Отчет по домашнему заданию 1

Капралов Степан Алексеевич

Задача 1. Обедающие философы (Dining philosophers problem)

Первая часть задачи, “устранение взаимных блокировок”, достаточно проста и имеет множество таких же простых решений, например проверка вилок перед взятием или нумерация вилок с правилом, что философ берет сначала вилку с меньшим номером. Поэтому прежде всего стоит отталкиваться от второй проблемы, “голодание” и попутно с ней решать проблему взаимоблокировок.

Основная причина появления голодания в том, что философы “отправлены в свободное плавание” и их работа никак не контролируется. Голодание возникает если два философа “устроят заговор” против философа между ними, то-есть будут есть и думать в такой очередности, что средний философ не сможет начать есть. Таким образом некоторые потоки выполняют больше полезной работы чем другие. Также время ожидания ресурсов будет сильно разниться.

Ключ к исправлению этих неполадок - ввод свода правили или контролирующих структур. Это могут быть: ввод в программу различных атомарных флагов состояния, использование барьеров для синхронизации потоков или ввод дополнительного контролирующего потока, “официанта”.

Метод талонов на еду. Суть метода в том, что философам раздается m талонов (m - максимальное число философов принимающих пищу в один момент времени), наличие которых разрешает прием пищи философам. Если философ голоден и у него есть такой талон, то он начинает есть, если нет талона, ждет. Если у философа есть талон и он собирается начать думать, он передает талон правому философу. Этот метод избавляет потоки от взаимоблокировок и должен обеспечивать равномерное использование ресурсов. Однако проведенные тесты после его реализации выявили, что этот алгоритм приводит к большому и неравномерному времени ожидания ресурсов, так-же это притянуло за собой падение числа приемов пищи философами.

Официант. Водим дополнительный поток контролирующий прием пищи философами, назовем его “официантом”. Официант занимается тем, что разрешает прием пиши философам. Реализуем официанта который непрерывно бегает вокруг стола и спрашивает у философов их состояние, с помощью набора атомарных флагов. У каждого философа есть флаги состояния “я ем”, “я голоден”. Официант подходи к философу и спрашивает, “ты голоден?”, если да то проверяет едят ли его соседи и если не едят разрешает ему есть, если если не голоден идет дальше. Пробегая каждый круг официант вычислят самого наевшегося философа и применяет к нему

карательные меры(пробегает мимо него). Наказывать объевшихся философов быстрее чем подгонять голодных, так как для наказания мы просто пропускаем обжору, а для подталкивания голодного нужно ждать или создавать ситуацию, что бы он мог есть. Взаимоблокировки устраняются тем, что официант не добегает круг до конца на одного философа. Но при этом этот философ не остается постоянно голодным так как новый круг всегда начинается с самого объевшегося.

Метод контроля по среднему значению числа приемов пищи. Суть метода в том, что поток перед тем как начать есть сравнивает количество своих приемов пищи и среднего числа приемов пищи среди всех философов. Такой подход позволяет жестко контролировать распределение еды между философами, не допускать голоданий и перееданий. Вилки берутся по принципу: берем левую вилку, теперь если правую вилку взять удалось идем есть, если не удалось бросаем обе вилки. Этот принцип борется с взаимоблокировками.

Тестирование:

Взаимоблокировки
Входные параметры “./phil 3 10 1 1 0”

Официант	Среднее значение	Исходная программа

[TID] <eat_count> <wait_time>
[1] 3531 5280
[2] 3532 5487
[3] 3532 5389
Total stats:
eat_avg=3531.67;
wait_avg=5385.33;
min/max=1.00;
jains=1.0000
	[TID] <eat_count> <wait_time>
	[1] 2407 6712
	[2] 2406 6671
	[3] 2407 6663
	Total stats:
	eat_avg=2406.67;
	wait_avg=6682.00;
	min/max=1.00;
	jains=1.0000
		Взаимоблокировка

Время еды и раздумий всегда равно единице, повышенный риск того, что все философу захотят одновременно начать есть.

Голодание
Входные параметры “./phil 12 5 10 1000 0”

Официант	Среднее значение	Исходная программа
[TID] <eat_count> <wait_time> [1] 5 2900 [2] 2 2764 [3] 4 2536 [4] 5 2982 [5] 2 2356	[TID] <eat_count> <wait_time> [1] 3 2538 [2] 3 2961 [3] 3 2176 [4] 3 2749 [5] 3 1668	[TID] <eat_count> <wait_time> [1] 3 4117 [2] 3 3394 [3] 2 3721 [4] 3 5264 [5] 3 4146

[6] 5 2686
[7] 4 2410
[8] 3 7
[9] 3 2884
[10] 5 1181
[11] 4 3326
[12] 4 1271
Total stats: eat_avg=3.83;
wait_avg=2275.25;
min/max=0.40;
jains=0.9281
	[6] 3 2846
	[7] 3 3419
	[8] 3 3266
	[9] 3 2599
	[10] 3 3296
	[11] 3 1878
	[12] 3 2716
	Total stats: eat_avg=3.00;
	wait_avg=2676.00;
	min/max=1.00;
	jains=1.0000
		[6] 2 4459
		[7] 3 3956
		[8] 3 7032
		[9] 5 6406
		[10] 4 6010
		[11] 4 4265
		[12] 3 5402
		Total stats: eat_avg=3.17;
		wait_avg=4847.67;
		min/max=0.40;
		jains=0.9401

Этот тест отражает ситуацию когда философы быстро думают и очень долго едят. Философам приходится очень долго ждать пока наедятся их соседи. По результатам теста видим, что метод официанта выдал схожие параметры с исходным методом, а метод среднего значения показал максимальное выравнивание по приемам пищи за счет балансировки временем ожидания.

Долгая работа

Входные параметры “./phil 5 240 100 100 0”

Официант	Среднее значение	Исходная программа

[TID] <eat_count> <wait_time>
[1] 1246 113267
[2] 1247 111888
[3] 1247 115225
[4] 1246 115658
[5] 1247 112458
Total stats: eat_avg=1246.60;
wait_avg=113699.20;
min/max=1.00;
jains=1.0000
	[TID] <eat_count> <wait_time>
	[1] 1476 91305
	[2] 1477 91277
	[3] 1477 92677
	[4] 1477 92337
	[5] 1477 92697
	Total stats: eat_avg=1476.80;
	wait_avg=92058.60;
	min/max=1.00;
	jains=1.0000
		[TID] <eat_count> <wait_time>
		[1] 1267 115571
		[2] 1267 112869
		[3] 1282 114481
		[4] 1260 115080
		[5] 1259 116601
		Total stats: eat_avg=1267.00;
		wait_avg=114920.40;
		min/max=0.98;
		jains=1.0000

Большое число философов

Входные параметры “./phil 10 20 100 100 0”

Официант	Среднее значение	Исходная программа
Total stats: eat_avg=119.40; wait_avg=7974.40; min/max=0.98; jains=0.9999	Total stats: eat_avg=113.80; wait_avg=8553.70; min/max=0.99; jains=1.0000	Total stats: eat_avg=100.80; wait_avg=10069.30; min/max=0.92; jains=0.9996

Входные параметры “./phil 100 20 100 100 0”

Официант	Среднее значение	Исходная программа

Total stats: eat_avg=66.03; wait_avg=12848.60; min/max=0.84; jains=0.9982	Total stats: eat_avg=63.99; wait_avg=13227.48; min/max=0.98; jains=1.0000	Total stats: eat_avg=98.47; wait_avg=10280.69; min/max=0.80; jains=0.9978

Падение числа приемов пищи при увеличении числа потоков в 10 раз: Официант - 1,8раз, среднее значение - 1,78раз, исходная программа - 1,02раз.

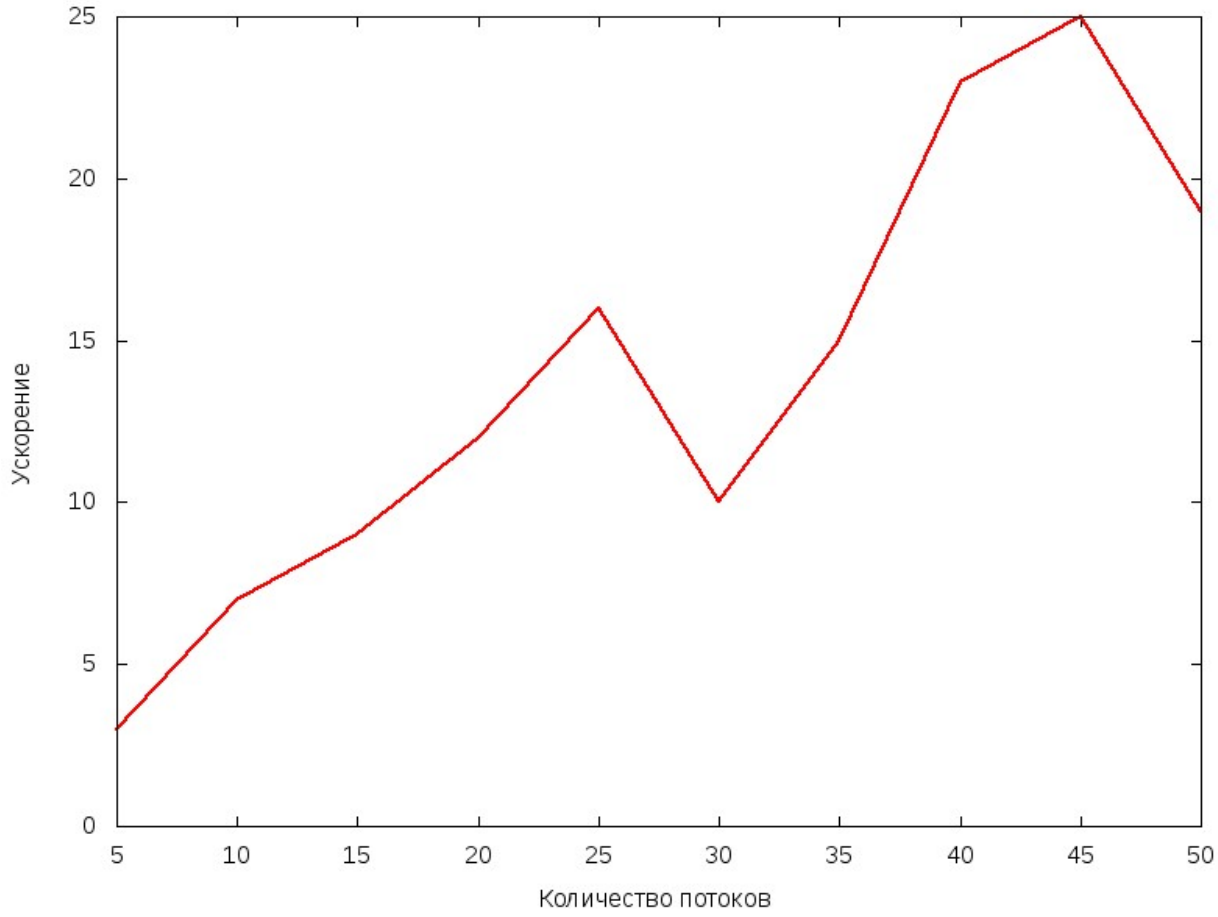
Вывод:

Исходя из проделанной работы, можно сделать вывод, что исправление проблемы голодания ведет к появлению дополнительных накладных расходов и не приводит к значительным улучшениям результата. При том, что в исходной программе голодание как таковое сложно отловить, в основном она распределяет ресурсы равномерно, присутствуют только небольшие колебания в зависимости от входных параметров. Единственное, что гарантируют методы борьбы с голоданием - невозможность ситуации заговора соседей против философа. Но вероятность такой ситуации в исходной программе очень и очень мала. Поэтому я считаю главным преимуществом модифицированных программ, борьбу с взаимоблокировками, которые могут носить действительно фатальные последствия для выполнения алгоритма.

Задача 2. Поисковый робот (Web crawler)

Тесты:

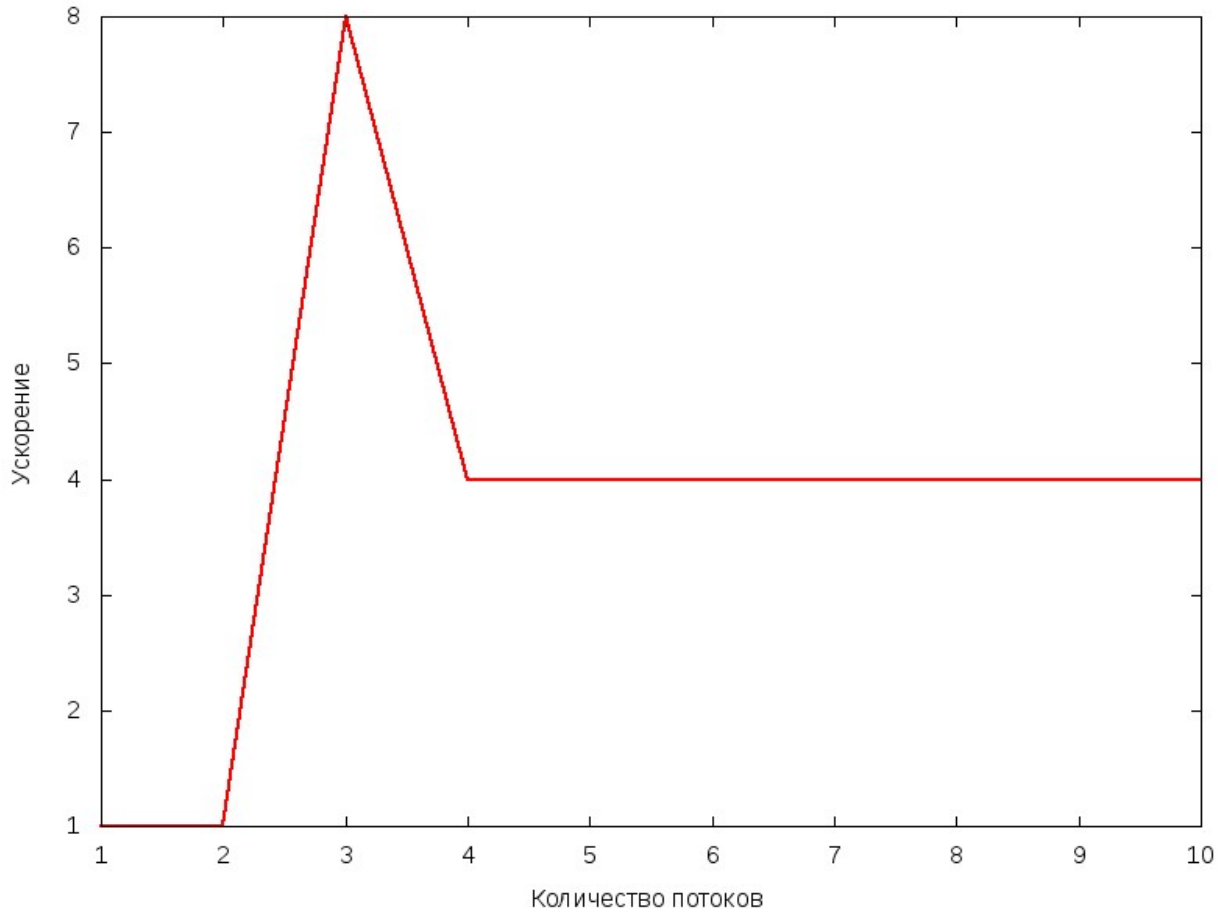
Проведем тесты на многопоточную загрузку страниц в string буфер при помощи библиотеки curl и многопоточное сохранение страниц на диск.
Вообще проводить тесты на быстродействия загрузки страниц из глобальной сети - занятие сомнительное. Так как перед нами очень переменчивая среда и быстродействие зависит от множества факторов помимо увеличения числа потоков в программе. Однако необходимо увидеть как ведет себя многопоточная программа загрузки страниц и выявить определенную закономерность. Составим тест который будет скачивать 100 различных страниц в буфер, последовательно или параллельно(страницы распределяются поровну, на сколько это возможно). Проведем тесты с разным числом потоков по нескольку раз и возьмем средние значения, получим вот такой странный график:



(максимально время загрузки - 109018 мс, минимальное - 4308 мс)

Все же используя график можно сделать вывод, о наличии некой масштабируемости у этого теста.

Теперь проведем тест на параллельное сохранение страниц на жесткий диск. Каждый поток будет загружать на диск текст в размере 5МБ с отключенным буфером вывода. Получившееся общее время разделим на количество потоков и получим среднее время загрузки одного текста.



(максимальное время сохранения - 8 мс, минимальное - 1 мс)

Судя по графику увеличение числа пишущих потоков мало сказывается на увеличении скорости записи и максимальное ускорение наблюдается при трех потоках.

Идея:

Теперь можно предсказать как будет вести себя многопоточный робот. Время загрузки страницы из сети значительно выше времени сохранения на диск. Время поиска новых ссылок будет незначительно мало так как размер html документа в среднем не будет превышать 150КБ. Значит главным приоритетом робота будет максимально быстрая загрузка страниц из сети в буфер. Добьемся этого путем разделения специализации потоков, большая часть которых будет заниматься скачиванием страниц из сети, а меньшая парсингом скачанных страниц и сохранением контента на диск. Первых назовем “загрузчиками”, а вторых “парсерами”.
Однако при таком подходе может возникнуть проблема голодания, если “загрузчикам” будет не хватать ссылок для загрузки либо “парсерам” контента для парсинга. Решим эту проблему научив каждый поток и загружать, и парсить, меняя свою специализацию

в зависимости от ситуации. Таким образом добьемся равномерного распределения потоков для решения каждой подзадачи.

Реализация:

Для реализации обхода графа в ширину необходимо использовать очередь ссылок загрузки контента. Так же, если я хочу распределить потоки по двум специализациям, мне необходима еще одна очередь где будет располагаться уже загруженный контент в ожидании парсинга. Получаем такую схему работы программы: поток загрузчик вынимает ссылку из очереди загрузки, скачивает ее и кладет контент в очередь для парсинга, далее снова обращается к очереди загрузки…, тем временем поток парсер вынимает из очереди парсинга контент, находит в нем ссылки, добавляет их в очередь загрузки и сохраняет контент на диск.

Для реализации очереди я воспользовался boost::lockfree::queue, так как очередь из stl не является потокобезопасной структурой. Еще один плюс boost::lockfree::queue это отсутствие внутренних блокировок, что обеспечивает максимально быстродействие добавления и извлечения элементов и избавляет от остальных проблем связанных с блокированием памяти. Однако по настоящему “lockfree” эту очередь можно назвать только если она имеет фиксированный размер, чем я и воспользовался.

Так как требуется постоянно перемещать по памяти большое число строк, введу в повсеместное использование std::shared_ptr, что бы один раз выделить память под строку, а потом передавать исключительно указатели на эту строку.

Для того, что бы обеспечить условие “робот не должен посещать одну и ту же страницу более одного раза”, я храню все уже использованные ссылки в виде их хеша в потокобезопасной хеш таблице “thread_safe_hash_set”. Такой подход был выбран для того что бы за константное время определять уникальность ссылки. Таблица хранит в себе только результат хеш функции размером 8байт вместо целой строки. Вероятность того что std::hash выдаст одинаковые значения для разных строк стремиться к 1.0/std::numeric_limits<size_t>::max(), она настолько мала, что можно гарантировать уникальность ссылки.

Класс parser. Этот класс реализует поиск ссылок в загруженном контенте, а именно находит обозначение ссылки в html документе “href=\”, вырезает ссылку из кавычек и проверяет ее с помощью регулярного выражения(используется std::regex), если ссылка подходит, добавляет ее в вектор найденных ссылок. Класс имеет метод для сохранения контента на диск. Имя файла задается по следующим правилам - ("num" + порядковый номер сохранения + "dp" + глубина страницы в дереве + "id" + номер потока сохраняющего страницу + ".html").

Класс downloader. Это класс реализует загрузку контента из сети по поданной ему ссылке. Если не удается загрузить страницу, возвращается пустой буфер.

Классы parser и downloader не являются потокобезопасными и их объекты создаются каждым потоком локально.

Конфигурационный файл conf.conf содержит записи:

- N_THREADS - число потоков вместе с основным.
- MAX_PARSE_THREAD - максимальное число потоков парсера в один момент времени.
- Q_D_MIN - минимальный размер очереди загрузки.
- Q_D_MAX - максимальный размер очереди загрузки.
- Q_P_MIN - минимальный размер очереди парсинга.
- Q_P_MAX - максимальный размер очереди парсинга.
- HASH_SIZE - размер хеш таблицы thread_safe_hash_set.
- TIMEOUT - время ожидания страницы в Классе downloader.

Специализация потоков. Если потоку требуется сменить специализацию с парсера на загрузчика или на оборот, он с легкостью это делает. Вся смена специализации заключается в изменении “bool status_flag”. Поэтому робот может не бояться накладных расходов при смене типа потока и менять его хоть на каждой итерации, если того требует ситуация.

Поток становится загрузчиком если:

- очередь парсинга меньше минимального размера, а очередь загрузки больше
- размер очереди загрузки дошел до максимума

Поток становится парсером если:

- очередь парсинга больше минимального размера и номер потока меньше MAX_PARSE_THREAD
- размер очереди загрузки меньше минимума, а размер очереди парсера больше
- размер очереди парсинга дошел до максимума

Программа поддерживает следующие аргументы командной строки:

- Адрес начальной страницы
- Максимальная глубина обхода
- Максимальное количество загружаемых страниц
- Путь к директории, в который сохраняются посещенные страницы
- Флаг отладки

Тестирование:

Параметры тестирующей машины:

- Процессор - Intel® Core™ i7-3537U CPU @ 2.00GHz × 4
- Память - 5,7 ГиБ DDR3 1600 МГц
- Операционная система - Ubuntu 14.04.2 LTS 64-bit
- Накопитель 1000 Гиб Serial ATA 5400 об/мин
- Сеть Wi-Fi IEEE 802.11n

Тестирование будем производить на следующих параметрах запуска:
(http://www.yandex.ru/ 2 100)

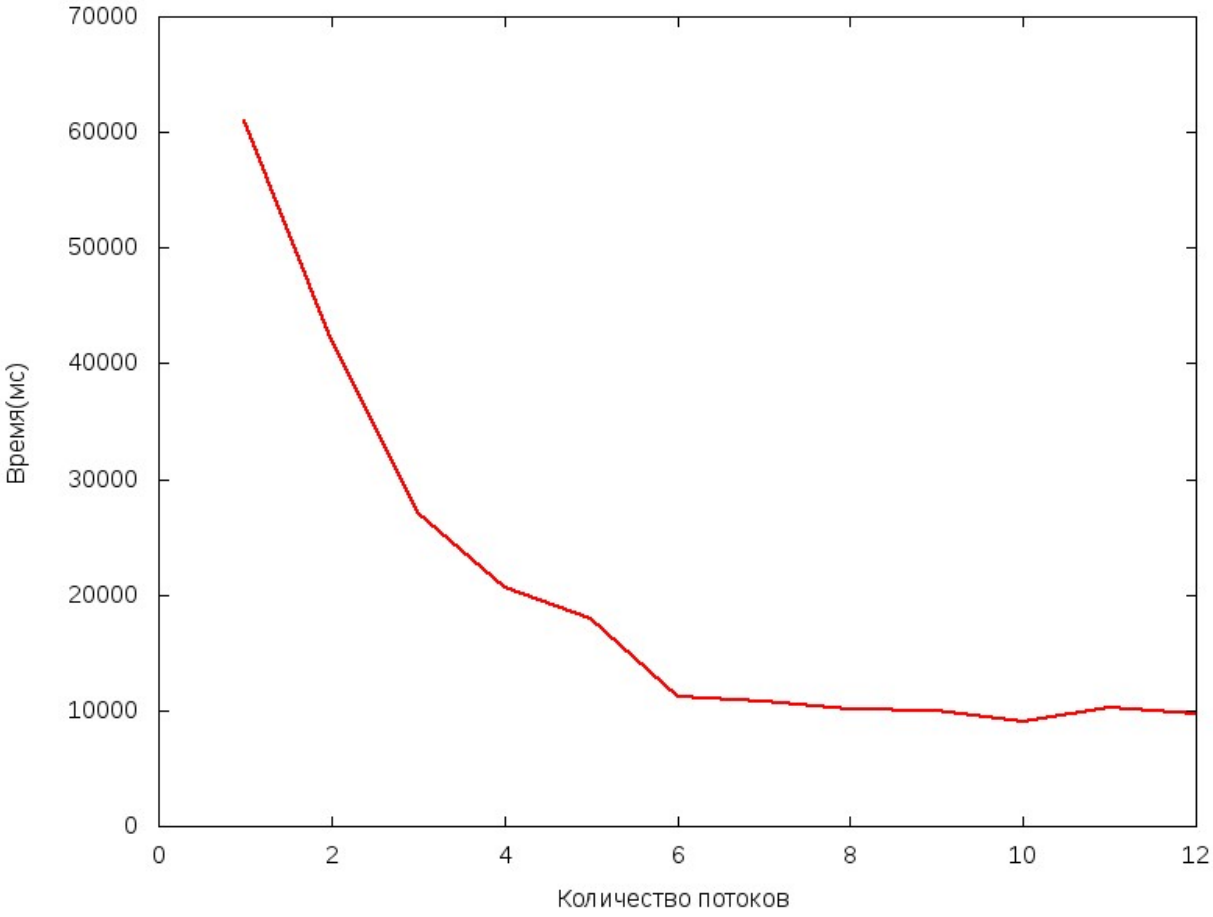
Сперва проведем тест на последовательной версии программы.
Три проведенных теста дали результаты 63484мс, 58744мс, 60549мс возьмем среднее значение за время выполнения последовательной программы (60926мс)

Теперь тестируем параллельный алгоритм.
Значения конфигурационного файла:
Q_D_MIN 5, Q_D_MAX 800, Q_P_MIN 10, Q_P_MAX 200, HASH_SIZE 1000, TIMEOUT 5

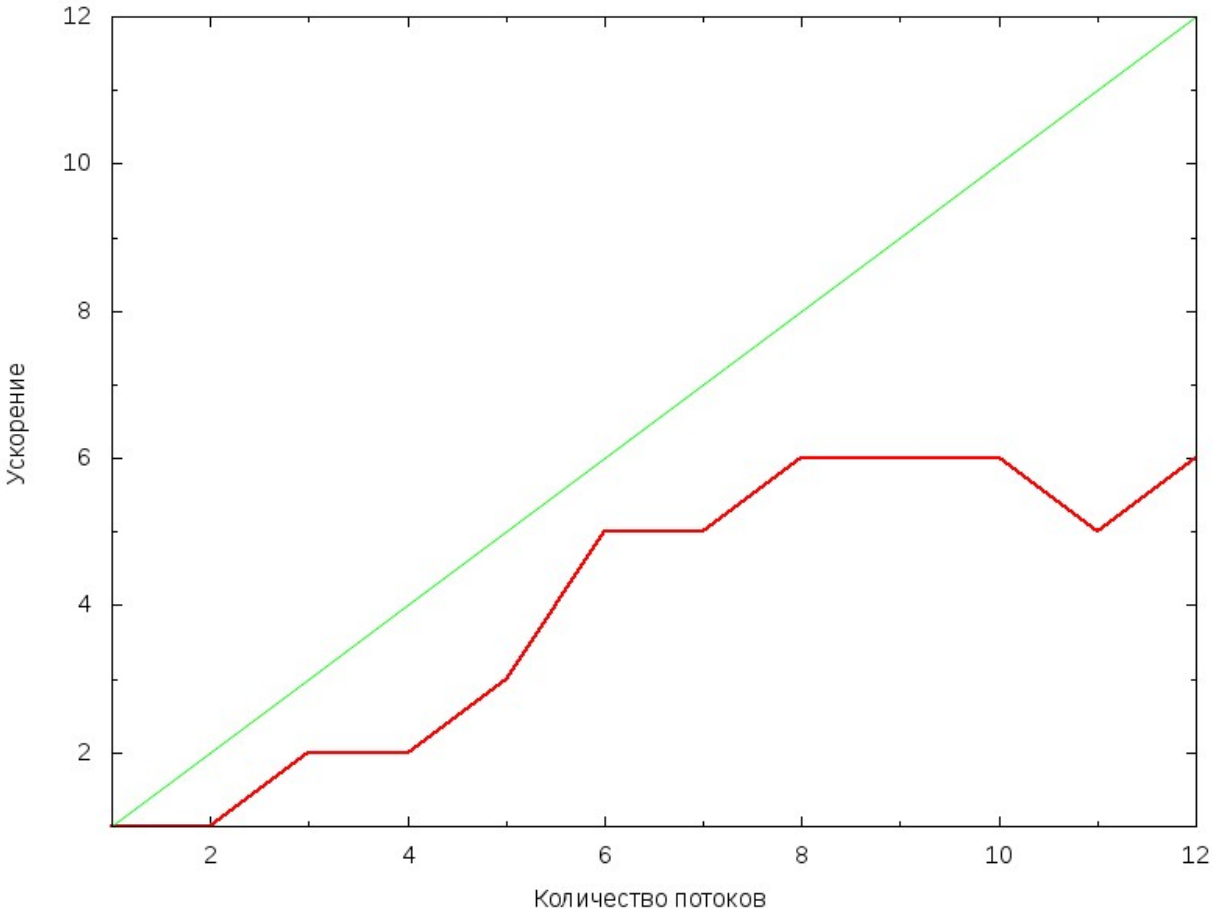
- N_THREADS 2, MAX_PARSE_THREAD 1 - 41937мс
- N_THREADS 3, MAX_PARSE_THREAD 1 - 27101мс
- N_THREADS 4, MAX_PARSE_THREAD 2 - 20671мс
- N_THREADS 5, MAX_PARSE_THREAD 2 - 17889мс
- N_THREADS 6, MAX_PARSE_THREAD 3 - 11131мс
- N_THREADS 7, MAX_PARSE_THREAD 3 - 10802мс
- N_THREADS 8, MAX_PARSE_THREAD 4 - 10142мс
- N_THREADS 9, MAX_PARSE_THREAD 4 - 9943мс
- N_THREADS 10, MAX_PARSE_THREAD 5 - 9095мс
- N_THREADS 11, MAX_PARSE_THREAD 5 - 10303мс
- N_THREADS 12, MAX_PARSE_THREAD 6 - 9718мс

По полученным результатам построим графики:

Зависимость времени загрузки страниц от числа потоков:



Масштабируемость алгоритма:


12
10
8
6
4
2
Ускорение
2
4
6
8
10
12
Количество потоков