


Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Нижегородский государственный университет им. Н.И. Лобачевского
Институт информационных технологий, математики и механики

ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ГРАФОВ

Лекция 3. Поиск в ширину в графе

Пирова А.Ю.
Кафедра МОСТ

Содержание

- Применение
- Постановка задачи
- Последовательный алгоритм
- Параллельный алгоритм для общей памяти
- Параллельные алгоритмы для распределенной памяти
  - Одномерное разделение матрицы смежности
  - Двумерное разделение матрицы смежности

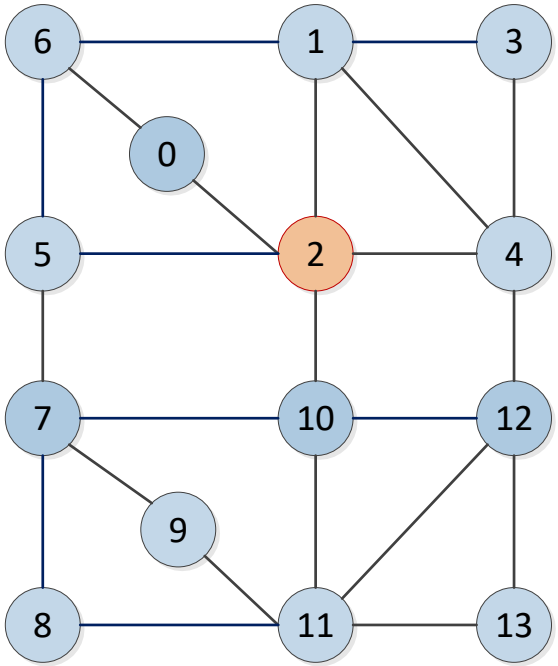
Применение

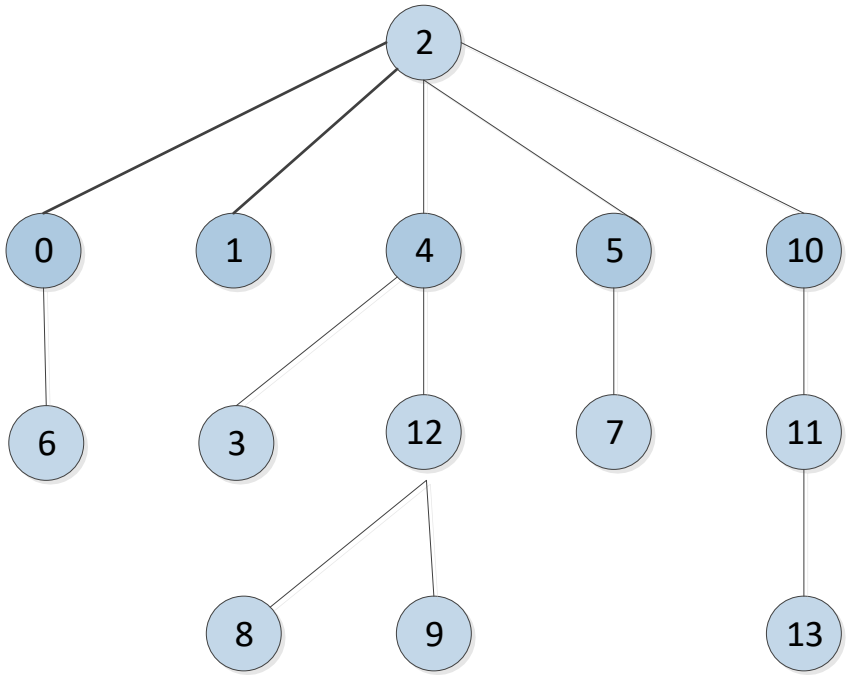
- Два типа обхода графа – поиск в глубину (depth-first search, DFS) и поиск в ширину (breadth-first search, BFS).
- Обход в глубину последовательный по построению. Обход в ширину используется в параллельных вычислениях на графах.
- BFS применяется как составная часть других алгоритмов:
  - Нахождение кратчайших путей между двумя вершинами
  - Алгоритм Форда-Фалкерсона поиска максимального потока в сети
  - Алгоритм растущего разделения графа
  - Вычисление betweenness centrality
  - Поиск компонент сильной связности и др.
- BFS используется как бенчмарк (например, рейтинг Graph500).

Постановка задачи

- Дан граф $G = (V, E)$, где V – множество вершин графа, E – множество ребер графа. Задана исходная вершина графа s. Необходимо найти все вершины графа, достижимые из s, в порядке увеличения расстояния от s.
- В процессе строится дерево поиска, в котором исходная вершина s – корень; вершины, находящиеся на расстоянии k ребер от s, находятся в дереве на глубине k.
- Результат процедуры BFS – массив, в котором для каждой вершины указан родитель (предшественник) в дереве поиска.
- Вариант постановки задачи: найти уровни смежности – массив вершин, в порядке их удаления от исходной.
- Вычислительная сложность $O(n + m)$, если $|V| = n, |E| = m$.

Пример





Parent: {2, 2, -2, 4, 2, 0, 5, 7, 7, 2, 10, 4, 11}

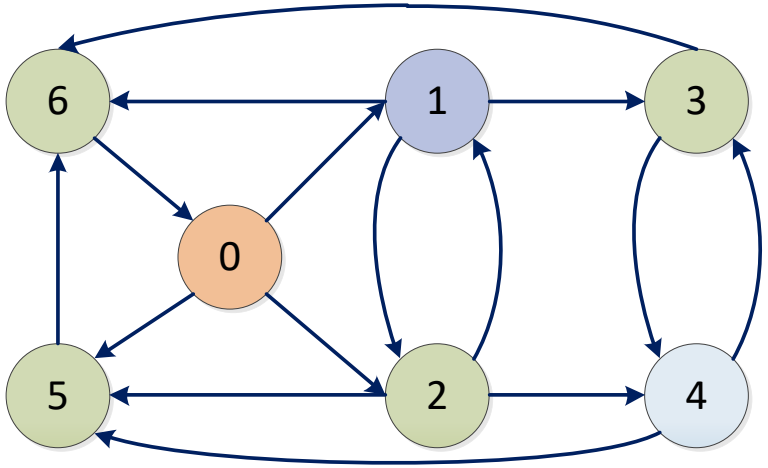
Levels: {2, 0, 1, 4, 5, 10, 6, 3, 12, 7, 11, 8, 9, 13}

LevelIndex: {0, 1, 6, 11, 14}

ПОСЛЕДОВАТЕЛЬНЫЙ АЛГОРИТМ

Базовый алгоритм

- В процессе обхода графа все вершины делятся на 3 типа:
  - уже рассмотренные (родитель и потомки найдены),
  - граничные (родитель найден, потомки еще не найдены),
  - не рассмотренные (родитель еще не найден).



parent:

0	0	0	1	-1	0	1

Граничные вершины: 2, 5, 3, 6

Итог parent:

0	0	0	1	2	0	1

Базовый алгоритм

- Способы построения дерева BFS:
  - «сверху вниз» (top-down) – классический подход. На каждой итерации алгоритма для граничных вершин ищутся потомки среди еще не рассмотренных вершин.
  - «снизу вверх» (bottom up) – На каждой итерации алгоритма для каждой вершины определяется, есть ли ее родитель среди граничных вершин.
  - Комбинированный (direction optimized, DO) – в зависимости от числа граничных вершин выбирается направление обхода.

Обход «сверху вниз» (top-down)

- Алгоритм

Algorithm 1 Sequential top-down BFS algorithm

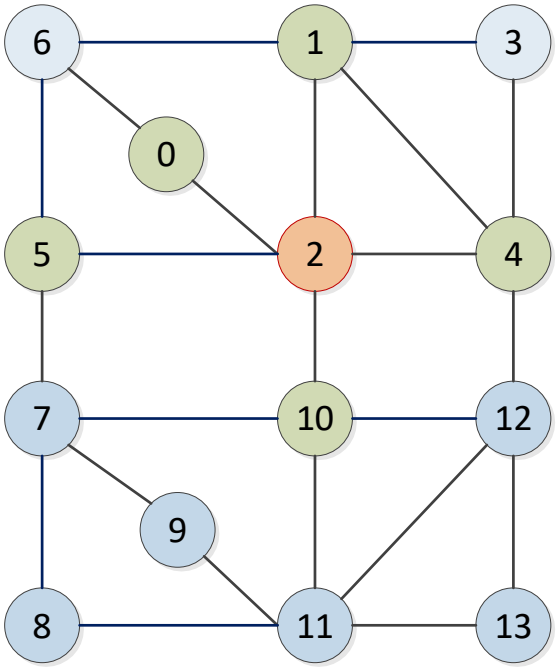
Input: $G(V, E)$, source vertex s

Output: $parent[1..n]$, where $parent[v]$ gives the parent of $v \in V$ in the BFS tree or -1 if it is unreachable from s

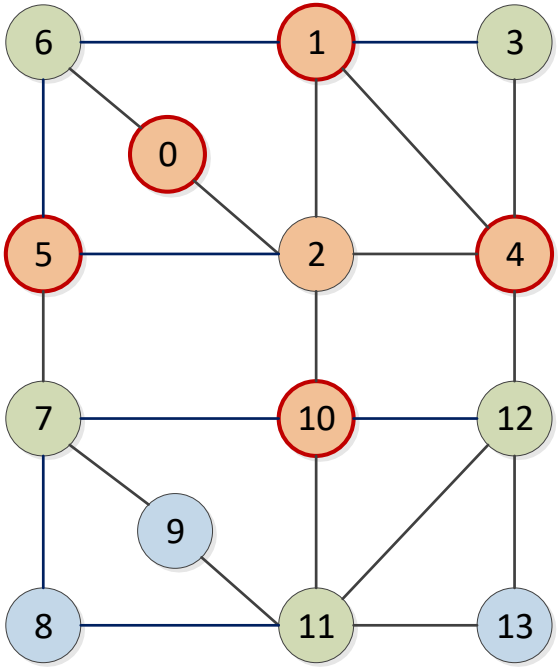
```
1: parent[:] ← −1, parent[s] ← s
2: frontier ← {s}, next ← φ
3: while frontier ≠ φ do
4:     for each u in frontier do
5:         for each neighbor v of u do
6:             if parent[v] = −1 then
7:                 next ← next ∪ {v}
8:                 parent[v] ← u
9:     frontier ← next, next ← φ
```

- Граничные вершины *frontier* хранятся в очереди
- Метки о посещении вершин можно хранить в битовой маске

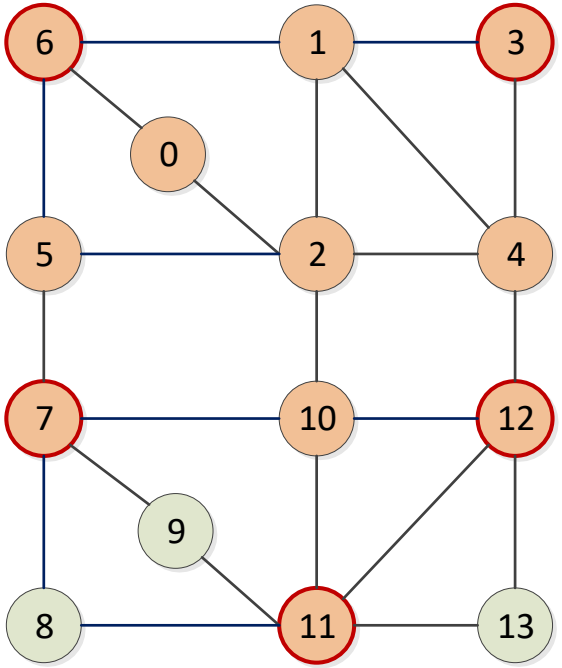
Обход «сверху вниз». Пример



Frontier: {2}
Next: {0, 1, 4, 5, 10}
Parent: {2, 2, -2, 4, 2, -1, -1,-1, -1, 2, -1, -1, -1}



Frontier: {0, 1, 4, 5, 10}
Next: {6, 3, 7, 11, 12}
Parent: {2, 2, -2, 4, 2, 0, 5, -1, -1, 2, 10, 4, -1}



Frontier: {6, 3, 7, 11, 12}
Next: {8, 9, 13}
Parent: {2, 2, -2, 4, 2, 0, 5, 7, 7, 2, 10, 4, 11}

Обход «сверху вниз»

- Размер множества граничных вершин (Бимер и др., 2012)

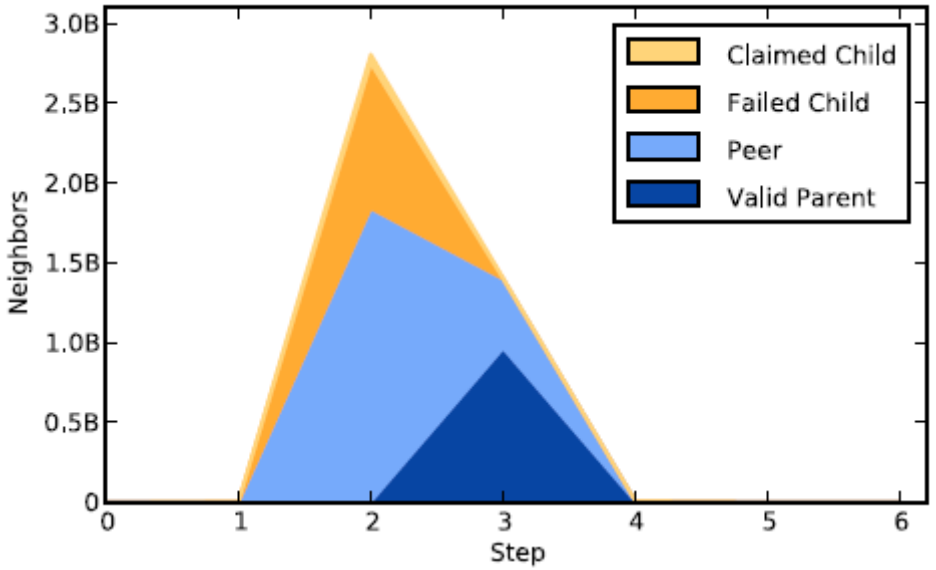


Fig. 3. Breakdown of edges in the frontier for a sample search on `kron27` (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

Для вершины на глубине *d*:
valid parent – любой сосед на глубине $d - 1$;
peer – любой сосед той же глубины
failed child – любой сосед глубины $d + 1$, уже использованный другой вершиной.
claimed child – выбранный вершиной потомок

Beamer S., Asanovic K., Patterson D. Direction-optimizing breadth-first search //SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. – IEEE, 2012. – С. 1-10.

Обход «сверху вниз»

- Размер множества граничных вершин (Бимер и др., 2012)

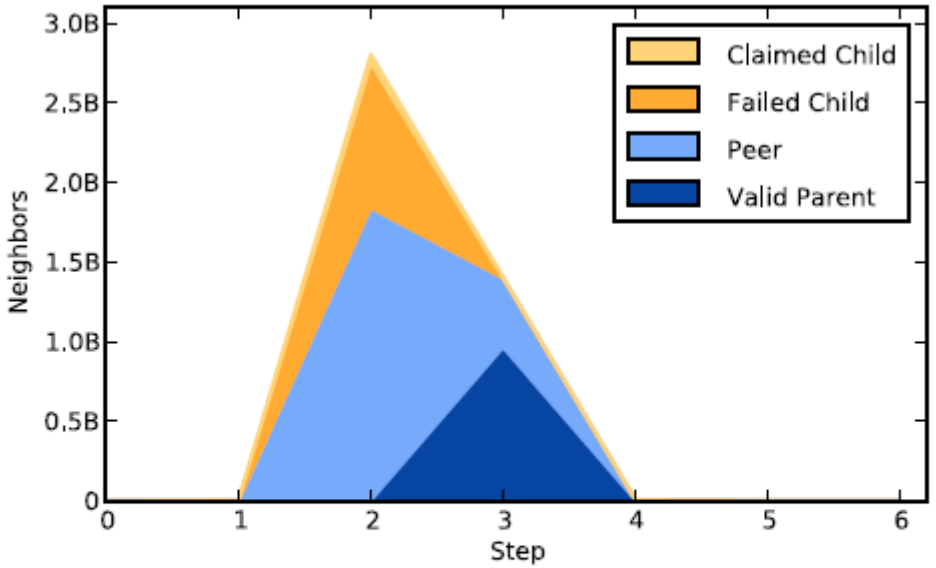


Fig. 3. Breakdown of edges in the frontier for a sample search on `kron27` (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

- Большинство проверок выполняется на шагах 2 и 3
- Лишние проверки, если вершина уже была посещена ранее

Beamer S., Asanovic K., Patterson D. Direction-optimizing breadth-first search //SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. – IEEE, 2012. – С. 1-10.

Обход «снизу вверх»

- Обход «снизу вверх» (bottom up)

Algorithm 2 Sequential bottom-up BFS algorithm

Input: $G(V, E)$, source vertex s

Output: $parent[1..n]$, where $parent[v]$ gives the parent of $v \in V$ in the BFS tree or -1 if it is unreachable from s

```
1:  parent[:] ← −1, parent[s] ← s
2:  frontier ← {s}, next ← φ
3:  while frontier ≠ φ do
4:      for each u in V do
5:          if parent[u] = −1 then
6:              for each neighbor v of u do
7:                  if v in frontier then
8:                      next ← next ∪{u}
9:                      parent[u] ← v
10:                     break
11:     frontier ← next, next ← φ
```

- Как хранить множество граничных вершин?

Обход «снизу вверх»

- Обход «снизу вверх» (bottom up)

Algorithm 2 Sequential bottom-up BFS algorithm

Input: $G(V, E)$, source vertex s

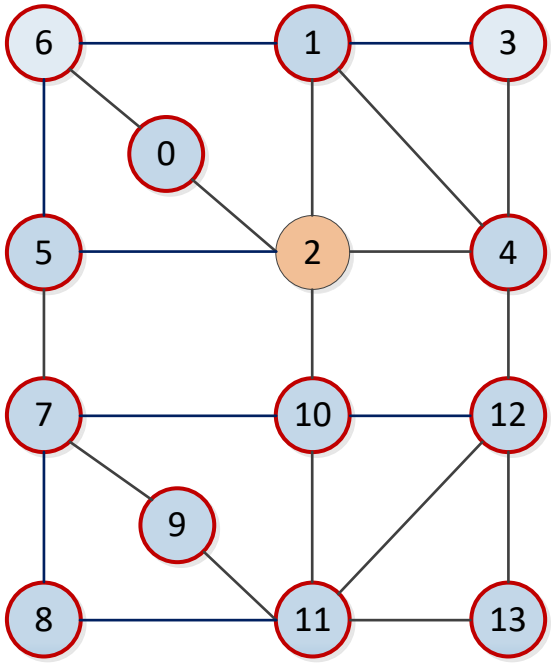
Output: $parent[1..n]$, where $parent[v]$ gives the parent of $v \in V$ in the BFS tree or -1 if it is unreachable from s

```
1:  parent[:] ← −1, parent[s] ← s
2:  frontier ← {s}, next ← φ
3:  while frontier ≠ φ do
4:      for each u in V do
5:          if parent[u] = −1 then
6:              for each neighbor v of u do
7:                  if v in frontier then
8:                      next ← next ∪{u}
9:                      parent[u] ← v
10:                     break
11:     frontier ← next, next ← φ
```

- Граничные вершины хранятся битовой маской

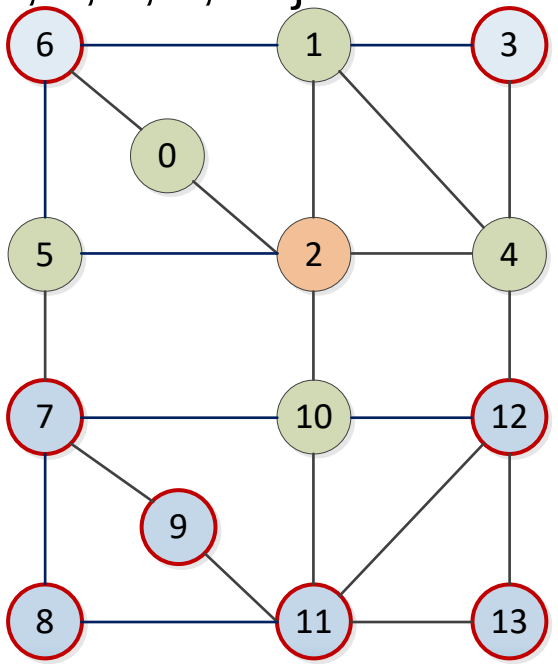
Обход «снизу вверх». Пример

- Итерация 1. Проверяем все вершины, являются ли они соседями *s = 2*



Frontier: {2}
Next: {0, 1, 4, 5, 10}
Parent: {2, 2, -1, 4, 2, -1, -1,-1, -1, 2, -1, -1, -1}

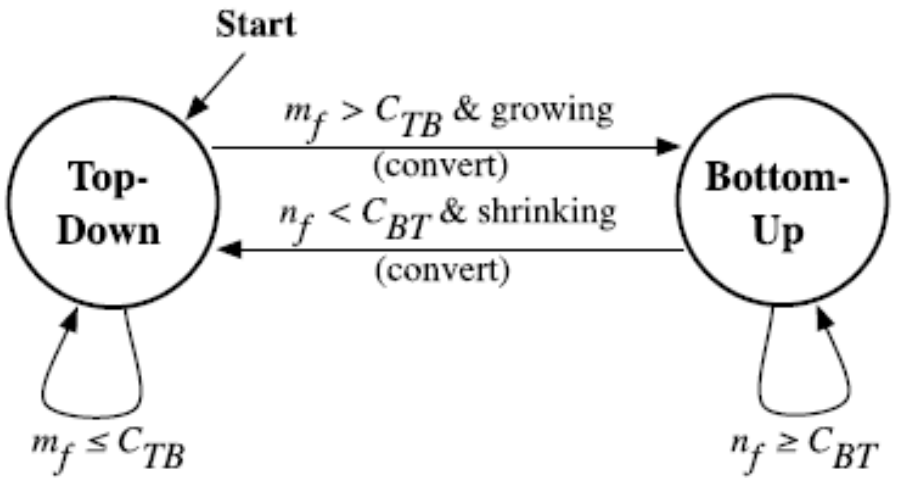
- Итерация 2. Проверяем вершины {6, 3, 7, 8, 9, 11, 12, 13}, являются ли они соседями вершин {0, 1, 4, 5, 10}



Frontier: {0, 1, 4, 5, 10}
Parent: {2, 2, -2, 4, 2, 0, 5, -1, -1, 2, 10, 4, -1}

Комбинированный алгоритм

- Для комбинированного алгоритма целесообразно начинать и заканчивать поиск обходом «сверху вниз», а промежуточные итерации выполнять обходом «снизу вверх».
- Пример схемы переключения между алгоритмами (Бимер и др., 2013):

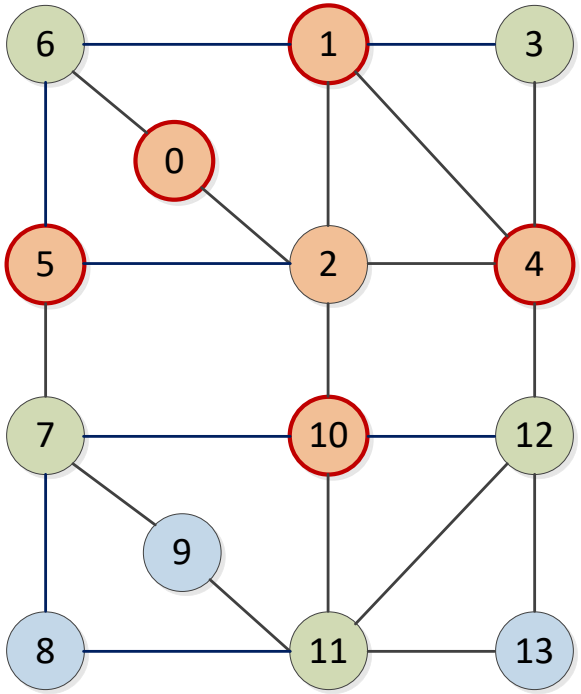


m_f - число ребер, которые надо проверить из граничных вершин
n_f - число граничных вершин
m_u - число ребер, которые надо проверить из непосещенных вершин
$C_{TB} = m_u/\alpha$, $C_{BT} = n/\beta$– пороги переключения, a, b – параметры.

- При переключении между алгоритмами выполняется конвертация представления граничных вершин

Комбинированный алгоритм

- Пример



$m_f = 17$ - число ребер, которые надо проверить из граничных вершин
$n_f = 5$ - число граничных вершин
$m_u = 24$ - число ребер, которые надо проверить из непосещенных вершин
$C_{TB} = m_u/\alpha$, $C_{BT} = n/\beta$ – пороги переключения, a, b – параметры.

Frontier: {0, 1, 4, 5, 10}
Next: {6, 3, 7, 11, 12}
Parent: {2, 2, 1, 4, 2, 0, 5, -1, -1, 2, 10, 4, -1}

ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ ДЛЯ ОБЩЕЙ ПАМЯТИ

Параллельный алгоритм для общей памяти

- Необходимо обрабатывать общую очередь граничных вершин
- Подходы к распараллеливанию:
  - Использование атомарных операций и общей очереди
  - Использование других структур данных для хранения – «карман» вместо очереди (Лейзерсон и Шардл)
  - Подход на основе линейной алгебры: BFS как умножение разреженной матрицы на плотный или разреженный вектор (Кепнер, Гильберт, 2011)

Параллельный алгоритм для общей памяти

Алгоритм 3. Параллельный поиск в ширину, обход **«сверху вниз»**

Input: граф $G(V, E)$, источник s

Output: $parent$ – массив размера n. $parent[v]$ – родитель вершины v, если она достижима из s, иначе $parent[v] = -1$.

1. $parent[s] = s, parent[i] = -1$ для $i \neq s$
2. $frontier = \{s\}$; $next = \emptyset$
3. **while** $(frontier \neq \emptyset)$ **do**:
4. **parallel for** для каждого $u \in frontier$
5. **for** для каждого $v \in Adj(u)$
6. **atomic read** $p = parent[v]$
7. **if** $p == -1$ **then**
8. $next = next \cup \{v\}$
9. **atomic write** $parent[v] = u$
10. $frontier = next, next = \emptyset$

Параллельный алгоритм для общей памяти

Алгоритм 4. Параллельный поиск в ширину, обход **«снизу вверх»**

Input: граф $G(V, E)$, источник s

Output: $parent$ – массив размера n. $parent[v]$ – родитель вершины v, если она достижима из s, иначе $parent[v] = -1$.

1. $parent[s] = s, parent[i] = -1$ для $i \neq s$
2. $frontier = \{s\}$; $next = \emptyset$
3. **while** $(frontier \neq \emptyset)$ **do**:
4. **parallel for** для каждого $u \in V$
5. **for** для каждого $v \in Adj(u)$
6. **if** $v \in frontier$ **then** // atomic read
7. $next = next \cup \{v\}$
8. **atomic write** $parent[v] = u$
9. $frontier = next, next = \emptyset$

Параллельный алгоритм для общей памяти

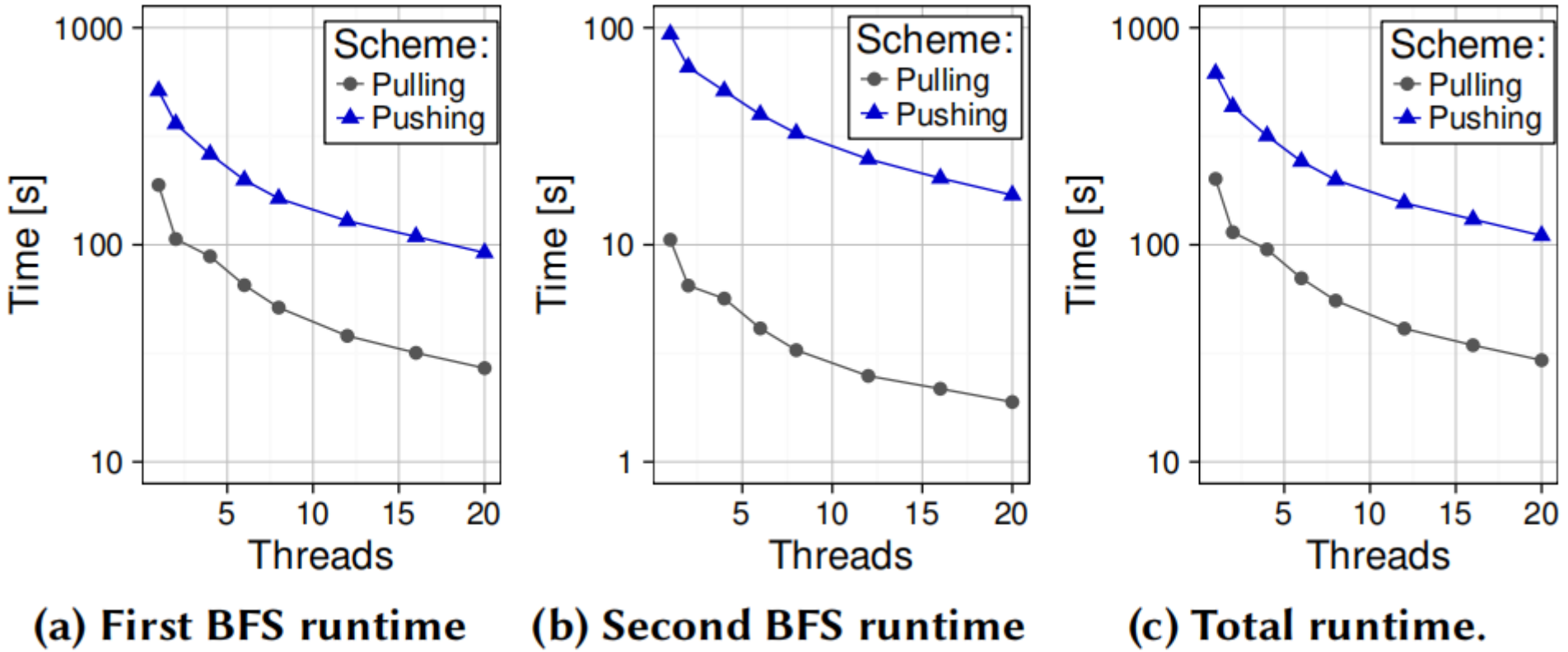
- Оптимизация:
  - Разделить списки смежности каждой вершины на локальную для потока и удаленную части.
  - Сократить барьерную синхронизацию.
  - Разделить фронт на порции, обрабатывать один уровень дерева поиска в несколько итераций.
  - Использовать битовые маски для хранения информации о посещенных и граничных вершинах.

Вычислительные эксперименты

- Источник:
  Besta M. et al. To push or to pull: On reducing communication and synchronization in graph computations //Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. – 2017. – С. 93-104.
- Тестовое окружение:
  Cray XC nodes from the CSCS supercomputing systems. We use XC50 and XC40 nodes from the Piz Daint machine. An XC50 node contains a 12-core Intel Xeon E5-2690 CPU with 64 GiB RAM. Each XC40 node contains an 18-core Intel Xeon E5-2695 CPU with 64 GiB RAM.
- Тестовые матрицы:
  orc, 3.07M вершин, 117M ребер, степени вершин от 9 до 39.

Вычислительные эксперименты

- BFS как один из этапов алгоритма betweeness centrality



(a) First BFS runtime **(b) Second BFS runtime** **(c) Total runtime.**

Besta M. et al. To push or to pull: On reducing communication and synchronization in graph computations //Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. – 2017. – С. 93-104.

ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ ДЛЯ РАСПРЕДЕЛЕННОЙ ПАМЯТИ

Параллельный алгоритм для распределенной памяти. Декомпозиция данных

- 1D разделение. Матрица смежности графа разделяется между процессами на полосы по числу процессов. Процесс P_i обрабатывает область A_i

$$A = \begin{pmatrix} A_1 \\ \hline A_{np} \end{pmatrix}$$

- 2D разделение. Матрица смежности графа разделяется между процессами на квадратные области. Строится квадратная (прямоугольная) сетка из $r \times c$ процессов. Процесс $P_{i,j}$ обрабатывает область $A_{i,j}$

$$A = \left(\begin{array}{c|c|c} A_{11} & \dots & A_{1,p_c} \\ \hline \dots & & \dots \\ \hline A_{1,p_r} & \dots & A_{p_r,p_c} \end{array}\right)$$

- Множество граничных вершин и массив родителей хранится распределенно.

Хранение данных

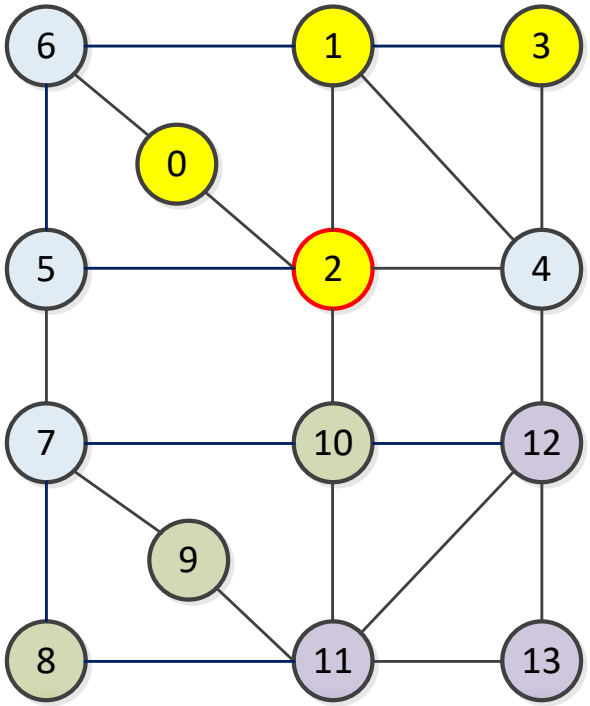
- Хранение подматрицы $A_{i,j}$:
  - CRS. Требует $O(n * p_c + m)$ памяти. Целесообразно использовать для 1D разделения ($p_c = 1$).
  - Doubly Compressed Sparse Columns (DCSC). Используется для 2D разделения (Булук и др., 2011).
  - Skip List + индекс – строки матрицы объединяются в список по несколько штук. Используется для 1D разделения (Чеккони, Петрини, 2014)
- Хранение множества граничных вершин f:
  - 1D разделение – очередь
  - 2D разделение – сортированный разреженный вектор

Параллельный алгоритм для распределенной памяти с 1D разделением. Обход сверху вниз

- Инициализация. $frontier = \{s\}, parent[s] = s.$
- Для всех процессов, Пока есть граничные вершины:
  1. Обойти списки смежности локальных вершин из $frontier$, сохранить в $next$. Сохранить родителя-кандидата вершин из $next$ в массиве $pnext$.
  2. Разослать вершины из $next$ и $pnext$ по процессам-владельцам (all-to-all коммуникация). На каждом процессе объединить полученные списки в $next_{loc}$, $pnext_{loc}$.
  3. Обновить уровень в дереве поиска или родителя для вершин из $next_{loc}$. Вершины, для которых выполнено обновление, сохранить в множество новых граничных f_{new}.
  4. Переход к следующему уровню: $frontier = f_{new}$. Проверить, остались ли еще граничные вершины на каком-либо процессе (allreduce).

Параллельный алгоритм для распределенной памяти с 1D разделением. Обход сверху вниз

- Пример. 4 процесса. **Шаг 1.**



Процесс 1.
front = {2}
Next = {0, 1, 4, 5, 10}
Pnext = {2, 2, 2, 2, 2}
Next_loc = {0, 1}
Parent = {2, 2, 2, -1}
Front_new = {0, 1}

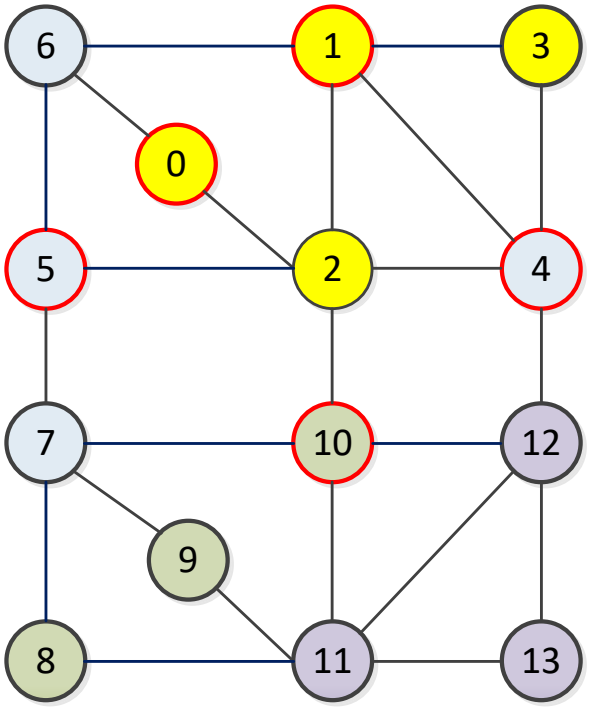
Процесс 2.
front = Ø
Next = Ø
Pnext = Ø
Next_loc = {4, 5}
Parent = {2, 2, -1, -1}
Front_new = {4, 5}

Процесс 3.
front = Ø
Next = Ø
Pnext = Ø
Next_loc = {10}
Parent = {-1, -1, 2}
Front_new = {10}

Процесс 4.
front = Ø
Next = Ø
Pnext = Ø
Next_loc = Ø
Parent = {-1, -1, -1}
Front_new = Ø

Параллельный алгоритм для распределенной памяти с 1D разделением. Обход сверху вниз

- Пример. 4 процесса. **Шаг 2.**



Процесс 1.
front = {0, 1}
Next = {3, 4, 6}
Pnext = {0, 1, 1}
Next_loc = {3, 1, 2}
Parent = {2, 2, 2, 1}
Front_new = {3}

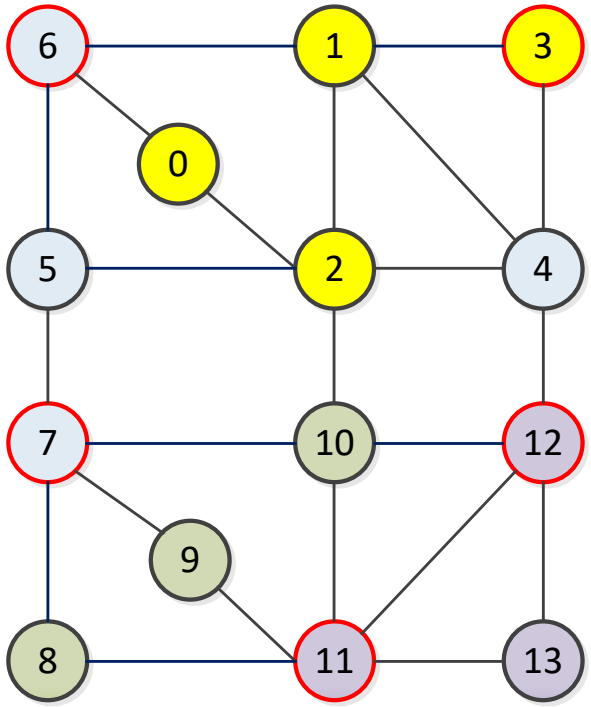
Процесс 2.
front = {4, 5}
Next = {1, 2, 3, 12, 6, 7}
Pnext = {4, 4, 4, 4, 5, 5}
Next_loc = {6, 7, 4}
Parent = {2, 2, 0, 5}
Front_new = {6, 7}

Процесс 3.
front = {10}
Next = {2, 7, 11, 12}
Pnext = {10, 10, 10, 10}
Next_loc = Ø
Parent = {-1, -1, 2}
Front_new = Ø

Процесс 4.
front = Ø
Next = Ø
Pnext = Ø
Next_loc = {11, 12}
Parent = {10, 4, -1}
Front_new = {11, 12}

Параллельный алгоритм для распределенной памяти с 1D разделением. Обход сверху вниз

- Пример. 4 процесса. **Шаг 3.**



Процесс 1.
front = {3}
Next = {4}
Pnext = {3}
Next_loc = {0, 1}
Parent = {2, 2, 2, 1}
Front_new = ∅

Процесс 2.
front = {6, 7}
Next = {0, 1, 5, 8, 9, 10}
Pnext = {6, 6, 6, 7, 7, 7}
Next_loc = {4, 5}
Parent = {2, 2, 0, 5}
Front_new = {6, 7}

Процесс 3.
front = ∅
Next = ∅
Pnext = ∅
Next_loc = {8, 9, 10}
Parent = {7, 7, 2}
Front_new = {8, 9}

Процесс 4.
front = {11, 12}
Next = {8, 9, 10, 13, 4}
Pnext = {11, 11, 11, 11, 12}
Next_loc = {13}
Parent = {10, 4, 11}
Front_new = {13}

Параллельный алгоритм для распределенной памяти с 2D разделением. Обход сверху вниз

Для всех процессов, Пока есть граничные вершины:

1. Построить текущее множество граничных вершин $frontier$, собрав его по процессам (allgather по столбцу процессов)
2. Обойти списки смежности локальных вершин из $frontier$. Для вершин следующего уровня сохранить родителей-кандидатов t. Операция аналогична умножению разреженной матрицы на разреженный вектор (SpMSV).
   $t[v] = u$ – для вершины v родитель-кандидат u
3. Обменяться списками родителей-кандидатов t с разных процессов (alltoallv по строке процессов). Объединить обновления для повторяющихся вершин.
4. Определить родителей для вершин следующего уровня ($t[v] \neq -1$). Определить новые граничные вершины как те, для которых обновлялся родитель.

Параллельный алгоритм для распределенной памяти с 2D разделением. Обход сверху вниз

Algorithm 3 Parallel 2D top-down BFS algorithm (adapted from the linear algebraic algorithm [6])

Input: A: graph represented by a boolean sparse adjacency matrix, s: source vertex id

Output: π: dense vector, where $\pi[v]$ is the predecessor vertex on the shortest path from s to v, or -1 if v is unreachable

1: $\pi(:) \leftarrow -1$, $\pi(s) \leftarrow s$
2: $f(s) \leftarrow s$ ▷ f is the current frontier
3: **for** all processors $P(i, j)$ **in parallel do**
4: **while** $f \neq \emptyset$ **do**
5: TRANSPOSEVECTOR(f_{ij})
6: $f_i \leftarrow$ ALLGATHERV($f_{ij}, P(:, j)$)
7: $t_{ij}(:) \leftarrow 0$ ▷ t is candidate parents
8: **for** each $f_i(u) \neq 0$ **do** ▷ u is in the frontier
9: **for** each neighbor v of u in A_{ij} **do**
10: $t_{ij}(v) \leftarrow u$
11: $t_{ij} \leftarrow$ ALLTOALLV($t_i, P(i, :)$)
12: $f_{ij}(:) \leftarrow 0$
13: **for** each $t_{ij}(v) \neq 0$ **do**
14: **if** $\pi_{ij}(v) \neq -1$ **then** ▷ Set parent if new
15: $\pi_{ij}(v) \leftarrow t_{ij}(v)$
16: $f_{ij}(v) \leftarrow v$

Параллельный алгоритм для распределенной памяти с 2D разделением. Обход сверху вниз

- На разных этапах алгоритма синхронизация выполняется по разным процессам (шаг 1 – по строке, шаг 3 – по столбцу). Это сокращает время коммуникации в сравнении с 1D разделением.
- Граничные вершины f и родители-кандидаты t хранятся как разреженные вектора

Параллельный алгоритм для распределенной памяти с 2D разделением. Обход снизу вверх

Algorithm 4 Parallel 2D bottom-up BFS algorithm

Input: A: unweighted graph represented by a boolean sparse adjacency matrix, s: source vertex id

Output: π: dense vector, where $\pi[v]$ is the parent vertex on the shortest path from s to v, or -1 if v is unreachable

1: $f(:) \leftarrow 0$, $f(s) \leftarrow 1$ ▷ bitmap for frontier
2: $c(:) \leftarrow 0$, $c(s) \leftarrow 1$ ▷ bitmap for completed
3: $\pi(:) \leftarrow -1$, $\pi(s) \leftarrow s$
4: **while** $f(:) \neq 0$ **do**
5: **for** all processors $P(i, j)$ **in parallel do**
6: TRANSPOSEVECTOR(f_{ij})
7: $f_i \leftarrow$ ALLGATHERV($f_{ij}, P(:, j)$)
8: **for** s in $0 \ldots p_c - 1$ **do** ▷ p_c sub-steps
9: $t_{ij}(:) \leftarrow 0$ ▷ t holds parent updates
10: **for** u in $V_{i,j+s}$ **do**
11: **if** $c_{ij}(u) = 0$ **then** ▷ u is unvisited
12: **for** each neighbor v of u in A_{ij} **do**
13: **if** $f_i(v) = 1$ **then**
14: $t_{ij}(u) \leftarrow v$
15: $c_{ij}(u) \leftarrow 1$
16: **break**
17: $f_{ij}(:) \leftarrow 0$
18: $w_{ij} \leftarrow$ SENDRECV($t_{ij}, P(i, j + s), P(i, j - s)$))
19: **for** each $w_{ij}(u) \neq 0$ **do**
20: $\pi_{ij}(u) \leftarrow w_{ij}(u)$
21: $f_{ij}(u) \leftarrow 1$
22: $c_{ij} \leftarrow$ SENDRECV($c_{ij}, P(i, j + 1), P(i, j - 1)$))

УНИВЕРСИТЕТ ЛОБАЧЕВСКОГО

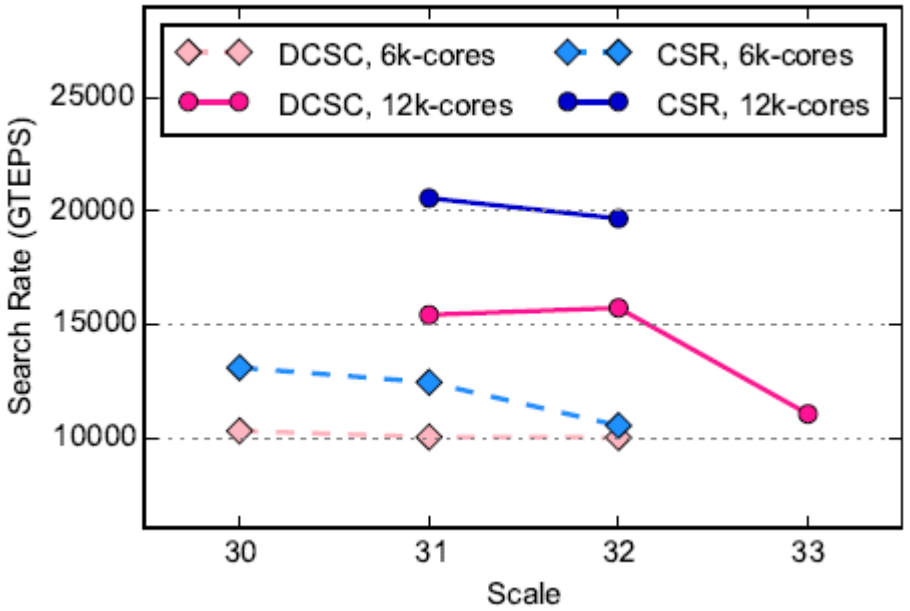
Результаты вычислительных экспериментов

- Источник: Buluç A. et al. Distributed-memory breadth-first search on massive graphs //arXiv preprint arXiv:1705.04590. – 2017.
- Тестовая инфраструктура: Cray XK7 (Titan), 2.2 GHz, 16 ядер на чип, память 22 Gb.
- Тестовые графы: R-MAT порядка 2^s, $s = 31, 32, 33$, со степенью вершин 16; граф Твиттер с 61.5 млн вершин и 1.47 биллионом ребер.
- Производительность измеряется в TEPS – Traversed Edges Per Second

  TEPS = число ребер в графе / время работы (в секундах)

Результаты вычислительных экспериментов

- Сравнение производительности при использовании форматов CRS и DCSC

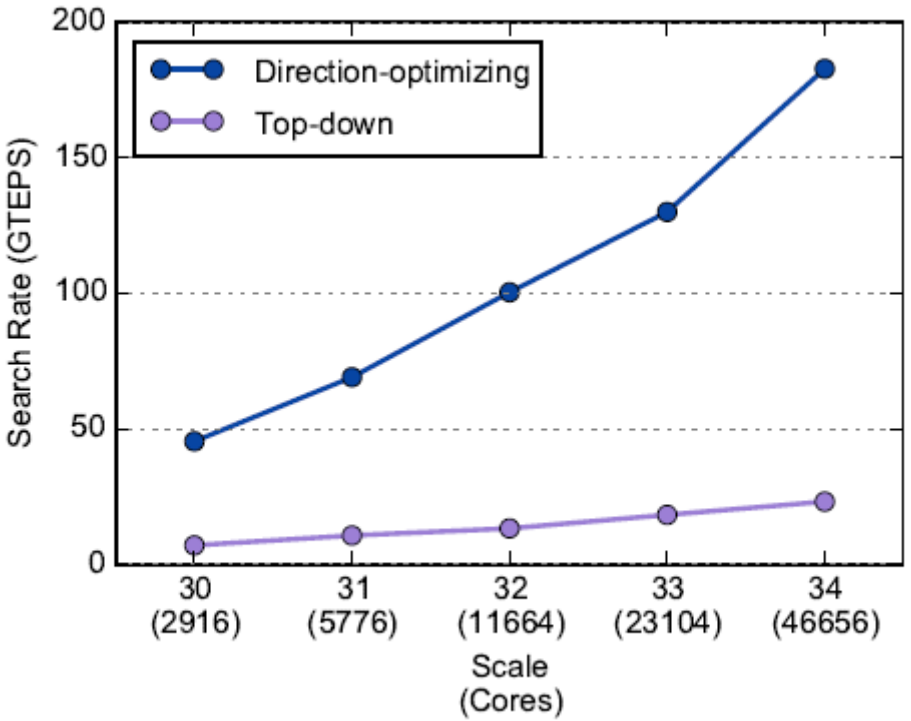


Сравнение производительности DCSC и CRS. 6k на решетке процессоров 78*78, 12k – на решетке 120*120.

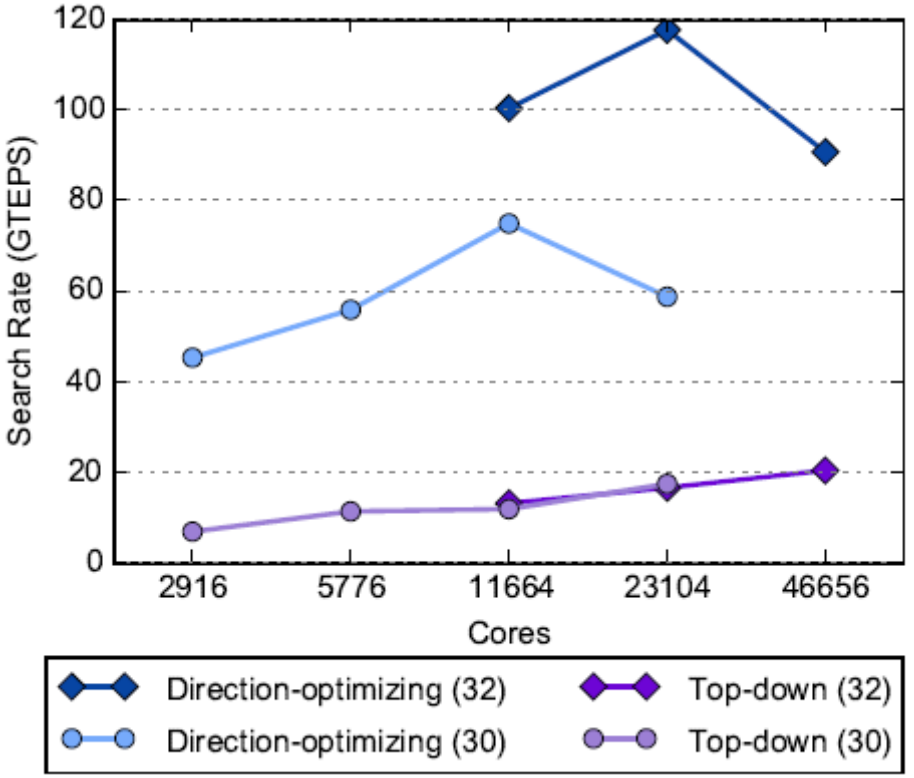
- В представлении CRS алгоритм работает быстрее, если граф убирается в память. Для этого необходимо поддерживать дополнительный массив размера $O(n)$. В DCSC вместо этого – дополнительная косвенная индексация.
- При росте порядка графа производительность снижается для CRS представления

Результаты вычислительных экспериментов

- Сравнение масштабируемости для разных видов обхода графа



Слабая масштабируемость на графах R-MAT, формат DCSC



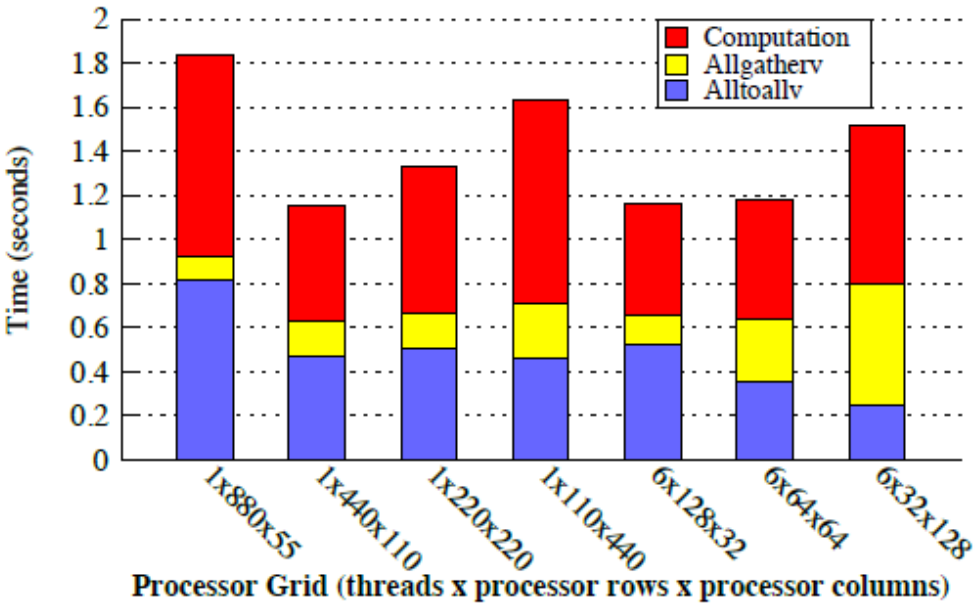
Сильная масштабируемость на графах R-MAT порядка 30 и 32, формат DCSC

Результаты вычислительных экспериментов

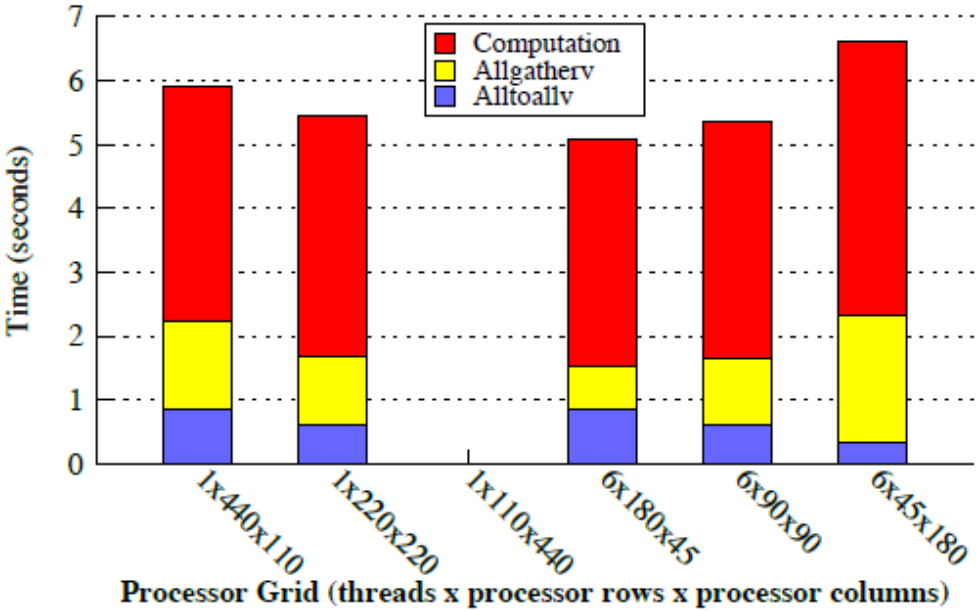
- Выводы:
  - Производительность комбинированного алгоритма от 6.5 до 7.9 раз больше, чем обхода «сверху вниз».
  - При увеличении числа вычислительных ядер снижается производительность комбинированного алгоритма. Большую долю времени занимает коммуникация между процессами.
  - Производительность обхода «снизу вверх» возрастает с ростом числа ядер, при этом рост коммуникаций между процессами компенсируется уменьшением числа вычислительных операций на ядро.

Результаты вычислительных экспериментов

- Время работы на прямоугольной решетке процессов



(a) scale 32



(b) scale 33

Алгоритм обхода «сверху вниз», формат хранения CRS, 48400 ядер для базовой MPI-реализации, 48600 ядер для реализации MPI + потоки.

Результаты вычислительных экспериментов

- Выводы:
  - Абсолютное время поиска различается незначительно для разных конфигураций
  - Для графа порядка 32 небольшое преимущество имеют «длинные вытянутые» прямоугольные решетки, для графа порядка 33 – квадратные решетки
  - Время работы на «коротких длинных» решетках больше из-за большего числа локальных коммуникаций и операций Allgatherv
  - При увеличении порядка графа наблюдается замедление в 4 раза. Это объясняется возрастанием числа локальных операций.

Сравнение библиотек

Paper	Problem	Graph	Memory	Hyper-threads	Nodes	Time
Mosaic [54]	BFS*	2014	0.768	1000	1	6.55
	Connectivity*	2014	0.768	1000	1	708
	SSSP*	2014	0.768	1000	1	8.6
FlashGraph [26]	BFS*	2012	.512	64	1	208
	BC*	2012	.512	64	1	595
	Connectivity*	2012	.512	64	1	461
	TC*	2012	.512	64	1	7818
BigSparse [45]	BFS*	2012	0.064	32	1	2500
	BC*	2012	0.064	32	1	3100
Slota et al. [85]	Largest-CC*	2012	16.3	8192	256	63
	Largest-SCC*	2012	16.3	8192	256	108
	Approx k-core*	2012	16.3	8192	256	363
Stergiou et al. [86]	Connectivity	2012	128	24000	1000	341
This paper	BFS*	2014	1	144	1	5.71
	SSSP*	2014	1	144	1	9.08
	BFS*	2012	1	144	1	16.7
	BC*	2012	1	144	1	35.2
	Connectivity	2012	1	144	1	38.3
	SCC*	2012	1	144	1	185
	k-core	2012	1	144	1	184
	TC	2012	1	144	1	1470

Dhulipala L., Blelloch G. E., Shun J. Theoretically efficient parallel graph algorithms can be fast and scalable //Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures. – 2018. – С. 393-404.

Заключение

- Алгоритм поиска в ширину используется как часть других алгоритмов обработки графов
- Существует два способа реализации обхода графа в ширину – обход «сверху вниз» и обход «снизу вверх». Эффективно применять комбинированный алгоритм, в котором первые и последние итерации выполняются обходом «сверху вниз», промежуточные – обходом «снизу вверх».
- Параллельный алгоритм для систем с общей памятью предполагает использование атомарных операций и работы с общей очередью вершин. Реализации плохо масштабируются
- Параллельный алгоритм для систем с распределенной памятью можно реализовать с одномерным или двумерным разделением матрицы смежности по процессам

Заключение

- При реализации алгоритма на распределенной памяти важен выбор формата хранения матрицы смежности. Как правило, используются форматы CRS, DCSC, DCRS и их модификации.
- Производительность реализаций для систем с распределенной памятью ограничена большой долей коммуникаций между процессами.
  - Производительность комбинированного алгоритма больше, чем обхода «сверху вниз».
  - При 2D разделении матрицы смежности меньше коммуникаций вида все-со-всеми, чем при 1D разделении.
  - Для графов больших порядков целесообразно использовать квадратные или прямоугольные «длинные вытянутые» решетки процессоров

Литература

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ, 3-е издание. – М.: «Вильямс», 2013. – 1328 с.
2. Buluç A. et al. Distributed-memory breadth-first search on massive graphs //arXiv preprint arXiv:1705.04590. – 2017.
3. Besta M. et al. To push or to pull: On reducing communication and synchronization in graph computations //Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. – 2017. – С. 93-104.
4. Beamer S., Asanovic K., Patterson D. Direction-optimizing breadth-first search //SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. – IEEE, 2012. – С. 1-10.
5. Ueno K. et al. Efficient breadth-first search on massively parallel and distributed-memory machines //Data Science and Engineering. – 2017. – Т. 2. – №. 1. – С. 22-35.

Контакты

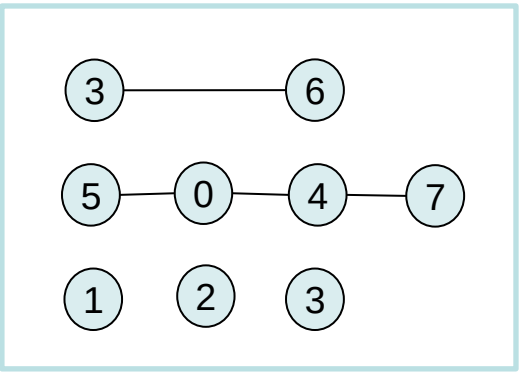
Нижегородский государственный университет
http://www.unn.ru

Институт информационных технологий, математики и механики
http://www.itmm.unn.ru

Пирова А.Ю.
anna.pirova@itmm.unn.ru

Форматы хранения графа

Edges list	SRC	0 0 6 7
	DST	4 5 3 1
CSR	Row-starts	0 2 2 2 2 2 2 3 4
	DST	4 5 3 1
Bitmap-based sparse matrix representation	Offset	0 1 3
	Bitmap	1 0 0 0 0 0 1 1
	Row-starts	0 2 3 4
	DST	4 5 3 1
DCSR	AUX	0 1 1 3
	JC	0 6 7
	Row-starts	0 2 3 4
	DST	4 5 3 1



Ueno K. et al. Efficient breadth-first search on massively parallel and distributed-memory machines //Data Science and Engineering. – 2017. – Т. 2. – №. 1. – С. 22-35.

Форматы хранения графа

- Требуемый объем памяти

Data structure	CSR		Bitmap-based CSR	
	Order	Actual	Order	Actual
Offset	–	–	$V'C/64$	32 MB
Bitmap	–	–	$V'C/64$	32 MB
Row-starts	$V'C$	2048 MB	$V'p$	190 MB
DST	$V'\hat{d}$	1020 MB	$V'\hat{d}$	1020 MB
Total	$V'(C+\hat{d})$	3068 MB	$V'(C/32+p+\hat{d})$	1274 MB
Data structure	DCSR		Coarse index + Skip list	
	Order	Actual	Order	Actual
AUX	$V'p$	190 MB	–	–
JC	$V'p$	190 MB	–	–
Row-starts	$V'p$	190 MB	$V'C/64$	32 MB
DST or skip list	$V'\hat{d}$	1020 MB	$V'\hat{d}+V'p$	1210 MB
Total	$V'(3p+\hat{d})$	1590 MB	$V'(C/64+p+\hat{d})$	1242 MB

We partition a Graph500 graph with 16 billion vertices and 256 billion edges into $64 \times 32 = 2048$ Nodes

Ueno K. et al. Efficient breadth-first search on massively parallel and distributed-memory machines //Data Science and Engineering. – 2017. – Т. 2. – №. 1. – С. 22-35.