Реализация и анализ work-stealing планировщика задач[1]

Р. И. Кучумов, студент[2]
Петрозаводский государственный университет
kuchumovri@gmail.com

Параллельные планировщики задач, работающие по стратегии work-stealing, дают распределение задач близкое к оптимальному и при этом обладают низкими накладными расходами по времени, используемой памяти и межпроцессорным синхронизациям. Основная идея этой стратегии в том, что, когда у процессора заканчиваются задачи для выполнения, то он их забирает (крадет) у другого процессора. Один из недостатков этой стратегии это большое количество краж, возникающих при выполнении относительно маленьких задач. В этой статье приводится реализация планировщика задач, в которой можно изменять количество задач, извлекаемых за одну кражу. Такая модификация при правильном выборе размера кражи позволяет значительно уменьшить общее количество краж.

Ключевые слова: планировщик задач, work-stealing, деки, параллельные вычисления.

1. Введение

Некоторые программы можно разбить на подпрограммы, и, если их выполнять одновременно, то можно получить ускорение времени работы всей программы.

Операционные системы для этого предоставляют интерфейс работы с потоками вычислений. Но их использование не всегда оправданно и эффективно. Во-первых, создание и завершение потоков достаточно долгая операция, поэтому, если подпрограммы слишком маленькие, то ускорения по времени не будет. Во-вторых, количество потоков ограниченно операционной системой. В-третьих, если создать потоков больше, чем имеется физических процессоров, то потоки не смогут выполниться одновременно: в этом случае операционная система будет равномерно распределять процессорное время между всеми потоками.

[1]Работа выполнена при финансовой поддержке РФФИ (грант №15-01-03404-а).

[2]©Р. И. Кучумов, 2016

Для того чтобы избавиться от вышеперечисленных проблем заранее создают фиксированное количество потоков, которые будут активны в течении всего времени работы программы и используют планировщик, который неким образом распределяет подпрограммы между потоками. В этом случае не будет накладных расходов на создание потоков для каждой подпрограммы, а их количество будет ограничено только используемым планировщиком.

1.1. Стратегии планирования

Стратегии планирования разделяют на статические и динамические.

Статические используются, когда известны все свойства и особенности выполняемых задач. В этом случае можно заранее определить оптимальное распределение задач. Такая задача считается NP-полной [1] и встречается достаточно редко.

При динамическом планировании во время работы планировщик использует некую относительно простую стратегию планирования, которая дает результат близкий к оптимальному. В свою очередь, в динамическом планировании выделяют централизованное, когда есть специальный поток для распределения работы между другими потоками [2, 3] и не централизованное.

При не централизованном планировании у каждого потока имеется своя очередь задач (подпрограмм) для выполнения, и потоки сами осуществляют распределение. Например, поток, у которого слишком много задач, может их перераспределить между другими потоками (work-dealing [4]). Или, когда у потока заканчиваются задачи для выполнения, он их запрашивает (work-requesting [5]) или крадет (work-stealing [6]) у другого потока.

Теоретически доказано [6], что стратегия work-stealing дает распределение задач близкое к оптимальному, и на практике она себя также зарекомендовала – ее реализацию можно встретить во многих планировщиках задач, например в Rust, Cilk, Intel TBB, .NET TPL. Основное преимущество этой стратегии в низких накладных расходах по времени, используемой памяти и синхронизации между процессорами. Но, в отличие от стратегии work-dealing, в ней

не учитывается близость процессоров и данных, что делает ее не самым лучшим решением для кластерных систем.

Еще одна проблема work-stealing планировщиков, это большое количество краж, возникающих при выполнении небольших задач. Так как при кражах процессор обращается к памяти другого процессора, то происходят межпроцессорные синхронизации и возникают задержки. Эта проблема в work-requesting стратегии разрешается перемещением по запросу сразу половины имеющихся задач. Но есть и реализация work-stealing планировщика [7], позволяющая извлекать половину задач за одну кражу.

В этой работе предложена реализация work-stealing планировщика на языке C++11, в которой возможно извлекать константное количество задач за одну кражу, и предложен способ определения частот и оценки вероятностей операций, выполняющихся в планировщике.

2. Работа планировщика

Задачи, выполняемые планировщиком, состоят из инструкций, которые должны быть выполнены последовательно. Но во время своей работы, они могут создавать подзадачи и ожидать их завершения. Таким образом задачи можно представить в виде графа, в котором каждая задача указывает на родительскую задачу, ожидающую завершения ее выполнения.

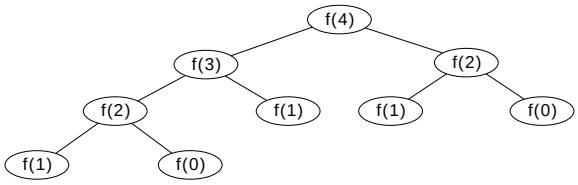

Рис. 1: Граф задач, для вычисления четвертого числа Фибоначчи. Каждая задача создает и ожидает завершения двух подзадач, т. е. ширина 2, и максимальная вложенность задач (глубина) 4.

Приведенный в приложении 7.1 код описывает задачу для вычисления n-го числа Фибоначчи. Каждая задача создает и ожидает

завершения двух своих подзадач и имеет максимальную глубину вложенности равную $n - 1$.

Задачи реализуются в виде объектов класса, наследуемого от класса *task* с абстрактным методом для их выполнения – *execute*. Дополнительно в классе *task* определен указатель на родительскую задачу и количество созданных подзадач. Функции *spawn* и *wait_for_all* также являются методами класса *task*.

Подзадачи, добавленные с помощью функции spawn помещаются в дек потока и могут быть украдены и выполнены другими потоками.

Функция *wait_for_all* передает управление планировщику, позволяя ему выполнять другие задачи, пока количество подзадач у родительской задачи не станет равным нулю. Задачи для выполнения поток сначала извлекает из своей очереди, а, когда они заканчиваются, начинает красть из очереди случайно выбранного потока. Алгоритм работы приведен в приложении 7.2.

Есть и другой подход к реализации алгоритма работы планировщика [8], в котором вместо задач в очередях хранятся состояния процессора. Тогда, при вызове функции *spawn* в очереди сохраняется текущее состояние процессора, а поток, вызвавший эту функцию, начинает выполнять переданную ей задачу. Когда другой поток украдет это состояние, то он его восстанавливает и продолжает выполнять родительскую задачу. При таком подходе используется намного меньше памяти, но его сложнее реализовать на практике.

3. Очереди задач

Есть разные реализации очередей задач, используемых в work-stealing планировщике.

Одна из первых работ [6] основана на массивах фиксированного размера. Одновременная работа с концами очереди происходит с неблокирующими синхронизациями, использующими атомарную операцию сравнения с обменом. Для этого вместе с индексами концов хранятся счетчики количества изменений, каждый поток их обновляет, используя эту операцию. Основной недостаток этой реализации это отсутствие динамического изменения размера.

В работе [9] для решения этой проблемы предлагают использовать двусвязные списки вместо массивов, но в результате появляются накладные расходы при работе со списками.

Есть и модификация [7], в которой реализована возможность извлекать половину задач за одну кражу. В [5] показана эффективность такого подхода для программ с большим количеством маленьких, независящих друг от друга задач. Такие задачи возникают, например, при реализации обхода графа в глубину.

В одной из последних и наиболее распространенных реализаций [10] используются очереди с двумя концами, т. н. деки (double ended queues). Нижний конец дека используется только его владельцем для добавления и извлечения задач, а верхний – другими потоками для краж. Такое ограничение позволяет реализовать дек на основе циклического массива, с использованием неблокирующей синхронизации и с возможностью динамического изменения размера.

Относительно других очередей задач, деки при работе нуждаются в меньших накладных расходах и их проще реализовывать. Работающие реализации можно можно встретить в планировщиках задач в библиотеке Intel TBB или в языке Rust. В работе [11] доказывается правильность реализации алгоритмов работы с деками на слабых моделях памяти.

В этой статье ниже будет приведена реализация деков, которая позволяет извлекать больше одной задачи за одну кражу.

3.1. Определение структуры данных

В планировщике деки представлены классом *task_deque*, который содержит методы *put*, *take*, *steal* для добавления, извлечения и кражи задач соответственно, а также методы для перемещения нескольких задач (*load_from*) и изменения размера (*resize*).

Для хранения индексов верхнего и нижнего конца дека, используются атомарные переменные *top* и *bottom*. Индекс *top* указывает на задачу, хранящуюся в верхнем конце дека, этот индекс увеличивается только при кражах, и не уменьшается. Индекс bottom указывает на свободную ячейку перед нижнем концом дека и изменяется

только владельцем дека при добавлении и извлечении задач. Дек считается пустым, когда top совпадает с *bottom*.

Индексы концов используются одновременно несколькими потоками, поэтому желательно, чтобы они одновременно не попадали в процессорный кэш первого уровня, так как будут происходить задержки из-за т. н. false-sharing. Для этого их нужно выровнять под размер кэш линии или добавить между ними отступы.

Массив дека (*array*) определяется структурой, содержащий буфер атомарных указателей на объекты задач (*buffer*), и маской размера буфера (*size_ mask*). Так как сам массив и его элементы будут использоваться одновременно разными потоками, то их необходимо сделать атомарными.

Поскольку массив циклический, то обращение к его элементам происходит по модулю от его размера. Но, чтобы не столкнуться с маловероятным переполнением индексов вершин и не использовать долгую операцию деления (для вычисления остатка), используется размер равный степени двойки и побитовая операцию "И" с маской размера вместо операции деления.

3.2. Операция кражи

Операция кражи (приложение 7.3) пытается извлечь k задач из дека потока. Если у жертвы нет k задач, то будет извлечена только одна задача. После успешной кражи функция возвращает указатель на первую задачу, а $k - 1$ задача помещаются в дек, потока, выполнявшего кражу.

Выполнение кражи состоит из следующих шагов:

1. Если в деке потока нет задач, то кража завершается неуспешно.
2. Загружаем задачу, на которую указывает верхний индекс дека.
3. Если у потока не имеется k задач, то пытаемся увеличить индекс верхнего конца дека на единицу, используя сравнение с обменом.

- Если его удалось увеличить, то это означает, что индекс не изменил никакой другой поток, и загруженная ранее задача считается украденной.
- Если сравнение с обменом завершилось неуспешно, то это означает, что задачу забрал другой поток, который изменил этот индекс, и операция завершается неуспешно.

4. Следующие шаги выполняются, если у потока имеется k задач. Загружаем $k-1$ задачу в дек потока, выполняющего кражу, (функция *load_from*), при этом не изменяем индекс нижнего конца, т.к. пока не известно успешна кража или нет.

5. Пытаемся увеличить индекс верхнего конца на k единиц, используя сравнение с обменом. Если увеличить не удалось, то кража завершается неуспешно.

6. При успешном увеличении верхнего индекса жертвы, увеличиваем нижний индекс на $k-1$ и кражу считаем успешной.

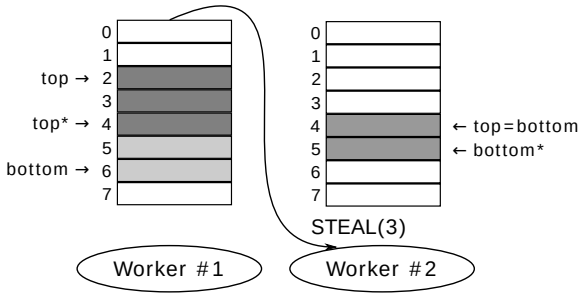

Рис. 2: Кража трех задач. После выполнения операции кражи индекс верхнего конца дека увеличивается на 3. При этом 2 задачи перемещаются в дек потока выполнявшего кражу.

3.3. Операция извлечения

Операция извлечения (приложение 7.4) задачи выполняется только владельцем дека. Если у дека было меньше задач, чем может быть украдено, то задача извлекается из нижнего конца без выполнения синхронизаций. В другом случае, т. к могло бы возникнуть

состояние гонки с операцией кражи, задача извлекается из верхнего конца дека.

Выполнение извлечения состоит из следующих шагов:

1. Увеличиваем индекс нижнего конца на единицу.
2. Если дек ранее был пустым, то восстанавливаем исходное пустое состояние ($top = bottom$).
3. Если у дека было задач больше, чем у него может быть украдено (k), то возвращаем задачу из нижнего конца. В этом случае состояние гонки с операцией кражи не может возникнуть, и не нужно выполнять никакие синхронизации.
4. Если у дека было не больше чем k задач, то загружаем задачу, на которую указывает верхний индекс.
5. Пытаемся увеличить верхний индекс на единицу, используя операцию сравнения с обменом. При неуспехе загруженную ранее задачу нельзя использовать, т. к. она была украдена потоком, который увеличил этот индекс.
6. Восстанавливаем предыдущее значение нижнего индекса, т. к. задача с нижнего конца не извлекалась.
7. Возвращаем взятую задачу или указатель на пустую задачу, в зависимости от успеха изменения верхнего индекса.

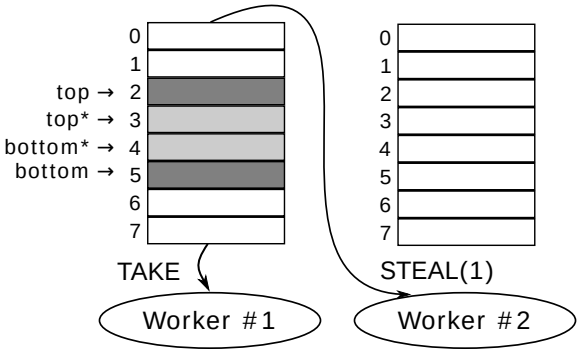

Рис. 3: Извлечение задачи владельцем. После выполнения операции извлечения индекс нижнего конца дека увеличивается единицу. В случае, когда в деке задач не больше чем может быть украдено, то задачи извлекаются из верхнего конца.

3.4. Операция добавления

Операция добавления (приложение 7.5) задачи помещает новую задачу по индексу bottom и увеличивает этот индекс на единицу. При ее выполнении не возникают состояния гонок, поэтому не нужно выполнять дополнительные синхронизации.

При необходимости операция добавления выполняет увеличение размера массива – вызывает функцию *resize*. Эта функция запрашивает у операционной системы память под массив большего размера, копирует в нее все элементы из старого массива, подменяет одновременно указатель на буфер и его маску размера и удаляет старый массив.

После увеличения размера массива индексы концов не изменяются, но изменится маска размера, которая к ним применяется. Так как индексы по прежнему будут указывать на задачи, на которые они указывали в старом массиве, то не нужно никак синхронизировать изменение размера с операцией кражи.

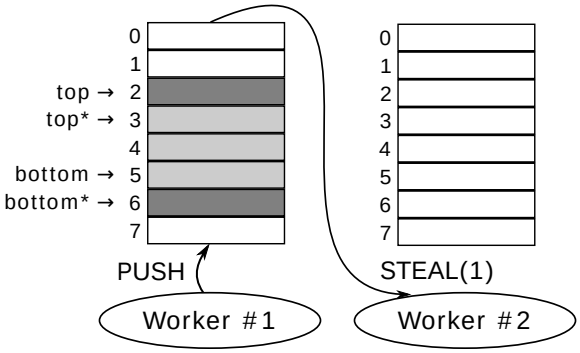

Рис. 4: Добавление задачи. После выполнения операции добавления индекс нижнего конца дека увеличивается на единицу. Состояние гонки при одновременной краже не возникает.

4. Сбор статистики работы планировщика

Для анализа работы планировщика были добавлены счетчики событий добавления, извлечения (успешного и неуспешного), кражи одной и нескольких задач (успешные и неуспешные) и счетчики количества изменений размеров. Эти счетчики являются локальными для каждого потока и увеличиваются на единицу при наступлении каждого события.

Кроме этого, были добавлены счетчики количества увеличений и уменьшений нижних концов при добавлении и извлечении задач. Значения этих счетчиков вместе со значением индекса верхнего конца сохраняются в каждый момент времени работы программы и выводятся в файл. Далее эти данные можно использовать, например, для определения размеров деков в каждый момент времени.

Для этого был создан специальный поток, который во время работы планировщика сохранял значения перечисленных выше счетчиков и моменты времени, когда выполнялись их взятие.

Для определения моментов времени использовалась команда на языке ассемблера RDTSCP, которая позволяет получить количество тактов работы процессора. При этом сама команда выполняется за десятки тактов [12], что на порядки быстрее функций работы со временем, предоставленных операционной системой.

Чтобы минимизировать недетерминизм при параллельной работе, планировщик запускался на ядрах процессора, изолированных от работы планировщика потоков операционной системы.

4.1. Оценка вероятностей изменений размеров деков

Для оценки вероятностей изменения размеров деков сначала определялись их размеры на каждом интервале времени. Для этого использовались значения счетчиков увеличения и уменьшения индекса нижнего конца (B_i^+ и B_i^-) и значения индекса верхнего конца (T_i) в соответствующие моменты времени t_i.

Интервал времени $d\tau$ выбирался не больше минимальной разности между соседними моментами времени. Если между t_i и t_{i+1} произошло k интервалов времени и значения счетчика изменялось на n единиц, то на каждом таком интервале времени предполагается, что значения изменялось на n/k. Тогда размер дека на каждом интервале времени вычислялся как $u(\tau_j) = B^+_{\tau_j} - B^-_{\tau_j} - T_{\tau_j}$.

Далее по изменению размера (du) определялось среднее количество изменений размеров на $0, \pm1, \pm2$ и т.д. единиц. Так как du вещественные числа, то количество изменений на $l \neq 0$ единиц за каждый интервал времени, увеличивалось на du/l.

Вероятности изменения размеров для задачи перемножения матриц и задачи о рюкзаке при интервале времени $d\tau = 100$ тактов приведены ниже. Далее эти вероятности можно использовать, например, как входные параметры для модели [13, 14].

Задача перемножения матриц:

l	-2	-1	0	1	2
Дек 1	0.0000047	0.0976776	0.8046355	0.0976741	0.0000080
Дек 2	0.0000000	0.1343020	0.7313987	0.1342942	0.0000044

Задача о рюкзаке:

l	-2	-1	0	1	2
Дек 1	0.0000000	0.0282798	0.9434404	0.0282795	0.0000002
Дек 2	0.0000000	0.0538008	0.8924009	0.0537951	0.0000031

4.2. Зависимость количества краж

Можно заметить явную линейную зависимость между общим количеством краж и количеством создаваемых подзадач.

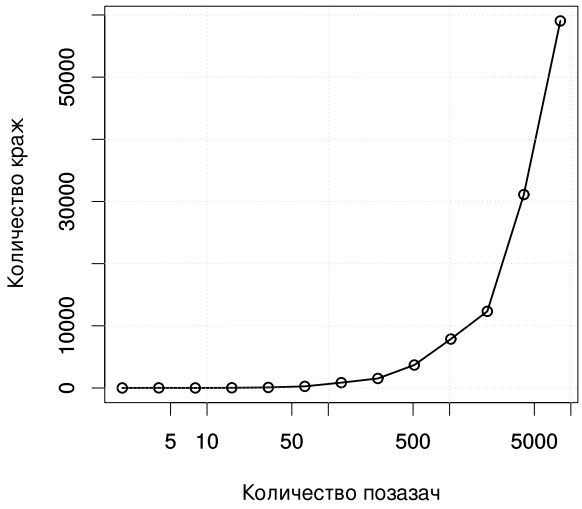

Рис. 5: Зависимость количества краж от количества подзадач.

Как было упомянуто ранее, при кражах происходит обращение к памяти другого потока. Не смотря на то, что все операции работы с деками неблокирующие, все равно происходят межпроцессорные синхронизации и из-за этого возможны задержки. Поэтому имеет смысл минимизировать количество краж.

Для задачи, которая создает 300 подзадач и завершается при глубине вложенности равной трем зависимость количества краж от размера кражи представлена на Рис. 6.

Можно заметить, что сначала количество краж резко уменьшается, а потом начинает увеличиваться. Это можно объяснить тем, что, когда у потока после кражи остается мало задач, то он в скором времени сам начинает выполнять кражу. Похожие графики были получены и для других задач.

К сожалению, корреляции размера кражи со временем работы

программы замечено не было. Но, возможно, она появится на других процессорных архитектурах или при других реализациях деков, в которых, например, оптимизированы операции добавления и извлечения в ущерб операции кражи.

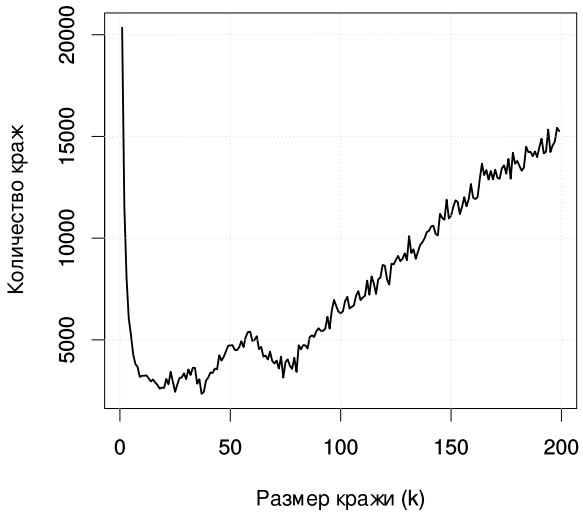

Рис. 6: Зависимость количества краж от размера кражи.

5. Сравнение с планировщиком задач из Intel TBB

Планировщик задач из библиотеки Intel TBB использует алгоритм планирования похожий на тот, что описан в этой статье. Очереди задач основаны на таких же деках, с разницей в том, что при краже извлекается только одна задача. Поэтому качество реализации следует сравнивать именно с Intel TBB.

В отличие от моей реализации в Intel TBB не используется интерфейс работы с потоками, определенный в C++11. Вместо этого используется POSIX Threads и свои реализации атомарных операций написанные на языке ассемблера для различных процессорных архитектур.

Для более точного сравнения оба планировщика были скомпилированы на компиляторе g++ 5.3.1 с одинаковыми флагами оптимизации (-O2). Путь к объектным файлам библиотек (.so) указывался на этапе компиляции с помощью параметра rpath. Дополнительно у Intel TBB были отключены собственные реализации атомарных операций (с помощью константы TBB_USE_GCC_BUILTINS), а вместо них использовались атомарные операции, предоставляемые компилятором.

5.1. Тестовые задачи

Для сравнения работы планировщиков были реализованы следующие задачи:

- Вычисление 35-го числа Фибоначчи по рекуррентной формуле
- Решение задачи о рюкзаке методом ветвей и границ для 26 элементов
- Перемножение матриц размером 256 на 256 разбиением и перемножением подматриц
- Сортировка 64 Мб равномерно и экспоненциально распределенных целых чисел методом слияния
- Обход графа задач, в котором задачи создают 300 подзадач и завершаются при достижении глубины равной 3.

5.2. Результаты

Сравнения производились на процессоре Intel Pentium N3530 2.16GHz, 4 CPU, 64 L1, с linux 4.4.0. и на процессоре ARMv7l rev 5, 4 CPU, 32 L1, с linux 4.4.8

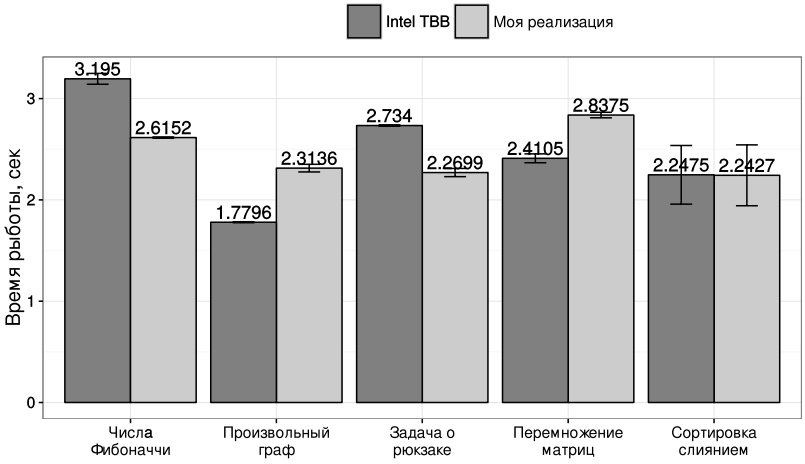

Рис. 7: Сравнение времени работы планировщиков на процессоре с архитектурой x86.

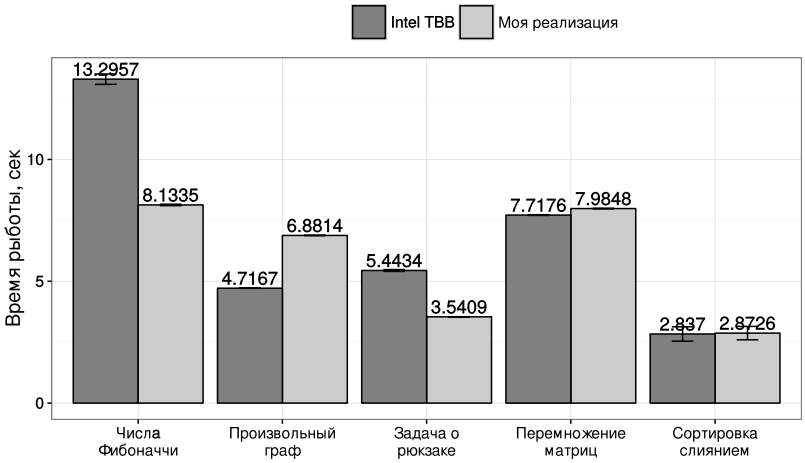

Рис. 8: Сравнение времени работы планировщиков на процессоре с архитектурой ARMv7.

6. Заключение

В этой работе приведена реализация work-stealing планировщика задач. В сравнении с планировщиком задач из библиотеки Intel TBB эта реализация дает приемлемые результаты, а на некоторых тестах показатели были даже лучше.

В этой реализации есть возможность извлекать более одной задачи за одну кражу. При правильном выборе размера кражи значительно уменьшается общее количество краж, возникающих в планировщике. К сожалению, корреляции этого параметра со временем работы замечено не было, но, тем не менее, это позволяет оптимизировать операции извлечения и добавления задач в ущерб операции кражи и строить модели для определения оптимального значения параметра.

Кроме этого, в статье приведен способ определения размеров очередей задач на каждом интервале времени работы и способ оценки вероятностей изменения размеров. Далее эти данные можно использовать для получения параметров моделей, описывающих работу планировщика.

7. Приложение

7.1. Задача вычисления чисел Фибоначчи

```
1  class FibTask: public task {
2  public:
3      FibTask (int n_, long *sum_): n(n_), sum(sum_) { }
4
5      void execute() {
6          if (n < 2) {
7              *sum = 1;
8              return;
9          }
10
11         long x, y;
12         FibTask *a = new FibTask(n - 1, &x);
13         FibTask *b = new FibTask(n - 2, &y);
14
15         spawn(a);
16         spawn(b);
```

```
17
18          wait_for_all();
19
20          *sum = x + y;
21      }
22
23  private:
24      int n;
25      long *sum;
26  };
```

7.2. Алгоритм работы планировщика

```
1   void scheduler::task_loop(task *parent) {
2       task *t = nullptr;
3
4       while (true) { // Stealing loop
5           while (true) { // Local tasks loop
6               if (t) {
7                   t->execute();
8
9                   if (t->parent != nullptr)
10                      dec_relaxed(t->parent->subtask_count);
11
12                  delete t;
13              }
14
15              if (parent != nullptr && load_relaxed(parent->
                    subtask_count) == 0)
16                  return; // All subtasks are finished
17
18              t = my_pool->take();
19
20              if (!t) // Local deque is epmty or failed race
21                  break;
22          }
23
24          if (parent == nullptr && !load_relaxed(is_active))
25              return; // Terminates workers threads
26
27          t = steal_task();
28      }
29  }
```

7.3. Операция кражи

```
task *task_deque::steal(task_deque *thief) {
    size_t t = load_acquire(top);
    atomic_fence_seq_cst();
    size_t b = load_acquire(bottom);

    if (t ≥ b) // Deque is empty
        return nullptr;

    array_t *a = load_consume(array);

    task *r = load_relaxed(a->buffer[t & a->size_mask]);

    if (tasks_per_steal == 1 || b - t < tasks_per_steal) {
        // Victim doesn't have required amount of tasks
        if (!cas_weak(top, t, t + 1))// Stealing one task
            return nullptr;

        return r;
    }

    // Moving tasks to thief deque without updating its
        bottom index (in case of failed steal)
    size_t thief_b = thief->load_from(a, t + 1,
        tasks_per_steal - 1);

    // Moved tasks could be stolen or one task could be
        taken by owner
    if (!cas_weak(top, t, t + tasks_per_steal))
        return nullptr;

    // Steal was successfull, updating thief bottom
    store_relaxed(thief->bottom, thief_b + tasks_per_steal -
        1);

    return r;
}
```

7.4. Операция извлечения

```
task *task_deque::take() {
    size_t b = dec_relaxed(bottom) - 1;
    array_t *a = load_relaxed(array);
    atomic_fence_seq_cst();
    size_t t = load_relaxed(top);
```

```

    if (t > b) { // Deque was empty, restoring to empty state
        store_relaxed(bottom, b + 1);
        return nullptr;
    }

    if ((t + (tasks_per_steal - 1)) ≥ b) {
        // Deque had less or equal tasks tnan steal amount

        task *r = load_relaxed(a->buffer[t & a->size_mask]);
        if (!cas_strong(top, t, t + 1)) // Check if they are not stollen
            r = nullptr;

        store_relaxed(bottom, b + 1);
        return r;
    }

    return load_relaxed(a->buffer[b & a->size_mask]);
}
```

7.5. Операция добавления

```
void task_deque::put(task *new_task) {
    size_t b = load_relaxed(bottom);
    size_t t = load_acquire(top);

    array_t *a = load_relaxed(array);

    if (b - t > a->size_mask) {
        resize();
        a = load_relaxed(array);
    }

    store_relaxed(a->buffer[b & a->size_mask], new_task);
    atomic_fence_release();
    store_relaxed(bottom, b + 1);
}
```

Список литературы

[1] *Yu-Kwong Kwok, Ishfaq Ahmad.* Static scheduling algorithms for allocating directed task graphs to multiprocessors // ACM. 1999.

[2] *Beaumont O., Carter L., Ferrante J., Legrand A., Marchal L., Robert Y.* Centralized versus distributed schedulers for multiple bag-of-task applications // 20th IEEE International Parallel & Distributed Processing Symposium. 2006.

[3] *Yinglong X., Prasanna V., Li James.* Hierarchical Scheduling of DAG Structured Computations on Manycore Processors with Dynamic Thread Grouping. Springer. 2010.

[4] *Hendler D., Shavit N.* Work dealing (extended abstract) // SPAA'02. 2002.

[5] *Acar U. A., Chargueraud A., Rainey M.* Scheduling parallel programs by work stealing with private deques // ACM. 2013.

[6] *Arora N. S. , Blumofe R. D. , Plaxton C. G.* Thread scheduling for multiprogrammed multiprocessors // SPAA'98. 1998.

[7] *Hendler D., Shavit N.* Non-blocking steal-half queues // ACM. 2002.

[8] *Guo Yi, Barik Rajkishore, Raghavan Raman, Sarkar Vivek.* Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. 2009.

[9] *Hendler D., Lev Y., Moir M., Shavit N.* A Dynamic-Sized Nonblocking Work Stealing Deque. Springer. 2005.

[10] *Chase D., Lev Y.* Dynamic circular work-stealing deque // SPAA'05. 2005.

[11] *Nhat Minh Le, Pop A., Cohen A., Nardelli F. Z.* Correct and efficient work-stealing for weak memory model // ACM. 2013.

[12] *Paoloni G.* How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. 2010.

[13] *Барковский Е.А.* Математические модели и методы оптимального управления Work-stealing деками в общей памяти // Стохастическая оптимизация в информатике. 2015. Т. 11. №1. С. 55–64.

[14] *Sokolov A.V., Barkovsky E.A.* The mathematical model and the problem of optimal partitioning of shared memory for work-stealing deques // Springer International Publishing, Proceedings of The International Conference of Parallel Computing Technologies, 2015. Lecture Notes in Computer Science. Vol. 9251, P. 102–106.