ПОТОКОВЫЙ БЛОЧНО-ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ НА ГРАФЕ

О.Н. КАРАСИК, А.А. ПРИХОЖИЙ

Белорусский национальный технический университет, Республика Беларусь

Поступила в редакцию 1 февраля 2018

Аннотация. Рассмотрена задача поиска кратчайших путей на взвешенных графах. Проанализированы варианты постановки задачи, известные алгоритмы решения, области практического применения и существующие проблемы, в частности проблема масштабируемости. Исследован класс блочно-параллельных алгоритмов, их достоинства и недостатки. Предложен быстрый потоковый блочно-параллельный алгоритм, ориентированный на графы большого размера и отличающийся изменением порядка вычислений блоков, сокращением критического пути, уменьшением времени работы на многоядерной системе, сокращением обменов данными между локальными кэш ядер и между уровнями памяти.

Ключевые слова: граф, кратчайший путь, блочный алгоритм, параллельные вычисления, потоковый алгоритм.

Abstract. The problem of finding the shortest paths on weighted graphs is considered. The variants of statement of the problem, known algorithms for it solving, areas of practical application and existing challenges, in particular, the challenge of scalability, are analyzed. The class of block-parallel algorithms, their advantages and disadvantages is investigated. A fast block-parallel threaded algorithm oriented to large-sized graphs is proposed. It differs by changing the order of block calculations, reducing the critical path and operating time on a multi-core system, decreasing the data exchanges among local caches of cores and between neighbor levels of hierarchical memory.

Keywords: graph, shortest path, blocked algorithm, parallel computing, multithreading.

Doklady BGUIR. 2018, Vol. 112, No. 2, pp. 77-84
Threaded block-parallel algorithm for finding the shortest paths on graph
O.N. Karasik, A.A. Prihozhy

Введение

Задачи поиска кратчайших путей во взвешенном графе [1–6] делятся на два больших класса: SSSP (Single Source Shortest Path) – нахождение всех кратчайших путей в графе из одной вершины в другие вершины и APSP (All Pair Shortest Path) – нахождение кратчайших путей между всеми парами вершин графа. Алгоритм Дейкстры является базовым для решения первой задачи, алгоритм Флойда-Уоршелла – для решения второй задачи. Решение задачи APSP и алгоритм Флойда-Уоршелла играют важную роль во многих приложениях: в системах баз данных для оптимизации обработки запросов, в системах автоматизированного проектирования, в микроэлектронике, в инструментальных средствах оптимизации конвейеров, в компьютерных играх, для анализа кластеров генов в биоинформатике, для планирования работы многоагентных систем, для распознавания речи и т. д. Во многих случаях размер реальных графов достигает таких огромных размеров, что алгоритм Флойда-Уоршелла и его известные модификации, имеющие полиномиальную степень вычислительной сложности, потребляют нереально большое процессорное время. В данной статье ставится цель разработки нового более быстрого блочно-параллельного алгоритма, построенного в виде множества кооперативно работающих потоков, выполняющих поблочный расчет матрицы кратчайших путей на многоядерной системе, использующей иерархическую организацию памяти.

Методика эксперимента

Пусть ориентированный взвешенный граф $G = (V, E)$ с множеством V из N вершин и множеством ребер E представлен матрицей W положительных весов ребер, в которой $w_{i,i} = 0$ при $i = 0 \ldots N-1$ и $w_{i,j} = \infty$ при $(i,j) \notin E$. Длины кратчайших путей между парами вершин описываются

матрицей D. Алгоритм Флойда-Уоршелла (ФУ) [1] на шагах $0…N–1$ пересчитывает матрицу $D^0 = W$ в результирующую матрицу $D^{N–1}$. Расчет элемента $d^{k+1}{}_{i,j}$ выполняется по формуле (1).

$$d^{k+1}{}_{i,j} = \min\left\{d^{k}{}_{i,j},\ d^{k}{}_{i,k} + d^{k}{}_{k,j}\right\}. \quad (1)$$

ФУ строится из трех циклов по k, i, j, работает на всех шагах с матрицей одинаковой размерности $N×N$ и имеет высокую однородность. Его вычислительная сложность равна $O(N^3)$.

Блочный алгоритм Флойда-Уоршелла (БФУ) [2–5] помог решить две важнейшие проблемы: 1) локализовать работу с многоуровневой памятью внутри блоков и тем самым сократить число операций обмена между уровнями; 2) организовать параллельное вычисление блоков на многопроцессорной системе. Разобьем матрицу D на блоки размерностью $B×B$ каждый с образованием матрицы блоков размерностью $M×M$, где $M = N/B$. Псевдокод БФУ показан на рис. 1. Его функционирование представляется циклом по $m = 0…M–1$, на каждой итерации которого выполняется упорядоченный однократный пересчет всех блоков алгоритмом *calcblock* (рис. 2, *а*), в котором аргумент B^1– пересчитываемый блок, аргументы B^2 и B^3 – блоки, через которые осуществляется пересчет.

На итерации m цикла каждый блок матрицы D рассчитывается до уровня $m+1$: сначала центральный блок $B_{m,m}$, затем блоки креста, лежащие на строке и столбце D с номером m, затем остальные блоки. Все вызовы *calcblock* можно заменить вызовами функции *calcblock_auto* (рис. 2, *б*). Все блоки креста вычисляются взаимно параллельно, но последовательно с $B_{m,m}$. Остальные блоки вычисляются взаимно параллельно, но последовательно с блоками креста. Всего БФУ пересчитывает M^3 блоков. Вычислительная сложность ФУ и БФУ одинакова.

Предлагаемый потоковый блочно-параллельный алгоритм (ПБПА) улучшает БФУ на основе следующих принципов. В основе лежит кооперативная модель выполнения потоков [7]. За каждым потоком закрепляется свое множество рассчитываемых блоков, а за каждой группой потоков закрепляется свой процессор. Способ закрепления локализует повторно используемые данные и сокращает обмен между уровнями памяти. Потоки взаимодействуют друг с другом так, что загрузка процессоров увеличивается за счет сокращения слотов времени ожидания и простаивания. В пределах одной группы потоки взаимодействуют путем прямой передачи управления. Главным отличием ПБПА от БФУ является комбинирование различных уровней вычисления блоков, что делает возможным реализацию перечисленных преимуществ.

```
1.  algorithm blocked_fw(D)
2.    for m=0 to M-1 do
3.      calcblock(D_m,m, D_m,m, D_m,m)
4.      for i=0 to m-1 do
5.          calcblock(D_i,m, D_i,m, D_m,m)
6.          calcblock(D_m,i, D_m,m, D_m,i)
7.      end for
8.      for i=m+1 to M-1 do
9.          calcblock(D_i,m, D_i,m, D_m,m)
10.         calcblock(D_m,i, D_m,m, D_m,i)
11.     end for
12.     for i=0 to m-1 do
13.         for j = 0 to m-1 do calcblock(D_i,j, D_i,m, D_m,j) endfor
14.         for j = m+1 to M-1 do calcblock(D_i,j, D_i,m, D_m,j) endfor
15.     end for
16.     for i=m+1 to M-1 do
17.         for j = 0 to m-1 do calcblock(D_i,j, D_i,m, D_m,j) endfor
18.         for j = m+1 to M-1 do calcblock(D_i,j, D_i,m, D_m,j) endfor
19.     end for
20.   end for
21. end algorithm
```

Рис. 1. Псевдокод алгоритма БФУ

```
1.  algorithm calcblock(B^1, B^2, B^3)
2.    for k = 0 to B-1 do
3.      var b3^rk = row(B^3, k)
4.      for i = 0 to B-1 do
5.          var b1^ri = row(B^1, i), b2 = B^2_i,k
6.          for j = 0 to B-1 do b1^ri_j = min(b1^ri_j, b2 + b3^rk_j) end for
7.      end for
8.    end for
9.  end algorithm
```

а

```
1.  algorithm calcblock_auto(m, i, j)
2.    if m ≠ i then
3.      if m = j then calcblock(D_i,j, D_i,j, D_m,m)
4.      else calcblock(D_i,j, D_i,m, D_m,j) end if
5.    else
6.      if m ≠ j then calcblock(D_i,j, D_m,m, D_i,j)
7.      else calcblock(D_i,i, D_i,i, D_i,i) end if
8.  end algorithm
```

б

Рис. 2. Алгоритмы блока: *а* – calcblock; *б* – calcblock_auto

Пусть для размера B блока выполняются равенства $N \bmod B = 0$ и $M \bmod P = 0$, где P – число процессоров. ПБПА вычисляет каждый блок M раз. Вычисление l блока $D_{i,j}$ назовем уровнем блока. Матрица L описывает уровни всех блоков на каждом шаге работы ПБПА. Для вычисления блоков вводится M потоков $t_0…t_{M–1}$. С целью локализации данных внутри потоков все блоки одной строки i матрицы D вычисляются одним потоком t_i. Потоки распределяются на P процессорах и образуют группы $g_0…g_{P–1}$. Поток t включается в группу g, если выполняется равенство $t \bmod P = g$. В группе g текущий поток обозначается c, первый поток обозначается

first(c) = *c mod P*, последний поток обозначается *last*(*c*) = *M*–*P*+(*c mod P*), поток *prev*(*c*) называется предыдущим для *c* если *prev*(*c*) = *c*–*P*, поток *next*(*c*) называется следующим для с если *next*(*c*) = *c*+*P*.

На способ передачи управления от блока *B* к вычисляемому блоку *A* влияет принадлежность блоков: 1) одному потоку; 2) разным потокам одной группы (процессора); 3) потокам разных групп (процессоров). В первом случае зависимость *A* от *B* разрешается в период компиляции. Во втором случае *A* и *B* принадлежат разным потокам одной группы, выполняемым последовательно на одном процессоре. Передача блоков достигается корректной передачей управления от одного потока к другому операцией *switch thread*. В третьем случае *A* и *B* принадлежат потокам разных групп, выполняемым параллельно на разных процессорах, а передача блока *B* из потока t_B в поток t_A достигается блокировкой потока t_A операцией *wait for* в ожидании по матрице уровней *L* окончания вычисления блока *B* потоком t_B с последующим уведомлением потока t_A операцией notify set.

Номера строк и столбцов *i*, *j* матрицы *D*, принимают значения от 0 до *M*–1, а уровень *l* принимает значение от 1 до *M*. Вычисление блока $D^l_{i,j}$ потоком t_i требует установления типа блока и разрешения зависимостей между блоками. В зависимости от значений *i*, *j* и *l* различают блоки четырех типов: центральный, горизонтальный, вертикальный и периферийный. Блок $D^l_{i,j}$ – центральный при $i = j = l–1$. Он зависит только от блока $D^{l–1}_{i,j}$. Блок $D^l_{i,j}$ – горизонтальный при $i = l$ -1 и $i \neq j$. Он зависит от блоков $D^{l–1}_{i,j}$ и $D^l_{l–1,l–1}$, которые находятся в одной строке матрицы *D* и вычисляются одним потоком t_i. Блок $D^l_{i,j}$ – вертикальный при $j = l–1$ и $i \neq j$. Он зависит от блоков $D^{l–1}_{i,j}$ и $D^l_{l–1,l–1}$. Разрешение зависимости $D^l_{i,j}$ от $D^l_{l–1,l–1}$ осуществляется посредством операции *switch thread*, если это блоки одной группы, иначе, посредством операций *wait for* и *notify set*, если это блоки разных групп. Блок $D^l_{i,j}$ – периферийный при $i \neq l–1$ и $j \neq l–1$. Он зависит от блоков $D^{l-1}_{i,j}$, $D^l_{i,l–1}$ и $D^l_{l–1,j}$. Блоки $D^l_{i,j}$ и $D^l_{i,l–1}$ находятся в одной строке матрицы *D*. Разрешение зависимости блока $D^l_{i,j}$ от блока $D^l_{l–1,j}$ осуществляется посредством операции *switch thread*, если это блоки одной группы, и посредством операций *wait for* и *notify set*, если это блоки разных групп. В процессе вычислений тип блока меняется. Центральный блок может быть только диагональным, горизонтальный или вертикальный блок не может быть диагональным, однако любой блок матрицы *D* может быть периферийным.

Рис. 3. Диаграмма переходов режимов потока

В ПБПА каждый поток рассчитывает блоки, находясь в одном из шести режимов: ведущий (master), ведомый (slave), дополняющий (compl), пассивный типа А (passiveA), пассивный типа Б (passiveB) и пассивный типа С (passiveC). Для каждого режима определяетсясвой набор, диапазон уровней и порядок расчета принадлежащих потоку блоков. Диаграмма переходов потока из одного режима в другой показана на рис. 3. Алгоритм ПБПА формируется из алгоритмов работы режимов. Его начальное состояние устанавливается при инициализации. Параллельный рассчет блоков потоками и взаимодействие потоков в ПБПА показаны на рис. 4 для матрицы блоков 4×4, вычисляемой на 2-х процессорах.

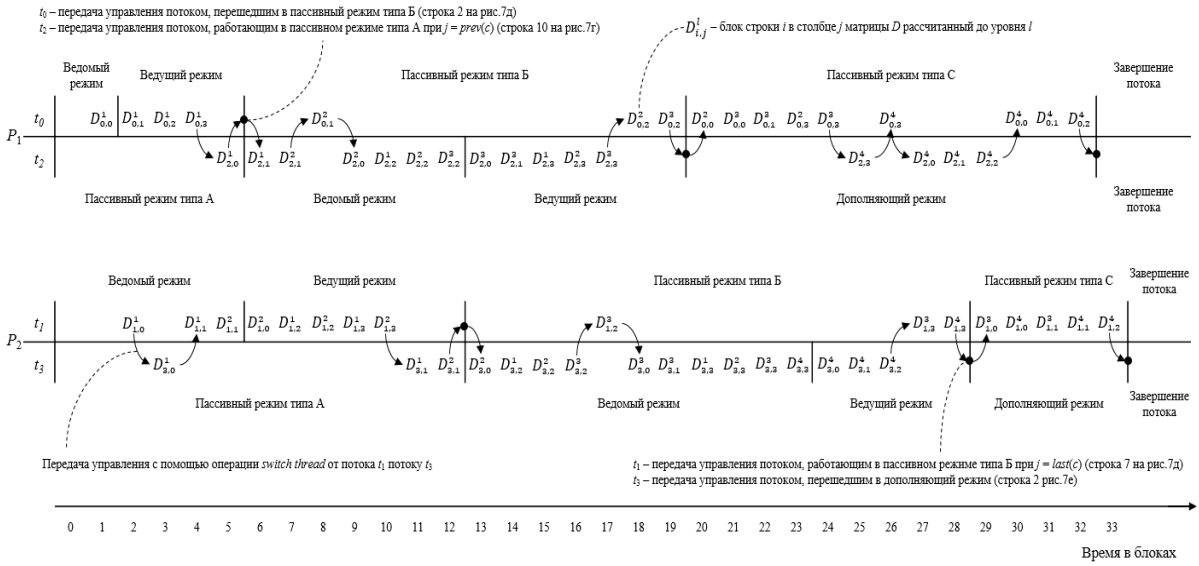


Рис. 4. Временная диаграмма работы ПБПА на матрице блоков 4×4 при использовании 2-х процессоров; единицей времени является время расчета одного блока на один уровень

Процесс развернут во времени, измеряемом в числе последовательно вычисленных блоков. На процессоре P_1 работают потоки t_0, t_2, на процессоре P_2 – потоки t_1, t_3. Потоки t_0, t_2, так же как и потоки t_1, t_3, выполняются взаимно последовательно. Пара потоков t_0, t_2 работает параллельно с парой потоков t_1, t_3.

В БФУ любой блок может быть вычислен до уровня l строго после того, как все блоки матрицы D вычислены до уровня l–1. В работе [3] при разработке рекурсивного блочного алгоритма доказано, что в формуле (1) элемент $d^{k+1}{}_{i,j}$ матрицы D может быть рассчитан через элементы $d^k{}_{i,j}$, $d^v{}_{i,k}$ и $d^u{}_{k,j}$ корректно, если $v \geq k$ и $u \geq k$. Многократно применяя это преобразование к вычислению блока $D^l{}_{i,j}$ через блок $D^{l-1}{}_{i,j}$ и блоки $D^l{}_{i,l-1}$ и $D^l{}_{l-1,j}$, приходим к выводу о том, что последние два блока могут быть заменены на $D^v{}_{i,l-1}$ и $D^u{}_{l-1,j}$ корректно, если $v \geq l$ и $u \geq l$. Это ослабление требований при построении ПБПА использовано авторами для переупорядочения вычислений блоков для более эффективного распараллеливания потоков, увеличения загрузки процессоров и повышения локализации обращений к данным в кэш. Например, на рис. 4 поток t_0 вычисляет блок $D^2{}_{0,2}$ в момент времени 18 не через блок $D^2{}_{1,2}$, а через блок $D^3{}_{1,2}$.

Режим работы потока меняется с течением времени. Так поток t_0 работает в ведомом режиме 1 единицу, в ведущем режиме – 4 единицы, в пассивном режиме типа Б – 14 единиц и в пассивном режиме типа С – 13 единиц времени. Передача управления по *switch thread* между потоками одного процессора показана на рис. 4 стрелками вниз и вверх. При получении управления поток выполняет пересчет назначенных на него блоков. Иногда, например, на процессоре P_1 в моменты времени 5, 19, 32, передача управления нужна для смены режима работы другого потока. Формальное описание алгоритмов работы потока в шести режимах показано на рис. 5. В каждой группе всегда ровно один поток находится в ведущем, ведомом или дополняющем режиме, остальные потоки работают в пассивном режиме типа А, Б или С. Ведущему и ведомому режиму одного потока аккомпанируют пассивные режимы типа А и Б остальных потоков группы. Дополняющему режиму одного потока аккомпанируют пассивные режимы типа С остальных потоков группы. Первый поток каждой группы начинает работу в ведомом режиме, остальные потоки начинают работу в пассивном режиме типа А. Всегда за ведомым следует ведущий режим, который сменяется на пассивный режим типа Б и далее на пассивный режим типа С для не последних потоков группы. Последний поток группы переходит в дополняющий режим, в котором и завершает работу. Смена режимов потоков одной группы, работающих на одном процессоре, синхронизирована.

```
1.  function master_thread_routine(c)
2.    for k = 0 to c–1 do
3.        calculate_block_auto(D,c,c,k)
4.        notify set L_c,k = c+1
5.    end for
6.    for j = c+1 to M–1 do
7.        forb = 0 to c–1 do
8.            wait for L_b,j = b + 1
9.            calculate_block_auto(D,b,c,j)
10.       end for
11.       calculate_block_auto(D,c,c,j)
12.       notify set L_c,j = c + 1
13.   end for
14.   if c ≠ first(c) then
15.       switchthreadfirst(c) //passiveB
16.   end if
17.   if c ≠ last(c) then
18.       switchthreadnext(c) //passiveA
19.   end if
20.   if c = last(c) then
21.       complement_thread_routine(c)
22.   else
23.       passive_thread_routine_type_B(c)
24.   end if
25. end function
```

а

```
1.  function slave_thread_routine(c)
2.    var r = (c<P ? 0 : prev(c))
3.    for k = 0 to r do
4.      for b = k+1 to r–1 do
5.          wait for L_b,k = b+1
6.          calculate_block_auto(D,b,c,k)
7.      end for
8.    end for
9.    for m= (c<P ?0 : r+1) to c–1 do
10.     for b = 0 to m do
11.         wait for L_b,m = b+1
12.         calculate_block_auto(D,b,c,m)
13.     end for
14.     if c ≠ first(c) then
15.         switch thread first(c) //passiveB
16.     end if
17.     if c ≠ last(c) then
18.         switch thread next(c) //passiveA
19.     end if
20.     for j = 0 to m–1 do
21.         wait for L_m,j = m+1
22.         calculate_block_auto(D,m,c,j)
23.     end for
24.   end for
25.   for b = 0 to c–1 do
26.       wait for L_b,c = b+1
27.       calculate_block_auto(D,b,c,c)
28.   end for
29.   calculate_block_auto(D,c,c,c)
30.   notify set L_c,c = c+1
31.   master_thread_routine(c)
32. end function
```

б

```
1.  function complement_thread_routine(c)
2.    switch thread first(c)     //passiveB
3.    for r = c+1 to M–1 do
4.      wait for L_r,r = r+1
5.      calculate_block_auto(D,r,c,r)
6.      switch thread first(c)  //passiveC
7.      for x = 0 to r–1 do
8.          wait for L_r,x = r+1
9.          calculate_block_auto(D,r,c,x)
10.     end for
11.     for y = r+1 to M–1 do
12.         wait for L_r,y = r+1
13.         calculate_block_auto(D,r,c,y)
14.     end for
15.     switch thread first(c)  //passiveC
16.   end for
17. end function
```

в

```
1.  function passive_thread_routine_type_A(c)
2.    var s = first(c)
3.    forj = 0 to prev(c) do
4.      for b = 0 to j do
5.        calculate_block_auto(D,b,c,j)
6.      end for
7.      if c ≠ last(c) then
8.        switch thread next(c) //passiveA
9.      else do
10.       switch thread s //master or slave
11.       if s = j then
12.         s = s+P
13.       endif
14.     end if
15.   end for
16.   slave_thread_routine(c)
17. end function
```

г

```
1.  function passive_thread_routine_type_B(c)
2.    switch thread next(c) //passiveA
3.    for j = c+1 to last(c) do
4.      for b = c+1 to j do
5.        calculate_block_auto(D,b,c,j)
6.      end for
7.      switch thread next(c)
        /* passiveB or slave or master */
8.    end for
9.    passive_thread_routine_type_C(c)
10. end function
```

д

```
1.  function complement_thread_routine(c)
2.    switch thread first(c) //passiveB
3.    for r = c+1 to M–1 do
4.      wait for L_r,r = r+1
5.      calculate_block_auto(D,r,c,r)
6.      switch thread first(c) //passiveC
7.      for x = 0 to r–1 do
8.          wait for L_r,x = r+1
9.          calculate_block_auto(D,r,c,x)
10.     end for
11.     for y = r+1 to M–1 do
12.         wait for L_r,y = r+1
13.         calculate_block_auto(D,r,c,y)
14.     end for
15.     switch thread first(c) //passiveC
16.   end for
17. end function
```

е

Рис. 5. Шесть режимов алгоритма работы потока: *а* – ведущий; *б* – ведомый; *в* – дополняющий; *г* – пассивный типа А; *д* – пассивный типа Б; *е* – пассивный типа С

Сравним между собой последовательный и два параллельных алгоритма: БФУ и ПБПА. Последовательный алгоритм рассчитывает блоки за $4^3 = 64$ единиц времени при загрузке одного процессора в 100%. Идеальный параллельный алгоритм мог бы рассчитать 64 блока на 2-х процессорах за 32 единицы времени. Однако БФУ (рис. 1) выполняет 4 итерации, на каждой из которых вычисляется 16 блоков, причем первый блок вычисляется последовательно с остальными 15 блоками, которые рассчитываются параллельно-последовательно. Это требует 9 единиц времени на одной итерации и 36 единиц на 4-х итерациях. Загрузка оборудования у БФУ составляет 88,9 %. Алгоритм ПБПА затратил 33 единицы времени (на 9,1 % меньше чем БФУ), а загрузка оборудования оказалась высокой и составила 97 %. Это говорит о значительном преимуществе ПБПА над БФУ в отношении организации параллелизма.

Для локализации обращений к данным принципиально важным является длительность $\tau_{i,j}$ интервала времени, на протяжении которого поток или процессор обращается к одному блоку $D^l_{i,j}$, рассчитываемому на разных уровнях $l = 1,\ldots,M$. В алгоритме БФУ длительность интервала $\tau_{i,j} \geq M \times M \times (M-1)/P$ для каждого из блоков, в частности $\tau_{i,j} \geq 24$ для матрицы 4×4 и 2-х процессоров. Алгоритм ПБПА дает меньший интервал для многих блоков. Например, вычисление блока $D^l_{2,3}$ на четырех уровнях выполняется на интервале времени 15–25, при этом $\tau_{2,3} = 11$. На производительность ПБПА влияет число переключений между потоками по switch thread. Частые переключения могут замедлить работу алгоритма. Так для матрицы блоков 4×4 было совершено 22 переключения по *switch thread* при общем числе вычислений блоков 64, что дает число переключений потоков на один вычисляемый блок $\lambda = 0{,}34$. Это значение λ достаточно большое, однако его отрицательное влияние на производительность ПБПА нивелируется большим размером B блока, расчет которого требует намного большего процессорного времени по сравнению со временем реализации операции *switch thread*. Большие размеры блоков характерны для графов с большим числом N вершин при малом числе M блоков в

строке матрицы *D*. Проведенные эксперименты показали, что с увеличением значения *M* значение λ быстро уменьшается, соответственно уменьшается отрицательное влияние переключений потоков по *switch thread* на производительность ПБПА. Так, при использовании 4-х процессоров значение λ = 0,1146 для матрицы блоков 12×12. Оно уменьшается до λ = 0,0238 для матрицы 48×48 и становится несущественным λ = 0,0054 для матрицы 192×192.

Результаты и их обсуждение

Реализация БФУ выполнена на базе OpenMP [4] для организации высокоэффективного позадачного параллелизма (task-based). Реализация ПБПА выполнена на базе разработанных в [7] средств для управления, синхронизации и организации передачи управления между кооперативно выполняющимися потоками. Эксперименты проведены на 4-ядерной системе, построенной на базе процессора Intel(R) Core(TM) i5-3450 CPU с частотой 3,10 GHz, архитектура IvyBridge. Многоуровневая кэш память включает локальный уровень *L*1 емкостью 256 KB, локальный уровень *L*2 емкостью 1,0 MB и разделяемый уровень *L*3 емкостью 6,0 MB. Исполняемый код сгенерирован компилятором Intel Compiler 18, настроенным на оптимизацию под архитектуру IvyBridge и применение векторизации Intel AVX (Advanced Vectorization Extensions). Установлен максимальный уровень оптимизации программного кода посредством включения высокоуровневой оптимизации от Intel – OV3 – High Level Optimization, при этом все опции настроены на максимальную производительность исследуемых программ.

Проведено экспериментальное исследование и сравнение двух алгоритмов: БФУ и ПБПА. В качестве входных данных использованы автоматически сгенерированные полные взвешенные графы размером 4800, 9600 и 14400 вершин. Достоинство такого решения состоит в том, что полные графы обеспечивают измерение параметров алгоритмов в условиях высокой нагрузки. Исходная матрица весов разделялась на блоки размером: 25×25, 50×50, 100×100, 120×120, 150×150, 200×200, 300×300 и 600×600 вершин.

Рис. 6 показывает характер зависимости времени выполнения от размера блока, который практически идентичен для обоих алгоритмов БФУ и ПБПА. Обе кривые имеют по два локальных минимума и одному локальному максимуму. Первый минимум связан с кэш *L*1, второй минимум – с кэш *L*2. Суть связи состоит в том, что как в БФУ, так и в ПБПА требуется для вычисления одного блока присутствие до трех исходных блоков в быстрой памяти верхнего уровня. При размере одного блока 120×120, который для типа long C++ размерностью 4 байта занимает 56 KB в кэш памяти, три блока занимают 169 KB. Этот размер приближается к емкости *L*1 в 256 KB, но не превышает ее, что гарантирует отсутствие активизации обмена данными между *L*1 и *L*2 в процессе вычисления целевого блока и обусловливает наличие первого локального минимума. В этой точке БФУ израсходовал 6,784 с, а ПБПА – 5,606 с процессорного времени. При размере одного блока 250×250, который занимает 244 KB кэш памяти, три блока занимают 733 KB. Этот размер приближается к емкости *L*2 в 1 MB, но не превышает ее, что гарантирует отсутствие активизации обмена данными между *L*2 и *L*3 и обусловливает появление второго локального минимума. При близком размере блока 200×200, БФУ израсходовал 7,028 с, а ПБПА израсходовал 6,301 с процессорного времени. Локальный максимум, расположенный между двумя минимумами, обусловлен тем, что, начиная с некоторого размера блока, *L*1 не вмещает одновременно три блока, при этом интенсивность обмена между *L*1 и *L*2 начинает возрастать, а время выполнения БФУ и ПБПА увеличивается до 7,503 с и 6,900 с соответственно. Начиная с размера блока 150×150, и до размера 250×250 решающим фактором является способность *L*2 вместить три блока, при этом время выполнения алгоритмов снова падает. Возрастание времени для малых и для больших размеров блоков (края графиков) обусловлено разными причинами. В первом случае возрастающее время тратится на учащающиеся переходы между потоками и переходы между вычислениями разных блоков. Во втором случае время тратится на учащающуюся доставку данных из *L*3 в *L*2 и далее в *L*1 из-за неспособности *L*1 и *L*2 разместить три блока.

Превосходство предлагаемого ПБПА над известным БФУ убедительно показывает рис. 7. ПБПА сокращает время выполнения по сравнению с БФУ на всех трех размерах графов (4800, 9600 и 14400 вершин) и на всех восьми размерах блока. Выигрыш составил от 0,99 % до 21,80 %. Главным фактором успеха явилась способность ПБПА загружать ядра процессора

близко к 100 %. Это подтверждается графиками загрузки четырех ядер, показанными на рис. 8. Другим фактором явилась локализация использования данных ядрами процессора и сокращение обращений к разделяемой памяти третьего уровня *L*3 благодаря возможности изменения порядка расчета блоков алгоритмом ПБПА. Следует отметить, что при размере блока меньше 300×300 алгоритм ПБПА дает почти одно и то же ускорение при разных размерах графа. Это позволяет использовать графы небольшого размера для нахождения наилучшего размера блока для каждой конкретной архитектуры вычислительной системы.

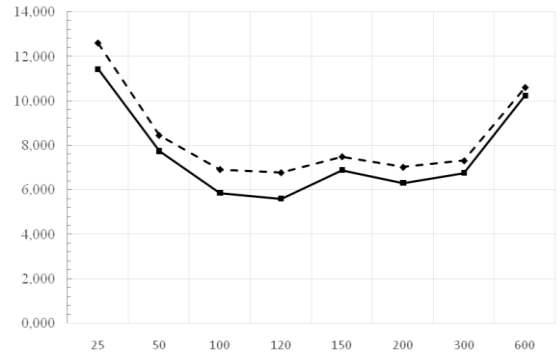


Рис. 6. Зависимость времени работы БФУ (пунктирная) и ПБПА (сплошная) от размера блока для графа 4800

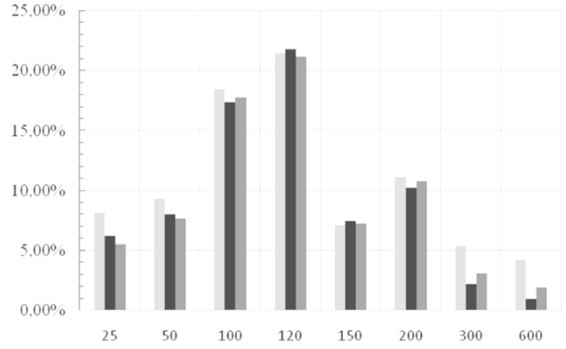
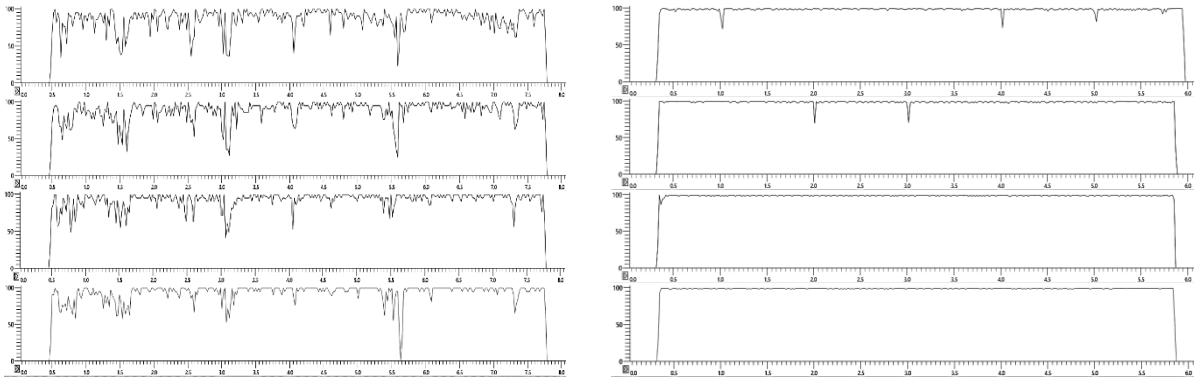


Рис. 7. Сокращение времени работы ПБПА по сравнению с БФУ в зависимости от размера блока



а *б*

Рис. 8. Графики загруженности четырех ядер процессора алгоритмами: *а* – БФУ; *б* – ПБПА для графа из 4800 вершин

Заключение

Предложен быстрый потоковый блочно-параллельный алгоритм поиска кратчайших путей между всеми парами вершин графа. Он локализует данные и вычисления внутри потоков и ядер и изменяет порядок вычисления блоков по сравнению с известным блочным алгоритмом Флойда-Уоршелла. Алгоритм увеличивает загрузку ядер, сокращает обмен данными между кэш памятью ядер и между уровнями иерархической памяти, уменьшает время поиска кратчайших путей. Он реализован на базе кооперативной модели и усовершенствованного планировщика выполнения потоков. Потоки одного процессора взаимодействуют посредством операций передачи управления друг другу согласно заранее разработанной схеме, потоки разных процессоров синхронизируются с минимальными слотами ожидания данных.

Список литературы

1. Floyd R.W. Algorithm 97: Shortest path // Communications of the ACM. 1962. № 5(6). P. 345.
2. Venkataraman G., Sahni S., Mukhopadhyaya S. A Blocked All-Pairs Shortest Paths Algorithm // Journal of Experimental Algorithmics (JEA). 2003. Vol. 8. P. 857–874.
3. Park J.S., Penner M., Prasanna V.K. Optimizing graph algorithms for improved cache performance // IEEE Trans. on Parallel and Distributed Systems. 2004. № 15 (9). P. 769–782.
4. Albalawi E., Thulasiraman P., Thulasiram R. Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0 // 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013). Los Angeles, CA, July 1–2, 2013. P. 109–112.

5. Прихожий А.А., Карасик О.Н. Разнородный блочный алгоритм поиска кратчайших путей между всеми парами вершин графа // Системный анализ и прикладная информатика. 2017. № 3. С. 68–75.
6. Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs / A. Prihozhy [et al.] // IEEE Trans. on CAD of Integrated Circuits and Systems. 2015. Vol. 34, No. 10. P. 1613–1626.
7. Прихожий А.А., Карасик О.Н. Кооперативная модель оптимизации выполнения потоков на многоядерной системе // Системный анализ и прикладная информатика. 2014. № 4. С. 13–20.

References

1. Floyd R.W. Algorithm 97: Shortest path // Communications of the ACM. 1962. № 5(6). P. 345.
2. Venkataraman G., Sahni S., Mukhopadhyaya S. A Blocked All-Pairs Shortest Paths Algorithm // Journal of Experimental Algorithmics (JEA). 2003. Vol. 8. P. 857–874.
3. Park J.S., Penner M., Prasanna V.K. Optimizing graph algorithms for improved cache performance // IEEE Trans. on Parallel and Distributed Systems. 2004. № 15 (9). P. 769–782.
4. Albalawi E., Thulasiraman P., Thulasiram R. Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0 // 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013). Los Angeles, CA, July 1–2, 2013. P. 109–112.
5. Prihozhij A.A., Karasik O.N. Raznorodnyj blochnyj algoritm poiska kratchajshih putej mezhdu vsemi parami vershin grafa // Sistemnyj analiz i prikladnaja informatika. 2017. № 3. S. 68–75. (in Russ.)
6. Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs / A. Prihozhy [et al.] // IEEE Trans. on CAD of Integrated Circuits and Systems. 2015. Vol. 34, No. 10. P. 1613–1626.
7. Prihozhij A.A., Karasik O.N. Kooperativnaja model' optimizacii vypolnenija potokov na mnogojadernoj sisteme // Sistemnyj analiz i prikladnaja informatika. 2014. № 4. S. 13–20. (in Russ.)

Сведения об авторах

Прихожий А.А., д.т.н., профессор, профессор кафедры программного обеспечения информационных технологий и автоматизированных систем Белорусского национального технического университета.

Карасик О.Н., аспирант кафедры программного обеспечения вычислительной техники и автоматизированных систем Белорусского национального технического университета.

Адрес для корреспонденции

220013, Республика Беларусь,
г. Минск, пр. Независимости, 65,
Белорусский национальный
технический университет
тел. +375-44-765-94-86;
e-mail: prihozhy@yahoo.com
Прихожий Анатолий Алексеевич

Information about the authors

Prihozhy A.A., D.Sci, professor, professor of software information technologies and automated systems department of Belarusian national technical university.

Karasik O.N., PG student of software of computer facilities and the automated systems department of Belarusian national technical university.

Address for correspondence

220013, Republic of Belarus,
Minsk, Nezavisimosti ave., 65,
Belarusian national
technical university
tel. + 375-44-765-94-86;
e-mail: prihozhy@yahoo.com
Prihozhy Anatoly Alekseevich