме Кернигана-Лина состоит в обмене вершинами между подмножествами имеющегося разбиения графа. Для формирования требуемого количества частей графа может быть использована, как и ранее, рекурсивная процедура деления пополам.
- Пример перестановки двух вершин (выделены серым) в методе Кернигана-Лина





- 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.





• 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 таблицу общая характеристика ряда алгоритмов разделения графов, рассмотренных в данном разделе.



- Сравнительная таблица некоторых алгоритмов разделения графов

		Необходимость координатной информации	Точность	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			

Заключение



- В разделе был рассмотрены ряд алгоритмов для решения типовых задач обработки графов:
 - Алгоритм Флойда,
 - Алгоритм Прима.
- Был приведен обзор методов разделения графа:
 - Геометрические методы:
 - Покоординатное разбиение,
 - Рекурсивный инерционный метод деления пополам,
 - Деление сети с использованием кривых Пеано.
 - Комбинаторные методы:
 - Деление с учетом связности,
 - Алгоритм Кернигана-Лина.



Вопросы для обсуждения



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



Темы заданий для самостоятельной работы



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

Обзор литературы...



- Дополнительная информация по алгоритмам Флойда и Прима может быть получена, например, в
 - Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.:МСНТОб 1999
- Подробное рассмотрение вопросов, связанных с проблемой разделения графов, содержится в работах
 - Berger, M., Bokhari, S. Partitioning strategy for nonuniform problems on multiprocessors. IEEE Transactions on Computers, C-36(5). P. 570-580, 1987.
 - George, A., Liu, J. Computer Solution of Large Sparse Positive Definite Systems. Prentice-Hall, Englewood Cliffs NJ, 1981.
 - Gilbert, J., Miller, G., Teng, S. Geometric mesh partitioning: Implementation and experiments. In Proceedings of International Parallel Processing Symposium, 1995.
 - Heath, M., Raghavan, P. A Cartesian parallel nested dissection algorithm. SIAM Journal of Matrix Analysis and Applications, 16(1):235-253, 1995.
 - Miller, G., Teng, S., Thurston, W., Vavasis, S. Automatic mesh partitioning. In A. George, John R. Gilbert, and J. Liu, editors, Sparse Matrix Computations: Graph Theory Issues and Algorithms. IMA Volumes in Mathematics and its applications. Springer-Verlag, 1993.
 - Nour-Omid, B., Raefsky, A., Lyzenga, G. Solving finite element equations on concurrent computers. -In A. K. Noor, editor, American Soc. Mech. P. 291-307, 1986.
 - Patra, A., Kim, D. Efficient mesh partitioning for adaptive hp finite element methods. In International Conference on Domain Decomposition Methods, 1998.
 - Pilkington, J., Baden, S. Partitioning with spacefilling curves. Technical Report CS94-349, Dept. of Computer Science and Engineering, Univ. of California, 1994.
 - Raghavan, P. Line and plane separators. Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61901, 1993.
 - Schloegel K., Karypis G., Kumar V. Graph Partitioning for High Performance Scientific Simulations.
 2000



Обзор литературы



- Параллельные алгоритмы разделения графов рассматриваются в
 - Barnard S. PMRSB: Parallel multilevel recursive spectral bisection // Proc. Supercomputing '95. 1995
 - Gilbert, J., Miller, G., Teng, S. Geometric mesh partitioning: Implementation and experiments. - In Proceedings of International Parallel Processing Symposium, 1995.
 - Heath, M., Raghavan, P. A Cartesian parallel nested dissection algorithm. SIAM Journal of Matrix Analysis and Applications, 16(1):235-253, 1995.
 - Karypis, G., Kumar, V. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of Parallel and Distributed Computing, 48(1), 1998.
 - Karypis, G., Kumar, V. Parallel multilevel k-way partitioning scheme for irregular graphs. Siam Review, 41(2): 278-300, 1999.
 - Raghavan, P. Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee, 1995.
 - Walshaw, C., Cross, M. Parallel optimization algorithms for multilevel mesh partitioning. Techical Report 99/IM/44, Uviversity of Greenwich, London, UK, 1999.



Следующая тема



• Параллельные методы решения дифференциальных уравнений в частных производных

О проекте



Целью проекта является создание национальной системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения.

Задачами по проекту являются:

- Задача 1. Создание сети научно-образовательных центров суперкомпьютерных технологий (НОЦ СКТ).
- **Задача 2**. Разработка учебно-методического обеспечения системы подготовки, переподготовки и повышения квалификации кадров в области суперкомпьютерных технологий.
- **Задача 3**. Реализация образовательных программ подготовки, переподготовки и повышения квалификации кадров в области суперкомпьютерных технологий.
- **Задача 4**. Развитие интеграции фундаментальных и прикладных исследований и образования в области суперкомпьютерных технологий. Обеспечение взаимодействия с РАН, промышленностью, бизнесом.
- Задача 5. Расширение международного сотрудничества в создании системы суперкомпьютерного образования.
- Задача 6. Разработка и реализации системы информационного обеспечения общества о достижениях в области суперкомпьютерных технологий.

См. http://www.hpc-education.ru

