ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Чухарева Мария Валентиновна

МНОГОПОТОЧНЫЕ АЛГОРИТМЫ СИСТЕМЫ НЕПЕРЕСЕКАЮЩИХСЯ МНОЖЕСТВ ДЛЯ NUMA АРХИТЕКТУР

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
Д.М. Кутенин

Руководитель
канд.физ.-мат.наук
А.И. Храбров

Консультант
Н.Д. Коваль

Санкт-Петербург 2022

Оглавление

В современных многопроцессорных системах часто используется архитектура с неоднородным доступом к памяти (NUMA), которая позволяет эффективно поддерживать работу большего числа ядер, чем классическая архитектура. NUMA – архитектура с неоднородным доступом к памяти – подразумевает разделение ядер на NUMA-сокеты, каждый из которых обладает своей локальной памятью и кешом. Доступ сохраняется ко всей памяти, однако каналы для взаимодействия между соектами обладают большей задержкой и меньшей пропускной способностью, так что взаимодействие с памятью других сокетов гораздо дороже, чем с локальной. Для оптимальной работы программ на такой архитектуре, базовые алгоритмы специально адаптируют, чтобы минимизировать взаимодействие между сокетами и большая часть взаимодействий с памятью проводилась именно внутри сокета

Данная работа направлена на оптимизацию одной из классических структур данных – системы непересекающихся множеств – которая широко применяется в графовых алгоритмах. Алгоритм уже хорошо изучен в своей последовательной и многопоточной версиях, однако еще не было попыток адаптировать структуру к NUMA-архитектуре – так что данная работа направлена на устранение этого пробела.

В результате работы было получено несколько алгоритмов-адаптаций осонованных на техниках репликации данных (с раличными механизмами синхронизации реплик, с разной гарантией на согласованность данных) и разделения данных. Было проведено сравнение полученных алгоритмов с существующими многопоточными решениями в разных сценариях использования, в результате которого выяснилось какие алгоритмы в каких ситуациях использовать оптимальнее. Так в некоторых случаях предложенные алгоритмы выигрывают до 35% производительности у известных алгоритмов без адаптации.

Ключевые слова: многопоточность, многопоточные алгоритмы, неоднородная архитектура (NUMA), система непересекающихся множеств (CHM)

Modern multiprocessor hardware is most often non-uniform memory access (NUMA) machines that allow more cores to be supported efficiently. In this architecture, systems contain multiple NUMA-nodes where each node has its local memory, a local cache, and processing cores. In such systems, all memory is visible and cache-coherent, but inter-node communication channels typically have higher latency than intra-node links. So, most algorithms need some redesign to reduce the number of inter-node communications and achieve the best possible performance on NUMA.

In this paper, we redesign the disjoint set union (DSU) – one of the fundamental algorithmic problems widely used in graph algorithms. DSU already has several sequential solutions and a bunch of concurrent algorithms but was never optimized for NUMA, and this paper fills the gap.

We introduced several possible designs of NUMA-aware DSU based on data replication and data separation principles. After performance comparison against known concurrent versions, we find out what algorithms are optimal in different cases. So, in some cases, our algorithms win up to 35% of the performance of known algorithms without adaptation.

Keywords: concurrency, parallel algorithms, non-uniforn memory access (NUMA), disjoint set union (DSU), Union-Find problem

Введение

Многие современные высокопроизводительные мультипроцессорные системы используют архитектуру с неоднородным доступом к памяти, котороая позволяет одновременно иметь больше ядер чем классическая архитектура памяти. NUMA (non-uniform memory access) — архитектура с неоднородной памятью — предполагает разделение ядер на NUMA-сокеты, внутри каждого из которых есть своя локальная когерентная память, кеш и ядра работающие с ними. В такой системы сохраняется доступ и когерентность всей памяти, однако взаимодействие между сокетами становится намного дороже, чем внутри одного.

Для работы программ на такой памяти обычно не требуется дополнительных действий, однако так не используется все ее преимущество — для этого программы специально адаптируют, а точнее – оптимизируют работу с памятью, чтобы сократить взаимодействие между сокетами. На данный момент существует ряд работ, адаптирующих примитивы синхронизации и стандартные алгоритмы к NUMA-архитектуре.

Задача поддержки системы непересекающихся множеств (СНМ) — одна из фундаментальных задач, которые еще не были адаптированы для работы на NUMA-архитектурах.

Задача: есть набор элементов, разделенный на множества (непересекающиеся) и на этом наборе требуется поддерживать три операции:

- `Union(u, v)` – объединить множества, содержащие эдлементы `u` и `v`
- `SameSet(u, v)` – проверить находятся ли элементы `u` и `v` в одном множестве
- `Find(u)` – найти репрезентативного представителя множества, в котором находится элемент `u` (в большинстве применений эта операция выполняет только вспомогательную роль)

Задача имеет классическое решение, основанное на представлении множеств в виде деревьев и такую структуру принято называть системой непересекающихся множеств. Кроме того, на классическом последовательном алгоритме основаны различные многопоточные алгоритмы (подробнее в главе 1).

Система непересекающихся множеств применяется как в чистом виде, так и как часть более сложных алгоритмов. Наиболее распространена в графовых задачах:

динамическое поддержание компонент связности графа, поиск компонент сильной связности, построение минимального остовного дерева или поиск минимального общего предка в дереве (LCA).

Постановка и актуальность проблемы

На данный момент еще не представлено ни одного алгоритма, адаптирующего систему непересекающихся множеств к неоднородной архитектуре. Кроме того, что задача очень важная, ее природа позволяет ожидать заметное улучшение результатов после оптимизации: запросы работают с разными частями структуры и хорошо параллелятся.

Цель работы: Оптимизировать алгоритм системы непересекающихся множеств для работы на архитектуре с неоднородной памятью (NUMA).

Задачи

- Изучить известные работы по созданию и адаптации различных алгоритмов к неоднородной архитектуре, выделить среди них подходы, применимые к системе непересекающихся множеств
- Спроектировать и реализовать алгоритм(ы) CHM для архитектуры NUMA
- Сравнить и проанализировать эффективность реализованных алгоритмов на синтетических и реальных примерах использования

1. Обзор литературы

1.1. Система непересекающихся множеств

1.1.1. Последовательный алгоритм

Классическая последовательная реализация системы непересекающихся множеств была предложена в 1964 [5], и чуть позже была доказана оценка сложности алгоритма [9, 10]. Алгоритм представлял каждое множество как дерево, в вершинах которого хранятся элементы, а корневой элемент дерева используется в качестве репрезентативного представителя своего множества. В классической реализации деревья представляются как список предков. Тогда операция объединения двух множеств представляется как: дерево одного множества подвешивается к корню второго (меняется один предок), а получение представителя множества по элементу становится операцией поиска корня дерева, а для этого достаточно подняться по дереву вверх от элемента до корня. Проверка лежат ли два элемента в одном множестве выражается через операцию поиска представителя – элементы находятся в одном и том же множестве тогда и только тогда, когда их представители совпадают.

Заметим, что время работы алгоритма зависит от высоты получившихся деревьев (из-за линейного поиска корня). Для контроля высоты применяются две классические эвристики – **эвристика сжатия путей** и **ранговая**. Эвристика сжатия путей отвечает за переподвешивание вершин из множества ближе к корню при подъеме по дереву, что обеспечивает логарифмическую сложность. Существует три варианта эвристики сжатия путей:

- классическое двухпроходное сжатие – первым проходом находится корень, вторым все вершины на пути переподвешиваются к нему
- halving – каждая вторая вершина переподвешивается к "дедушке" (предку предка), тем самым путь делится пополам
- splitting – каждая вершина переподвешивается к "дедушке" (предку предка)

Ранговая эвристика отвечает на вопрос какой из двух корней станет представителем множества при выполнении операции $Union$. Дерево с меньшей высотой выгодно подвешивать к дереву с большей. Ранг – это высота дерева или некоторое

ее приближение (например, через размер поддерева). Но вместо классической детерминированной ранговой эвристики так же принято использовать **случайный выбор корня**: при объединении корень для подвешивания определяется случайно.

Дополнительная неасимптотическая эвристика – **немедленная проверка родителей** (immediate parent check). При малом числе множеств высока вероятность, что вершины находятся в одном множестве и даже имеют одного родителя – отсюда в операции $SameSet$ может быть выгодно не искать корень дважды, а заранее проверить предков на равенство.

1.1.2. Многопоточные алгоритмы

Первая многопоточная версия алгоритма впервые была предложена в 1994 [2] — неблокирующий алгоритм на основе примитива синхронизации `CAS`, основанный на классической последовательной реализации с использованием ранговой эврстики (подсчет высоты дерева) и эвристики разделения пути методом halving.

В 2016 году была предложена аналогичная версия алгоритма [6], но уже с рандомизированным выбором корня (ее преимущество в том, что не требуется дополнительных затрат по памяти на подсчет ранга) и эвристикой разделения пути методом splitting, а так же была доказана оценка сложности данного алгоритма, показывающая его асимптотическое преимущество. Кроме того, позже авторы [7] предложили возможность использовать в алгоритме ранговую эвристику, реализованную с использованием примитива синхронизации `DCAS` (double compare-and-swap) и доказали равенство верхней оценки для такого алгоритма и для алгоритма, предложенного в первой статье 2016 года.

Однако эти алгоритмы не единственные: по сути, многопоточный алгоритм может также быть основан на любом наборе из классических эвристик.

1.1.3. Сравнение алгоритмов

В 2019 году была опубликована масштабная статья-сравнение различных реализаций системы непересекающихся множеств [1]: вариаций классического алгоритма с различными эвристиками и всевозможные многопоточные реализации. В результате было выявлено какие эвристики наиболее эффективны на практике:

- Из эвристик сжатия путей хуже всего себя показала классическая двух-

проходная эвристика, в то время как алгоритмы, использующие splitting и halving, показали схожие результаты.

- Эвристика случайного выбора корня оказалась эффективнее ранговых благодаря тому, что использует меньше пямяти, что уменьшает затраты на промахи кеша.

Кроме того, была проанализирована эвристика немедленной проверки родителей: как и ожидалось, она улучшает результаты на малом числе множеств, но и не ухудшает их в общем случае.

Большая часть многопоточных алгоритмов реализуется с использованием примитива синхронизации `CAS`. Однако было выявлено, что при реализации эвристики сжатия путей выгоднее использовать простую запись: алгоритм остается корректен, а время работы уменьшается значительно, так как процент невыполненных операций `CAS` достаточно мал, а операция простой записи намного быстрее.

Наиболее эффективным на большом числе множеств оказался алгоритм, использующий транзакционную память, а в остальных случаях – использующий случайный выбор корня, немедленную проверку родителя и простые записи вместо операций `CAS`.

1.2. NUMA-алгоритмы

На данный момент не существует специального алгоритма системы непересекающихся множеств, ориентированного на неоднородную архитектуру (NUMA). Более того, на данный момент представлено не так много NUMA-алгоритмов. Большая часть работ по адаптации алгоритмов на неоднородную память связана с примитивами синхронизации. Использование адаптированных примитивов синхронизации помогают адаптировать весь алгоритм, который их использует – однако, это плохо подходит для нашей задачи, так как все эффективные многопоточные алгоритмы системы непересекающихся множеств являются неблокирующими.

Так же есть ряд статей, адаптирующих различные алгоритмы и структуры данных к NUMA-памяти, которые не связаны с нашей задачей, но могут служить примером и представляют некоторые общие методы. Два основных приема – это репликация и разделение данных.

Разделение данных предполагает разделение структуры на непересекающиеся части, расположенные на разных NUMA-сокетах – при таком методе предполага-

ется построить алгоитм так, что по большей части исполняющий поток работает именно с локальной памятью. Хорошим примером применения разделения данных является реализация многопоточной хеш-таблицы, представленная в статье 2012 года [8]. Метод эффеетивен, если есть возможность так данные разделить, однако для системы непересекающихся множеств этот подход мало применим, т.к. единственное естественное разделение данных – разделение по множествам, но оно не статично и потребует дополнительных затрат на пере-разделение и перемещение данных между сокетами.

Репликация – второй, более общий, подход подразумевает сохранение копии данных на каждом сокете и некоторый механизм синхронизации этих копий.

В 2017 году был предложен метод адаптации алгоритмов к архитектуре NUMA [3]. Метод состоит из трех концепций: репликация структуры данных на каждом NUMA-сокете, техники flat-combining [4] и использования журнала операций для синхронизации между копиями. Однако данный метод подразумевает адаптацию однопоточного алгоритма, что мало применимо к алгоритму системы непересекающихся множеств – в нашем случае запросы почти никогда не работают с одной частью структуры, так что хорошо распараллеливаются в неблокирующих многопоточных алгоритмах, а при использовании техники flat-combining любой новый запрос будет ожидать пока не выполнятся все предыдущие. Для нас может быть хорошим вариантом использование журнала операций и метода репликации данных вместе с одним из известных многопоточных алгоритмов системы непересекающихся множеств.

Еще один подход – делегирование. Для реализации на неоднородной памяти важно увеличить долю запросов к локальной памяти, а в идеале – оставить только такие запросы. Делегирование подразумевает выполнение запросов только в локальной памяти, а точнее – перенаправление всех запросов на нужного исполнителя так, чтобы это поддерживалось. Обычно этот метод подразумевает наличие специальных потоков-исполнителей, в то время как все остальные потоки выполняют роль роутеров. В таком случае запросы в некотором роде "выстраиваются в очередь" на стороне потока исполнителя, что мжет оказать влияние на распараллеливание запросов в алгоритме системы непересекающихся множеств (и многих других, в которых запросы редко друг-друга блокируют).

1.3. Выводы по главе

Алгоритмы CHM Благодаря статье, сравнивающей всевозможные вариации многопоточных алгоритмов системы непересекающихся множеств, можно сделать вывод какой алгоритм следует брать за основу при работе над созданием NUMA-алгоритма для данной задачи. В последующей работе будет подразумеваться, что мы адаптируем к неоднородной памяти многопоточный алгоритм, использующий эвристики случайного выбора корня, splitting и немедленную проверку родителя.

NUMA-алгоритмы Не существует стандартного способа создания или адаптации алгоритма к NUMA-архитектуре, кроме единственного метода предложенного в статье 2017-го года [3]. Предложенный метод не совсем соответствует специфике системы не пересекающихся множеств, потому плохо пригоден в данном случае. Однако некоторые идеи из этой и других статей могут быть применены при создании NUMA-алгоритмы системы непересекающихся множеств: делегирование, репликация и разделение данных. Главной из которых можно считать идею репликации данных – на нее и будет сделан упор в дальнейшей работе.

2. Адаптация CHM к NUMA архитектуре

В этом разделе я представлю все алгоритмы-адаптации CHM к неоднородной архитектуре, которые я спроектировала и реализовала на языке `C++` .

В основе всех алгоритмов лежит идея репликации данных и основное различие состоит в методе синхронизации реплик. Для каждого алгоритма я выписала основные его преимущества и недостатки (перед базовым многопоточным алгоритмом и перед предыдущими моими идеями).

2.1. NR Black-Box метод (`NRBB`)

Первым этапом моей работы была реализация предложенного в статье 2019 года [3] метода применительно к задаче системы непересекающихся множеств. Итак, метод заключался в следующем:

- на каждом NUMA-сокете будет храниться копия данных
- для синхронизации копий поддерживается журнал операций
- на каждом сокете запросы будут выполняться классическим последовательным алгоритмом по технике flat-combining: в один момент только один поток работает с данными, а остальные только ставят свои запросы в очередь выполнения (в данном случае записывают в журнал операций) и ожидают результат запросы

Чуть подробнее про flat-combining: по умолчанию каждый поток, когда хочет выполнить операцию, дописывает ее в очередь запросов и дальше может начать выполнять накопившиеся запросы, если никто этим еще не занимается или же ожидать результат, если известно, что другой поток занимается выполнением запросов. Так что в нашем случае запрос `Union` представляется как (и аналогично для `SameSet`):

```
void Union(u, v) {
    operation_log.add("union", u, v);
    if (updates_in_progress_on_node[node].try_lock()) {
        // будем выполнять запросы, если еще никто не
        do_all_from_log();
        updates_in_progress_on_node[node].unlock();
    } else {
```

```
8           // ждем пока не выполнено
9       }
10  }
```

Преимущества этого алгоритма:

1. вся работа со структурой происходит в локальной памяти (но не работа с журналом операций)
2. потоки друг другу не мешают, так как с данными в один момент работает только один поток.

Недостатки алгоритма:

1. при выполнении любого запроса нам все равно приходится обращаться к общей памяти (к журналу операций то есть)
2. в один момент на каждом сокете выполняется не больше одного запроса (в то время как в простой многопоточной версии параллельно выполняется множество запросов)

2.2. Flat-combining алгоритм без гарантии на согласованность данных между сокетами (`FC`)

(попытка исправить недостатки прошлого алгоритма)

Операции `SameSet` по сути своей – читающие. Если не выполнять сжатие путей во время работы этой операции, ее можно выполнять сразу, а не ждать пока выполнятся все предыдущие.

Итак, сделаем неблокирующие читающие операции `SameSet`. Возьмем за базу предыдущий алгоритм: будем хранить реплитку данных на каждом сокете и поддерживать общий журнал операций (теперь только модифицирующих структуру, т.е. `Union`), и обработка запросов на объединение не изменится – только один поток в один момент будет обрабатывать накопившиеся в журнале операции. Но для того, чтобы сделать валидные параллельные чтения, заменим последовательный алгоритм известным нам многопоточным с условием, что эвристика сжатия путей применяется только при операциях объединения.

Преимущества такого алгоритма:

1. все запросы обрабатываются в локальной памяти
2. чтения теперь выполняются параллельно

Недостатки:

1. все еще остаются долгие обращения к журналу, но уже только на модифицирующих стр-ру запросах
2. есть ограничение на одно объединение в один момент для каждого сокета
3. нет гарантии на согласованность данных между сокетами (если читатель сделает два одинаковых запроса `SameSet` на разных сокетах, может получить разный ответ)

2.3. Алгоритм с самостоятельной синхронизацией копий (`LateSync`)

В основе у нас репликация данных – на каждом сокете храним копию структуры. С каждой локальной копией может работать базовый многопоточный алгоритм, но все объединения должны происходить не только в локальной копии, но и во всех остальных.

Все операции чтения будут выполняться в локальной копии как обычно. А при объединении поток-инициатор будет выполнять запрос не только в локальной копии, но и во всех остальных тоже, при чем:

- т.к. работа с другими копиями дорогая, при выполнении объединения в чужой памяти, не будет осуществляться эвристика сжатия путей (получим меньше дорогих записей)
- и для сокращения чтений в чужой памяти, будем сначала находить корни объединяемых множеств локально

Стоит заметить, что в этом алгоритме у нас также нет гарантии на согласованность данных, т.к. есть моменты, когда в части копий объединение уже выполнилось, а в остальных – нет.

Преимущества этого алгоритма:

1. все операции чтения работают только с локальной памятью

2. теперь все операции (`SameSet` и `Union`) могут выполняться параллельно

Недостатки:

1. при операции объединения приходится выполнять объединение несколько раз, при чем не в локальной памяти
2. нет гарантии на согласованность данных между сокетами

2.4. Алгоритм с синхронизацией по переменной (`OneUnion`)

Все проблемы с согласованностью данных исходят из ситуации, когда читатель видит на одном сокете результат после обновления, а на другом – до (когда объединение еще не успело выполниться).

2.4.1. Ожидающие операции

Первая мысль, что с этим делать – запретить читателям читать во время выполнения объединения.

Будем поддерживать статус – выполняется ли объединение прямо сейчас. Тогда перед выполнением любой операции будем ждать пока этот статус не будет отрицательным и только тогда выполнять новый запрос. Тогда реализация будет выглядеть так:

```
bool union_in_progress;

void wait_previous() {
    while (union_in_progress.load()) {
        // wait
    }
}

bool SameSet(u, v) {
    wait_previous();
    return do_sameset(u, v);
}

void Union(u, v) {
    wait_previous();

    while {
```

```
18          // пробуем начать выполнять наше объединение
19          if (union_in_progress.CAS(false, true)) {
20              // если получилось, приступаем
21              break;
22          } else {
23              // иначе снова ждем чужое
24              wait_previous();
25          }
26      }
27
28      do_union(u, v); // выполняем уже обычное объединение на каждом сокете
29      union_in_progress.store(false); // отмечаем, что закончили объединять
30  }
```

Преимущества такого алгоритма:

1. все операции чтения работают только с локальной памятью
2. алгоритм линеаризуем, т.к. все читатели получают информацию после выполнения объединений

Недостатки:

1. в один момент выполняется только одно объединение
2. объединение делается также и в чужой памяти
3. читатели тоже ждут

Этот алгоритм также можно соптимизировать, применив распространенный в многопоточных алгоритмах подход, когда один поток "помогает" другому выполнить или завершить операцию. В нашем случае, его можно применить так: в функции `wait_previous` вместо ожидания поток начнет выполнять объединение на своем сокете. Для этого кроме статуса потребуется хранить какое именно объединение сейчас происходит.

Преимуществом такой оптимизации будет то, что потоку-инициатору, возможно, не придется выполнять объединение в чужой памяти, т.к оно произошло без его участия. Но может случиться и так, что очень много потоков будут пытаться выполнятьб одно и то же объединение, что может даже ухудшить ситауцию.

Однако, на практике выяснилось, что эта оптимизация не улучшает и не ухудшает производительность алгоритма.

2.4.2. Параллельные чтения

Проблема предыдущих алгоритмов заключалась в том, что все операции "ожидали" завершения предыдущих объединений. Однако, это узкое место можно исправить, сделав без ожидания хотя бы операции чтения (`SameSet`).

Сделаем так, чтобы объединение, которое еще не произошло на всех сокетах было не видимо читателям. Для этого модифицируем способ получения предка (а значит и корня, т.е. представителя множества):

- будем поддерживать статуст текущего объединения в `union_in_progress`
- у каждого элемента будем кроме предка хранить дополнительно статус действительно ли это текущий предок (или тот, что будет после текущего объединения)

Тогда при выполнении операции `Union` нужно:

1. отметить, что объединение началось
2. выполнить объединение на каждом сокете и при этом пометить, что оно еще не завершилось и новый предок окончательно не установился (в массиве предков)
3. отметить статус объединения как выполненный
4. проставить на каждом сокете, что предок окончательно поменялся

И функция получения предка теперь будет выглядеть как:

```
<int, bool> parents[]; // храним предка и статус действительно ли это уже предок
<int, int, bool> union_in_progress; // два множества, которые нужно объединить
                                    // и статус закончилось ли это объединение

int get_parent(u) {
    parent, status = parents[u].load();
    if (status.is_done()) { // если родитель уже утвержден
        return parent;  // его и возвращаем
    } else { // если оказалось, что проставлен новый родитель
        union_u, union_v, union_status = union_in_progress.load();
        if (parent != union_u && parent != union_v) { // если сейчас идет
            // другое объединение, значит затрагивающее нас уже прошло
```

```
13              // и просто не успел проставиться правильный статус у предка
14              return parent;
15          } else { // если сейчас идет объединение с нашим множеством
16              if (union_status.is_done()) { // если отмечено, что оно закончилось
17                  // значит просто не успел проставиться правильный статус у предка
18                  return parent;
19              } else { // если оно все еще идет
20                  // значит предок не актуален
21                  // заметим, что вся эта ситуация может возникнуть
22                  // только с корнем дерева, то есть представителем множества
23                  // если объединение еще не случилось, значит корень все еще корень
24                  return u;
25              }
26          }
27      }
28  }
```

Преимущества такого алгоритма:

1. теперь операции чтения выполняются без ожидания
2. алгоритм все еще линеаризуем

Недостатки:

1. все еще в один момент выполняется только одно объединение
2. объединение делается также и в чужой памяти
3. читателям приходится читать общую память (`union_in_progress`)

2.5. Алгоритм с (почти) параллельными объединениями (`llUnions`)

Заметим, что операции объединения мешают друг другу только если работают с одними и теми же множествами (если хотя бы одно из множеств для объединения повторяется). Надо научиться выполнять параллельно объединения, которые друг другу не мешают!

Оставим прошлую идею с параллельными чтениями и сохранением знания о том, какое объединение сейчас просиходит. И переформатируем это знание в новое – для каждого множества, а точнее для каждого корня будем хранить находится

ли оно в объединении прямо сейчас (то есть `union_in_progress` будет не просто глобальной переменной, а массивом, где для каждого элемента хранится статус). И если находится, соответственно возвращать предка до объединения, а если нет – реального (все как в прошлом алгоритме).

Соответственно, операция объединения будет проверять находятся ли интересующие ее множества уже в другом объединении. И если не находятся, выполнять это объединение, а иначе – ждать пока закончатся предыдущие.

Преимущества такого алгоритма:

1. параллельные операции чтения без ожидания
2. параллельные независимые объединения (на практике большинство объединений именно такие)
3. алгоритм все еще линеаризуем
4. в сравнении с предыдущим алгоритмом хорошо еще и то, что все потоки-инициаторы объединения теперь не стучатся в одну и ту же переменную

Недостатки:

1. некоторые объединения все же ожидают завершения предыдущих
2. объединение делается также и в чужой памяти
3. читателям приходится каждый раз читать общую память (`union_in_progress`)

2.6. Техника разделения данных

До этого все алгоритмы основывались на том, что на каждом сокетк хранится своя копия структуры и алгоритм эти копии синхронизирует некоторым способом – при запросе на объединение, оно (так или иначе) выполнится на каждом сокете. Однако, исходные данные могут быть распределены так, что нет необходимости поддерживать все объединения на каждой из копий. Если точнее, на некоторых сокетах даже никогда и не потребуется информация о части данных.

Например. Пусть структура должна поддерживать связность городов в железнодорожной сети, где элементами множеств будут города. Запросы будут генерироваться из разных стран и за каждой страной будет закреплен конкретный сокет,

при этом запрос из страны A может запрашивать/модифицировать информацию только о городах и дорогах страны A. В такой конфигурации, данные о ситуации, например, в Австралии достаточно поддерживать только на одном сокете.

Это и есть основная идея – не будем на сокете хранить всю структуру, а только ее часть. Тогда для каждого элемента будем хранить информацию о том, какие сокеты в нем "заинтересованы" и при обновлении данных (объединении множеств), будем обновлять только те копии, которым "интересны" эти данные.

При чем это именно техника, которую можно применять к разным алгоритмам.

Преимущества:

1. меньше обновлений
2. меньше данных хранится на каждом сокете

Недостатки:

1. дополнительные затраты на проверку "кому интересно", которые не оправданы, если пересечение интересов слишком большое

Кроме того, если заранее не известно разделение "интересов", его придется поддерживать динамически: когда на сокете впервые появляется запрос к данным, которые раньше на нем не содержались, и х придется откуда-то достать, скопировать с другого сокета – и это дополнительные затраты.

2.6.1. Применение

(В нашем случае)

Если все интересы известны заранее, алгоритм при применении техники меняется минимально: при запросе на объединение нужно объединять не во всех сокетах, а только в тех, кому это объединение интересно.

Интересен случай, если интересы заранее не известны.

Тогда все еще остается объединение только в нужных сокетах. Но еще и перед любым запросом нужно проверить, что информация о нужных множествах на сокете уже есть. И получить ее, если еще нет. Получить можно, скопировав с сокете, на котором эта информация есть.

В нашей задаче может потребоваться информация о множестве, содержащем конкретный элемент `u`, получить ее можно, посмотрев какому сокету она интересна и скопировав весь путь в дереве от вершины `u` до корня. Нам никогда не интересно все дерево, т.к. мы работаем только с корнями – объединяем именно их и их считаем представителями множеств, так что такой информации должно быть достаточно.

Функция копирования зависит от алгоритма, но общая структура такая:

1. узнать на каком сокете информация есть
2. пройти весь путь от вершины до корня, попутно копируя его себе (получается, заодно скопируется не только информация об запрашиваемой вершине и корне, но и все на пути)
3. отметится, что теперь на этом сокете тоже должна быть актуальная информация об этом множестве

Чтобы не потерять изменения, на время копирования, объединения этого множества придется приостановить – добавляется к **недостаткам** этой техники.

Получается, что для этой техники нам дополнительно нужно хранить информацию об интересах и постоянно к ней обращаться (в общей памяти).

В целях оптимизации, я еще дополнительно в локальной памяти сокета по каждому множеству храню является ли текущий сокет единственным владельцем информации о нем.

2.7. Выводы по главе

Я представила несколько вариантов алгоритмов в порядке от применения известного метода, адаптации его под нашу задачу и, затем, оптимизации различными методами. Все алгоритмы представлены в подглавах 2.1 - 2.5.

Основное различие алгоритмов строится на методах синхронизации реплик и можно выделить основные отличительные черты:

- Возможность/невозможность параллельно осуществлять операции `SameSet`
- Возможность/невозможность параллельно осуществлять операции `Union`

- Различные гарантии на согласованность данных
- Осуществление объединений только в своей памяти или в чужой

Кроме того, в 2.7 представлена техника, которую можно применить к любому из этих алгоритмов.

Общая черта всех алгоритмов — они подразумевают репликацию данных, а значит, все объединения выполняются не один раз, а несколько (по числу сокетов, если нет разделения данных); в то же время они облегчают операции `SameSet` тем, что выполняют их в своей быстрой локальной памяти. Получается, мы утяжелили один вид операций и ускорили другой.

В условиях нашей задачи это оправдано: почти всегда на одну операцию объединения приходит несколько операций `SameSet`, не говоря уже о том, что сама операция `Union` подразумевает проверку не находятся ли элементы уже в одном множестве.

Кратко зафиксируем алгоритмы, которые получились:

1. `NRBBM` алгоритм (2.1): на каждом сокете в один момент только одна операция и все запросы выстраиваются в очередь, в порядке которой затем и выполняются; все операции выполняются в локальной памяти, общая используется только для очереди
2. `FC` алгоритм (2.2): параллельные чтения, но на каждом сокете в один момент только одно объединение + все та же очередь, только состоящая из операций объединения; все операции выполняются в локальной памяти, общая используется только для очереди
3. `LateSync` алгоритм (2.3): копии синхронизуются когда-нибудь, но в моменте нет гарантии на согласованность данных, все операции выполняются параллельно; операции объединения каждый раз выполняются в своей локальной памяти и в памяти всех остальных сокетов
4. `OneUnion` алгоритм (2.4.2): параллельные чтения в локальной копии, но использующие общую память, одно объединение в один момент; операции объединения каждый раз выполняются в своей локальной памяти и в памяти всех остальных сокетов

5. `llUnions` алгоритм (2.5): параллельные чтения в локальной копии, но использующие общую память, объединения могут выполняться параллельно, если работают с разными частями структуры; операции объединения каждый раз выполняются в своей локальной памяти и в памяти всех остальных сокетов

Каждый следующий алгоритм – некоторое развитие предыдущего, но не всегда однозначное улучшение:

- `FC` – прямое развитие `NRBBM`. Здесь вопрос в том, что задача CHM подразумевает много читающих операций, которые хорошо параллелятся и позволяют улучшить общее время работы. Все остальное не меняется, разве что в реализации появляются `atomic` переменные для хранения многопоточной структуры, которые несколько утяжеляют все операции.

  Коме того, в алгоритме `FC` теряется гарантия на согласованность данных.

- `LateSync` (как и `FC`) – алгоритм без гарантии на согласованность данных в моменте, в котором при этом разрешено параллельно объединять все, но объединения работают в чужой памяти.

- `OneUnion` – алгоритм, в котором уже сохраняется гарантия на согласованность данных, запросы на чтение выполняются параллельно, но только одно объединение в один момент.

- `llUnions` – прямое улучшение предыдущего алгоритма, разница только в объеме дополнительных данных, которые в итоге позволяют нам сделать некоторые (на практике – большую часть) объединений параллельно

Ожидается, что все алгоритмы, в которых есть строгое ограничение на число одновременных запросов – проигрывают остальным, т.к. в задаче CHM запросы редко пересекаются и могут выполняться параллельно. Далее мы это и проверим на практике.

Получилось 5 алгоритмов, к каждому из которых еще можно применить технику разделения данных и получить новый. Однако, в ходе экспериментов, некоторые исходные алгоритмы показали свою неэффективность, так что технику разделения я применила только к некоторым из них, если точнее, к одному линеаризуемому алгоритму и к одному алгоритму без гарантии на согласованность данных. Подробнее об этом – в следующей главе.

3. Сравнение и анализ эффективности алгоритмов

3.1. Сценарии сравнения

Так как система непересекающихся множеств наиболее широко применяется в графовых алгоритмах, мы будем тестировать алгоритмы на задачах с ними связанных. А именно на задаче динамического определения компонент связности и задаче построения минимального остовного дерева.
Далее подробнее о сценариях тестирования по каждой из задач:

3.1.1. Компоненты связности

Поиск и поддержание компонент связности – самое распространенное применение алгоритма системы непересекающихся множеств, которое встречается на практике как в чистом виде, так и в качестве подхадачи в более сложных алгоритмах.
Поэтому в данной работе, основной упор сделан именно на этом сценарии использования.

Формат запуска:

- на вход получаем граф
- случайно делим ребра графа на две категории: 1) те, что действительно будут использоваться как ребра графа, то есть по которым будет вызываться операция `Union` 2) те, что будут использоваться как проверка находятся ли вершины в одной компоненте связности, то есть для операции `SameSet`
- получается, ребра стали запросами, делим их случайно между потоками и запускаем: каждый поток выполняет свой набор запросов по-очереди

При чем у такого запуска есть разные вариации:

Различный баланс операций Как, описывалось выше, большая часть алгоритмов эффективно выполняет операции чтения (`SameSet`), но делает дополнительную работу на операциях модификации структуры (`Union`). Поэтому важным параметром будет баланс операций чтения и модификации.

Соответственно, для такой вариации необходимо разбивать ребра на категории со строго заданным соотношением.
Ожидается, что если 100% запросов — это `SameSet`, предложенные NUMA-алгоритмы буду эффективнее известных многопоточных алгоритмов. А при обратной ситуации (100% операций `Union`) NUMA-алгоритмы на каждой операции будут выполнять дополнительную работу для синхронизации, что может заметно снизить их эффективность.

На самом деле, из-за специфики предложенных NUMA-алгоритмов, а именно сложных операций объединения и легкий операций чтения, именно проверка производительности при разном соотношении операций – самая важная и интересная часть. Все последующие вариации опциональны, но работу при разном балансе операций будем проверять во всех бенчмарках по задаче компонент связности.

Начальное состояние До добавления новых ребер граф не всегда пустой. Проверим эффективность работы алгоритмов в зависимости от того, какой граф был изначально до выполнения запросов.

Для этого сценария входной граф разбивается на две части: первая будет использоваться для заполнения структуры до замера времени (то есть инициализации графа и его стартовых компонент связности), а на второй будет запускаться описанный выше бенчмарк.

Разделение вершин и запросов между сокетами Эффективность основанных на неполном обновлении и хранении алгоритмов зависит от пересечения "интересов". Важно посмотреть на зависимость производительности алгоритмов от размера этого пересечения – для этого распределим вершины графа между сокетами и запросы между потоками так, чтобы пересечение было заданного размера.

Для этого был реализован итеративный алгоритм разделения графа на несколько частей, минимизирующий число ребер между частями до нужного параметра. На каждом этапе алгоритма выбирается вершина, перекладывание которой в другую часть даст наибольший эффект. При этом поддерживается условие, что размеры частей отличаются не более, чем на 10%. Алгоритм завершается, когда достигается заданный процент ребер между частями.

Однако, на реальных графах маленький процент труднодостижим, поэтому для таких случаев была реализована отдельная генерация случайного графа с заданным пересечением.

3.1.2. Остовные деревья

Еще одно из распространненых применений системы непересекающихся множеств (которое первым приходит в голову) — поиск минимальных остовых деревьев.
Так что для второго сценария тестирования я реализовала многопоточную версию алгоритма Борувки [] для построения минимального остовного дерева:

- добавляем ребра в остов итеративно, пока он не построится:
- независимо для каждой вершины находится наименьшее по весу из ее ребер, которое не будет образовывать цикл, если его добавить в остов
- затем для каждого связного множества вершин определяется минимальное выходящее из него ребро
- и все эти ребра (то есть по ребру от каждого множества) добавляются в остовное дерево, при этом выполняется объединение компонент связности (то есть множеств в СНМ)

Здесь будем сравнивать время работы алгоритма построения остовного дерева в зависимости от использованной реализации системы непересекающихся множеств и от входных данных (графа).

3.1.3. Входные данные

Для тестирования алгоритмов, был реализован алгоритм случайной генерации графов по заданным параметрам (число вершин, ребер, число компонент связности и, для тестирования алгоритмов с техникой разделения, по числу сокетов и размеру пересечения между частями). Кроме того были использованы некоторые естественные графы разной природы:

- Графы дорог
  - `Roads0`: `V=24млн E=58млн` (граф дорог США)
  - `Roads1`: `V=2млн E=5.5млн` (граф дорог Калифорнии)

- Web-графы
  - **Web0**: V=41млн E=1150млн (web-граф 2004 года)
  - **Web1**: V=0.7млн E=7.6млн
    (граф связей web-страниц на доменах berkely.edu и stanford.edu)
- Социальные графы
  - **LiveJournal0**: V=4.8млн E=69млн (граф социальной сети LiveJournal)
  - **LiveJournal1**: V=4млн E=3.4млн (граф социальной сети LiveJournal)
  - **StackOverflow**: V=2.6млн E=63.5млн
    (граф взаимодействий на сайте stackoverflow.com)
  - **Pokec**: V=1.6млн E=30.5млн (граф социальной сети Pokec)
  - **Orkut**: V=3млн E=106.3млн (граф социальной сети Orkut)
- Граф коллабораций актеров **Hollywood**: V=1.1млн E=56.3млн

3.2. Сравнение и анализ работы алгоритмов

Техническая информация Все алгоритмы и бенчмарки были реализованы на языке C++ и запускались на машинке с процессором Intel Xeon Gold и 4 NUMA-сокетами по 18 ядер в каждом.

3.2.1. Компоненты связности

Алгоритмы NRBB, FC, OneUnion заслуживают отдельного рассмотрения, так как в каждом из них есть строгое ограничесние на число выполняющихся операций в одимн момент: в NRBB одна операция (любого типа) на сокете, в FC одно объединение на сокете, а в OneUnion одно объединение на всю структуру.
Ожидается, что эти алгоритмы будут проигрывать базовому многопоточному при большом числе операций Union, но на операциях SameSet выигрывать за счет того, что они происходят в локальной памяти.

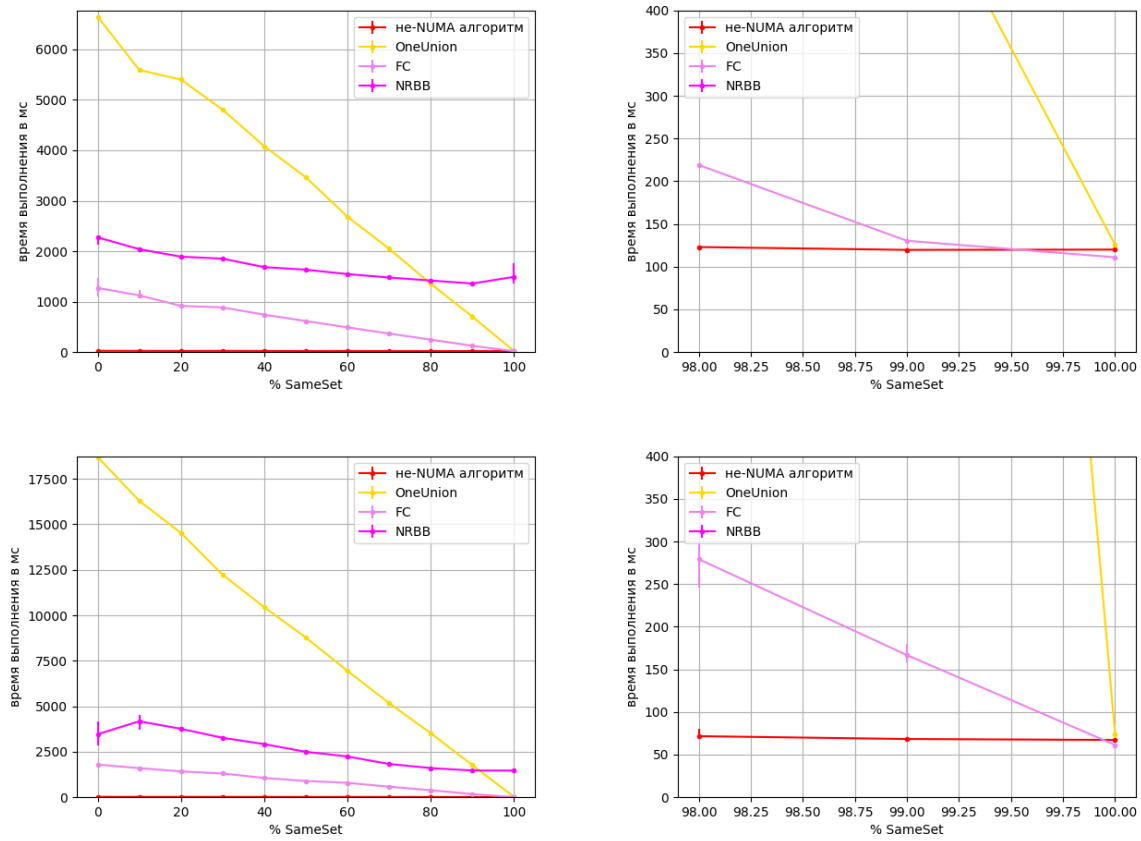

Рис. 1: Запуск алгоритмов `NRBB, FC, OneUnion` для сценария динамического определения компонент связности и различным балансом операций `SameSet` и `Union` на 2 (сверху) и 4 (снизу) NUMA-сокетах. На графиках слева запуск проводился на проценте операций `SameSet` от 0 до 100 с шагом 10; на графиках справа – от 98 до 100 с шагом 1; запуски производились на разных входных данных

На практике (рис. 1) видим, что алгоритмы проигрывают не адаптированной к NUMA-архитектуре версии:

- Алгоритм `NRBB`, в котором все операции ожидают выполнения предыдущих, проигрывает в любой ситуации и отрабатывает заметно дольше.
- Алгоритм `OneUnion` работает хуже всех других на проценте операций `Union` от 20; в ситуации меньше 20% отрабатывает лучше алгоритма `NRBB`, но хуже `FC` и не адаптированной многопоточной версии. При чем даже в случае только операций `SameSet` он проигрывает остальным алгоритмам благодаря тому, что в реализации кроме чтения локальных данный проверяет статус происходит ли объединение (в общей памяти).

- Алгоритм `FC` работает лучше других предложенных алгоритмов, но практически всегда хуже бзового многопоточного алгоритма – начинает работать эффективнее только когда операций объединения практически нет. И в ситации 100% `SameSet` в некоторых выигрывает до 25% при запуске на 2 сокетах и до 35% на 4х (приложение 1).

Алгоритмы `LateSync` и `llUnions` не требуют у операций ожидания выполнения предыдущих (почти всегда), так что рассмотрим их отдельно на сценарии определения компонент связности при разном балансе операций `SameSet` и `Union`, сначала результаты при запуске на двух NUMA-сокетах (рис. 2)

Как и ожидалось, видим проигрыш на большом проценте операций `Union`, для которых алгоритмы совершают объединение несколько раз (на каждом сокете) и выигрыш на большом проценте операций `SameSet`.

Если смотреть на результаты на случайных графах, алгоритм `LateSync` начинает выигрывать базовый многопоточный алгоритм при проценте операций `SameSet` от 70% и выше (а на некторых примерах даже от 60), а алгоритм `llUnions` – начиная с 80-90%. Однако, на естественных графах ситуация не всегда повторяется — почти всегда вигрыш начинается позже и природа графиков явно зависит от структуры входного графа (рис. 3 и рис. 14 в приложении 2).

Максимальный выигрыш достигается на 100% запросов `SameSet` и может состовлять до 25%, в то время как максимальный проигрыш получается в ситуации, когда запросы только на объединение и алгоритм `llUnions` может проигрывать до 2х раз, а алгоритм `LateSync` до 1.6 (по данным запусков на описанных графах).

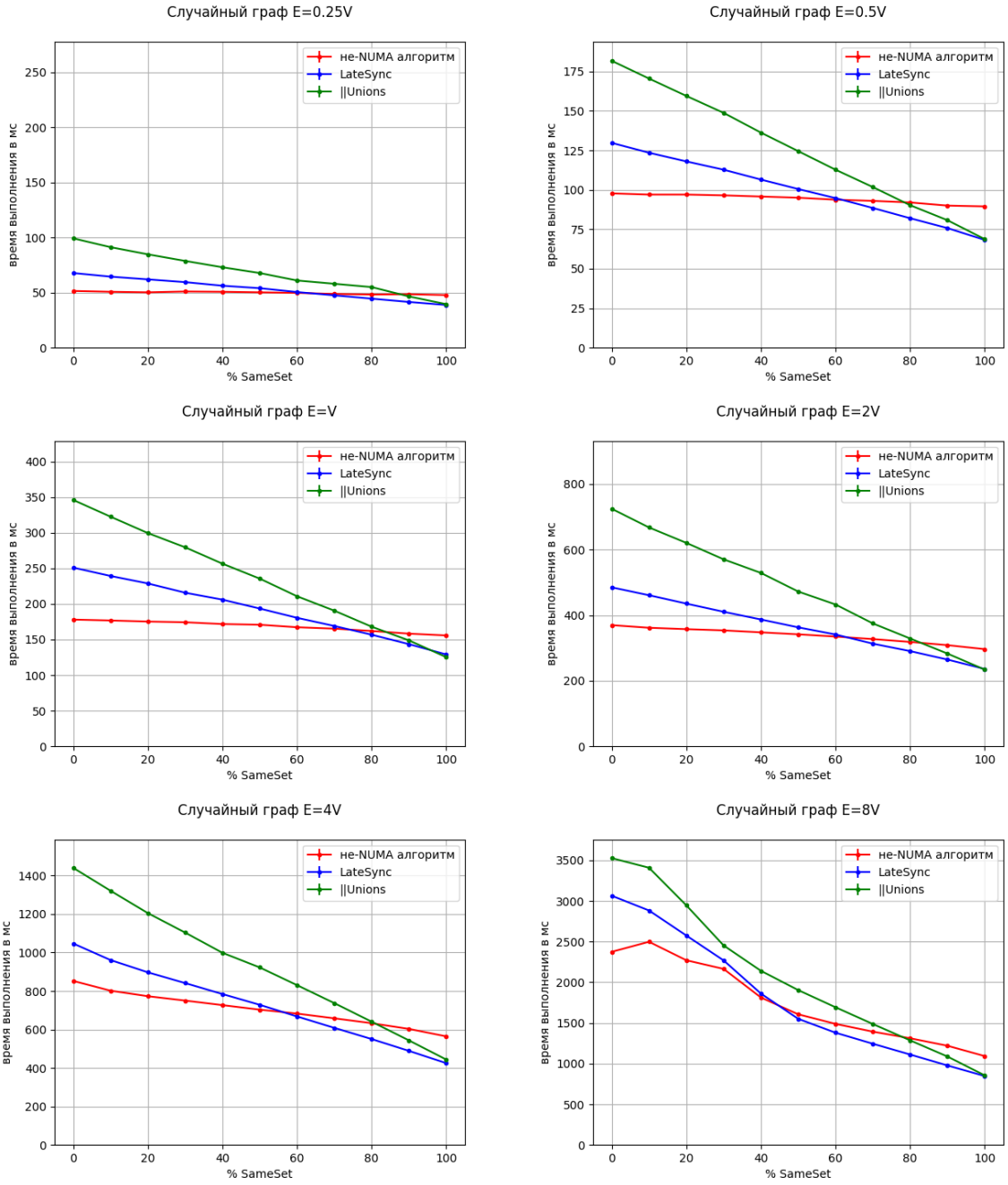

Рис. 2: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 2x NUMA-сокетах на задаче определения компонент связности с разным балансом операций чтения `SameSet` и объединения `Union`.

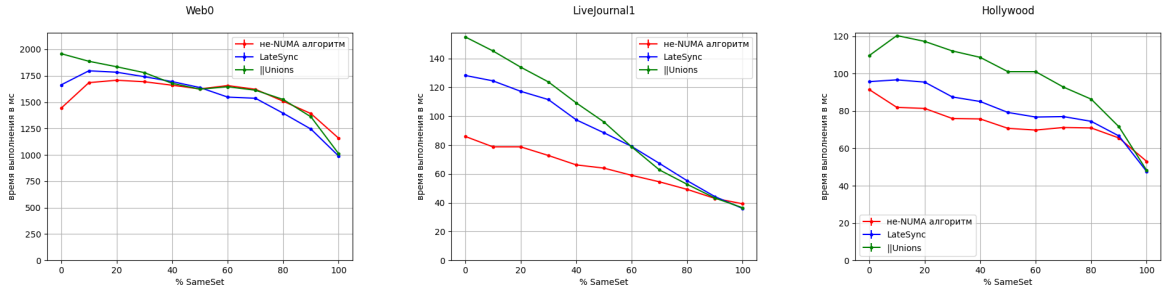

Рис. 3: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 2х NUMA-сокетах на задаче определения компонент связности с разным балансом операций чтения `SameSet` и объединения `Union` на естественных графах.

Интересно, что в некоторых кейсах алгоритм `llUnions` выигрывает у алгоритма `LateSync`, который на самом деле проще первого. Предполагаю, что так случается из-за повышенного спроса на одни и те же данные во втором случае, так как эту ситуацию можно заметить только в плотных графах. В то время как в алгоритме `llUnions` некоторые объединения будут ожидать, в `LateSync` все они будут пытаться выполниться и, при затрагивании одних и тех же данных, мешать друг другу.

Если рассматривать ситуацию при запуске на 4х сокетах (рис. 15 в приложении 2), здесь результаты схожи — выигрыш начинается на тех же долях и достигает уже до 30%. При чем максимальное замедление так же остается до 2х раз, что даже лучше ожиданий, ведь теперь надо обновлять не две копии, а четыре.

Можно также заметить, что результаты на графах с одинаковым соотношением чила ребер и вершин очень похожи (в нашем случае, это графы дорог и соответствующие им случайные) (рис. 4).

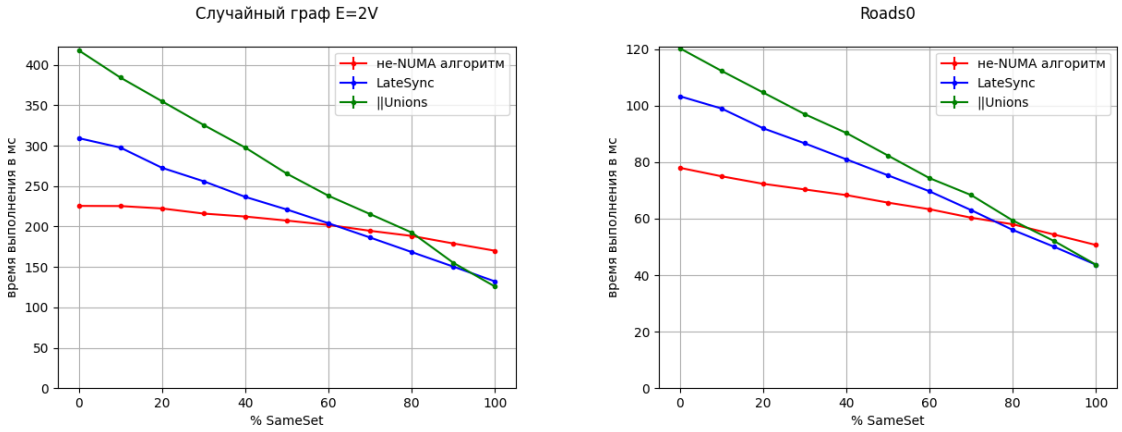

Рис. 4: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 4х NUMA-сокетах на задаче определения компонент связности с разным балансом операций чтения `SameSet` и объединения `Union`.

Техника разделения вершин в нашем случае была применена только к "успешным" алгоритмам `LateSync` и `llUnions`.

Сначала рассмотрим случай полного разделения графа между сокетами (без пересечения) (рис. 5).

Здесь для сравнения реализована дополнительная версия системы непересекающихся множеств, представляющая собой некоторый идеал, лучше которого наши алгоритмы не отработают: на каждом сокете будет работать независмая копия системы непересекающихся множеств, которая будет работать со своей копией данных без синхронизации между этими копиями; то есть все операции выполняются локально и нет проверки на необходимость синхронизации данных о множестве, котороая предполагается в технике разделения данных.

В зависимости от графа этот алгоритм выигрывает у базового многопоточного на 25-35% при запуске на двух сокетах и на 30-46% при запуске на четырех.

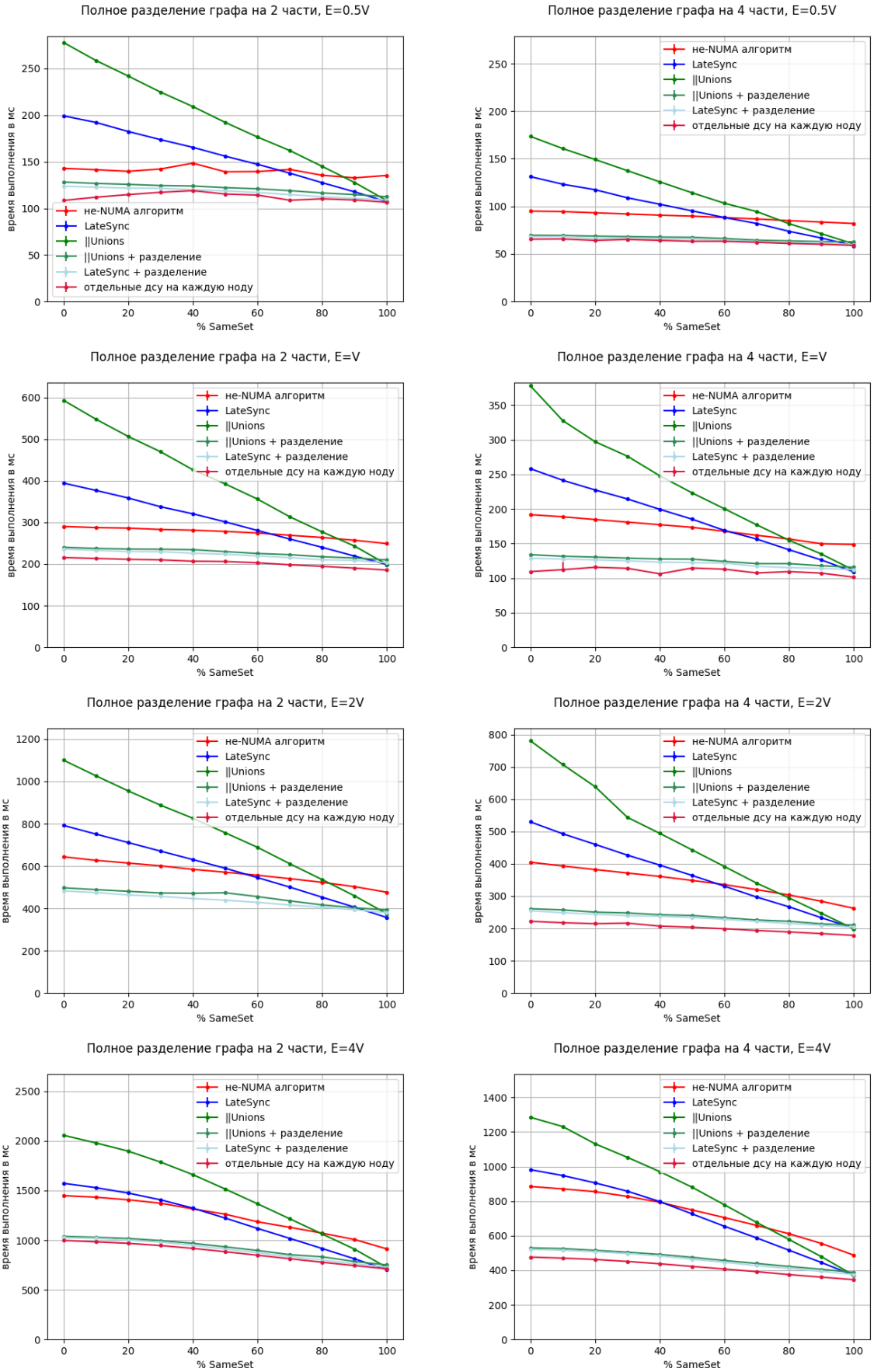

Рис. 5: Сравнение алгоритмов в сценарии разделения вершин графа и запросов на непересекающиеся части. Сравниваются алгоритмы `LateSync` и `llUnions` в совокопности с примененной к ним техникой разделения данных. Результаты представлены для запусков на 2х (слева) и 4х (справа) сокетах.

Заметим, что версии алгоритмов `LateSync` и `llUnions` с использованием техники разделения ведут себя практически одинаково, только версия основанная на `llUnions` на 1-5% отстает от версии, в основе которой алгоритм `LateSync` из-за более долгой модифицированной функции получения предка.

Далее будем сравнивать алгоритм основанный на `LateSync` с остальными, помня, что второй алгоритм с разделением проигрывает нра несколько процентов у него.

В сравнении с "идеальным" алгоритмом с полным разделением без синхронизации, алгоритм `LateSynnc + разделение` проигрывает на 8-10% при запуске на двух сокетах и на 5-14% при запуске на четырех. А в сравнении с простым многопоточным — выигрывает на 18-30% при запуске на двух сокетах и на 24-42% при запуске на четырех.

И если сравнивать предложенные алгоритмы с самими собой с техникой разделения и без нее, можно заметить что при большом числе операций `SameSet` алгоритмы без техники разделения немного выигрывают. Так происходит из-за того, что в алгоритмах с разделением приходится дополнительно проверять информацию есть ли конкретные данные на сокете перед началом работы с ними.

Теперь посмотрим как производительность зависит от размера пересечения:

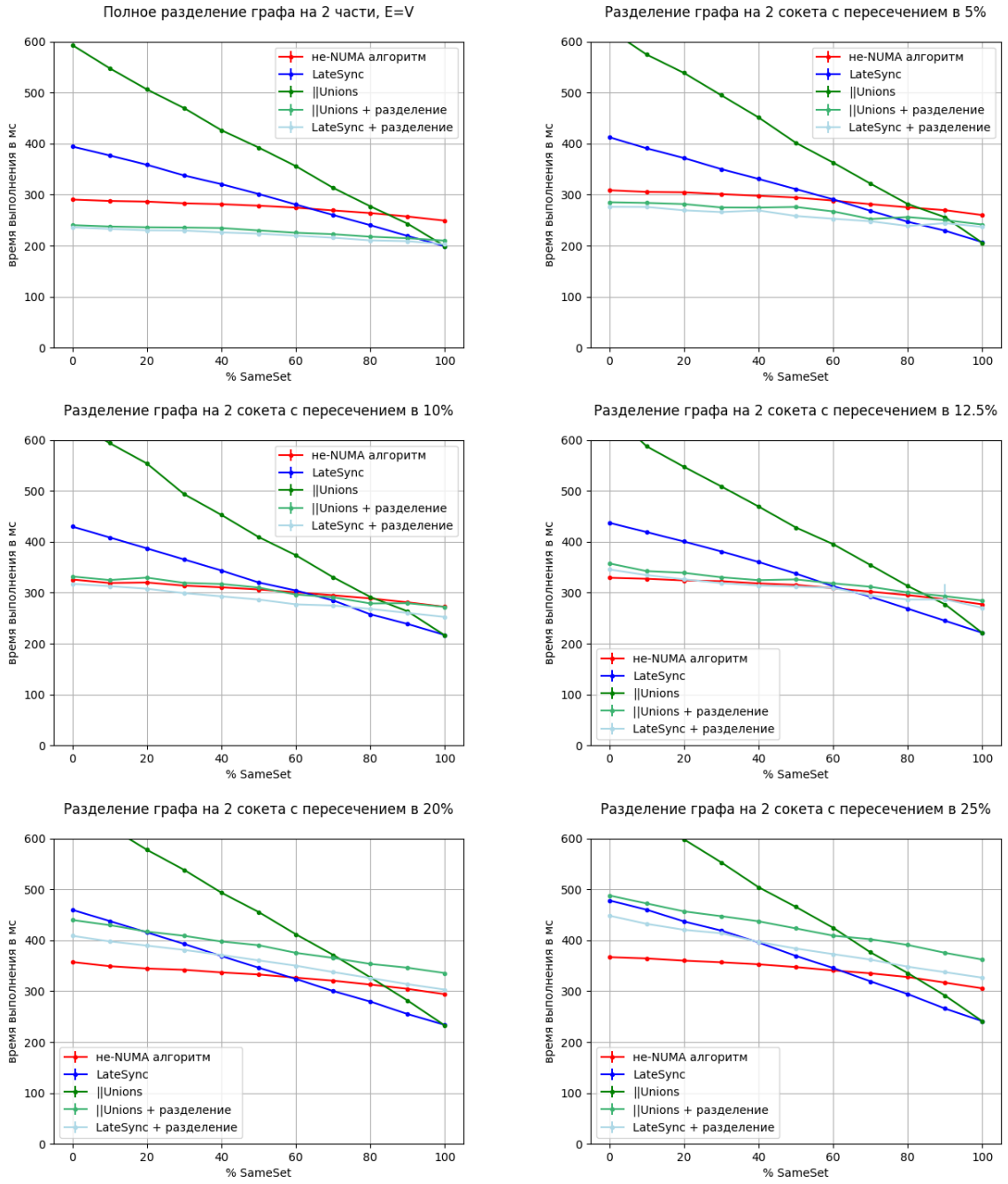

Рис. 6: Сравнение времени работы алгоритмов с разделением данных на сценарии динамического поддержания компонент связности в зависимости от размера пересечения данных между разными сокетами (запуск на машинке с 2 NUMA-сокетами)

Видим, что уже при 10% пересечения алгоритм `llUnions + разделение` начинает проигрывать базовому многопоточному алгоритму из-за дорогой синхронизации. Алгоритм `LateSync + разделение` выступает несколько лучше и на 10% пересечения все еще работает не хуже базового, однако на следующем запуске с 12.5%

пересечения уже уступает ему, особенно при большом числе операций `Union`.

И чем больше пересечение, тем выгоднее может быть использование алгоритма без техники разделения. Например, такая ситуация получилась при запуске на графе с пересечением частей в 25% — алгоритм `LateSync` почти на всем запуске отрабатывает лучше без техники разделения, чем с ней.

Можно также заметить, что время работы алгоритма зависит от числа операций объединения, так как при обработке таких запросов скачивание недостающей информации становится сложнее. Однако, разница не такая явная, как в алгоритмах без разделения и при большом проценте операций `SameSet` алгоритмы без разделения выигрывают у своих версих с разделением.

Далее я приведу результаты работы алгоритмов на естественных графах: я разделила графы так, чтобы получилось наименьшее возможное пересечение при условии, что размеры частей отличаются не более, чем на 10%; далее запустила стандартный бенчмарк, но с фиксированным процентом операций `SameSet`: 20%, 50% и 80% (приложение 3). Ниже приведен наиболее интересный случай с 80% (рис. 7) – соотношение результатов базового алгоритма и алгоритмов с разделением всегда похожи, но интересно еще и посмотреть с какого момента алгоритмы без разделения могут начать выигрывать у аналогичных алгоритмов с разделением.

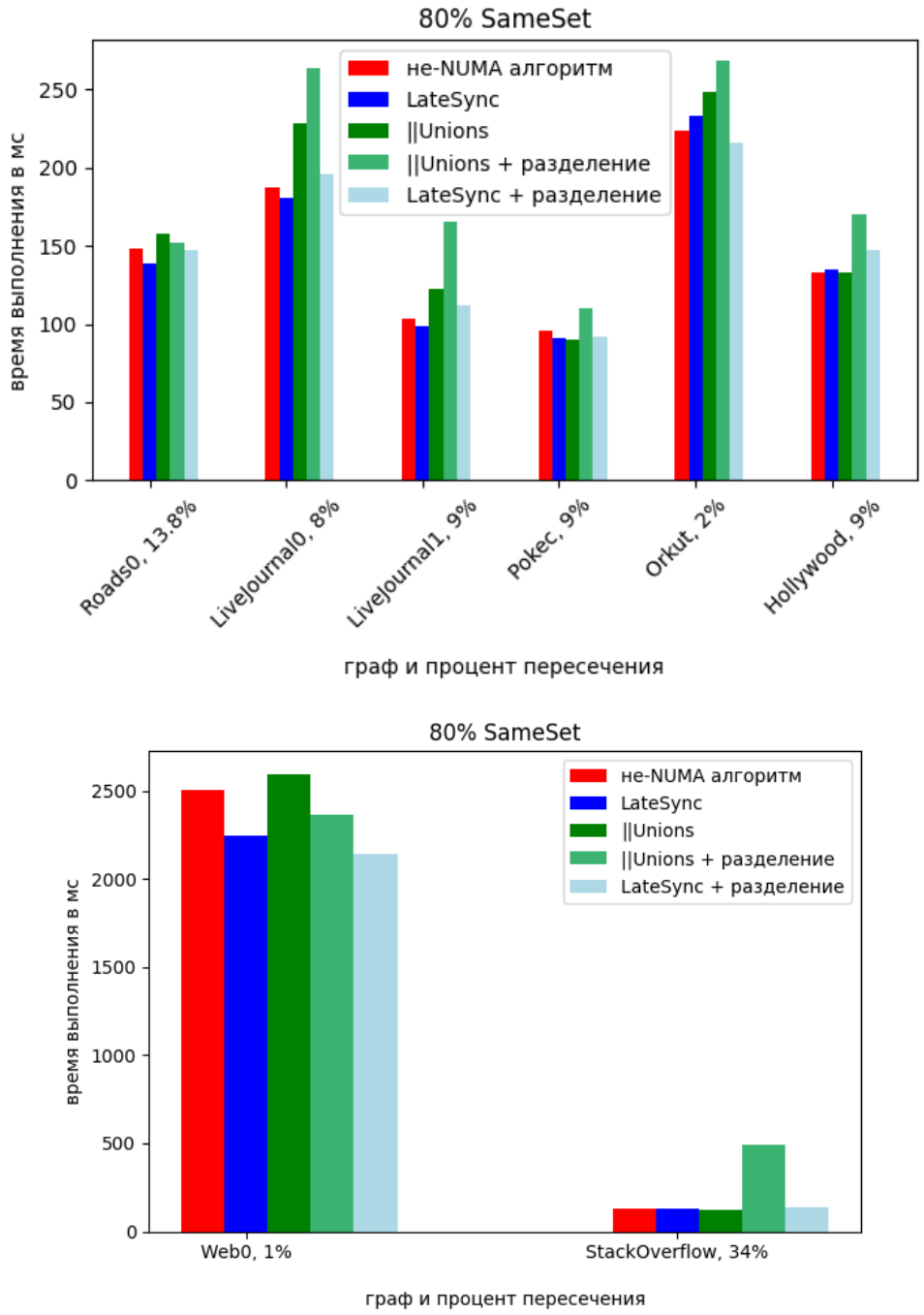

Рис. 7: Сравнение времени работы алгоритмов с разделением данных на естественных разделенных графах (запуск на машинке с 2 NUMA-сокетами)

К сожалению, на момент работы с этим сценарием тестирования, у меня уже не было доступа к машине с 4 NUMA-сокетами, поэтому все результаты представлениы только для 2х.

Только два графа оказалось возможно разделить на мало-пересекающиеся части (до 2х процентов): это web-граф `Web0` и граф социальной сети `Orkut`. И результаты на этих графах оказались очень разными:

- Для графа `Web0` оба алгоритма с разделением показали себя хорошо: алгоритм `LateSync + разделение` отработал лучше всех остальных, а алгоритм `llUnions + разделение` оказался лучше базового многопоточного и своей версии без разделения, однако на запуске с большим количеством операций `SameSet` показал себя хуже алгоритма `LateSync` (без техники разделения)

- В случае же графа `Orkut` алгоритм `LateSync + разделение` повел себя все так же хорошо, в то время как алгоритм `llUnions + разделение` на всех этапах отработал хуже базового многопоточного алгоритма;

  Так произошло из-за того, что алгоритм разделения графа считает процент пересечения частей по ребрам (сколько ребер лежат не внутри части); В данном графе получилось, что небольшой процент ребер затронул большой процент вершин (синхронизация потребовалась не только тем вершинам, которые непосредственно связаны с ребром, но и тем, что просто лежат с ними в одной компоненте связности)

Один из графов (`StackOverflow`) удалось разделить только на части с пересечением в 34% – и на нем ожидаемо алгоритм `llUnions + разделение` проигрывает остальным (даже аналогичному алгоритму без разделения), однако алгоритм `LateSync + разделение` показал себя не хуже оставльных.

У графа `Roads0` лучшее разделение содержало процент пересечения 13.8, при котором алгоритмы с разделением данных также показали себя хорошо: `LateSync + разделение` во всех случаях выигрывает на пару процентов, а аолгоритм `llUnions + разделение` – немного проигрывает.

Все остальные графы в разделенном виде остались с пересечением в 8-9%, интересно, что все это оказались социальные графы, но результаты на них получились разные. В большинстве случаев алгоритм `LateSync + разделение` не проигрывает остальным, а вот алгоритм `llUnions + разделение` ведет себя нестабильно и иногда отрабатывает хуже других.

Начальное состояние структуры должно повлиять на результаты: если граф изначально не пустой и структура CHM уже как-то заполнена, алгоритмы с раз-

делением данных будут иметь больше информации об ”интересах” сокетов уже в начале и потребуют меньше скачиваний; для остальных алгоритмов такого сильног влияющего фактора нет, однако может произойти больше сжатий путей и как-то повлиять на результат.

В данном сценарии все запросы делятся на две части: те, что выполняются до замера времени и наполняют структуру и те, на которых происходит обычный замер.

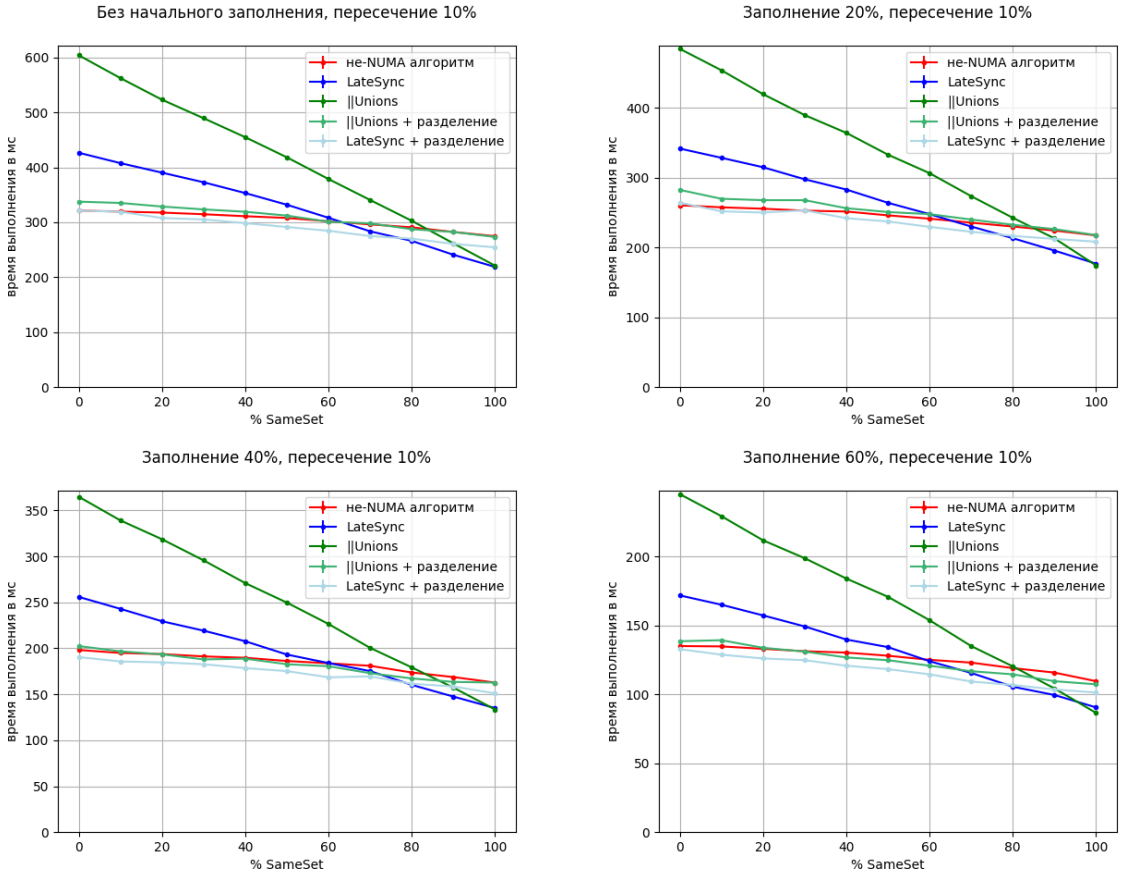

Рис. 8: Сравнение времени работы алгоритмов в зависимости от уровня начального заполнения структуры в случае с разделением данных при пересечении в 10% (запуск на машинке с 2 NUMA-сокетами)

Видим (приложение 4), что алгоритмы без разделения данных не меняют свое поведение. А вот результаты алгоритмов с разделением при наличии предварительного заполнения, как и ожидалось, начинают показывать себя лучше относительно базового многопоточного алгоритма, это хорошо видно на графиках случая с пересечением в 10% (рис. 8) — без заполнения алгоритм `llUnions + разделение` ведет себя хуже базового многопоточного, а в случае заполнения на 40 или 60% –

уже выигрывает у него. Смысл такого большого предзаполнения не в том, чтобы большая часть запросов выполнилась заранее, а в том, чтобы все интересы набиболее полно были известны к началу выполнения задачи – то есть если все интеерсы известны заранее, можно о них сразу сообщить и результаты будут схожи.

3.2.2. Остовные деревья

Задача построения минимального остовного дерева. Для начала, протестируем на случайно генерируемых графах и посмотрим сразу на зависимость времени работы построения от плотности графа:

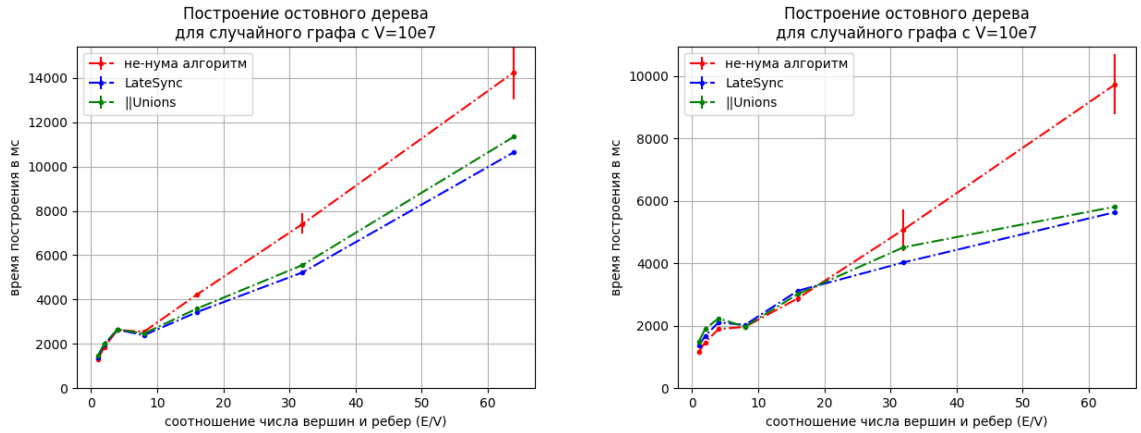

Рис. 9: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 2х (слева) и 4х (справа) NUMA-сокетах на задаче построения остовного дерева на графах с разным соотношением числа вершин и ребер.

Видим, что предложенные NUMA-адаптации выигрывают у базовой версии на графах с большим числом ребер, при чем на графике результатов на 4х сокетах отчетливо видно, что чем плотнее граф, тем больше выигрываем.

Что ожидаемо, так как соотношение числа операций `SameSet` и `Union` зависит от числа вершин, числа ребер и числа компонент связности: на выходе алгоритма получаем набор деревьев, то есть всего $V - components\_number$ ребер – а это значит, что ровно столько операций `Union` в системе непересекающихся множеств было совершено, в то время как каждое ребро хотя бы раз вызывало операцию `SameSet`. А из предыдущих результатов мы уже знаем, что чем меньше процент операций `Union`, тем выгоднее использовать предложенные NUMA-адаптации системы непересекающихся множеств.

В запуске на двух NUMA-сокетах видно, что предложенные алгоритмы выигры-

вают при числе ребер в 10 или более раз превышающем число вершин, а в запуске на 4х сокетах – в 20 и более.

Далее представлены результаты запусков построения минимального остовного дерева на реальных графах. Для невзвешенных графов веса ребер определялись случайным образом.

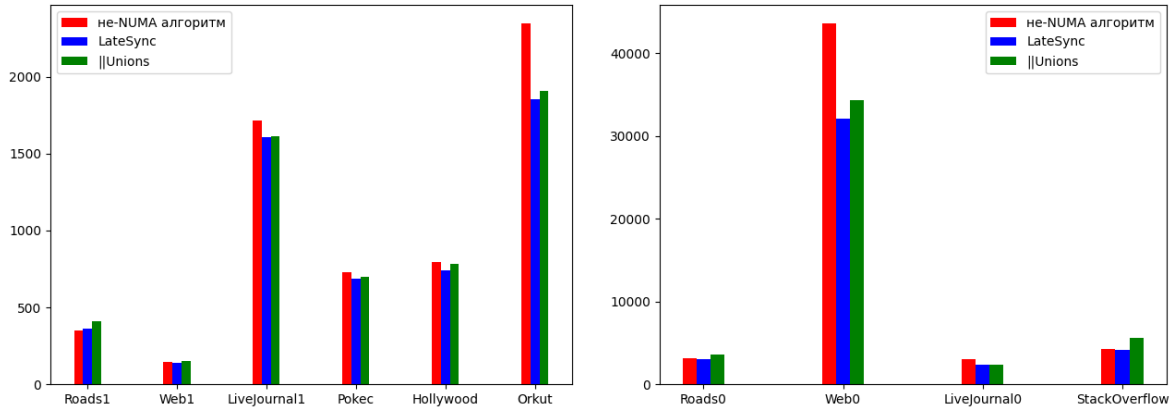

Рис. 10: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 2х NUMA-сокетах на задаче построения остовного дерева на графах разного размера и разной природы (описание графов см в 3.1.3).

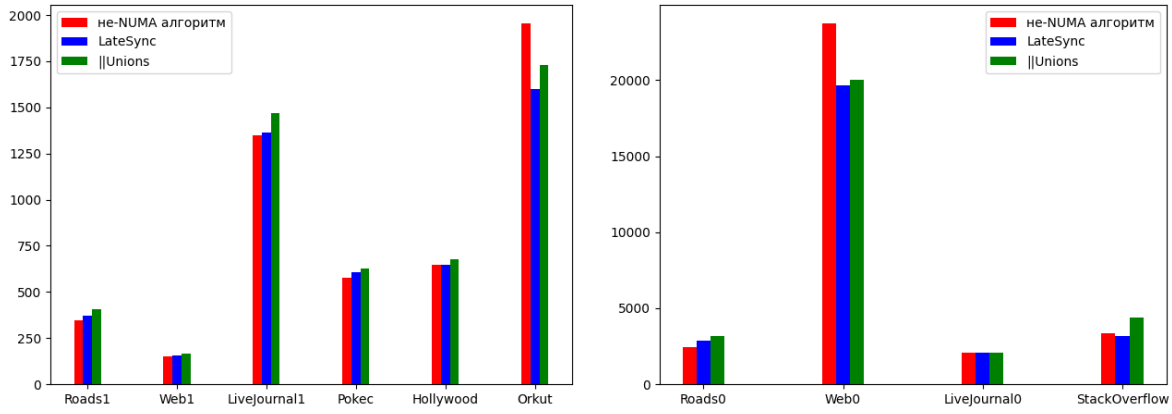

Рис. 11: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 4х NUMA-сокетах на задаче построения остовного дерева на графах разного размера и разной природы (описание графов см в 3.1.3).

Слева представлены результаты запуска на графах поменьше; можно заметить, что общее время выполнения мало отличается, что означает, что время работы системы непересекающихся множеств составляет малую часть времени работы всего

построения. При этом на двух сокетах результаты предложенных алгоритмов почти всегда лучше, а на четырех – лучше только в одном случае.

На графиках справа видим, что алгоритм `LateSync` выигрывает стандартный многопоточный алгоритм почти всегда. При чем у этого алгоритма есть ньюанс — отсутствие гарантии на согласованность данных в моменте, что совершенно не мешает в данной ситуации, так как действия чтения и объединения строго разделены, а гарантия на согласованность данных "в итоге" у нас есть.

При этом результаты хорошо соотносятся с полученным выводом о зависимости выигрыша от разности числа ребер и вершин — чем больше ребер относительно вершин, тем лучше показывают себя предложенные NUMA-алгоритмы.

3.3. Выводы по главе

В результате сравнения на практике, выяснилось, что алгоритмы `NRBB, FC, OneUnion` не улучшают время работы относительно базовой многопоточной версии из-за того, что одновременно в таких алгоритмах выполняться может ограниченное число запросов.

Алгоритмы `LateSync` и `llUnions` показали хорошие результаты в некоторых сценариях:

- на задаче динамического определения компонент связности графа алгоритмы тем лучше работают, чем меньше требуется операций объединения (то есть добавления ребер) в процентном соотношении; на случайных не плотных графах при проценте операций `Union` меньше 40 алгоритм `LateSync` работает лучше базового многопоточного и выигрывает до 25% производительности при запуске на 2х сокетах и до 30% при запуске на 4х; а алгоритм `llUnions` начинает выигрывать при проценте операций меньше 20 и максимальный выигрыш в производительности составляет столько же; на практике же для естественных графов результаты не такие стабильные и результаты сравнения зависят также от природы и плотрности графа

- на задаче построения минимального остовного дерева алгоритмы тем больше выигрывают, чем плотнее входной граф и при запуске на естественных графах алгоритм `LateSync` всегда выигрывает базовую многопоточную версию; алгоритм `llUnions` справляется несколько хуже, но так как его преимущество перед `LateSync` только в линеаризуемости, а в этой задаче она

не требуется, можно его даже в сравнении не учитывать; более того, когда алгоритм `LateSync` не выигрывает, он и не проигрывает или проигрывает малый процент производительности

Более того, к этим алгоритмам я применила технику разделения данных и протестировала на разных входных результатах и выяснила, что:

- алгоритмы `LateSync + разделение` и `llUnions + разделение` на кейсах с полным разделением данных (то есть без пересечения) стабильно работают лучше базового многопоточного алгоритма и проигрывают "идеальному" алгоритму из нескольких независимых копий CHM до 10% производительности на 2х сокетах и до 15 – на 4х

- алгоритм `LateSync + разделение` выигрывает у базового многопоточного до 30% и до 42% на 2х и 4х сокетах соответственно, а алгоритм `llUnions + разделение` отстает от него на несколько процентов, но все еще выигрывает у базового

- при небольшом пересечении данных между сокетами (этот процент зависит от самого графа, на случайных не очень плотных составляет 10%) предложенные алгоритмы с разделением отрабатывают быстрее базовой версии, однако при малом числе операций объединения на практике может быть выгоднее использование версий этих алгоритмов без разделения

Получается, предложенные алгоритмы стабильно лучше работают в задаче построения минимального остовного дерева, а именно алгоритм `LateSync`. В задаче динамического определения компонент связности графа алгоримы `LateSync` и `llUnions` работают лучше при условии, что ребра в граф добавляются не очень часто.

И при возможности полного разделения данных на части между сокетами, алгоритмы отрабатывают на 30-40% лучше базового многопоточного алгоритма. И даже если разделение неполное, он работают эффективнее при проценте пересечения до 10%.

Заключение

Результаты Главным результатом данной работы являются спроектированные и реализованные алгоритмы, ориентированные на неоднородную архитектуру. По результатам тестирования в разных сценариях были выявлены условия, в которых использование некоторых из предложенных алгоритмов будет оптимальнее, чем использование известных многопоточных алгоритмов системы непересекающихся множеств. А именно:

- в задаче построения минимального остовного дерева оптимально использовать предложенный алгоритм `LateSync`
- в задаче динамической поддержки компонент связности предложенные алгоритмы `LateSync, llUnions` работают лучше базового алгоритма в случае, когда процент операций объединения небольшой
- при возможности полного разделения данных алгоритмы `LateSync + разделение` и `llUnions + разделение` работают сильно лучше базового многопоточного алгоритма и между собой отличаются на 1-5% (линеаризуемый алгоритм чуть-чуть проигрывает версии без полной гарантии на согласованность данных)
- при неполном разделении данных предложенные алгоритмы могут быть оптимальнее базового, если пересечение достаточно маленькое (в среднем, до 10%, однако это варьируется в зависимости от входного графа)

Не смотря на то, что алгоритмы тестировались на двух задачах, также можно выявить общий принцип: чем меньше процент операций, требующих синхронизации (объединений), тем быстрее алгоритмы отрабатывают.

Однако, важно также упомянуть, что из-за необходимости синхронизации копий, при большом числе объединений, предложенные алгоритмы могут рабоать хуже (в случае работы на 2х сокетах – до 2х раз).

Дальнейшие исследования по теме могут развиваться в нескольких направлениях:

- Техника разделения данных содержит две идеи:

1. неполное обновление — нам не нужно выполнять объединения на сокетах, которые не заинтересованы ни в одном из объединяемых множеств
2. неполное хранения — можно на сокете не держать реплику вскех данных, а хранить информацию только о тех элементах, которые на этом сокете нужны

Мои реализации этой техники были направлены только на неполное обновление, но на каждом сокете все равно про каждый элемент хранилась информация в явном виде интересен он этому сокету или же нет.

Соответственно, это самое первое направление, в котором можно улучшать предложенные алгоритмы.

- Другое направление — оптимизация реализаций предложенных алгоритмов с помощью различных техник.
- И последнее, самое общирное, направление — проектирование новых алгоритмов: основанных на технике репликации и использующих новые методы синхронизации реплик, или основанных на технике разделения данных с динамическим определением "интересов".

Список литературы

[1] Alistarh D., Fedorov A., Koval N. In Search of the Fastest Concurrent Union-Find Algorithm. — 2019.

[2] Anderson R. J., Woll H. Wait-free Parallel Algorithms for the Union-Find Problem. — 1994.

[3] Black-Box Concurrent Data Structures for NUMA Architectures / Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, Marcos K. Aguilera // SIGARCH Comput. Archit. News. — 2017. — Apr. — Vol. 45, no. 1. — P. 207–221. — Access mode: https://doi.org/10.1145/3093337.3037721.

[4] Flat Combining and the Synchronization-Parallelism Tradeoff / Danny Hendler, Itai Incze, Nir Shavit, Moran Tzafrir // Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures. — SPAA '10. — New York, NY, USA : Association for Computing Machinery, 2010. — P. 355–364. — Access mode: https://doi.org/10.1145/1810479.1810540.

[5] Galler B. A., Fisher M. J. An improved equivalence algorithm. — 1964.

[6] Jayanti S.V., Tarjan R.E. A Randomized Concurrent Algorithm for Disjoint Set Union. — 2016.

[7] Jayanti S.V., Tarjan R.E. Concurrent Disjoint Set Union. — 2020.

[8] Metreveli Z., Zeldovich N., Kaashoek M. F. CPHASH: A Cache-Partitioned Hash Table. — 2012.

[9] Tarjan R. E. Efficiency of a Good But Not Linear Set Union Algorithm // J. ACM. — 1975. — Apr. — Vol. 22, no. 2. — P. 215–225. — Access mode: https://doi.org/10.1145/321879.321884.

[10] Tarjan R. E. A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. — 1979.

Приложение 1

Сравнение алгоритма `FC` с базовой многопоточной выерсией при малом числе операций `Union`

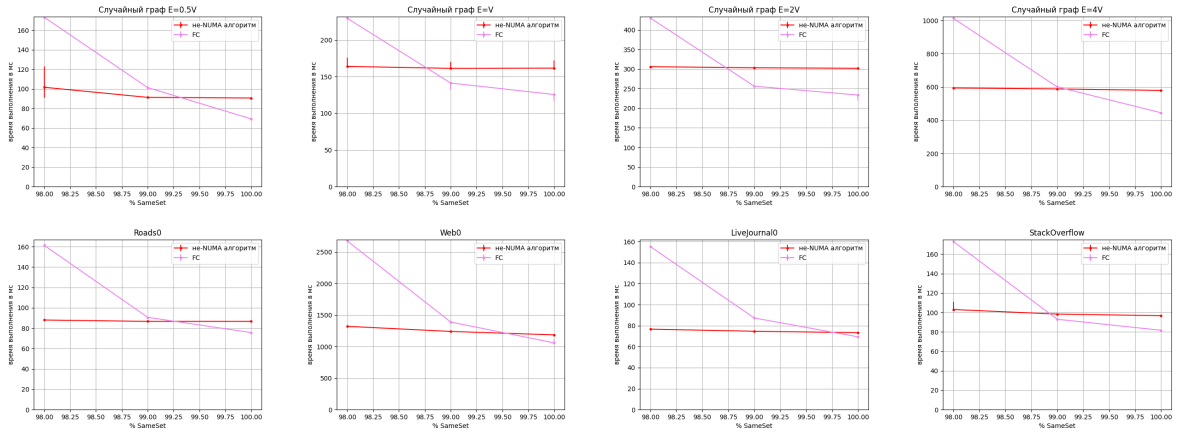

Рис. 12: Сравнение `FC` алгоритма с базовой многопоточной версией на сценарии малого числа операций `Union` (от 0 до 2%) на разных входных графах при запуске на 2х сокетах.

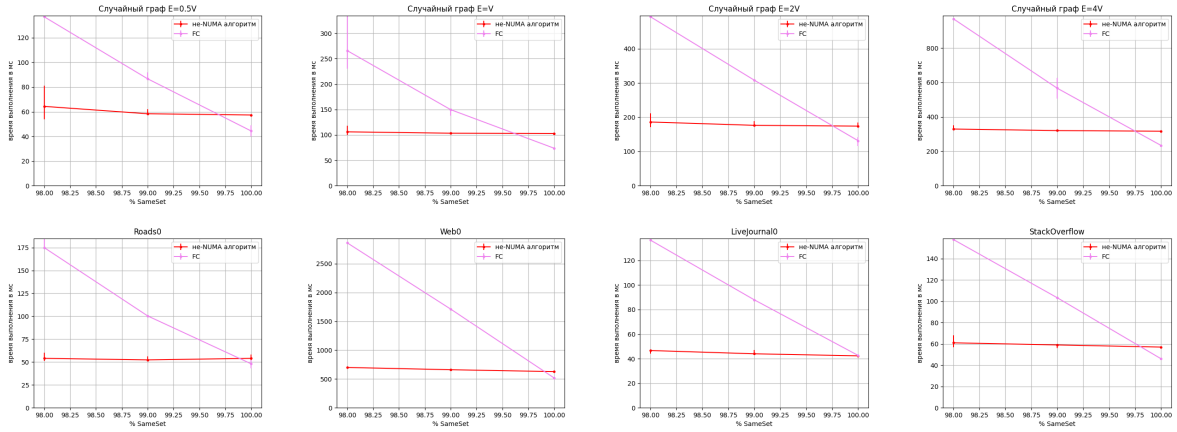

Рис. 13: Сравнение `FC` алгоритма с базовой многопоточной версией на сценарии малого числа операций `Union` (от 0 до 2%) на разных входных графах при запуске на 4х сокетах.

Приложение 2

Сравнение алгоритмов `LateSync, llUnions` с базовой многопоточной версией на задаче компонент связности графа

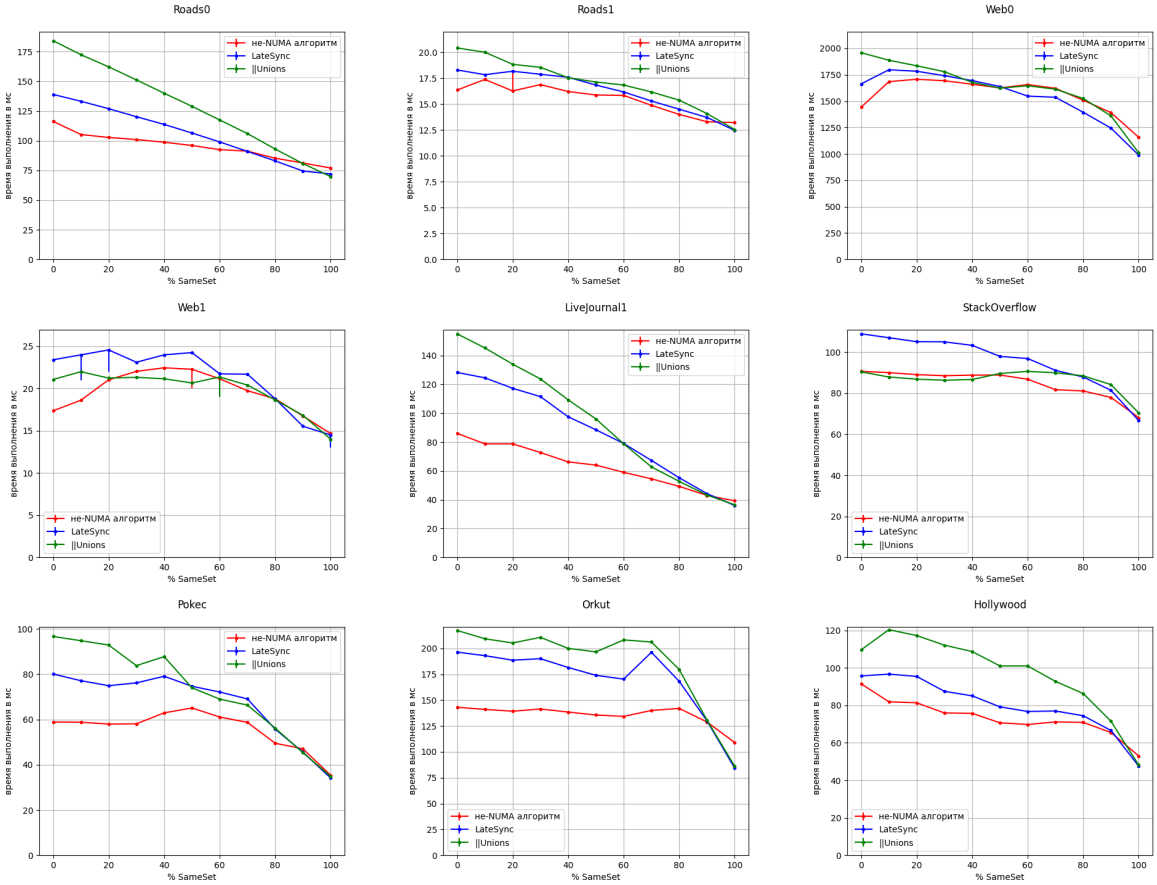

Рис. 14: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 2х NUMA-сокетах на задаче определения компонент связности с разным балансом операций чтения `SameSet` и объединения `Union` для естественных графов.

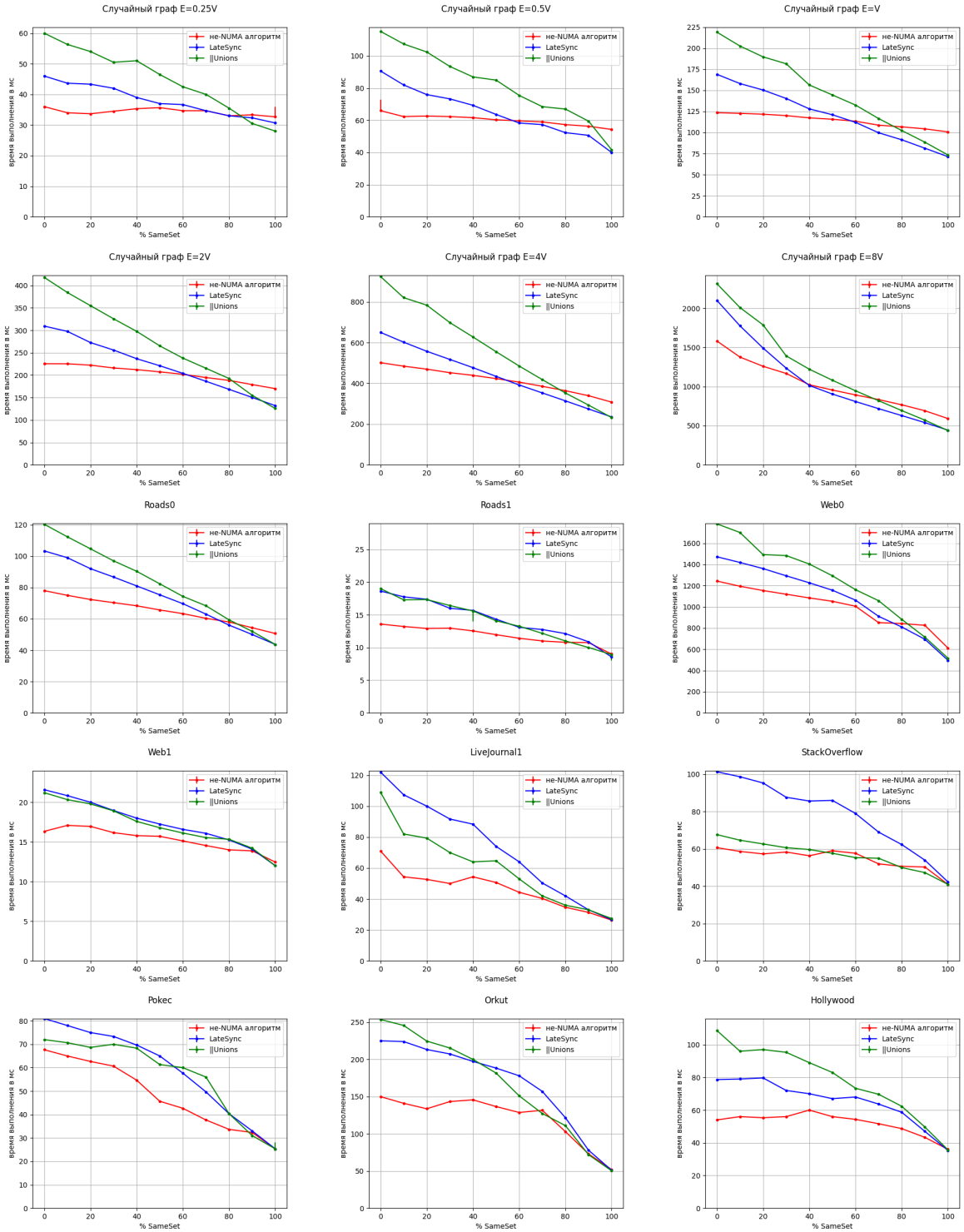

Рис. 15: Сравнение алгоритмов `LateSync` и `llUnions` с базовой многопоточной версией при запуске на 4х NUMA-сокетах на задаче определения компонент связности с разным балансом операций чтения `SameSet` и объединения `Union`.

Приложение 3

Сравнение алгоритмов с разделением данных для задачи компонент связности в естественных графах

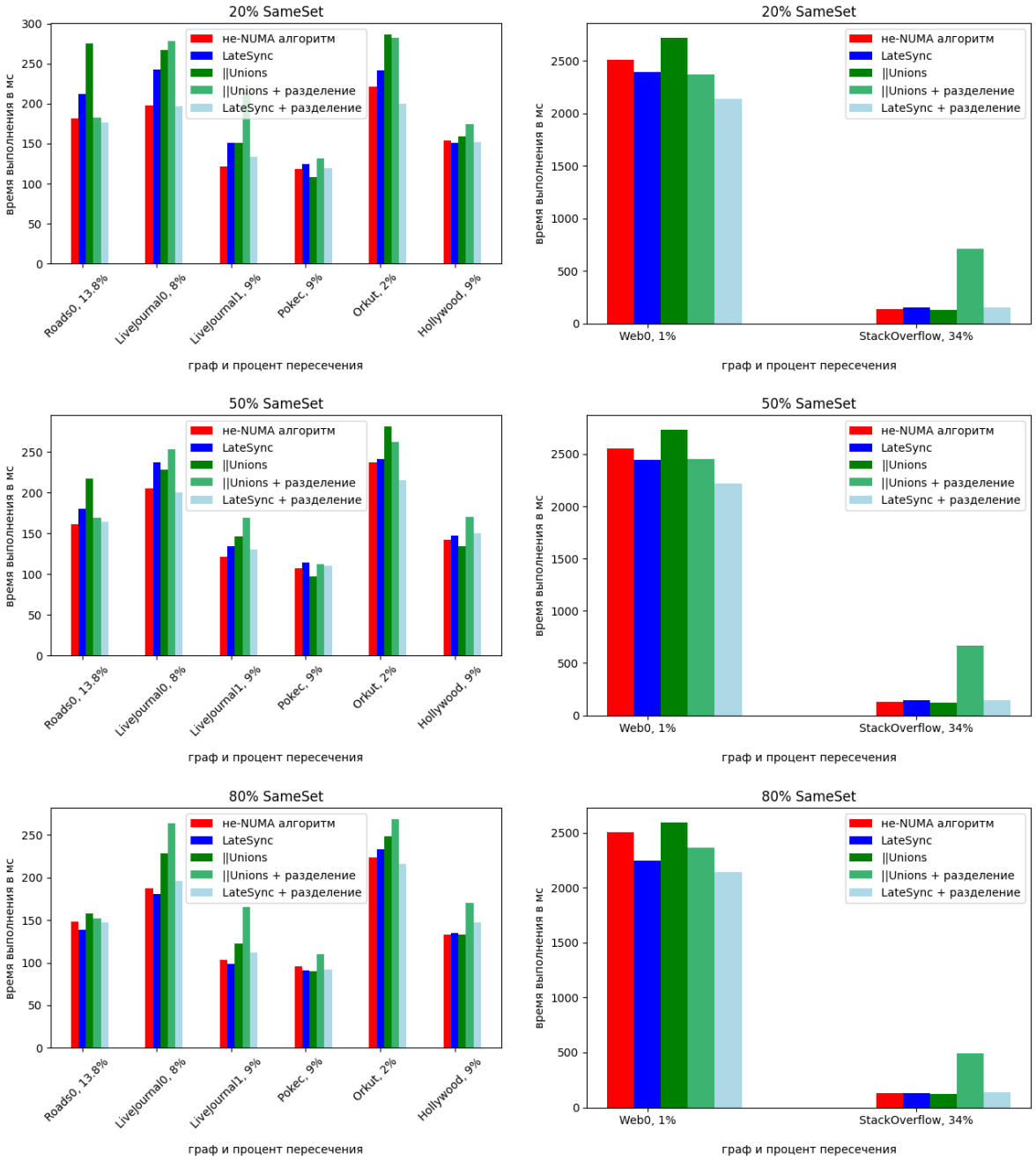

Рис. 16: Сравнение времени работы алгоритмов с разделением данных на естественных разделенных графах (запуск на машинке с 2 NUMA-сокетами)

Приложение 4

Зависимость производительности от уровня начального заполнения структуры

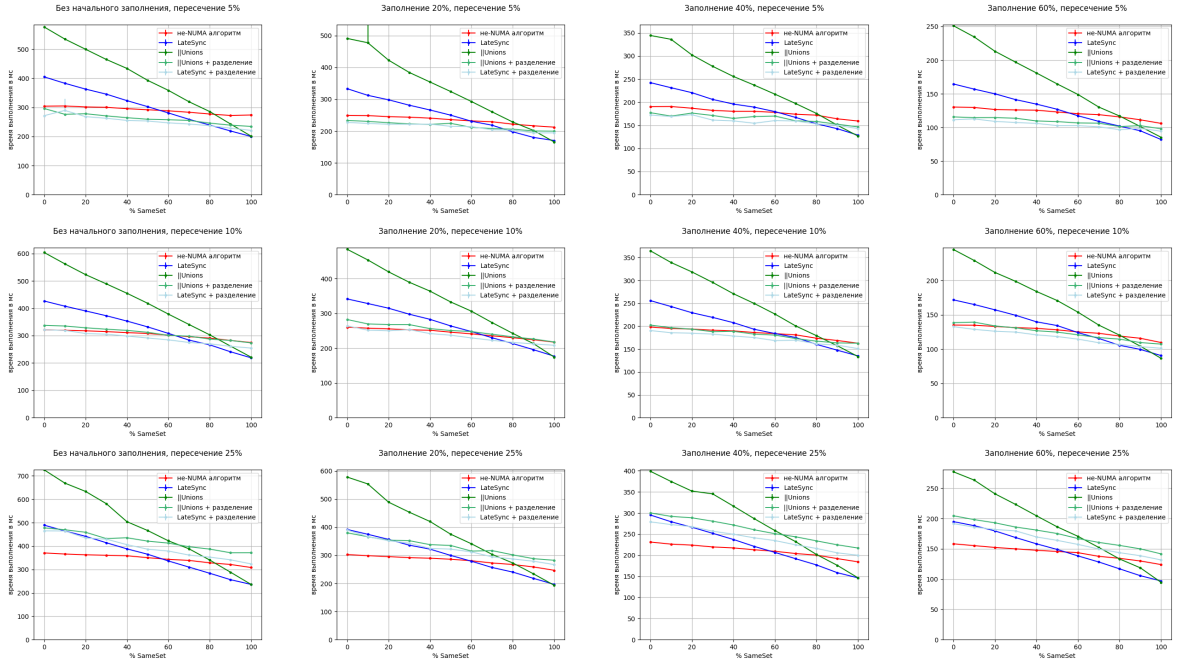

Рис. 17: Сравнение времени работы алгоритмов в зависимости от уровня начального заполнения структуры в трех случаях: с разделением данных при пересечении в 5, 10 и 25% (запуск на машинке с 2 NUMA-сокетами)