УДК 004.272.2 (075.8)

А. А. ПРИХОЖИЙ, О. Н. КАРАСИК

РАЗНОРОДНЫЙ БЛОЧНЫЙ АЛГОРИТМ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ МЕЖДУ ВСЕМИ ПАРАМИ ВЕРШИН ГРАФА

Белорусский национальный технический университет

Рассматривается проблема поиска кратчайших путей между всеми парами вершин взвешенного ориентированного графа. Известны алгоритмы Дейкстры и Флойда-Уоршелла, однородные блочные и параллельные алгоритмы и другие алгоритмы решения этой проблемы. Предлагается новый разнородный блочный алгоритм, рассматривающий различные типы блоков и учитывающий разделяемую иерархическую организации памяти и многоядерность процессоров при вычислении блока каждого типа. На теоретическом и экспериментальном уровнях проводится сравнение предлагаемых разнородных алгоритмов вычисления блоков с общепринятым однородным универсальным алгоритмом пересчета блока. Основной акцент делается на использовании вариантов неоднородности, взаимодействия блоков во время вычислений и вариаций в размере блока, размере матрицы блоков и общего количества блоков с целью выявления возможности сокращения объема вычислений, производимых при расчете блока, сокращения активности работы с кэш памятью процессора и выявления влияния времени расчета каждого типа блока на общее время выполнения разнородного блочного алгоритма. Предложен рекуррентный ресинхронизированный алгоритм расчета диагонального блока (D0), улучшающий использование кэш памяти процессора и сокращающий количество итераций и размер данных, необходимых для расчёта диагонального блока до 3 раз, что дает ускорение в расчете диагонального блока до 60%. Для более эффективной работы с кэш памятью предложены варианты перестановки основных циклов k-i-j алгоритмов расчета блоков креста (C1, C2) и обновляемых блоков (U3), использование которых в комбинации с алгоритмом расчета диагонального блока сокращает общее время работы разнородного блочного алгоритма на 13% в среднем по сравнению с однородным блочным алгоритмом.

Ключевые слова: *алгоритм Флойда-Уоршелла, кратчайший путь, разнородный блочный алгоритм, многоядерная система, разделяемая кэш память.*

Введение

Задача поиска кратчайших путей на взвешенном графе [1–10] формулируется в различных постановках: для ориентированного или неориентированного, разреженного или плотного графа, со взвешенными ребрами и/или взвешенными вершинами, положительными или возможно отрицательными весами, между парой вершин или всеми парами вершин, при обязательном проходе всех вершин (задача коммивояжера) или необязательном проходе и т. д. Вычислительная сложность различных постановок задач различная: поиск кратчайшего пути между парой вершин решается за квадратичное время алгоритмом Дейкстры, поиск кратчайших путей между всеми парами вершин решается за кубическое время алгоритмом Флойда-Уоршелла, поиск кратчайшего пути при обязательном проходе всех вершин полного графа является NP-трудной задачей.

Задача поиска кратчайших путей на графе находит практическое применение при решении многих задач в микроэлектронике, компьютерных сетях и программировании, компьютерных играх, транспортной отрасли и многих других. Размер графов может достигать таких больших размеров, что даже для решения задачи поиска алгоритмами полиномиальной степени сложности может потребоваться нереально большое процессорное время на современной вычислительной технике. Так алгоритм Флойда-Уоршелла, который совершенствуется и развивается в настоящей статье, затрачивает время на поиск, пропорциональное величине $1.25{\times}10^{11}$для графа на 5000 вершинах, и практически мало приемлем уже для такого размера графа.

Быстродействие алгоритмов поиска кратчайших путей сильно зависит от вычислительной платформы, на которой они реализуются. Учет архитектурных особенностей вычислительных систем может значительно повысить производительность алгоритмов. К важнейшим особенностям систем относится многоядерность процессоров, разделяемая кэш память, разнородность на программном и аппаратном уровне.

Алгоритм Флойда-Уоршелла

Пусть ориентированный взвешенный граф $G = (V, E)$ с множеством V из N вершин и множеством ребер E представлен матрицей W положительных весов ребер. Петли и параллельные ребра отсутствуют, при этом $w_{ii} = 0$ при $i = 0...N–1$ и $w_{ij} = \infty$ при $(i, j) \notin E$. Длины кратчайших путей между парами вершин опишем матрицей D. Алгоритм Флойда-Уоршелла [1, 2] пересчитывает матрицу D на шагах $0...k...N–1$, при этом образуется последовательность матриц $D(0)...D(k)...D(N–1)$, в которой $D(0) = W$, $D(k)$ – матрица кратчайших расстояний после нахождения кратчайших путей, проходящих через вершину k, и $D(N–1)$ – результирующая матрица кратчайших расстояний. Переход от шага $k–1$ к шагу k иллюстрируется на рис. 1, при этом кратчайший путь $D_{ij}(k)$ от вершины i до вершины j пересчитывается через матрицу $D(k–1)$ и k-ые строку и столбец матрицы $D(k)$:

$$D_{ij}(k) = \min\left\{D_{ij}(k-1),\ D_{ik}(k) + D_{kj}(k)\right\} \quad (1)$$

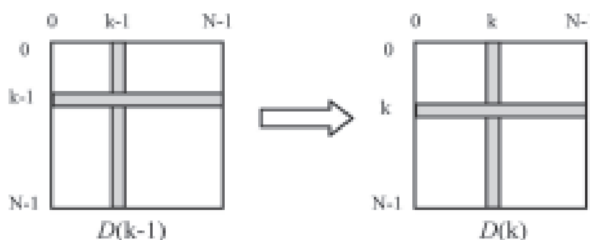

Рис. 1. Иллюстрация работы алгоритма Флойда-Уоршелла

```
Algorithm Floyd_Warshall {
    D = W;
    for k=0…N-1 {
        for i=0…N-1 {
            for j=0…N-1 {
                s = D_ik + D_kj;
                if(D_ij > s) D_ij = s;
            }
        }
}}
```

Рис. 2. Псевдокод алгоритма Флойда-Уоршелла

С увеличением k строка и столбец смещаются по матрице $D(k)$ сверху вниз слева направо. Псевдокод алгоритма Флойда-Уоршелла (рис. 2) состоит из трех циклов по i, j и k. На всех шагах он работает с матрицей одинаковой размерности $N \times N$. Алгоритм имеет высокую однородность, а его вычислительная сложность равна $O(N^3)$. Заметим, что перестановка цикла по k с циклами по i и j приводит к неверным результатам.

Однородный блочный алгоритм поиска кратчайших путей на графе

Построение блочного алгоритма поиска кратчайших путей на графе помогает решить две важнейшие проблемы: 1) локализовать работу с многоуровневой памятью и тем самым ускорить выполнение операций записи и чтения из памяти; 2) организовать параллельную многопоточную работу блоков на многоядерной системе. В работах [3, 4] алгоритм Флойда-Уоршелла расширен до блочного алгоритма поиска кратчайших путей на графе, а в работах [5, 6] показаны возможности распараллеливания этого блочного алгоритма. Матрица D размерностью $N \times N$ разбивается на блоки размерностью $B \times B$, при этом образуется матрица блоков размерностью $M \times M$, где $M = N/B$. Псевдокод блочного алгоритма показан на рис. 3. Функционирование блочного алгоритма *Blocked_FW* представляется циклом по блокам $m = 0...M–1$, на каждой итерации которого выполняется упорядоченный однократный пе-

```
Algorithm Blocked_FW {
    for m=0…M-1 {
        CalBlock (B_m,m,B_m,m,B_m,m);        // type D0
        for i=0…m-1 {
            CalBlock (B_i,m,B_i,m,B_m,m);    // type C1
            CalBlock (B_m,i,B_m,m,B_m,i);    // type C2
        };
        for i=m+1…M-1 {
            CalBlock (B_i,m,B_i,m,B_m,m);    // type C1
            CalBlock (B_m,i,B_m,m,B_m,i);    // type C2
        };
        for i=0…m-1 {
            for j=0…m-1
                CalBlock (B_i,j,B_i,m,B_m,j);   // type U3
            for j=m+1…M-1
                CalBlock (B_i,j,B_i,m,B_m,j);   // type U3
        };
        for i=m+1…M-1 {
            for j=0…m-1
                CalBlock (B_i,j,B_i,m,B_m,j);   // type U3
            for j=m+1…M-1
                CalBlock (B_i,j,B_i,m,B_m,j);   // type U3
        };
}}
```

Рис. 3. Псевдокод блочного алгоритма

```
Algorithm CalBlock(B1, B2, B3) {
    for k=0…B-1 {
        b3rk = row(B3,k);
        for i=0…B-1 {
            b1ri = row(B1,i);
            b2 = B2ik;
            for j=0…B-1 {
                bij= b2 + b3rkj;
                if (b1rij > bij) { b1rij = bij;}
            }
}}}
```

Рис. 4. Алгоритм вычисления блока

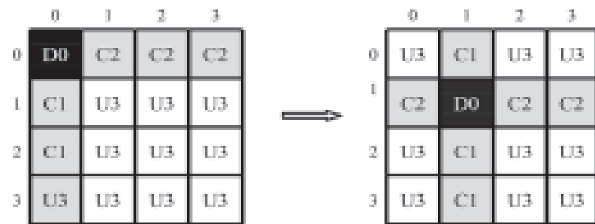

Рис. 5. Иллюстрация процесса вычисления блоков блочным алгоритмом

ресчет всех блоков одним и тем же универсальным однородным алгоритмом *CalBlock*, представленным на рис. 4, где аргумент B^1 – вычисляемый блок, а аргументы B^2 и B^3 – блоки, через которые осуществляется вычисление. На M итерациях цикла каждый блок пересчитывается M раз. Алгоритм блока имеет вычислительную сложность $O(B^3)$, которая достаточно быстро растет с увеличением размера блока. Заметим, алгоритм *CalBlock* из-за универсальности трудно поддается преобразованию и модификации.

Процесс работы блочного алгоритма иллюстрируется рис. 5. Тело главного цикла алгоритма можно разбить на три последовательно реализуемых шага. Первый шаг рассчитывает диагональный (D0 – Diagonal) блок $B_{m,m}$. Второй шаг рассчитывает через $B_{m,m}$ блоки, лежащие на кресте (C1 и C2 – Cross), образованном строкой m и столбцом m матрицы блоков с пересечением на блоке $B_{m,m}$. Третий шаг обновляет (U3 – Update) остальные блоки посредством блоков креста. При увеличении m крест перемещается из левого верхнего в правый нижний угол матрицы блоков. На одной итерации цикла по m вычисляется один диагональный блок D, вычисляется $2×M–1$ блоков креста C и вычисляется $M×(M–2)$ обновляемых блоков U. Всего выполняется M^3 пересчетов блоков.

Анализ последовательно-параллельного выполнения блоков показывает, что блок типа D0 работает последовательно с остальными блоками, все блоки типа C1 и C2 могут работать взаимно параллельно, все блоки типа U3 могут работать также взаимно параллельно, но последовательно с блоками C1 и C2.

Разнородный блочный алгоритм

Внимательный анализ всех вызовов алгоритма *CalBlock* вычисления блока показывает, что в них различаются четыре профиля аргументов B^1, B^2 и B^3. Профили аргументов тесно взаимосвязаны со спецификой работы самого алгоритма. Тип 0 профиля, когда $B^1 = B^2 = B^3$, встречается только для диагонального блока D0 (рис. 3). Тип 1 профиля, когда $B^1 = B^2 \neq B^3$, встречается для блоков C1 столбца креста. Тип 2 профиля, когда $B^1 = B^3 \neq B^2$, встречается для блоков C2 строки креста. Тип 3 профиля, когда $B^1 \neq B^2 \neq B^3$, встречается для остальных блоков U3. Идентичность или неидентичность аргументов алгоритма вычисления блока и особенности его поведения в каждом случае можно с пользой использовать для поиска методов сокращения вычислительных ресурсов, потребляемых во время выполнения алгоритма. В частности, целью является сокращение активности работы с кэшем, а также сокращение времени вычисления блока. Новые более эффективные алгоритмы вычисления блоков построим путем формальных преобразований исходного алгоритма с учетом особенностей вычисления блока каждого типа.

Новый алгоритм вычисления диагонального блока

В однородном блочном алгоритме все блоки вычисляются на основе классического алгоритма Флойда-Уоршелла (рис. 4) независимо от типа блока. Главный принцип заключается в последовательном рассмотрении вершин k от 0 до $B–1$, ассоциируемых с блоками B^2 и B^3, с целью сокращения расстояния от вершины i до вершины j, находящегося в блоке B^1. При вычислении диагонального блока типа $D0$ блоки B^2 и B^3идентичны блоку B^1. Изменим главный принцип алгоритма Флойда-Уоршелла. Последовательность вершин k от 0 до $B–1$ будем ассоциировать с процессом пошагового добавления очередной вершины к графу G'. В результате образуется последовательность

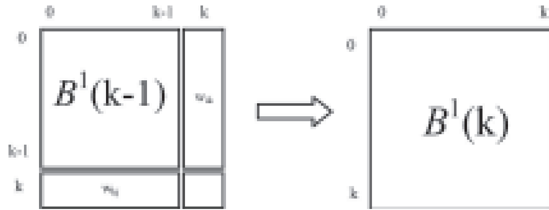

Рис. 6. Рекуррентная процедура вычисления диагонального блока

графов $G'(0)$, $G'(1)\dots G'(k)\dots G'(B\text{–}1)$. Матрицу расстояний между парами вершин в графе $G'(k)$ обозначим $B^1(k)$. Представим алгоритм рекуррентной процедурой, вычисляющей матрицу $B^1(k)$ по матрице $B^1(k\text{–}1)$ и весам w_{ik} и w_{kj} ребер, соединяющих добавляемую вершину k с вершинами $i, j \in \{1, ..., k\text{-}1\}$ (рис. 6). Сначала процедура вычисляет длины $B^1{}_{\mathrm{ik}}(k)$ в столбце k по формуле

$$B^1{}_{ik}(k) = \min_{j=1\dots k-1}\left(B^1{}_{ij}(k-1) + w_{jk}\right). \quad (2)$$

Длины $B^1_{kj}(k)$ в строке k вычисляются по аналогичной формуле. Затем процедура рассчитывает длины $B^1{}_{\mathrm{ij}}(k)$ по длинам $B^1{}_{\mathrm{ij}}(k\text{-}1)$ при $i, j = 0, ..., k\text{–}1$, используя формулу

$$B^1{}_{ij}(k) = \min\left\{B^1{}_{ij}(k-1),\, B^1{}_{ik}(k) + B^1{}_{kj}(k)\right\}. \quad (3)$$

Рекуррентная процедура циклически выполняется для всех вершин графа G'. На каждой ее итерации поочередно выполняются две операции: добавление A (Adding) новой строки и столбца для вершины k; обновление U (Updating) матрицы $B^1(k\text{–}1)$ до матрицы $B^1(k)$. Выполнение цикла по k порождает последовательность выполнения пар операций A и U:

$$A_0U_0 – A_1U_1 – A_2U_2 – \dots – A_{N–1}U_{N–1}. \quad (4)$$

Псевдокод рекуррентной процедуры *CalBlock_D*0 показан на рис. 7. Значения кратчайших путей от вершин 1, ..., k–1 до вершины k накапливаются в векторе b^{r}, а от вершины k до вершин 1, ..., k–1 – в векторе b^{c}. В отличие от классического алгоритма (рис. 3), в котором число итераций наиболее вложенного из трех циклов по i, j и k равно 3^B, в модифицированном алгоритме количество итераций самого вложенного цикла и объем обрабатываемых данных сокращаются до трех раз из-за сокращения числа значений индексов i и j. Это является важным источником уменьшения вычислительной сложности нового алгоритма пересчета диагонального блока.

*Ресинхронизация рекуррентной процедуры***.** Анализ информационных зависимостей между операциями A_kU_k в (4) показывает, что их нельзя совместить во вложенных циклах по i и j. В то же время пара операций U_kA_{k+1} совместима в этих циклах, что предоставляет возможность ресинхронизации последовательности пар операций (4) до последовательности операций

$$U_0A_1 – U_1A_2 – \dots – U_{N–2}A_{N–1} – U_{N–1}. \quad (5)$$

В последовательности (5) операция A_0 удалена за счет присваивания нулевого значения матрице $B^1(0)$, которая описывает граф на одной вершине. Ресинхронизация позволяет повысить уровень параллелизма и сократить критические пути в теле вложенных циклов алгоритма. В паре U_kA_{k+1} операция U_k есть отложенное обновление матрицы $B^1(k)$, которое выполняется одновременно с добавлением вершины k+ 1.

Псевдокод ресинхронизированной рекуррентной процедуры *CalBlock_D*0*R* показан на рис. 8. Ее вызов заменяет первый вызов алгоритма *CalBlock* в алгоритме *Blocked_FW*, показанном на рис. 3. Чтобы выполнить операцию U_k, вычисленные для вершины k строка и столбец матрицы $B^1(k)$ сохраняются в векторах d^{r} и d^{c} соответственно. На первой итерации цикла по k значения элементов этих векторов равны ∞. Значения весов входных для вершины k ребер доступны через вектор w^r, заранее сгенерированный на предыдущей итерации цикла по k. Операция $row(B^1, k)$ возвращает строку k матрицы B^1.

Скалярная переменная $b^r_{\min}$ используется для формирования значения элемента вектора

```
Algorithm CalBlock_D0(B1) {
  for k=1…B-1  {
    for i=0…k-1  { bci = ∞; }
    for i=0…k-1  {    // Adding Ak
        brmin = ∞;
        for j=0… k-1  {
            s' = B1ij + B1jk;  if(brmin > s ) brmin = s';
            s'' = B1ji + B1kj;  if(bcj > s'') bcj = s'';
        }  brj = brmin;
    }
    for i=0…k-1   { B1ik = bri;  B1ki = bci; }
    for i=0…k-1  {    // Updating Uk
        for j=0… k-1  {
            z = B1ik + B1kj;  if(B1ij > z) B1ij = z;
        }
}}}
```

Рис. 7. Псевдокод рекуррентной процедуры

```
Algorithm CalBlock_D0R(B^1) {
  d^r_0 = ∞;   w^r_0 = B^1_01;
  for k=1…B-1  {
    d^c = row(B^1,k-1);   w^c = row(B^1,k);
    for i=0…k-1   { b^c_i = ∞; }
    for i=0…k-1  {
      b^r_min = ∞;
      for j=0… k-1  {          // Operations U_k A_k+1
        z = d^r_i + d^c_j;   if(B^1_ij > z) B^1_ij = z;
        s_0 = B^1_ij + w^r_j;   if(b^r_min > s_0) b^r_min = s_0;
        s_1 = B^1_ij + w^c_i;   if(b^c_j > s_1) b^c_j = s_1;
      }
      b^r_j = b^r_min;
    }
    for i=0…k-1   {
      B^1_ki = b^c_i;   B^1_ik = d^r_i = b^r_i;   w^r_j = B^1_i,k+1;
    }  w^r_k = B^1_k,k+1;
  }
  for i=0…B-2  {                  // Updating U_N-1
    for j=0… B-2  {
      z = d^r_i + d^c_j;   if(B^1_ij > z) B^1_ij = z;
}}}
```

Рис. 8. Новый ресинхронизированный алгоритм вычисления диагонального блока

b^r_j. Значение элемента b^c_j формируется в самом векторе b^c. Значение переменной $b^r_{\min}$ рассчитывается по элементу B^1_{ij} матрицы и весу ребра w_{jk}, находящемуся в векторе w^r. Значение переменной b^c_j рассчитывается по элементу B^1_{ji} матрицы и весу ребра w_{kj}, находящемуся в векторе w^c. Значения $b^r_{\min}$ и b^c_j рассчитывается на одной и той же итерации, следовательно, значение B^1_{ij} должно быть предварительно обновлено на этой же итерации. Сформированные векторы b^r и b^c записываются в матрицу B^1. Конечная одиночная операция U_{N-1} выполняется двумя дополнительными вложенными циклами после завершения цикла по k.

Преобразование алгоритмов вычисления блоков креста и обновляемых блоков

После того, как с алгоритма *CalBlock* снята реализация блока типа D0, этот алгоритм продолжает реализовывать блоки типа C1, C2 и U3. При этом *CalBlock* приобретает новые свойства, важнейшим из которых является появившаяся возможность произвольной перестановки циклов по переменным i, j и k. Всего возможно шесть вариантов перестановки, из которых перестановка k–i–j использована в *CalBlock*. Важнейшие оставшиеся варианты перестановки рассмотрим с точки зрения повышения производительности алгоритма.

Выбор варианта i–k–j и устранение идентичных блоков во входных аргументах дает три алгоритма вычисления блоков типа $C1$, $C2$ и U3 (рис. 9). В каждый из них, с целью ускорения вычислений, введены по три дополнительные векторные переменные и одной скалярной переменной. Достоинством алгоритмов является последовательная обработка соседних элементов векторов и отсутствие перескакивания на отдаленные участки памяти, что облегчает работу кэш.

Преобразование алгоритма *CalBlock* для порядка i–j–k переменных цикла дает три алгоритма вычисления блоков типа C1, C2 и U3 (рис. 10). В каждый из них введены две дополнительные векторные переменные и одна скалярная переменная $d_{\min}$. Векторные переменные ускоряют просмотр элементов строк матриц, а скалярная переменная может быть размещена на регистре и способна уменьшить число операций записи в матрицу B^1 (и соответственно в кэш) в B раз. Недостатком алгоритмов является перескакивание по отдаленным участкам памяти при чтении значений индексных переменных $B^3{}_{kj}$ и $B^1{}_{kj}$. Алгоритмы вычисления блоков эффективно реализуются средствами адресного программирования.

```
Algorithm CalBlock_C1(B^1,B^3) {
  for i=0…B-1  {
    b1^r = row(B^1,i);
    d1^r = row(B^1,i);
    for k=0… B-1  {
      b1 = b1^r_k;
      b3^r = row(B^3,k);
      for j=0…B-1  {
        s = b1 + b3^r_j;
        if(d1^r_j > s) d1^r_j = s;
      }
}}}
```

а

```
Algorithm CalBlock_C2(B^1,B^2) {
  for i=0…B-1  {
    b2^r = row(B^2,i);
    d1^r = row(B^1,i);
    for k=0… B-1  {
      b2 = b2^r_k;
      b1^r = row(B^1,k);
      for j=0…B-1  {
        s = b2 + b1^r_j;
        if(d1^r_j > s) d1^r_j = s;
      }
}}}
```

б

```
Algorithm CalBlock_U3(B^1,B^2,B^3) {
  for i=0…B-1  {
    b2^r = row(B^2,i);
    d1^r = row(B^1,i);
    for k=0… B-1  {
      b2 = b2^r_k;
      b3^r = row(B^3,k);
      for j=0…B-1  {
        s = b2 + b3^r_j;
        if(d1^r_j > s) d1^r_j = s;
      }
}}}
```

в

Рис. 9. Преобразованные по варианту i–k–j алгоритмы остальных блоков: *а* – тип C1, *б* – тип C2, *в* – тип U3

```
Algorithm CalBlock_C1(B^1,B^3) {
  for i=0…B-1 {
    b1^r = row(B^1,i);
    d1^r = row(B^1,i);
    for j=0…B-1 {
      d_min = ∞;
      for k=0… B-1 {
        s = b1^r_k + B^3_kj;
        if(d_min > s) d_min = s;
      }
      d1^r_j = d_min;
}}
```

а

```
Algorithm CalBlock_C2(B^1,B^2) {
  for i=0…B-1 {
    b2^r = row(B^2,i);
    d1^r = row(B^1,i);
    for j=0…B-1 {
      d_min = ∞;
      for k=0… B-1 {
        s = b2^r_k + B^1_kj;
        if(d_min > s) d_min = s;
      }
      d1^r_j = d_min;
}}
```

б

```
Algorithm CalBlock_U3(B^1,B^2,B^3) {
  for i=0…B-1 {
    b2^r = row(B^2,i);
    d1^r = row(B^1,i);
    for j=0…B-1 {
      d_min = ∞;
      for k=0… B-1 {
        s = b2^r_k + B^3_kj;
        if(d_min > s) d_min = s;
      }
      if(d1^r_j > d_min) d1^r_j = d_min;
}}
```

в

Рис. 10. Алгоритмы блоков, преобразованные по варианту *i–j–k*: *а* – тип C1, *б* – тип C2, *в* – тип U3

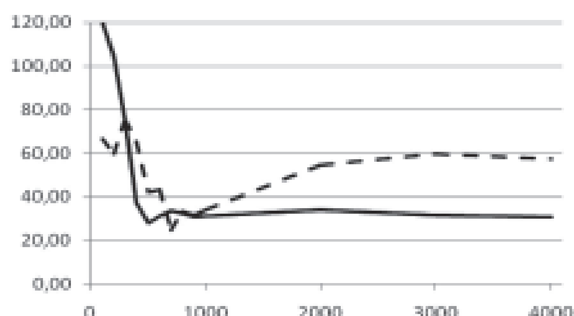

Рис. 11. Ускорение в% нового алгоритма диагонального блока по сравнению с алгоритмом Флойда-Уоршелла в зависимости от размера блока на двух архитектурах процессора: *cpu1* (сплошная), *cpu2* (пунктирная)

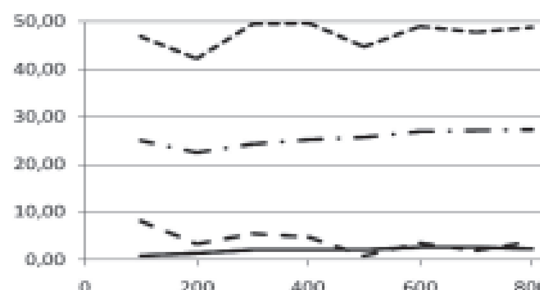

Рис. 12. Ускорение в % алгоритма разнородного блока по сравнению с алгоритмом однородного блока в зависимости от размера *B* блока при *M* = 4 и *N*= 4**B*: блоки всех типов на *cpu*1 (сплошная), блоки всех типов на *cpu*2 (пунктирная), диагональный блок на *cpu*1 (штрих-пунктирная), диагональный блок на *cpu*2 (точечная)

Результаты вычислительных экспериментов

Целью экспериментов явилось выявление зависимости времени работы всего разнородного блочного алгоритма и алгоритмов вычисления блоков четырех типов от размера графа, размера блока, числа блоков, а также сравнение нового разнородного блочного алгоритма с известным однородным блочным алгоритмом. Использованы полные графы со случайными весами на ребрах, для которых задача о кратчайших путях наиболее трудоемка. Эксперименты проведены на многоядерных процессорах Intel® Core™ i3 CPU 550 @ 3.20 GHz 3.19 GHz (в дальнейшем *cpu*1) и Intel(R) Core(TM) i5-6200UCPU @ 2.20 GHz (в дальнейшем *cpu*2).

Наилучшие результаты дал новый ресинхронизированный алгоритм *CalBlock_D*0*R* вычисления диагонального блока. Рис. 11 показывает его ускорение по сравнению с алгоритмом *CalBlock* Флойда-Уоршелла. При увеличении размера графа до 4000 вершин ускорение стабильно составляет чуть больше 30% на *cpu*1 и до 60% на *cpu*2. Благодаря такому значительному ускорению разнородный блочный алгоритм выгоднее использовать при небольшом числе блоков, так как число диагональных блоков равно *M* при общем числе вычисляемых блоков M^3. Так при *M*= 2 число диагональных блоков составляет 25%, при *M* = 4–6.25%, при *M* = 8 – только 1.56%.

Рис. 12 сравнивает неоднородный блочный алгоритм с однородным блочным алгоритмом при одновременном росте размера блока со 100 до 800 и росте размера графа с 400 до 3200 вершин, при этом размер матрицы блоков постоянен и составляет 4×4. Новый алгоритм вычисления диагонального блока показал значительное ускорение: на *cpu*1 оно в среднем выросло на 25.5%, на *cpu*2 – 47.29%. С учетом блоков всех четырех типов разнородный блочный алгоритм дал значительно меньшее ускорение, которое составило от 0.84% до 2.63% на *cpu*1 и составило от 0.74% до 8.05% на *cpu*2. Это свидетельствует о том, что алгоритмы *CalBlock_C*1, *CalBlock_C*2 и *CalBlock_U*3, изображенные на рис. 9, дают ускорение на

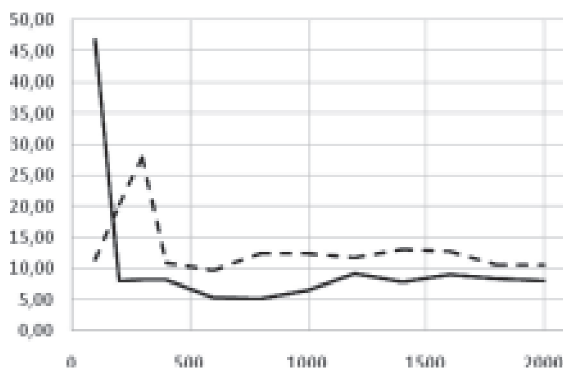

Рис. 13. Ускорение в % разнородного блочного алгоритма по сравнению с однородным блочным алгоритмом в зависимости от размера блока $B = 100–2000$ при $M = 2$ и $N = 2*B$ на *cpu*1 (сплошная) и *cpu*2 (пунктирная)

блоках типа *C*1, *C*2 и U3 значительно меньшее, чем *CalBlock_D*0*R* по сравнению с *CalBlock*. При этом отмеченные выше свойства алгоритмов *CalBlock_C*1, *CalBlock_C*2 и *CalBlock_U*3 составляют потенциал для дальнейшего повышения производительности разнородного блочного алгоритма. Действительно, уменьшение матрицы блоков с 4×4 до 2×2 дает ускорение нового разнородного блочного алгоритма по сравнению с однородным блочным алгоритмом в среднем на 10.88% на *cpu*1 и на 13.67% на *cpu*2 (рис. 13).

Заключение

Алгоритм Флойда-Уоршелла и построенный на его основе блочный алгоритм поиска кратчайших путей между всеми парами вершин взвешенного графа отличаются однородностью структуры и универсальностью построения базовых компонентов. В статье предложен неоднородный блочный алгоритм, различающий четыре типа блоков матрицы кратчайших путей и четыре отдельных более эффективных алгоритма для вычисления блоков каждого типа. Алгоритм вычисления диагонального блока дал наибольший прирост производительности (до 60%), а производительность всего разнородного блочного алгоритма при выполнении на одном процессоре с кэш памятью повысилась на 13.67% в среднем.

References

1. **Floyd, R. W.** Algorithm 97: Shortest path / R. W. Floyd // Communications of the ACM, 1962, 5(6), p. 345.
2. **Hofner, P. Dijkstra,** Floyd and Warshall Meet Kleene/ P. Hofner and B. Moller // Formal Aspect of Computing, Vol.24, No.4, 2012, № 2, pp. 459–476.
3. **Venkataraman, G. A** Blocked All-Pairs Shortest Paths Algorithm / G. Venkataraman, S. Sahni, S. Mukhopadhyaya // Journal of Experimental Algorithmics (JEA), Vol 8, 2003, pp. 857–874
4. **Park, J. S.** Optimizing graph algorithms for improved cache performance / J. S. Park, M. Penner, and V. K. Prasanna // IEEE Trans. on Parallel and Distributed Systems, 2004, 15(9), pp. 769–782.
5. **Albalawi, E.** Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0 / E. Albalawi, P. Thulasiraman, R. Thulasiram // 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013), 2013, Los Angeles, CA, July 1–2, 2013, pp. 109–112.
6. **Tang, P.** Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-Core Computers / P. Tang // IEEE SOUTHEASTCON 2014, pp. 1–7.
7. **Solomonik, E.** Minimizing Communication in All Pairs Shortest Paths / E. Solomonik, A. Buluc, and J. Demmel // IEEE 27th International Symposium on Parallel & Distributed Processing, 2013, pp. 548–559.
8. **Singh, A.** Performance Analysis of Floyd Warshall Algorithm vs Rectangular Algorithm / A. Singh, P. K. Mishra // International Journal of Computer Applications, Vol.107, No.16, 2014, pp. 23–27.
9. **Madduri, K.** An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances / K Madduri,. D. Bader, J. W. Berry, J. R. Crobak // Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp. 23–35.
10. **Prihozhy, A.** Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs / A. Prihozhy, E. Bezati, H. Rahman, M. Mattavelli // IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 34, No. 10, 2015, pp. 1613–1626.

Поступила *01.07.2017* | *После доработки* *06.07.2017* | *Принята к печати* *10.09.2017*

Prihozhy A., Karasik O.

HETEROGENIOUS BLOCKED ALL- PAIRS SHORTEST PATHS ALGORITHM

Belarusian National Technical University

The problem of finding the shortest paths between all pairs of vertices in a weighted directed graph is considered. The algorithms of Dijkstra and Floyd-Warshall, homogeneous block and parallel algorithms and other algorithms of solving this

problem are known. A new heterogeneous block algorithm is proposed which considers various types of blocks and takes into account the shared hierarchical memory organization and multi-core processors for calculating each type of block. The proposed heterogeneous block computing algorithms are compared with the generally accepted homogeneous universal block calculation algorithm at theoretical and experimental levels. The main emphasis is on using the nature of the heterogeneity, the interaction of blocks during computation and the variation in block size, the size of the block matrix and the total number of blocks in order to identify the possibility of reducing the amount of computation performed during the calculation of the block, reducing the activity of the processor's cache memory and determining the influence of the calculation time of each block type on the total execution time of the heterogeneous block algorithm. A recurrent resynchronized algorithm for calculating the diagonal block (D0) is proposed, which improves the use of the processor's cache and reduces the number of iterations up to 3 times that are necessary to calculate the diagonal block, which implies the acceleration in calculating the diagonal block up to 60%. For more efficient work with the cache memory, variants of permutation of the basic loops k-i-j in the algorithms of calculating the blocks of the cross (C1 and C2) and the updated blocks (U3) are proposed. These permutations in combination with the proposed algorithm for calculating the diagonal block reduce the total runtime of the heterogeneous block algorithm to 13% on average against the homogeneous block algorithm.

***Key words**: Floyd-Warshall algorithm, all pairs shortest paths, heterogeneous block algorithm, multi-core system, shared cache memory.*

Прихожий Анатолий Алексеевич – профессор кафедры программного обеспечения вычислительной техники и автоматизированных систем БНТУ, доктор технических наук (1999), профессор (2001). Научные интересы в области языков программирования и описания цифровой аппратауры, распараллелтвающих компиляторов, инструментальных средств проектирования программных и аппаратных систем на логическом, поведенческом и системном уровнях. Имеет более 300 научных публикаций в Восточной и Западной Европе, США и Канаде.

Anatoly Prihozhy is a full professor at the Computer and system software department of Belarusian national technical university, doctor of science (1999) and professor (2001). His research interests include programming and hardware description languages, parallelizing compilers, and computer aided design tools for software and hardware at logic, high and system levels. He has over 300 publications in Eastern and Western Europe, USA and Canada.

Карасик Олег Николаевич – аспирант кафедры «Программное обеспечение вычислительной техники и автоматизированных систем» БНТУ, ведущий инженер программист компании «EPAM Systems», научные интересы в области параллельных многопоточных приложений и распараллеливания для многоядерных и многопроцессорных систем.

Karasik Aleh is a postgraduate of the Computer and system software department of Belarusian national technical university, and a leading software engineer at EPAM Systems. His research interests include parallel multithreaded applications and the parallelization for multicore and multiprocessor systems.