Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

Направление подготовки: «Прикладная математика и информатика»

ОТЧЕТ

по производственной практике
«Научно-исследовательская работа»

Программная реализация параллельного алгоритма k-truss разложения графа

Выполнила:
студентка группы 382003_3

_____Семенова В.А.

Проверил:
к.н.т., доцент

_____Мееров И.Б.

Нижний Новгород

2023

Оглавление

Введение

Графы распространены повсеместно. Любой набор взаимодействующих объектов может быть представлен в виде графа, где объекты являются вершинами, а взаимодействия – ребрами. Такое представление данных в виде графа используется во многих прикладных областях, например: биологические и транспортные системы, графы социальных сетей и многие др. Масштаб данных, используемых в графовой аналитике, растет беспрецедентными темпами, более чем когда-либо экспертам из разных предметных областей требуются эффективные и параллельные алгоритмы. Разработке таких реализаций для высокопроизводительных систем уделяется большое внимание в научном и техническом сообществе. Учитываются особенности обработки больших объемов таких данных: ограниченный параллелизм задач из-за работы с памятью, низкая арифметическая интенсивность, зависимость по данным между итерациями.

Чтобы понять структуру графа, часто бывает полезно найти плотно связанные наборы вершин и ребер или подграфы в графе. Существует несколько известных задач на нахождение связанных подграфов, однако, поскольку большинство из них NP-полные, точное решение требует больших вычислительных ресурсов для графов больших порядков.

Один из методов нахождения связанных подграфов – найти k_truss разложение графа. K-truss представляет собой иерархическую декомпозицию ребер графа и тесно связан с задачей перечисления треугольников. Такая формулировка связных подграфов полезна на практике, потому что k-truss может быть точно вычислен с использованием простых алгоритмов за полиномиальное время. В частности, k-truss — это мощный инструмент для определения структуры сообществ в графе, который может обеспечить понимание во время анализа графов. Способность эффективно вычислять декомпозицию k_truss имеет важное значение, поскольку на практике графы становятся все более большими и разреженным.

Целью работы является программная реализация и экспериментальное исследование последовательных алгоритмов Х. Кабир и К. Маддури для вычисления разложения k_truss графа.

1. Понятие k-truss разложения

K-truss в графе определяется как максимальный нетривиальный однокомпонентный подграф такой, что каждое ребро в подмножестве опирается по крайней мере на k-2 других ребра, которые образуют треугольники с этим конкретным ребром. Другими словами, каждое ребро k-truss должно быть частью k-2 треугольников, состоящих из узлов, которые являются частью k-truss.

Например, на следующем рисунке показано два k-truss третьего порядка. По определению, 3-truss – это треугольники графа *(*Рисунок 1. K_truss третьего порядка*рисунок 1.* к_truss третьего порядка *).*

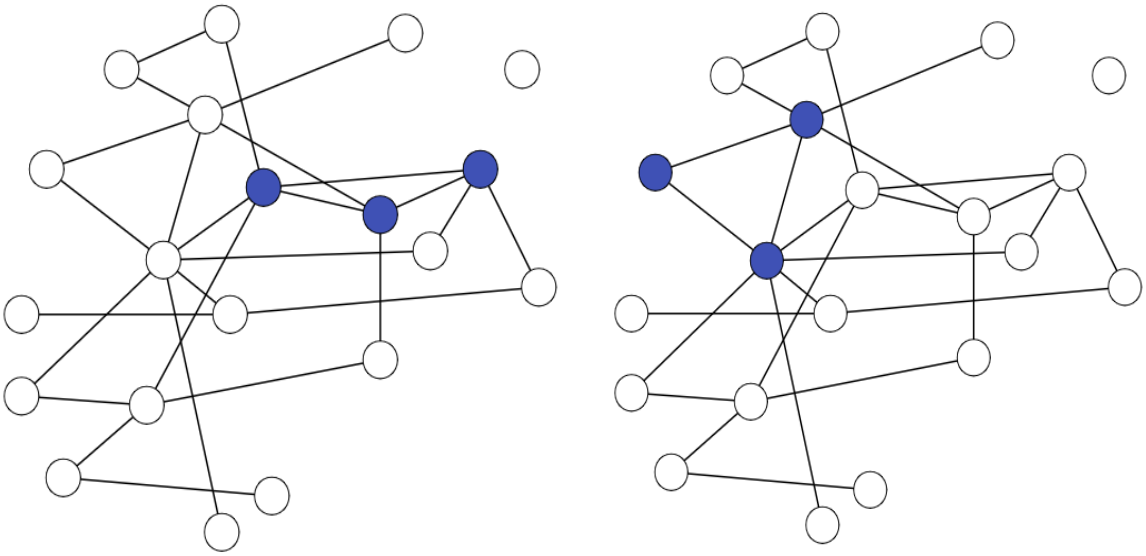


Рисунок 1. K_truss третьего порядка

Аналогично, на следующем рисунке *(Рисунок 2)* вы можете увидеть 4-truss — это набор таких ребер, что каждое ребро является частью двух треугольников, составленных из узлов truss.

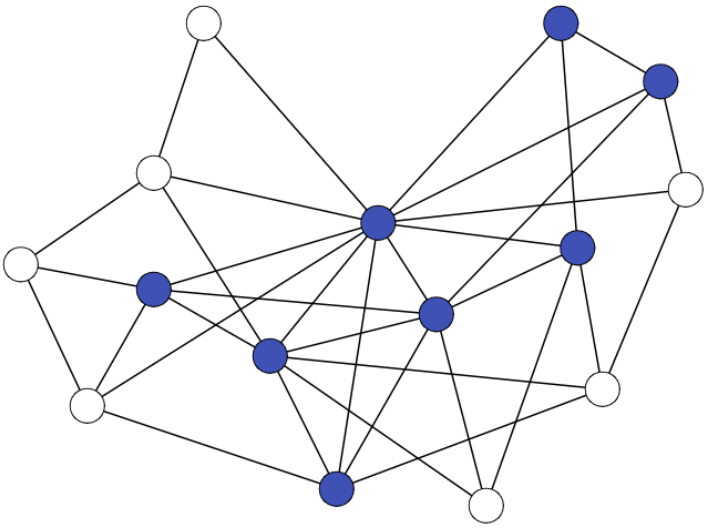


Рисунок 2. K_truss четвертого порядка

K-truss считается максимальным, когда не является подграфом другого k-truss: ребро принадлежит k-truss, если оно принадлежит k-truss графа, но не (k-1)-truss.

Так же для вычисления k-truss будет необходим подсчет *поддержки* для каждого ребра графа. Введем определение поддержки, согласно [1].

Пусть $G = (V; E)$ - неориентированный и невзвешенный граф с одной связной компонентой, V обозначает множество вершин, а E – множество ребер графа. Число вершин и ребер обозначим

как $n = |V|$ и $m = |E|$. *Поддержка* $S\ (e, G)$ ребра $e = (u;\ v)\ \epsilon\ G$ – количество треугольников, в которых оно содержится.

K-truss в таком случае будет определяется следующим образом: k-truss $T_k\ (k > 2)$ является максимальным связным подграфом G таким, что для каждого ребра e из этого подграфа $S(e, T_k) > k - 2$.

На следующем рисунке *(Рисунок 3)* приведен пример подсчета поддержки для ребер неориентированного графа G.

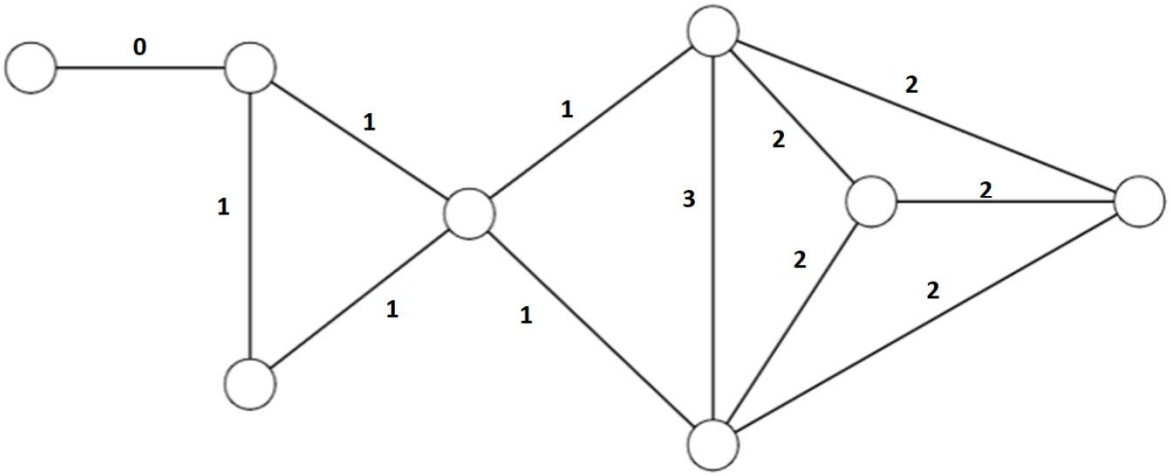


Рисунок 3. Пример подсчета поддержки для графа с 8 вершинами и 12 ребрами

Далее *(Рисунок 4)* представлено разложение k_truss на основе проведенных расчётов.

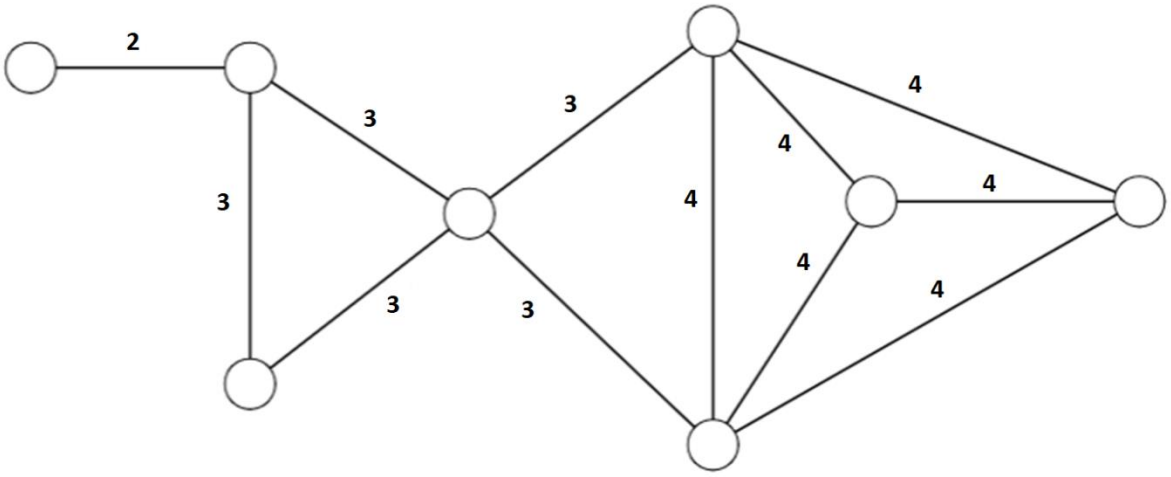


Рисунок 4. Разложение графа по уровням k_truss

Одним из основных алгоритмов для разложения k-truss является алгоритм Коэна Дж. [5] для нахождения максимального k-truss на основе обхода снизу вверх. В 2012 Ван и Ченг [6] представили усовершенствование алгоритма Коэна: ребра сортируются в порядке возрастания их поддержки с использованием линейной сортировки, затем обрабатываются в порядке возрастания поддержки. В работе рассмотрена модификация алгоритма Х. Кабир и К. Маддури на основе подсчета треугольников, так как вычисление поддержки ребер тесно связано с проблемами точного подсчета треугольников и [1, 2].

2. Последовательный алгоритм

2.1. Основной алгоритм вычисления k-truss

Большинство алгоритмов для вычисления разложения k-truss основаны на концепции отсечения: на каждом этапе удаляются ребра со значением меньше k из графа, затем инцидентность ребер обновляется. Далее приведен алгоритм, предложенный Джонатаном Коэна вместе с определением понятия k-truss.

Алгоритм 1:

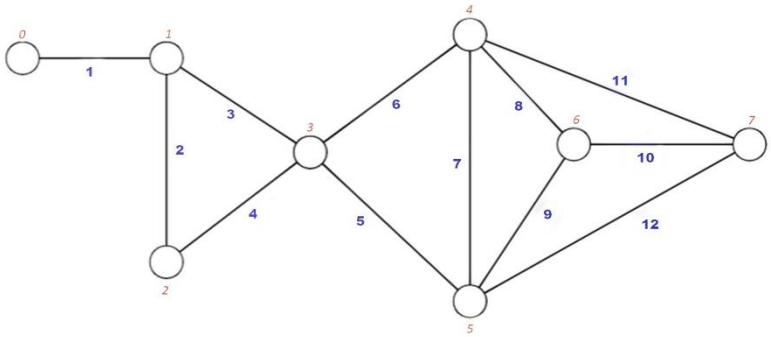
K_TRUSS(G, k):

1. **until** no change **do**
2. **for** each edge e = (a,b) in G:
3. **if** (size of CROSS(NEIGHBOUR(a), NEIGHBOUR(b) < k - 2:
4. remove e from g
5. **return** связанные компоненты оставшегося графика

В приведенном выше алгоритме G это граф и k это порядок k-truss, функция NEIGHBOUR(a) возвращает соседние вершины для вершины a, функция CROSS(s1, s2) находит пересечение двух наборов s1 и s2 .

Идея, лежащая в основе алгоритма, состоит в том, чтобы уменьшить граф так, чтобы в конечном итоге связанные компоненты графа были максимальными k-truss.

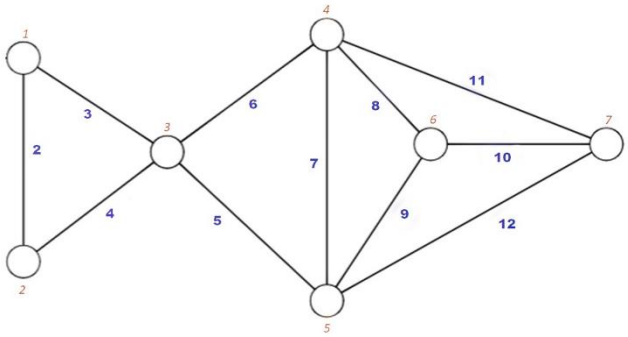
Рассмотрим данный алгоритм на приведенном ранее графе, состоящим из восьми вершин и 12 неориентированных ребер, в качестве структуры хранения используется список смежности.



G=(12,8)
0: 1
1: 0, 2, 3
2: 1, 3
3: 1, 2, 4, 5
4: 3, 5, 6, 7
5: 3, 4, 6, 7
6: 4, 5, 7
7: 4, 6, 5

Рисунок 5. Исходный граф для вычисления k_truss

При k=3 из графа будет удаленно ребро 1-(0,1), т.к. $|(1: 0, 2, 3) \cup (0: 1)| = 0 < k - 2 = 1$. Следовательно удаленное ребро будет принадлежать (k-1)-truss;



G=(11,7)
1: 2, 3
2: 1, 3
3: 1, 2, 4, 5
4: 3, 5, 6, 7
5: 3, 4, 6, 7
6: 4, 5, 7
7: 4, 6, 5

Рисунок 6. Граф для вычисления k-truss при k=3

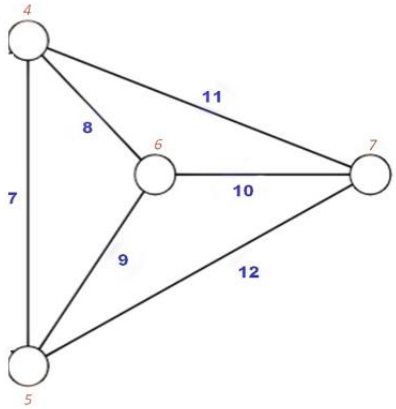
При k=4, из графа будет удаленно ребро 2-(1,2), 3-(1,3), 4-(2,3), 5-(3,5), 6-(3,4) так как

$|(1: 2, 3) \cup (2: 1, 3)| = 1 < k - 2 = 2;$

$|(1: 2, 3) \cup (3: 1, 2, 4, 5)| = 1;$ $|(2: 1, 3) \cup (3: 1, 2, 4, 5)| = 1;$

$|(3: 1, 2, 4, 5) \cup (5: 3, 4, 6, 7)| = 1;$ $|(3: 1, 2, 4, 5) \cup (4: 3, 5, 6, 7)| = 1;$

Все удаленный ребра на данной итерации составляют 3-truss;



G=(7,4)
4: 5, 6, 7
5: 4, 6, 7
6: 4, 5, 7
7: 4, 6, 5

Рисунок 7. Граф для вычисления k-truss при k=4

При k=5 из графа будут удалены все оставшиеся ребра, соответственно максимальный k-truss будет иметь порядок 4 и состоять из ребер 7-(4,5), 8-(4,6), 9-(5,6), 10-(6,7), 11-(4,7) и 12-(5,7). После этой итерации граф остается пустым, алгоритм поиска максимального k-truss в графе завершен.

Полученный алгоритм эффективен в вычислительном отношении, но его сложно распараллелить из-за неструктурированного характера вычислений и динамической природы графа. Алгоритм не предназначен для высокопроизводительных вычислительных систем, поскольку не учитывает детали реализации, которые имеют решающее значение для достижения высокой пропускной способности на современных архитектурах.

2.2. Модификация алгоритма Коэна Х. Кабир и К. Маддури.

В данной работе для разложения k_truss будет использоваться более эффективный алгоритм, в котором вычисляется поддержка для каждого ребра графа. Ключевой идеей этого подхода является подсчет треугольников в графе. Для вычисления числа треугольников разработан ряд алгоритмов, что предоставляет возможность сравнить несколько разных реализаций, которые могут подойти для параллельных вычислений на высокопроизводительных системах.

В основе реализации будет лежать следующий алгоритм (алгоритм 2).

Алгоритм 2:

PKT(G, S):

1. Initialize array S
2. SUPPORT(G, S)
3. curr ← φ; next ← φ;
4. flag ← φ;
5. todo = nE; k_level = 0
6. **while** todo > 0 **do**
7. CURR_INIT(S, k_level, curr)
8. **while** (curr > 0) **do**
9. todo ← todo - |curr|
10. SUBLEVEL(curr, S, k_level, next, flag)
11. curr ← next
12. next ← φ
13. k_level++

В алгоритме PKT значения k-truss вычисляются восходящим способом: сначала обрабатываются ребра, принадлежащие к классу (k), перед обработкой ребер, принадлежащих (k + 1) классу. Выходной массив S содержит значения поддержки всех ребер. Алгоритм начинается с параллельного вычисления поддержки ребер и сохраняет поддержку в массиве S. Далее он использует процедуры CURR_INIT и SSUBLEVEL (алгоритм 3).

Алгоритм 3:

procedure CURR_INIT(S, k_level, curr):

1. **for** (e = 0 to m - 1) **do**
2. **if** (S[e] = k) **then**
3. curr [e] ← true

procedure SUBLEVEL(curr, S, k_level, next, flag):

1. **for** (e_1 ϵ curr) **do**
2. (u,v) ← e_1
3. **for** (j = Es[u] to Es[u + 1] - 1) **do**
4. w ← N[j]
5. X[w] ← j + 1
6. **for** (j = Es[v] to Es[v + 1] - 1) **do**
7. W ← N[j]
8. **if** (X[w] = 0) **then**
9. **continue**
10. e_2 ← eid[j]
11. e_3 ← eid[X[w] - 1]
12. **if** (flag [e_2] or flag [e_3]) **then**
13. **continue**
14. **if** (S[e2] > l) **then**
15. S[e2] = S[e2] - l;
16. **if** (S[e3] > l) **then**
17. S[e3] = S[e3] - l;
18. **for** (j = Es [u] to Es [u + 1] - 1) do

19. w ← N[j]
20. X[w] ← 0
21. flag[e_1] ← true

Алгоритм использует функцию CURR_INIT для сканирования массива S и поиска ребер с поддержкой k–2. В функции SUBLEVEL рассматриваются найденные ребра, если вершин ребра имеют пересечения списков смежности (то есть образуется треугольник), то поддержку этих ребер следует уменьшить на один, она больше чем k-2, и если новое значение поддержки равно k-2, то ребро добавляется в очередь на обработку. После обработки ребра его помечают (удаляют из графа, чтобы не рассматривать их на следующем уровне k-truss). Это продолжается до тех пор, пока в k-truss нельзя будет добавить больше ребер. Массивы curr и next используются для отслеживания ребер на текущем уровне.

Если мы обратимся к рассмотренному нами ранее примеру подсчета максимального k-truss для графа на Рис.5 и так же учтем поддержку для данного графа на Рис.3, то очевидна зависимость между поддержкой ребра и его принадлежность к уровню k-truss, за исключением ребра 7-(4,5), которое одновременно входит в треугольники, принадлежащие разным уровням. Обработка этого случая происходит, путем добавления ребра в очередь (в массив curr) повторно, после уменьшения поддержки ребра на единицу.

Далее внимание будет сосредоточено на описании подхода к вычислению поддержки.

2.3. Подсчет поддержки пересечением

Далее приведен алгоритм подсчета поддержки в симметричном невзвешенном графе методом пересечения множества соседних вершин, представленный Ванг Дж., Ченг Дж[4].

Алгоритм 4:

SUPPORT_AI(G)

1. X ← φ
2. **for** all $e = (u, v)\ \epsilon\ E$
3. **for** all $w\ \epsilon\ N(u)$
4. if (w != v)
5. X[w] = e;
6. **for** all $w\ \epsilon\ N(v)$
7. if (w != u)
8. if (X[w] == e)
9. sup++;
10. S[e] = sup;

Алгоритм построен следующим образом. Для каждой вершины u отмечаем соседей $N(u)$, используя массив X. Затем мы рассматриваем список смежности $w\ \epsilon\ N(v)$. Если помечено w, то вершины $v - u - w$ образуют треугольник, соответственно увеличиваем поддержку ребра e. При подсчете треугольников массив X может быть битовым вектором. В качестве оптимизации можно рассматривать разделение вершин на два множест $N^+(u)$, такое, что в него входят все вершины, номер которых больше u, и $N^-(u)$ – множество соседних вершин для вершины, с номерами меньше u. Тогда при выполнении алгоритма 3 первый цикл выполняется по всем вершинам, принадлежащим $N^+(u)$, а второй – по всем вершинам, принадлежащим $N^-(v)$, что сократит общее число итераций.

2.4. Подсчет поддержки маркировкой

Представлен алгоритм подсчета поддержки в симметричном невзвешенном графе методом маркировки удаленных ребер, представленный Коэна Х. Кабир и К. Маддури[1].

Алгоритм 5:

SUPPORT_AM(G)

```
1.  X ← φ
2.      for all u ϵ V
3.          for all w ϵ N+(u)
4.              X[w] = j + 1;
5.          for all v ϵ N−(v)
6.              for all w ϵ N+(v)
7.                  if (X[w])
8.                      S(u) ++;
9.                      S(v) ++;
10.                     S(w) ++;
11.         for all w ϵ N+(u)
12.             X[w] ← 0
```

Алгоритм подсчета поддержки маркировкой так же основан на подсчете треугольников в графе, и использует разделение множества соседних вершин на $N^+(u)$ и $N^-(u)$ для каждой вершины u.

Ожидается, что реализация алгоритма, основанного на маркировки смежности, окажется быстрее, чем алгоритма, основанного на пересечении, так как алгоритмы подсчета треугольников используют порядок вершин на основе степени и сочетают это с ориентацией ребер. С увеличением порядка вершин каноническое треугольное представление $v < u < w$ дает малое число операций.

3. Параллельный алгоритм

В работе выполнено распараллеливание алгоритма Хабир и Маддури для систем с общей памятью, согласно работе [1].

Параллельный подсчет поддержки ребер графа аналогичен описанному последовательному алгоритму 5 на основе маркировки смежности (алгоритм 6).

Алгоритм 6:

P_SUPPORT(G)

```
1.  X ← φ
2.    for all u ϵ V  in parallel do
3.      for all w ϵ N+(u)  do
4.          X[w] = j + 1;
5.      for all v ϵ N−(v)  do
6.          for all w ϵ N+(v) do
7.              if (X[w]) then
8.                  S(u) ++;
9.                  S(v) ++;
10.                 S(w) ++;
11.     for all w ϵ N+(u) do
12.         X[w] ← 0
```

В параллельной реализации алгоритма PKT основная идея остается прежней. Значения k-truss вычисляются в порядке возрастания: сначала обрабатываются ребра, принадлежащие к классу k, затем уже ребра, принадлежащие к классу k+1, начиная с k равным 2. На каждой итерации обрабатываются ребра с поддержкой $k - 2$, до тех пор, пока все ребра в графе не буду отмечены как обработанные (алгоритм 7).

Алгоритм 7:

PKT(G, S):

```
1.  Initialize array S
2.  SUPPORT(G, S)
3.  curr ← φ; next ← φ;
4.  flag ← φ;
5.  todo = nE; k_level = 0
6.  while todo > 0 do
7.          CURR_INIT(S, k_level, curr)
8.          while (curr > 0) do
9.          todo  ← todo - |curr|
10.         SUBLEVEL(curr, S, k_level, next, flag)
11.         curr ←  next
12.         next  ← φ
13. k_level++
```

Алгоритм 7 подсчета k-truss сканирует массив поддержки ребер S и ищет ребра с поддержкой $k - 2$ с помощью функции CURR_INIT. Если такие ребра найдены, то функция SUBLEVEL проверяет, образуется ли треугольник из вершин этих ребер, с помощью пересечения списков смежности. Если образуется, то поддержка этих ребер уменьшается на один. Если новое значение поддержки становится равным $k - 2$, то ребро добавляется в очередь на обработку.

Функции CURR_INIT и SUBLEVEL выполняются параллельно несколькими потоками (алгоритм 8), между потоками распределяются вершины.

Алгоритм 8:

procedure P_CURR_INIT(S, k_level, curr):

```
1.  Initialize a thread-local array buff of size s
2.  i ← 0
3.  for (e = 0 to m - 1) in parallel do
4.          if (S[e] = k) then
5.                  buff[i] ←e; i ← i + 1;
6.                  curr [e] ← true
7.          if (i=s) then
8.                  Atomically update end of curr
9.                  Copy buff  to curr
10.                 buff ← φ; i ← 0
11.   if (i>0) then
12.         Atomically update end of curr
13.         Copy buff  to curr
14.         buff ← φ; i ← 0
```

procedure P_SUBLEVEL(curr, S, k_level, next, flag):

```
1.  Initialize a thread-local array buff of size s
2.  i ← 0
3.  for (e1 ϵ curr) in parallel do
4.          (u,v) ← e1
5.          for (j = Es[u] to Es[u + 1] - 1) do
6.                  w ← N[j]
7.                  X[w] ← j + 1
8.          for (j = Es[v] to Es[v + 1] - 1) do
9.                  W ← N[j]
10.                 if (X[w] = 0) then
11.                         continue
12.                 e2 ← eid[j]
13.                 e3 ← eid[X[w] - 1]
14.                 if (flag [e2] or flag [e3]) then
15.                         continue
16.                 if (S[e2] > l) then
17.                         if (((e1 < e3) and Curr[e3]) or (Curr[e3] = false)) then
18.                                 a←atomicSub(S[e2], 1)
19.                                 if (a = (l + 1)) then
20.                                         buff[i] ← e2; i ← i + 1
21.                                         inNext[e2] ← true
22.                                         if (i = s) then
23.                                                 Atomically update end of curr
24.                                                 Copy buff  to curr
25.                                                 buff ←φ; i←0
26.                                         if (a <= l) then
27.                                                 atomicAdd(S[e2], 1)
28.         for (j = Es [u] to Es [u + 1] - 1) do
29.                 w ← N[j]
30.                 X[w] ← 0
31.         if (i>0) then
32.                 Atomically update end of curr
33.                 Copy buff  to curr
34.                 buff ← φ; i ←0
```

35. **for** (*e* ϵ curr) in parallel do
36. flag[e] ← true
37. inCurr[e] ← false

Стоит обратить внимание на то, что каждый треугольник обрабатывается только один раз, это связано с тем, что треугольники, содержащие общее ребро, обрабатываются, когда алгоритм вычисляет k-класс. Ребро удаляется после завершения обработки и треугольник больше не существуют в графе. Однако, это приводит к гонке между потоками при обработке треугольника и обновлении поддержки ребер, так как при вычислении k-класс ребра, имеющие поддержку равную k - 2, обрабатываются параллельно в SUBLEVEL.

Обработка треугольника u-v-w параллельно с ребрами e1 = (u;v), e2 = (u; w) и e3 = (v;w) происходит по следующему алгоритму (P_SUBLEVEL). Можно выделить три случая:

1) только одно ребро находится в curr. Поток, обрабатывающий его, также может обрабатывать треугольник.

2) все три ребра находятся в curr. Поддержка всех ребер равна k - 2, и треугольник может быть посещен тремя разными потоками. Однако поддержка ребер обновляться не будет, так как ни одно ребро не имеет поддержки больше, чем k - 2, поэтому любой поток может обрабатывать треугольник.

3) два ребра находятся в curr. Чтобы не обрабатывать треугольник дважды, он обрабатывается потоком, который получает идентификатор нижнего края. Таким образом, обрабатывает поток T1, если e1 < e2, и обрабатывает поток T2, если e2 < e1.

Поскольку все потоки добавляют ребра к массиву curr, эта операция должна выполняться атомарно. Чтобы уменьшить количество атомарных операций, каждый поток использует буфер buff, и ребра добавляются в curr из buff, когда buff становится полным. Также функция CURR_INIT помечает в массиве inCurr ребра, которые добавляются в curr. Аналогично, в процедуре SUBLEVEL потоки должны использовать атомарные операции для добавления ребер в массив curr. Количество атомарных операций уменьшается за счет назначения buff каждому потоку и копирования ребер из buff в следующий, когда buff заполняется.

Аналогичным образом была реализована буферизация в алгоритме подсчета поддержки для сокращения одновременных добавлений в массив и числа атомарных операций.

4. Программная реализация

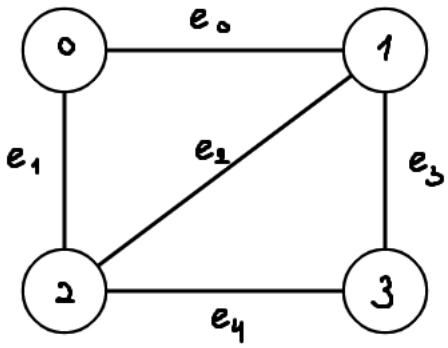
Реализация выполнена на языке программирования C++, использовалась технология OpenMP. Файловая структура программы:

1. `graphio.h` – заголовочный файл ввода-вывода графовой конструкции
2. `graphio.c` – файл с реализацией функций ввода-вывода
3. `ktruss.h` – заголовочный файл с описанием функций алгоритма Х. Кабир и К. Маддури последовательного разложения k-truss в графе
4. `ktruss.cpp` – файл с реализацией последовательного алгоритма Х. Кабир и К. Маддури разложения k-truss в графе
5. `P-ktruss.h` – заголовочный файл с описанием функций алгоритма Х. Кабир и К. Маддури параллельного разложения k-truss в графе
6. `P-ktruss.cpp` – файл с реализацией параллельного алгоритма Х. Кабир и К. Маддури разложения k-truss в графе
7. `main.c` – общий файл запуска алгоритмов с замерами времени

4.1. Структуры данных

Пусть дан граф $G = (V, E)$, где V обозначает множество ребер в граф, E – множество ребер графа. Обозначим $n = |V|$ и $m = |E|$. Структуры данных, используемые для хранения графа проиллюстрированы на рисунке 5 *(Рисунок 8).* Граф хранится в формате CRS, дополнительно используются три масси eid размером 2m используется для хранения идентификатора ребра, соответствующего каждому соседу вершины; Support размера m используется для хранения поддержки каждого ребра; edTo размера m используется для хранения списка вершин, соответствующих каждому ребру.

Таким образом, предполагая 4-байтовые целые числа, затраты памяти составляют (n+2m+2m+m+ 2m) × 4 байта = 28m+7n байт.



Xadj	0	2	5	8	10

Adjncy	1	2	0	2	3	0	1	3	1	2
eid	0	1	0	2	3	1	2	4	3	4

Support	1	1	2	1	1

edTo	0	0	1	1	2
	1	2	2	3	3

Рисунок 8. Структуры данных для хранения графа

4.2. Модульная структура программы

Файл graphio.cpp

`int init_graph(crsGraph* gr);` // инициализация графа
Входные параметры: указатель на структуру графа
Результат: собирает “пустой” граф, если удалось, возвращает нуль

`int free_graph_pointers(crsGraph* gr);` // освобождение указателей
Входные параметры: указатель на структуру графа
Результат: если граф пустой, выдает сообщение и возвращает единицу, иначе освобождает все массивы структуры и возвращает нуль

`int read_mtx_to_crs(crsGraph* gr, const char* filename);` // чтение mtx в списки смежности
Входные параметры: указатель на структуру графа, строка с именем файла mtx
Результат: собирает граф в структуру crs из файла mtx, если удалось, возвращает нуль

`int read_gr_to_crs(crsGraph* gr, const char* filename);` // чтение графа в списки смежности
Входные параметры указатель на структуру графа, строка с именем файла
Результат: собирает граф в структуру crs из файла, если удалось, возвращает нуль

`int write_crs_to_mtx(crsGraph* gr, const char* filename);` // запись графа в mtx
Входные параметры: указатель на структуру графа, строка с именем файла mtx
Результат: записывает граф crs в файл mtx, если удалось, возвращает нуль

`int read_arr_from_bin(double* arr, int size, const char* filename);` // чтение массива из бинарного файла
Входные параметры: указатель на структуру графа, размер файла, имя файла bin
Результат: собирает граф в структуру crs из файла bin, если удалось, возвращает нуль

`int write_arr_to_bin(double* arr, int size, const char* filename);` // запись массива в бинарный файл
Входные параметры: указатель на структуру графа, размер файла, имя файла bin
Результат: записывает граф crs в файл bin, если удалось, возвращает нуль

`int write_arr_to_txt(double* arr, int size, const char* filename);` // запись массива в текстовый файл
Входные параметры: указатель на структуру графа, размер файла, имя файла txt
Результат записывает граф crs в файл txt, если удалось, возвращает нуль

Файл ktruss.cpp

`void getEid(crsGraph* gr, int* eid, Edge* idEdge);` // инициализация массива идентификаторов ребер, соответствующеих каждому соседу вершины, и массива для хранения списка вершин, соответствующих каждому ребру для данного графа

Входные параметры: указатель на структуру графа, указатель на массив идентификаторов ребер, указатель на список вершин, соответствующих каждому ребру

Результат: записывает в массивы eid, idEdge данные о ребрах графа

`void SupAM(crsGraph* gr, int* eid, int* EdgeSupport);` // подсчет поддержки методом маркировке ребер для данного графа

Входные параметры: указатель на структуру графа, указатель на массив идентификаторов ребер, указатель на массив поддержки для каждого ребра

Результат: записывает в массивы EdgeSupport поддержку для каждого ребра

`void SupAI(crsGraph* gr, Edge* edTo, int* EdgeSupport);` // подсчет поддержки методом пересечения для данного графа

Входные параметры: указатель на структуру графа, указатель на указатель на список вершин, соответствующих каждому ребру, указатель на массив поддержки для каждого ребра

Результат: записывает в массивы EdgeSupport поддержку для каждого ребра

`void Curr_init(long nE, int* EdgeSupport, int k_level, int* curr, long* Tail);` // сканирования массива `EdgeSupport` и поиска ребер с поддержкой k_level - 2.

Входные параметры: указатель на массив поддержки для каждого ребра, значение текущего уровня разложения k_level для r-truss

Результат: записывает в массивы curr ребра с поддержкой k_level - 2.

`void SubLevel(crsGraph* gr, int* curr, long Tail, int* EdgeSupport, int k_level, int* next, long* nextTail, bool* flag, Edge* edTo, int* eid);` // Обработка помеченных ребер из curr и добавление новых ребер к k-truss

Входные параметры: указатель на структуру графа, указатель на массив curr ребер с поддержкой k_level - 2, текущее ребро для обработки, указатель на массив поддержки для каждого ребра, указатель на следующий элемент обработки, указатель на элемент для обработки следующего уровня, указатель на массив маркировки, указатель на идентификаторов ребер, указатель на список вершин, соответствующих каждому ребру.

Результат: обрабатывает треугольники и записывает новые ребра в k-truss

`int K_Truss(crsGraph* gr, int* EdgeSupport, Edge* edTo, int* eid);` // основная функция разложения k-truss для данного графа

Входные параметры: указатель на структуру графа, указатель на массив идентификаторов ребер, указатель на массив поддержки для каждого ребра

Результат: записывает в массивы EdgeSupport разложение k-truss графа.

P-ktruss

`void SupP(crsGraph* gr, int* eid, int* EdgeSupport);` // подсчет поддержки методом маркировке ребер для данного графа

Входные параметры: указатель на структуру графа, указатель на массив идентификаторов ребер, указатель на массив поддержки для каждого ребра

Результат: записывает в массивы EdgeSupport поддержку для каждого ребра

`void PCurr_init(long nE, int* EdgeSupport, int k_level, int* curr, long* Tail, bool* InCurr);` // сканирования массива `EdgeSupport` и поиска ребер с поддержкой k_level - 2.

Входные параметры: указатель на массив поддержки для каждого ребра, значение текущего уровня разложения k_level для r-truss

Результат: записывает в массивы curr ребра с поддержкой k_level - 2.

`void PSubLevel(crsGraph* gr, int* curr, bool* InCurr, int Tail, int* EdgeSupport, int k_level, int* next, bool* InNext, int& nextTail, bool* flag, Edge* edTo, int* eid);` // Обработка помеченных ребер из curr и добавление новых ребер к k-truss

Входные параметры: указатель на структуру графа, указатель на массив curr ребер с поддержкой k_level - 2, текущее ребро для обработки, указатель на массив поддержки для каждого ребра, указатель на следующий элемент обработки, указатель на элемент для обработки следующего уровня, указатель на массив маркировки, указатель на идентификаторов ребер, указатель на список вершин, соответствующих каждому ребру.

Результат: обрабатывает треугольники и записывает новые ребра в k-truss

`int PK_Truss(crsGraph* gr, int* EdgeSupport, Edge* edTo, int* eid);` // основная функция разложения k-truss для данного графа

Входные параметры: указатель на структуру графа, указатель на массив идентификаторов ребер, указатель на массив поддержки для каждого ребра

Результат: записывает в массивы EdgeSupport разложение k-truss графа

5. Результаты вычислительных экспериментов

5.1. Последовательные алгоритмы

Тестирование проводилось на симметричных невзвешенных графах с количеством вершин от 5 300 до 1 000 000 из коллекции SuiteSparse Matrix Collection (htt//sparse.tamu.edu/) на компьютере со следующими характеристика Intel(R) Core(TM) i3-7100U CPU, частота 2.40 GHz, память 16 GB.

Было проведено сравнение времени работы последовательных алгоритмов Х. Кабир и К. Маддури с применением подсчета поддержки пересечением и маркировкой смежности. Результаты представлены далее (таблица 1).

Таблица 1. Сравнение времени работы алгоритма вычисления k-truss с применением подсчета поддержки пересечением и маркировкой смежности.

№	Название графа	Число вершин	Число ребер	k_truss min		k_truss max		Маркировка смежности (секунды)		Пересечение смежности (секунды)	
				K	Ребра	k	Ребра	поддержка	k-truss	поддержка	k-truss
1	HB/bcspwr10	5 300	8 271	2	6 526	5	38	0,007	0,0022	0,001	0,0024
2	HB/bcsstk17	10 974	208 838	3	208 838	24	33 303	0,0495	0,1656	0,1029	0,2477
3	GHS_indef/ncvxbqp1	50 000	149 984	2	149 984	4	9 401	0,0107	0,0361	0,0114	0,0384
4	Rothberg/cfd2	123 440	1 482 229	6	1 482 229	10	13 020	0,1257	0,6693	0,3887	0,9178
5	Wissgott/parabolic_fem	525 825	1 574 400	3	1 574 400	3	1 574 400	0,0721	0,3622	0,1106	0,4051
6	McRae/ecology1	1 000 000	1 998 000	2	1 998 000	2	1 998 000	0,0614	0,2098	0,0944	0,2487

Рисунок 9. Сравнение алгоритмов пересечения и маркировки смежности

Алгоритм разложения k-truss графа с применением подсчета поддержки маркировкой смежности оказался быстрее, чем вариант алгоритма с методом пересечения. Из результатов экспериментов наглядно видно, что подсчет поддержки ускорился значительно, в некоторых случая более чем вдвое, это же ускорило и сам алгоритм разложения k-truss графа. Для тестирования были подобранным графы с различными характеристиками, разность скоростей вычисления поддержки не зависит от числа вершин, числа ребер и связанности графа. Однако заметно, что скорость самого алгоритма зависит от максимального уровня k-truss разложения и количества ребер на этом уровне.

Далее было проведено тестировано на наборе симметричных невзвешенных графов из коллекции SuiteSparse Matrix Collection с приблизительно равном количеством вершин и разными процентом разрежённости матрицы, чтобы оценить зависимость времени работы алгоритма от количества вершин, ребер, уровня разложения k-truss.

Таблица 2. Сравнение времен работы последовательных алгоритмов с применением подсчета поддержки пересечением и маркировкой смежности

№	Название графа	Число вершин	Число ребер	Разреже-ность %	k_truss min		k_truss max		Маркировка смежности (секунды)		Пересечение смежности (секунды)	
					k	Ребра	k	ребра	поддержка	k-truss	Поддержка	k-truss
1	HB/can_62	62	78	2,02	2	72	3	6	$0{,}14 \times 10^{-4}$	$0{,}46\times 10^{-4}$	$0{,}33\times 10^{-4}$	$0{,}44\times 10^{-4}$
2	HB/lap_25	25	72	11,52	4	72	4	72	$0{,}09\times 10^{-4}$	$0{,}28\times 10^{-4}$	$0{,}17\times 10^{-4}$	$0{,}41\times 10^{-4}$
3	FIDAP/ex5	27	126	17,28	6	126	9	102	$0{,}18\times 10^{-4}$	$0{,}68\times 10^{-4}$	$0{,}4\times 10^{-4}$	$0{,}85\times 10^{-4}$
4	Pajek/Journals	124	5 972	38,83	16	5 972	70	2 903	$0{,}45\times 10^{-2}$	$1{,}87\times 10^{-2}$	$0{,}96\times 10^{-2}$	$2{,}35\times 10^{-2}$
5	HB/bcsstk02	66	2 145	49,24	66	2 145	66	2 145	$0{,}69\times 10^{-2}$	$4{,}503\times 10^{-2}$	$0{,}99\times 10^{-2}$	$5{,}04\times 10^{-2}$

Из результатов экспериментов, представленных выше *(таблица 2)*, видно, что при примерно одинаковом количестве вершин на время работы алгоритма сильно влияет разреженность матрицы. С увеличением числа ребер в графе время обработки так же увеличивается. Так же явно видна зависимость между k-уровнем обработки и количеством ребер на этом уровне, с длительностью разложения k-truss в графе. Это объясняется подходом к вычислению k-truss восходящим способом.

Также была собрана и протестирована параллельная реализация k-truss из библиотеки NetworkX (htt//networkx.org), проведенно сравнении времени работы двух алгоритмов. Данные тестирования представлены далее *(таблица 3)*.

Таблица 3. Сравнение времени работы алгоритма с применением подсчета маркировкой смежности разложения k-truss (PKT) с аналогичным алгоритмом из библиотеки NetworkX

№	Название графа	Число вершин	Число ребер	k_tuss min		k_tuss max		PKT (секунды)	NetworkX (секунды)
				K	ребра	k	Ребра		
1	HB/bcspwr10	5 300	8 271	2	6 526	5	38	0.0022	0.1358
2	HB/bcsstk17	10 974	208 838	3	208 838	24	33 303	0.1656	141.1181
3	GHS_indef/ncvxbqp1	50 000	149 984	2	149 984	4	9 401	0.0361	5.89
4	Rothberg/cfd2	123 440	1 482 229	6	1 482 229	10	13 020	0.6693	152.48
5	Wissgott/parabolic_fem	525 825	1 574 400	3	1 574 400	3	1 574 400	0.3622	1499.27

При тестировании найденные наборы вершины алгоритмом Х. Кабир и К. Маддури для всех уровней разложений k-truss совпали с множеством вершин, найденным алгоритмом **NetworkX.** Результаты одинаковые. Алгоритм Х. Кабир и К. Маддури работает существенно быстрее.

5.2. Параллельные алгоритмы

В тестировании были использованы симметричные невзвешенные графы с количеством вершин от 100 196 до 4 847 571 из коллекции SuiteSparse Matrix Collection (htt//sparse.tamu.edu/), представленные далее.

Таблица 4. Характеристики тестовых графов для параллельной версии

№	Название графа	Число вершин	Число ребер	Разреженность %	k max	Max degree	Max k-core	Lower bound of Maximum Clique	Number of triangles
1	GHS_psdef /ford2	100 196	222 246	0,221%	27	29	27	27	210,3K
2	TSOPF/TSOPF_FS_b39_c30	120 216	1 545 520	1,069%	21	60K	30	21	1,3M
3	DNVS/shipsec8	114 919	3 269 240	2,476%	30	131	48	24	2,4K
4	PARSEC/Ga10As10H30	113 081	3 001 276	2,347%	390	697	390	390	121,7K
5	as-skitter	1 696 415	11 095 298	0,039%	68	35,5K	112	48	564,4K
6	in-2004	1 382 908	8 269 821	0,043%	4 253	21,9K	947	465	1,9M
7	soc-LiveJournal1	4 847 571	222 246	0,221%	3 533	22,9K	-	-	-

Тестирование проводилось на узле кластера «Лобачевский» с со следующими характеристиками: процессор 2x AMD EPYC 7742 (64 ядра), память 512.00 GB DDR4, частота 3200.00 MGHz. Параллельные алгоритмы запускались в 1, 2, 4, 8, 16 потоков. Были получены следующие результаты (Таблица 5, Таблица 6):

Таблица 5. Сравнение времени работы параллельного алгоритма подсчета поддержки на узле кластера при разном количестве потоков. Время указано в секундах. Расписание параллельных секций - static

Число потоков	GHS_psdef /ford2	TSOPF/TSOPF_FS_b39_c30	DNVS/ shipsec8	PARSEC /Ga10As10H30	as-skitter	in-2004	soc-LiveJournal1
1	0,0049	18,31	0,171	0,516	16,9	0,62	8,021
2	0,0088	13,75	0,155	0,329	16,67	0,442	8,571
4	0,0055	13,71	0,052	0,223	15,57	0,329	6,724
8	0,0047	7,972	0,032	0,147	11,88	0,255	4,606
16	0,0047	4,377	0,023	0,087	9,179	0,282	2,882

Таблица 6. Сравнение времени работы параллельного алгоритма подсчета k-truss на узле кластера при разном количестве потоков. Время указано в секундах. Расписание параллельных секций - static

Число потоков	GHS_psdef /ford2	TSOPF/TSOPF_FS_b39_c30	DNVS/ shipsec8	PARSEC /Ga10As10H30	as-skitter	in-2004	soc-LiveJournal1
1	0,0169	46,03	2,378	8,716	68,03	34,74	118,601
2	0,0234	24,48	1,010	2,106	55,58	28,42	87,62
4	0,0134	12,04	0,661	1,122	27,49	18,77	47,81
8	0,0092	6,208	0,423	0,604	20,11	15,52	28,48
16	0,0051	4,077	0,2805	0,459	11,63	10,88	22,98

Если рассматривать зависимость времени работы алгоритма от числа потоков, задействованных программой, то получим, что на маленьких графах мы наблюдаем относительно небольшой рост ускорения при подсчете и поддержки, и k-truss с увеличением числа потоков, в то

время как на сравнительно больших графах – рост ускорения с ростом числа запущенных в работу потоков. Наглядно зависимость отражена следующими диаграммами (Рисунок 11, Рисунок 12).

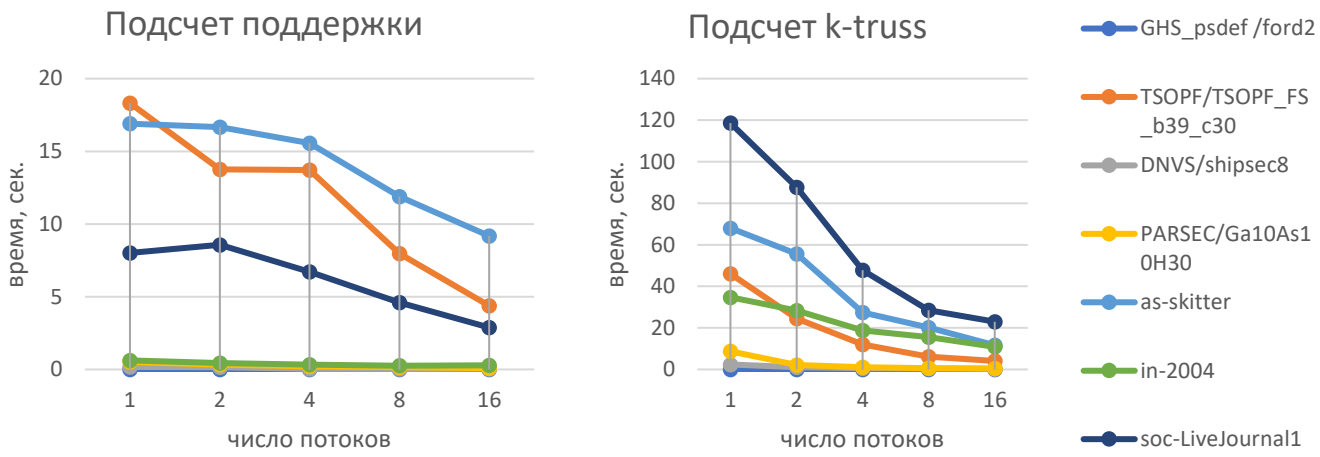


Рисунок 10. Сравнение времени работы алгоритмов РКТ при разном количестве потоков. Время указано в секундах. Расписание параллельных секций - static

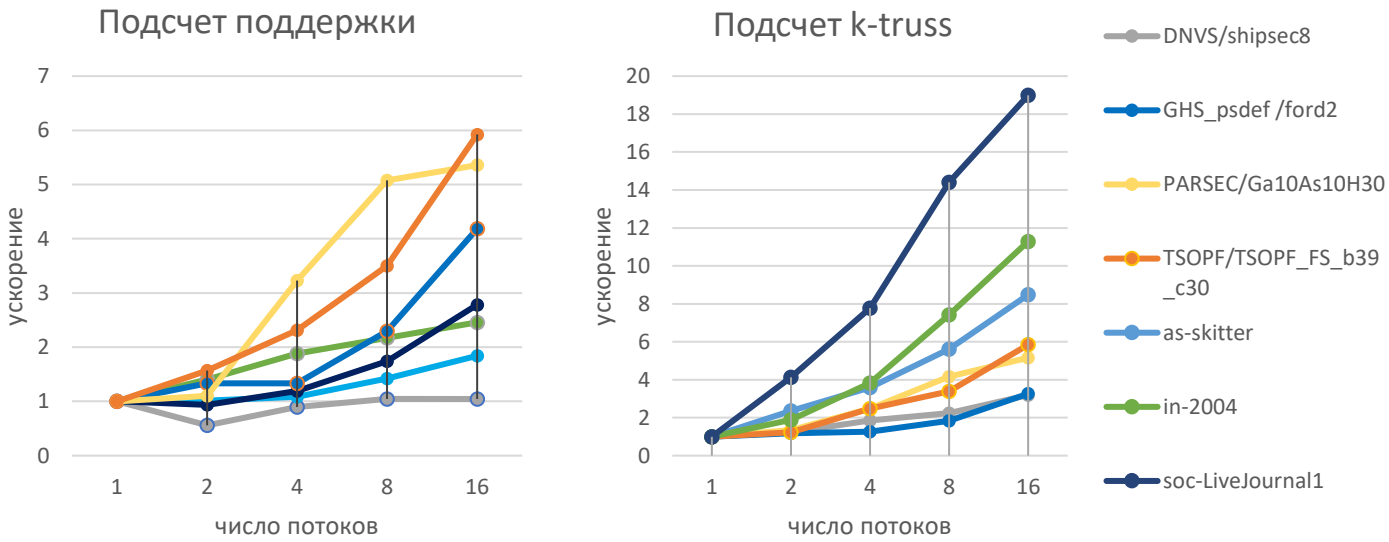


Рисунок 11. Сравнение масштабируемости алгоритмов РКТ. Расписание параллельных секций - static

Параллельный цикл в OpenMP позволяет задать опцию schedule, изменяющую способ распределения итераций между потоками. Поддерживается следующие опции планирования:

- schedule(static) — статическое планирование. При использовании такой опции итерации цикла будут поровну (приблизительно) поделены между потоками.
- schedule(static, K) — блочно-циклическое распределение итераций. Каждый поток получает заданное число K итераций в начале цикла, затем (если остались итерации) процедура распределения продолжается. Планирование выполняется один раз.
- schedule(dinamic), schedule(dynamic, K) — динамическое планирование. Каждый поток получает заданное число итераций K, выполняет их и запрашивает новую порцию. В отличии от статического планирования, во время выполнения программы распределение итераций выполняется многократно. Конкретное распределение итераций между потоками зависит от темпов работы потоков и трудоемкости итераций.

При удачном выборе параметров планирования параллельных секций в openMP время использования CPU может значительно увеличиться, благодаря правильному распределению нагрузки между потоками, что позволяет эффективно использовать ресурсы процессора и ускорить выполнение алгоритма. Однако, неудачный выбор параметров планирования может снизить производительность. Например, неправильный выбор типа планирования задач или размера блоков может привести к тому, что некоторые потоки будут простаивать, а другие будут перегружены работой. Также стоит учитывать, что время использования CPU зависит от сложности алгоритма и количества данных, которые нужно обработать. Поэтому необходимо уделить достаточно внимания настройке параметров и выбору оптимального режима работы для конкретной задачи.

Было проведено тестирование различных типов расписания для параллельных секций на наборе симметричных невзвешенных графов из SuiteSparse Matrix Collection (Таблица 4).

Таблица 7. Сравнение времени работы параллельного алгоритма PKT на узле кластера при разных параметрах планирования на графе TSOPF/TSOPF_FS_b39_c30.
Время указано в секундах

		число потоков						число потоков				
	Поддержка	**1**	**2**	**4**	**8**	**16**	k-truss	**1**	**2**	**4**	**8**	**16**
static												
без chunk		18,32	13,76	13,72	7,97	4,38		46,03	24,49	12,05	6,21	4,08
chunk=10		18,26	9,22	4,66	2,40	1,32		46,30	23,66	12,00	6,34	3,88
chunk=16		18,28	9,22	4,58	2,39	1,93		46,04	23,48	11,86	6,20	3,50
chunk=32		18,30	9,39	4,78	2,69	1,43		46,08	23,67	12,19	6,45	3,66
chunk=100		18,51	9,21	4,82	2,49	1,33		47,37	23,54	11,95	6,46	3,35
chunk=200		18,49	9,23	4,81	2,50	1,71		46,83	23,22	12,02	6,26	3,30
dynamic												
без chunk		18,28	13,82	17,39	6,79	3,28		46,05	23,46	12,10	6,78	3,52
chunk=4		18,26	13,80	11,54	7,77	4,08		46,02	23,25	11,65	6,21	3,38
chunk=10		18,28	13,85	12,18	9,54	4,32		46,07	23,25	11,78	6,06	3,59
chunk=16		18,28	13,90	11,64	8,73	5,34		46,05	23,25	11,89	6,18	3,38
chunk=32		18,26	13,94	16,37	9,03	5,43		46,05	23,11	11,89	6,18	3,44

Из результатов экспериментов, представленных выше, видно, что максимальное ускорение в алгоритме подсчёта поддержки достигается при использовании статистического расписания с блоком маленького размера, в алгоритме подсчета k-truss для достижения минимального времени работы следует использовать динамического расписание.

Статический тип планирования в openMP рекомендуется использовать в тех случаях, когда размеры задач равномерны, так же стоит учесть, что динамический тип планирования имеет более высокие накладные расходы, чем статический, поскольку он динамически распределяет итерации во время выполнения, однако динамическое расписание полезно, когда потоки получают различные вычислительные ресурсы, что оказывает существенное воздействие на разные объемы работы для каждой итерации.

Аналогичное поведения прослеживается и при тестирование остальных матриц, как и при подсчёте поддержи, так и при подсчёте k-truss, что наглядно видно из Таблицы 8 и Таблицы 9.

Таблица 8. Сравнение времени работы параллельного алгоритма подсчета поддержки на узле кластера при разных параметрах планирования с различным количеством потоков

	as-skitter						soc-LiveJournal1				
число потоков	**1**	**2**	**4**	**8**	**16**		**1**	**2**	**4**	**8**	**16**
без chunk	16,90	16,68	15,57	11,89	9,18		8,02	8,57	6,72	4,61	2,88
chunk=10	17,28	8,96	4,51	2,39	2,13		10,11	5,44	3,16	1,83	1,49
chunk=16	16,97	8,72	4,53	2,38	1,32		7,24	4,93	2,77	1,63	1,41
chunk=32	16,88	8,70	4,45	2,39	1,89		8,14	4,55	2,71	1,81	1,58
chunk=100	18,68	8,89	4,49	2,44	1,56		10,32	5,16	2,85	1,71	1,47

Таблица 9. Сравнение времени работы параллельного алгоритма подсчета k-truss на узле кластера при разных параметрах планирования с различным количеством потоков

	as-skitter						soc-LiveJournal1				
число потоков	**1**	**2**	**4**	**8**	**16**		**1**	**2**	**4**	**8**	**16**
без chunk	67,68	36,00	18,28	9,53	5,33		118,60	87,62	47,82	28,49	22,99
chunk=10	67,73	35,83	18,34	9,47	5,36		118,12	71,49	58,79	37,98	12,82
chunk=16	69,44	35,53	18,02	9,68	5,54		118,15	103,35	63,94	16,79	38,61
chunk=32	70,53	36,21	18,92	10,47	6,40		150,79	90,49	63,94	37,05	29,12
chunk=100	67,68	36,00	18,28	9,53	5,33		121,58	85,84	54,03	37,67	25,10

При правильном подборе параметров расписания параллельного блока ускорение алгоритма значительно, при подсчете поддержки ускорение составляет в 3-6 раз, и при подсчете k-truss в 2, то наглядно видно из Рисунка 12.

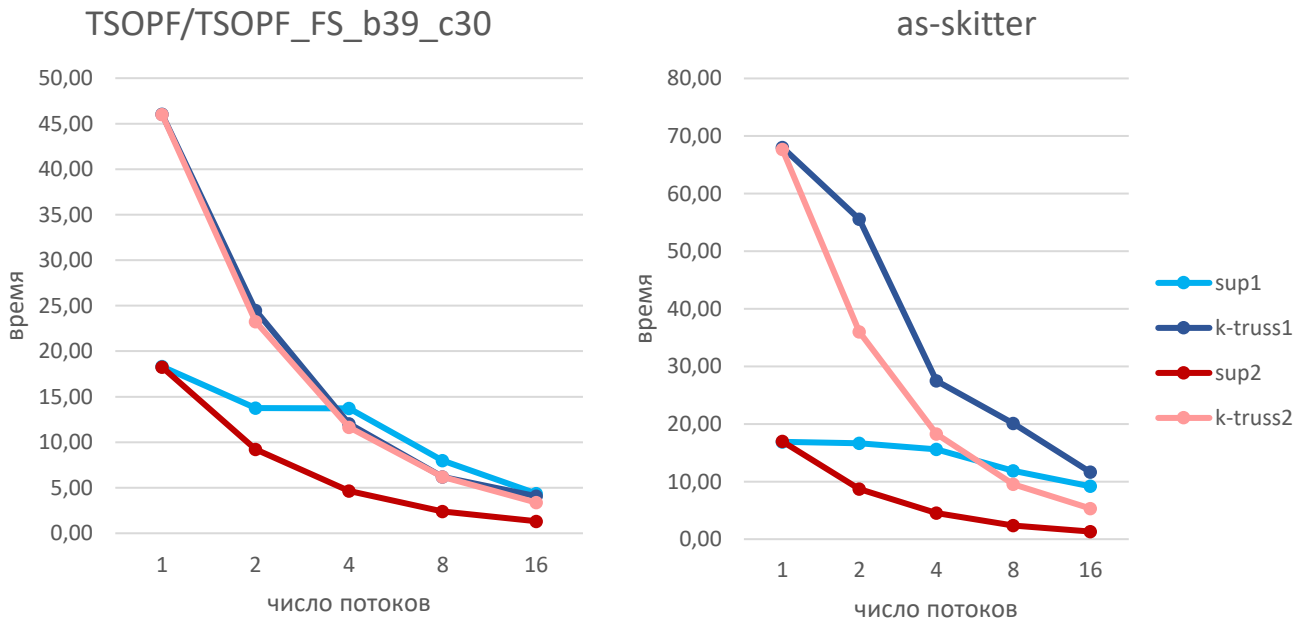


Рисунок 12. Сравнение алгоритмов РКТ с оптимальными параметрами планирования параллельных секций

Таблица 10. Сравнение процессорного времени для функций алгоритма разложения k-truss (PKT) при оптимальных параметрах планирования параллельных секций

Function	CPU (1) Time	CPU (2) Time		% of CPU (1) Time	% of CPU (2) Time
PSubLevel	443,667	449,383		77,80%	69,40%
SupP	99,583	170,691		17,50%	26,30%
_stdio_common_vfscanf	17,088	16,505		3,00%	2,50%
Malloc	4,987	5,412		0,90%	0,80%
free_dbg	2,866	3,156		0,50%	0,50%
[Others]	2,371	2,64		0,40%	0,40%

В Таблице 9 представлены результаты работы профилировщика производительности Intel VTune Profiler. Видно, что при грамотном выборе расписания для параллельных секций лучше распределяется процессорное время, что и является причиной ускорения работы алгоритма.

Заключение

В работе были рассмотрены и реализованы последовательные алгоритмы Х. Кабир и К. Маддури разложения k-truss графа. Были получены результаты вычислительных экспериментов произведено сравнение времени работы полученных реализаций.

Было рассмотрено два типа последовательных алгоритмов, основанных на применении подсчета поддержки пересечением и маркировкой смежности. Производительность первого алгоритма оказалось ниже, чем второго так как алгоритм подсчета треугольников рассматривает вершины в порядке уменьшения степени на основе степени и сортировку ребер в порядке возрастания их поддержки. С увеличением порядка вершин используемое представление дает малое число операций. По результатам сравнения с библиотекой NetworkX видно, что производительность представленного алгоритма Х. Кабир и К. Маддури сопоставима с библиотечной реализацией. Полученные результаты соответствуют ожидаемым.

Выполнено распараллеливание алгоритма Х. Кабир и К. Маддури разложения k-truss графа для систем с общей памятью с применением технологии OpenMP. Проведены вычислительные эксперименты с графами большего размера на узле кластера «Лобачевский», произведено сравнение времени работы полученных последовательных и параллельных реализаций, найдены в ходе экспериментов оптимальные параметры для расписания параллельных секций кода.

Литература

1. Kabir H. Madduri K. Parallel k-truss decomposition on multicore systems // 2017 IEEE High Performance Extreme Computing Conference (HPEC). – IEEE 2017. – С. 1-7. November 2017; d 10.1109/HPEC.2017.8091052
2. Kabir H. Madduri K. Shared-memory graph truss decomposition // 2017 IEEE 24th International Conference on High Performance Computing (HiPC). – IEEE 2017. – С. 13-22; d 10.1109/HiPC.2017.00012
3. Smith S. et al. Truss decomposition on shared-memory parallel systems // 2017 IEEE high performance extreme computing conference (HPEC). – IEEE 2017. – С. 1-6; d 10.1109/HPEC.2017.8091049
4. Wang J. Cheng J. Truss decomposition in massive networks //arXiv preprint arX1205.6693. – 2012.
5. Cohen J. Truss Cohesive subgraphs for social network analysis //National security agency technical report. – 2008. – Т. 16. – №. 3.1.