УДК 004.451.33

О параллельной разметке графов объектов

Щербина С.А. (Институт систем информатики СО РАН)

Михеев В.В. (Институт систем информатики СО РАН)

В статье рассматривается задача параллельной разметки графа объектов в рамках системы автоматического управления памятью. Авторы формулируют набор ограничений для решения данной задачи и строят алгоритм параллельной разметки, учитывающий эти ограничения. Предложенный алгоритм был реализован в Java-машине Excelsior RVM и апробирован на реальных Java-приложениях. Полученные результаты показывают значительное ускорение разметки графа объектов в большинстве случаев.

Ключевые слова: распределение памяти, сборка мусора, управляемые среды, производительность, параллельные алгоритмы, синхронизация

1. Введение

В реализациях современных языков программирования используется техника автоматического управления памятью, также называемая сборкой мусора[9]. Одной из известных проблем такого подхода является недостаточная производительность приложений, а также, зачастую, возникновение значительных пауз во время исполнения, вызванных работой сборщика мусора. С ростом объёмов используемой приложениями памяти эта проблема только усиливается.

Разметка графа объектов занимает центральное место в задаче сборки мусора. Самые тривиальные алгоритмы сборки мусора состоят именно в разметке, и поэтому её производительность определяет производительность всего сборщика мусора. На практике часто используются инкрементальные[1, 13] сборщики мусора. Несмотря на то, что их малые циклы сборки могут не столь сильно зависеть от производительности разметки, она занимает значительную долю во времени работы полных циклов сборки, то есть влияет на время самых длительных пауз в работе приложения, вызываемых сборщиком мусора. Время разметки актуально и для конкурентных (*concurrent*)[5] сборщиков мусора, которые могут выполняться параллельно с работой приложения, не вызывая вообще или вызывая лишь небольшие паузы в ней, но от длительности разметки графа объектов зависит производительность приложения[5].

С другой стороны, прогресс в росте вычислительной мощности компьютеров свернул с пути увеличения производительности одного процессора в сторону наращивания числа процессоров и ядер. Сочетание этих соображений и послужило мотивацией для выполнения данной работы. Иными словами, целью является исследование возможности применения параллельных алгоритмов для сокращения длительности разметки графа объектов и, как следствие, улучшения производительности и отзывчивости приложений, использующих автоматическое управление памятью.

Работа структурирована следующим образом. Раздел 2 описывает проблемы параллельной разметки. В разделе 3 конструируется и анализируется алгоритм, который призван эти проблемы решать. В разделе 4 описаны проводимые с алгоритмом эксперименты и приведены их результаты. Раздел 5 содержит краткий обзор предшествующих работ. В разделе 6 подводятся краткие итоги и обрисовывается предмет дальнейших исследований.

2. Проблемы параллельной разметки

Для задачи параллельной разметки графов объектов существует ряд серьёзных ограничений.

1. **Отсутствие общего решения**.

   Задача разметки графов в общем случае не решается параллельно. В качестве примера достаточно рассмотреть граф, соответствующий односвязному списку, одной из широко используемых структур данных (Рис. 1).

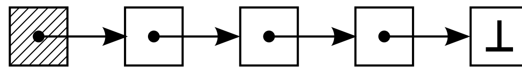
   

   Рис. 1. Контрпример для параллельной разметки графа (заштриховано корневое множество).

   В подобных случаях распределение нагрузки между ядрами принципиально невозможно.

2. **Мелкозернистый параллелизм**

   Этой задаче свойственен *мелкозернистый* параллелизм. Размер индивидуальных неделимых подзадач может оказаться мал, вплоть до нескольких инструкций процессора. Поэтому для эффективного параллельного решения задачи необходимо обеспечить балансировку нагрузки путём взаимодействия между потоками с крайне низкими издержками на синхронизацию. Использование стандартных методов синхронизации, таких, как атомарные операции процессора и, тем более, блокировки потоков

средствами операционной системы, не подходит.

3. **Деградация производительности**

   Параллельный алгоритм решения задачи необходим не «ради параллелизма», а как средство достижения изначально поставленной цели — ускорения разметки графа объектов. Это значит, что параллельный алгоритм не должен быть заметно медленнее последовательного в худшем случае. Учитывая контрпример, приведённый в пункте 1, накладные расходы на взаимодействие между потоками, связанные с балансировкой нагрузки, должны быть минимальными.

4. **Ограниченность дополнительной памяти**

   Ещё одна проблема относится скорее к задаче сборки мусора в общем и связана с требованием ограниченности памяти для структур данных самого сборщика мусора. Традиционно такой структурой является стек разметки (*mark stack*) [9]. С одной стороны, если использовать один только стек, его размер в общем случае должен быть пропорционален размеру кучи, так как количество памяти, необходимое для разметки графа в худшем случае, пропорционально количеству его вершин[1]. С другой стороны, резервирование такого объёма памяти заранее противоречит основной задаче эффективного управления памятью. Поэтому сборщики мусора используют вспомогательные структуры данных и алгоритмы, которые применяются в тех случаях, когда стека фиксированного размера не хватает. В случае параллельного алгоритма такие структуры должны быть дополнительно адаптированы к многопоточной работе.

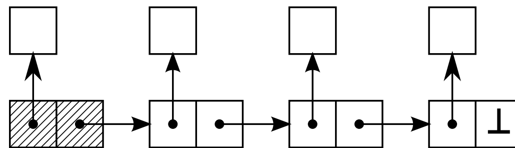


Рис. 2. Граф, требующий $O(n)$ памяти для разметки

5. **Пропускная способность памяти**

   Масштабируемость любого параллельного алгоритма, основное время работы которого занимает работа с памятью, ограничена пропускной способностью памяти.

   В этом можно убедиться, реализовав простую параллельную программу, каждый

[1]Поясним это на примере. Рассмотрим алгоритм обхода графа в глубину. Построим граф, состоящий из одной длинной цепи из n вершин, причём каждая вершина ссылается на ещё одну (листовую) вершину так, что при сканировании объектов в стек попадает сначала листовая вершина, а затем — следующая вершина цепи (Рис. 2). Тогда для разметки такого графа стек должен вмещать n элементов.

поток которой выполняет лишь операции чтения больших объёмов памяти.[2] С увеличением количества работающих потоков при фиксированном объёме работы эта программа работает быстрее (время выполнения становится меньше). Однако, предел этого увеличения меньше, чем количество ядер. Так, например, на четырёхъядерном процессоре Intel Core i5-3470 предельное увеличение достигло примерно 3.19 раз. В аналогичном тесте на запись в память подобный коэффициент оказался ещё ниже: 1.85. Последнее связано с тем, что выполняется не только запись, но и чтение целой линии кэша[6].

Хотя разметка графа объектов состоит не только из операций чтении и записи памяти, она требует интенсивной работы с памятью, поэтому эти константы призваны показать, что предел масштабируемости параллельного алгоритма для неё будет заведомо меньше, чем количество ядер, даже в идеальном случае – при отсутствии зависимостей по данным и издержек синхронизации.

3. Алгоритм

Описанные проблемы накладывают достаточно сильные ограничения на возможные реализации. Именно исходя из этих ограничений мы попробуем построить алгоритм, который будет эффективен для решения поставленной задачи.

Для начала заметим, что синхронизация мелких, связанных с каждым объектом действий недопустима: такой алгоритм не может быть эффективным, поскольку не существует методов синхронизации, накладные расходы которых были бы слабо заметны на фоне работы, связанной с одним объектом.[3]

Требование 1. *Алгоритм не должен запрещать повторную (и даже одновременную) разметку одного и того же объекта несколькими потоками, как и регистрацию его в структурах данных сборщика.*

Поэтому структурой данных для разметки графа был выбран набор стеков: раз синхронизация на уровне каждого объекта недопустима, у каждого потока исполнения должен быть свой стек.

[2]Описание этой программы выходит за рамки данной работы. Отметим лишь, что программа учитывает процессорный кэш, предзагрузку памяти (*memory prefetching*) и TLB[6].

[3]Мы исходим из того, что большинство объектов содержит небольшое количество полей ссылочного типа, сканируемых сборщиком мусора

3.1. Наивный алгоритм

Так мы пришли к простому, «наивному» параллельному алгоритму разметки графа: корневое множество равномерно распределяется между потоками, и затем каждый из них размечает граф, начиная от собственной части корневого множества и используя собственный стек.

Нетрудно, однако, привести пример, на котором такой метод будет неэффективен: размеры подграфов, достижимых из равномощных частей корневого множества, могут быть различными, и тогда, в худшем случае, вся работа достанется одному потоку. Поэтому необходимо распределение нагрузки в ходе разметки графа, т.е. применение какой-либо техники балансировки нагрузки (*load balancing*).

3.2. Алгоритм с балансировкой

Как упоминалось ранее, в худшем случае параллельный алгоритм разметки графа не должен работать заметно медленнее, чем последовательный. Напомним также, что работа может быть распределена по объектам неравномерно.

Требование 2. *Распределением нагрузки должны заниматься не те потоки, у которых есть работа, а те, у которых её нет.*

Предположим обратное, т.е. распределением нагрузки занимаются сами работающие потоки. Неделимым элементом работы будем считать те объекты, которые уже размечены (и помещены в стек), но ещё не просканированы для обнаружения исходящих ссылок. В соответствии с нотацией Дейкстры[9], назовём такие объекты *серыми*. Определение объёма работы, связанной с одним объектом, фактически требует выполнения этой работы, поэтому на этапе распределения нагрузки оценить её невозможно. В худшем случае объекты на стеке вообще не ссылаются на другие объекты, и тогда передача такого объекта другому потоку займёт больше времени, чем просто сканирование. Как следствие, такой алгоритм в худшем случае будет работать заметно медленнее, чем тривиальный однопоточный.

Поэтому в качестве подхода к балансировке нагрузки была выбрана техника, называемая «кража работы» (*work-stealing*)[4, 8].

3.2.1. Техника «кража работы»

Поскольку в нашем алгоритме работа — это сканирование серых объектов со стека разметки, кража работы — это обработка серых объектов с чужого стека.

Для дальнейшего изложения обозначим поток с непустым стеком, выполняющий работу, потоком O (owner), а другой, который пытается украсть у него работу — потоком T (thief).

При отсутствии синхронизации с потоком O, у нас нет никакой возможности явно удалять «украденные» потоком T объекты со стека O. Действительно, между моментом, когда поток T читает элемент стека потока O, и моментом, когда он этот элемент очищает или помечает как «украденный», поток O может переиспользовать этот же элемент стека, помещая туда ссылку на другой серый объект. В этом случае помещённая ссылка будет стёрта, и объект не будет просканирован, т.е. возникнет рассинхронизация данных (*data race*).

Требование 3. *Кража должна состоять лишь в сканировании части объектов с чужого стека. При этом ссылки на эти объекты не должны стираться со стека.*

Таким образом, ссылки на украденные потоком T объекты не стираются, и поток O будет повторно сканировать их. Мы полагаем, что это не должно быть большой проблемой: повторное (но не одновременное) сканирование должно проходить быстрее, так как все объекты, на которые сканируемый объект ссылается, уже помечены при предыдущем сканировании, а значит, повторное сканирование просто ограничится проверкой пометок на объектах-потомках.[4]

При краже поток T сканирует объекты со стека потока O, складывая результаты сканирования (ссылки на объекты-потомки) на свой стек. Можно было бы обойтись простым копированием ссылок со стека O на стек T, но из-за неизбежности повторного сканирования обоими потоками это лишь привнесло бы дополнительные издержки.

Прежде чем перейти к дальнейшему описанию, необходимо сделать небольшое отступление. Нельзя говорить о параллельных алгоритмах, не упомянув синхронизацию кэшей исполняющих устройств (ядер). В рамках данной работы нам будет достаточно знать, что запись в память, копия которой есть в кэше другого ядра, требует синхронизации между кэшами и поэтому вызывает накладные расходы[6]. Отсюда можно вывести ещё два

[4] В качестве альтернативы безусловному повторному сканированию можно явно помечать *чёрные* в нотации Дейкстры[9] (отсканированные) объекты.

важных принципа:

1. Нужно избегать ситуаций, когда разные потоки пытаются размечать одни и те же объекты.
2. Нельзя допускать, чтобы поток T часто читал память, в окрестность[5] которой часто пишет поток O. Это касается, например, верхних элементов стека и указателя на вершину стека.

Для соблюдения этих принципов, введём следующие требования.

Требование 4. *Красть ссылки нужно начиная со дна стека.*

Поток O дойдёт до них в последнюю очередь, поэтому есть больше шансов избежать одновременной работы двух потоков над одними и теми же объектами.

Требование 5. *Нужно избегать кражи ссылок, близких к вершине стека.*

Они будут размечаться потоком O в первую очередь, и поток O часто пишет в память около вершины стека.

Требование 6. *Нельзя допускать одновременную кражу работы одного потока несколькими другими*

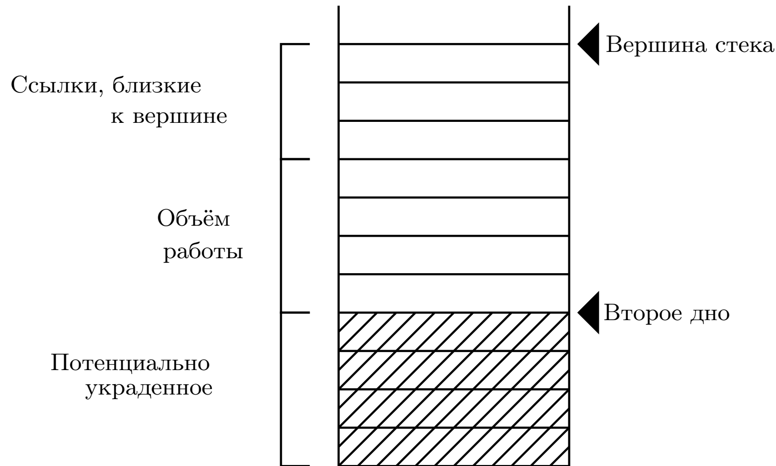
Иными словами, для каждого потока O в каждый момент времени существует только один поток T. Это ограничение упрощает весь алгоритм в целом и задачи синхронизации в частности.

Требование 7. *Необходимо обеспечить, чтобы разные потоки не крали одни и те же объекты.*

Для этого, помимо указателя на вершину стека, введём ещё один указатель (назовём его «вторым дном» стека). Его семантика будет такова: «Выше второго дна ещё никто ничего не крал» (Рис. 3).

---

[5] Имеется ввиду та же линия кэша



Ссылки ниже второго дна могли быть уже украдены. Между вторым дном и вершиной стека гарантированно ещё ничего не украдено, и поэтому ссылки из этого промежутка (кроме ссылок, близких к вершине стека) являются целью для кражи.

Рис. 3. Стек с двойным дном

Изменения этого указателя будут двух видов:

1. **Повышение**. Поток T устанавливает его после кражи в положение выше последней украденной ссылки. Заметим, что мы не можем гарантировать, что ниже этого указателя всё украдено, так как стек при этом используется ещё и потоком O.
2. **Понижение**. Поток O не даёт этому указателю превысить указатель на вершину стека. В случае, когда вершина стека опускается ниже, второе дно устанавливается в то же положение, т.е. равным вершине стека. Этим мы обеспечиваем то, что новые объекты на стеке смогут попадать выше второго дна.

Если потоки крадут работу начиная со второго дна вверх по стеку, то они гарантированно избегают кражи тех объектов, которые уже были украдены ранее.

Требование 8. *Количество украденных за один раз объектов нужно ограничить.*

Если поток T крадёт за один раз большое количество объектов, он может получить больше работы, чем требовалось, оставив поток O без работы. С другой стороны, нельзя красть слишком мало, иначе накладные расходы, связанные с поиском работы и началом кражи, будут заметны. В прототипе за один раз поток T сканировал не больше 64-х объектов.

3.2.2. Поиск работы

Когда у потока T заканчивается работа, он ищет, у кого её можно украсть. Для этого он проверяет каждый поток на предмет того, можно ли украсть работу у него. Если такого

потока не находится, то поток T выполняет короткое ожидание[6], а затем повторяет процесс поиска. В этой простой схеме существует две проблемы. Первая связана с определением момента остановки работы алгоритма разметки, и она обсуждается в разделе 3.2.3. Вторая связана с неэффективностью определения того, есть ли у другого потока работа, которую можно украсть.

Согласно 3.2.1, можно сказать, что потоку T есть что воровать у потока O в том случае, если между вершиной стека и вторым дном есть большой промежуток (объём работы на рис. 3). Проблема в том, что проверять это напрямую неэффективно: оба указателя активно изменяются потоком O в процессе работы, и поэтому частое их чтение другими потоками[7] приводит к существенному замедлению работы потока O.

Вместо чтения указателей поток T будет читать специальный флаг *hasWork*, выставляемый потоком O (*O.hasWork*), означающий, что у потока O есть работа, которую можно украсть. Чтобы не оказывать большого влияния на производительность потока O, выставление *hasWork* будет производиться раз в некоторое количество операций со стеком (в прототипе используется 256). Кроме того, *hasWork* сбрасывается потоком T после кражи в том случае, если он украл всю имеющуюся работу. Этот флаг расположен на отдельной линии кэша, чтобы он не оказался в той же линии кэша, что и другие часто используемые в работе данные, для того, чтобы избежать проблемы *false-sharing*[3]. Псевдокод схемы работы потока O приведён на рис. 4.

```
цикл
    если свой стек пуст то
        выйти из цикла
    конец
    извлечь объект со своего стека и просканировать его
    если второе дно > указатель на вершину стека то
        второе дно ← указатель на вершину стека
    конец
    выставить hasWork в зависимости от вершины стека и второго дна
конец
```

Рис. 4. Схема работы потока O

Наконец, в соответствии с требованием 6, у каждого потока в каждый момент времени красть работу может только один поток. Для этого каждый поток имеет мьютекс (*O.mutex*), который исключает одновременные кражи у потока O. Этот мьютекс захва-

[6] В целях производительности такое ожидание должно быть реализовано без блокировки потока средствами операционной системы

[7] А их чтение будет частым в том случае, когда у большого числа потоков нет работы

тывается и освобождается потоком T и может быть тривиальным образом реализован через атомарный флаг. Псевдокод схемы кражи работы потоком T у потока O приведён на рис. 5.

если $O.hasWork$ **то**
  попробовать захватить $O.mutex$
  если удалось захватить **то**
    украсть работу
    освободить $O.mutex$
  конец
конец

Рис. 5. Схема работы потока T

3.2.3. Проблема останова

Каждый работающий поток должен завершиться вовремя, то есть

1. Поток не должен завершиться раньше, чем просканированы все объекты
2. После того, как все объекты просканированы, все потоки должны завершиться.

Для решения этой проблемы введено два возможных состояния потока:

- **Занят**.

  Поток работает. К этому относится как кража работы, так и сканирование объектов с собственного стека.

- **Свободен**.

  Поток ищет работу.

Переход из первого состояния во второе происходит после того, как поток закончил обработку собственного стека, из второго в первое — в момент, когда он нашёл, у кого можно красть работу, перед непосредственной кражей.

Как только бездействующий поток видит, что в какой-то момент времени все остальные потоки также бездействуют, он должен завершиться. Для того, чтобы сделать такую проверку потоково-безопасной, вводится *counter* – атомарный счётчик[8] числа потоков, находящихся в состоянии "занят". Как только очередной поток выходит из состояния "занят", он уменьшает счётчик и проверяет, не стал ли он равен нулю. В последнем случае выставляется *exit* – глобальный флаг завершения. Каждый поток периодически (между попытками кражи) проверяет значение этого флага.

[8] Для проблемы останова было найдено решение, не использующее атомарные операции процессора, но оно не принесло видимых улучшений в производительности и поэтому, будучи более сложным, в работе не описывается

На рис. 6 приведён псевдокод общей схемы алгоритма. Изначально *counter* равен количеству используемых потоков, а *exit* равен $false$. Вложенный цикл — это цикл, в котором поток находится, пока ищет работу или ждёт завершения.

```
цикл
    обработать собственный стек согласно рис. 4
    val ← атомарно уменьшить counter на 1 и взять новое значение
    если val = 0 то
        exit ← true
    конец

    цикл
        если exit то
            завершиться
        конец

        если поток может украсть работу согласно рис. 5 то
            атомарно увеличить counter на 1
            украсть работу
            выйти из цикла
        конец

        ожидание
    конец

конец
```

Рис. 6. Общая схема алгоритма (тело рабочего потока)

Теорема 1 (о корректности). *После завершения алгоритма все достижимые объекты помечены.*

Доказательство. Пусть алгоритм завершился. Тогда флаг *exit* был выставлен, следовательно, в какой-то момент времени *counter* оказался равен нулю после очередного уменьшения. Это значит, что на этот момент все стеки пусты. Рассмотрим совокупность стеков как очередь серых объектов в алгоритме разметки графа. Поскольку каждый объект удаляется из этой очереди только после того, как был просканирован, и в данный момент очередь пуста, то все достижимые объекты уже помечены. □

Теорема 2 (о завершимости). *Алгоритм завершается.*

Доказательство. Пусть *counter* оказался равен нулю после очередного уменьшения. Тогда выставляется флаг *exit*. Поскольку все стеки на этот момент пусты, и ни один поток не находится в процессе кражи, все дальнейшие операции в псевдокоде будут выполняться

быстро (поскольку станут тривиальными), и поэтому через короткое время каждый поток прочитает выставленный флаг *exit* и завершит работу.

Таким образом, все потоки завершат работу вскоре после того, как счётчик *counter* обнулится. Покажем, что рано или поздно это произойдёт. Действительно, каждый объект может попасть на каждый стек не более одного раза (в момент маркировки). При этом каждый элемент каждого стека может быть просканирован не больше двух раз (один раз – хозяином стека, второй – вором). Таким образом, каждый объект может быть просканирован не более $2p$ раз, где p — это число потоков. Пусть n – это число достижимых объектов, тогда общее число операций сканирования ограничено сверху $2pn$. Пока счётчик не равен нулю, найдётся поток, который занят сканированием объектов. Таким образом, общее число выполненных операций сканирования монотонно возрастает. А значит, раз ограничение не может быть превышено, рано или поздно счётчик обнулится, и тогда все потоки будут завершены. □

Но как скоро после фактического выполнения всей работы *counter* обнулится? Мы не можем гарантировать, что это случится *сразу после того*, как все достижимые объекты будут помечены и просканированы: раз ссылки не удаляются при краже, потоки ещё просканируют и снимут все объекты со своего стека, не кладя ничего нового.

3.3. Анализ худших случаев

Несмотря на все попытки избежать проблем, связанных с неравномерностью распределения работы по стекам, предложенный алгоритм им подвержен, так что возможно построить очевидные контрпримеры, основанные на этой неравномерности.

Один из таких примеров изображён на рисунке 2. При разметке такого графа объектов поток O будет размечать список: каждый раз, снимая со стека очередной элемент списка, он кладёт на стек листовую вершину и следующий элемент списка, который будет снят с вершины стека на следующей итерации. Таким образом, на стеке накапливаются листовые объекты. Любой поток T, который будет красть работу у потока O, будет сканировать листовые вершины, не совершая никакой полезной работы. При этом каждый из этих объектов будет повторно просканирован потоком O. Учитывая ограниченность пропускной способности памяти, выполнение лишних операций с памятью и взаимодействие между потоками приводит к деградации производительности (см. раздел 4.3).

Подойдём к этому вопросу более формально. Отношение времени работы классическо-

го однопоточного алгоритма ко времени работы параллельного алгоритма назовём *ускорением* этого параллельного алгоритма. Величина ускорения, меньшая 1, означает, что наблюдается не ускорение, а замедление (деградация производительности). Тогда контрпримером будем считать входные данные, на которых ускорение минимально.

Пусть однопоточный алгоритм работает за время $c_1 \cdot N + c_2 \cdot M$, где N – это число объектов, а M – число ссылочных полей в этих объектах.

Рассмотрим алгоритм с балансировкой. В общем случае анализ времени его работы достаточно затруднён. Будем искать худшие случаи среди тех, где есть поток О, который всё время занят обработкой своего стека, т.е. ничего не крадёт. Тогда время работы всего алгоритма равно времени работы потока О. Пусть поток О обрабатывает n объектов, у которых есть m ссылок, и за это время у него украли (т.е. просканировали с его стека) s объектов. Во-первых, часть времени его работы занимает обработка объектов, т.е. $c_1 \cdot n + c_2 \cdot m$, как у однопоточного алгоритма. Но у алгоритма также есть и накладные расходы. Во-первых, это расходы на проверки, нужно ли выставлять флаг наличия работы (см. раздел 3.2.2). Поскольку они привязаны к операциям со стеком, мы будем считать, что они пропорциональны количеству объектов, т.е. n. Во-вторых, непосредственная кража объектов также вызывает накладные расходы, причём, не только у вора, но и у хозяина стека. Будем считать, что они пропорциональны числу украденных объектов. Таким образом, время работы потока О составит $c_1 \cdot n + c_2 \cdot m + c_3 \cdot n + c_4 \cdot s$.

Мы ищем худшие случаи, то есть, пытаемся минимизировать ускорение. Иными словами, мы пытаемся максимизировать отношение:

$$max \longleftarrow \frac{c_1 \cdot n + c_2 \cdot m + c_3 \cdot n + c_4 \cdot s}{c_1 \cdot N + c_2 \cdot M} \tag{3.1}$$

При фиксированных N и M это отношение тем больше, чем больше n, m и s. Значит, максимум следует искать среди тех случаев, где поток О обрабатывает весь граф, т.е.

$$max \longleftarrow \frac{c_1 \cdot N + c_2 \cdot M + c_3 \cdot N + c_4 \cdot s}{c_1 \cdot N + c_2 \cdot M} = 1 + \frac{c_3 \cdot N + c_4 \cdot s}{c_1 \cdot N + c_2 \cdot M} \tag{3.2}$$

Зафиксируем N. Тогда чем меньше M и чем больше s, тем больше (3.2). Но, с другой стороны, $s \leq N$ и $M \geq N - 1$. Таким образом, если мы представим граф объектов, в котором M и s близки к N, его можно будет считать худшим случаем.

Вернёмся к рисунку 2. Этот граф соответствует тому требованию, что все объекты

фактически обрабатываются одним потоком. При этом $s = N/2$, а $M = N$. Можно ли составить граф, где эти показатели были бы ещё хуже?

Пусть в графе на рисунке 2 каждый элемент списка непосредственно ссылается не на один листовой объект, а на k. Здесь по-прежнему $M = N$. При этом каждый листовой объект будет украден, поэтому $s = N \cdot \frac{k}{k+1}$. Таким образом, увеличивая k, можно составить сколь угодно близкий к худшему случаю граф.

4. Экспериментальные результаты

4.1. Методология эксперимента

Описанный алгоритм был реализован в рамках исследовательской виртуальной Java машины Excelsior RVM[10], которая поддерживает полный стандарт платформы Java 8.

Измерения проводились следующим образом. В качестве длительности разметки графа объектов, на основании которой подсчитывается ускорение (см. раздел 3.3), использовалась суммарная длительность фаз разметки по всем вызовам сборщика мусора во всех запусках теста. Каждый из тестов запускался по 3-4 раза. Поскольку разброс результатов между различными запусками тестов был не слишком велик, такого числа измерений хватает для оценки.

Для измерений виртуальная машина была сконфигурирована так, чтобы выполнялись только полные циклы сборки мусора, во время которых размечается весь граф объектов. Это было сделано для увеличения нагрузки.

4.2. Тестовый набор

Измерения проводились на тестах двух категорий. Первая состоит из разметки синтетических графов, таких как односвязные и двусвязные списки, бинарные деревья и леса, и так далее. Несмотря на то, что алгоритм предназначен для разметки графов объектов реальных приложений, а не синтетических тестов, использование таких тестов помогает выявить патологические случаи и улучшить множество аспектов алгоритма. Это позволит в общем случае получить прирост производительности и на реальных примерах.

В качестве синтетических графов использованы двусвязный список (DList), наборы односвязных списков (SList, SListMultiple), односвязный список с листовыми вершинами (SListML) (см. раздел 3.3), односвязный список с присоединёнными к элементам небольшими подграфами (SListD), бинарные леса (Tree2, Tree2Multiple), тернарное дерево (Tree3)

и «список с вкраплениями алмазов» (DiamondList), который строится таким образом: вводится последовательность основных вершин, и каждые две соседние соединяются поочерёдно одним либо двумя односвязными списками (Рис. 7).

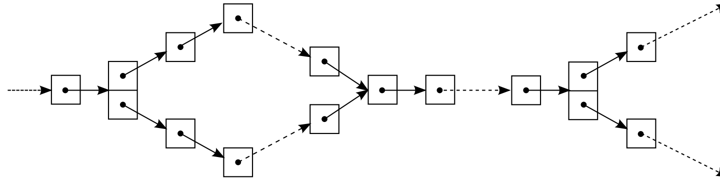


Рис. 7. DiamondList

Число вершин и дуг, а также мощность корневого множества для графов синтетических тестов приведены в таблице 1. Тесты с «маломощным» корневым множеством предназначены для оценки качества балансирования нагрузки.

Тест	**Кол-во вершин**	**Кол-во дуг**	**Корневое множество**
DiamondList	15250021	15500020	1
DList	20000001	40000001	1
SList	19000002	19000000	2
SListD	16000032	32000063	1
SListL	2000002	2000001	1
SListML	1050021	1050020	1
SListMultiple	20000040	20000000	40
Tree2	67108863	134217726	1
Tree2Multiple	20971480	41942960	40
Tree3	21523360	64570080	1

Таблица 1

Характеристики синтетических тестов

Вторая категория — это разметка графов объектов реальных приложений. В список приложений входят тесты производительности DaCapo.ps[2] (рендеринг PostScript-документа), SPECjbb2000[12] (трёхуровневая клиент-серверная система) и SPECjvm2008.derby[11] (нагрузочный тест для СУБД Apache Derby).

В синтетических тестах сборка мусора вызывалась вручную по 10 раз. В тестах с графами объектов реальных приложений вызовы сборки мусора осуществлялись так же, как и при обычной работе этих приложений.

4.3. Результаты и обсуждение

Измерения проводились на компьютере, оснащённом четырёхъядерным процессором Intel Core i5-3470 с частотой ядер 3200 MHz с 8 GB оперативной памяти под управлением

64-битной операционной системы Windows 7. Процессор содержит 64 Кб кэша первого уровня и 256 Кб кэша второго уровня для каждого ядра и 6 Мб общего кэша третьего уровня.

В таблице 2 приведены результаты измерений. Первая колонка содержит название теста, вторая и третья — ускорения (см. разд. 3.3) наивного параллельного алгоритма и алгоритма с балансировкой по отношению к однопоточному, соответственно.

Тест	**naïve**	**stealing**
DList	1.02	0.91
DiamondList	1.00	1.23
SList	1.95	1.80
SListD	1.00	1.79
SListML	0.99	**0.88**
SListMultiple	2.64	2.65
Tree2	1.03	2.54
Tree2Multiple	3.60	3.50
Tree3	1.06	3.40
SPECjbb2000	2.28	2.57
SPECjvm2008.derby	1.28	**2.46**
DaCapo.ps	1.00	1.76

Таблица 2

Ускорение разметки

Наличие небольших ускорений в тех тестах, где их вообще быть не должно[9], объясняется тем, что помимо размечаемого синтетического графа в куче есть также и другие объекты в небольшом количестве, и их удаётся размечать параллельно. На тесте SListML наблюдается ожидаемое замедление. Напомним, что тест SListML является контрпримером к алгоритму. Он основан на том, что динамическое распределение работы на таком графе не приносит никакой выгоды (см. раздел 3.3). На тесте DList работу распределить нельзя, поэтому в качестве причины такой деградации можно подозревать только дефекты в реализации протокола опроса работающего потока ворующими (раздел 3.2.2) или в алгоритме останова (раздел 3.2.3). Исследование причин, вызывающих такое замедление, оставлено на будущее.[10] Тесты Tree2 и Tree3 показывают хорошую работу механизма балансировки нагрузки: наивный алгоритм на них не даёт преимуществ, тогда как алгоритм с балансировкой показывает значительное увеличение производительности. На тестах SListD и DiamondList, которые можно считать хорошими случаями для алгоритма,

[9]Например, для наивного алгоритма это тесты с одноэлементным корневым множеством

[10]Есть предположение, что одной из основных причин замедления здесь является *false sharing*[3]

также можно видеть преимущество балансирующего алгоритма над наивным. На тестах SListMultiple и Tree2Multiple видно, что балансировка нагрузки не всегда вносит значительные деградации в наивный параллельный алгоритм: на каждом из этих тестов оба ускорения сравнимы. На рис. 8 приведены графики ускорений обоих алгоритмов. Наконец, тесты SPECjbb2000, SPECjvm2008.derby и DaCapo.ps показывают, что балансировка нагрузки даёт существенный выигрыш на графах реальных приложений. На рис. 9, 10 и 11 приведено сравнение роста ускорений алгоритмов на этих тестах.

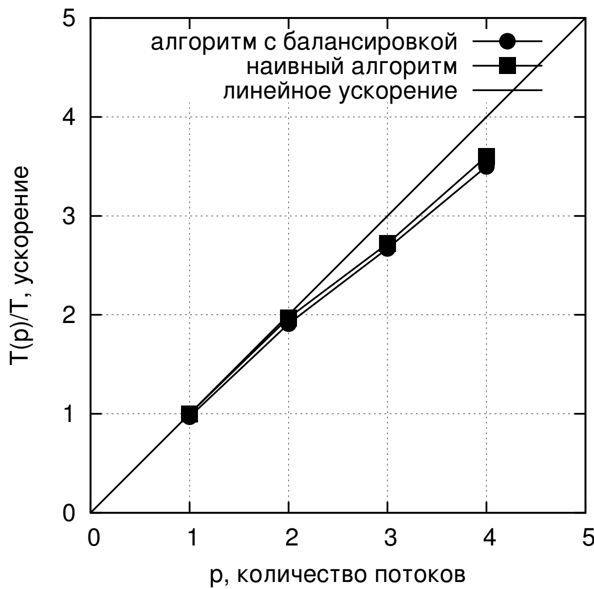


Рис. 8. Ускорения разметки на тесте Tree2Multiple

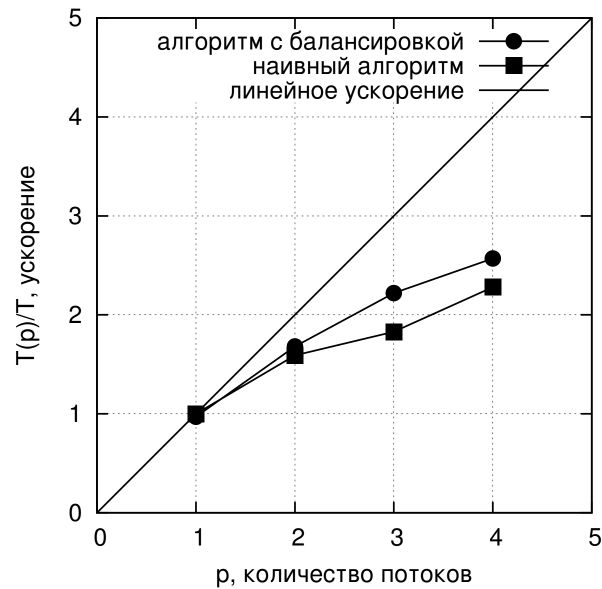


Рис. 9. Ускорения разметки на тесте SPECjbb2000

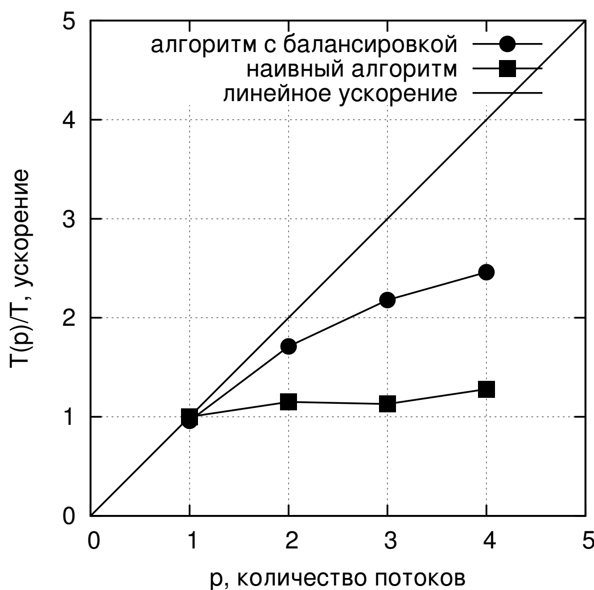


Рис. 10. Ускорения разметки на SPECjvm2008.derby

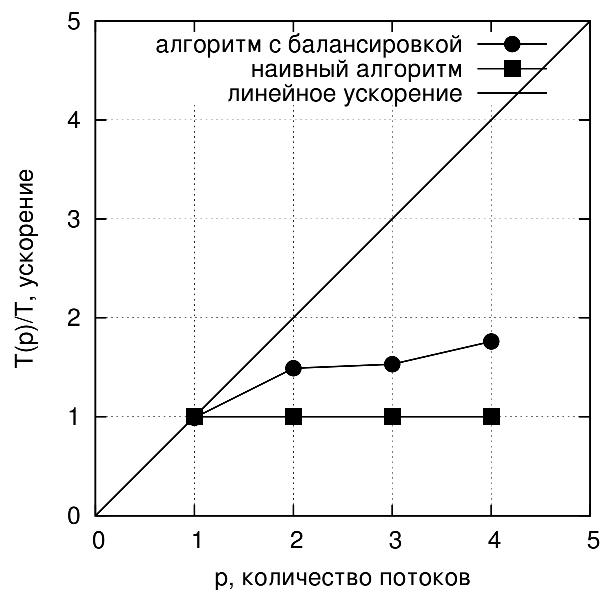


Рис. 11. Ускорения разметки на тесте DaCapo.ps

Предложенный алгоритм показывает здесь хороший результат, что подтверждает адек-

ватность положенных в его основу идей.

5. Обзор предшествующих работ

В данном разделе приведён критический обзор работ, предлагающих различные подходы к параллельной разметке графа объектов.

В работе [7] описаны параллельные версии разных алгоритмов, используемых классическими сборщиками мусора. Авторы отмечают необходимость динамической балансировки нагрузки для разметки графа объектов и так же, как и мы, выбирают технику кражи работы. В работе предлагается снабдить каждый поток отдельной структурой данных для разметки. В качестве таковой выбирается не стек, а специальная очередь с тремя операциями:

1. Добавление в начало потоком O; не требует синхронизации
2. Извлечение из начала потоком O; синхронизация используется в том случае, когда извлекается последний элемент
3. Извлечение с конца произвольным потоком T; синхронизация используется всегда

В работе потока O используются первые две операции, поэтому связанные с синхронизацией накладные расходы для потока O незаметны.

Однако, операция извлечения элемента из начала такой очереди гораздо сложнее, чем операция извлечения элемента с вершины обычного стека, что должно отрицательно сказываться на производительности в худшем случае. В статье не приведены измерения работы алгоритма на синтетических графах объектов (например, на односвязном списке), поэтому оценить степень деградации не представляется возможным. В целом подход выглядит удачным по результатам измерений на реальных приложениях, однако размеры некоторых из графов объектов этих приложений недостаточно велики.

В статье [14] представлена иная техника динамической балансировки. Авторы описывают специальную структуру данных — очередь для передачи задач от одного потока к другому, которая не требует синхронизации. Основная идея такой очереди в том, что в ней разрешены только две операции: один фиксированный поток может извлекать элементы из начала очереди, а другой — добавлять в конец. Авторы предлагают использовать очередь малой длины, но при этом не адресуют проблему замедления, вызванную синхронизацией кэшей процессоров. При этом результаты измерений на синтетических тестах, которые показали бы масштаб проблемы, в работе не приведены, не анализируются контрпримеры,

а общее количество тестов мало. Кроме того, в результатах встречаются малоправдоподобные, с нашей точки зрения, данные. Так, например, согласно графикам, добавление атомарной операции для разметки каждого объекта влияет на производительность лишь незначительно, что разительно расходится с нашим опытом. Таким образом, данная работа после изучения оставляет ряд вопросов, и для сравнения результатов с другими работами необходимо реализовать предлагаемый алгоритм и измерить его характеристики независимо.

6. Заключение

В настоящей работе описаны основные проблемы, возникающие при параллельной разметке графов объектов в трассирующих сборщиках мусора, и приведён алгоритм, который призван эти проблемы решать. Алгоритм был реализован в управляемой среде Excelsior RVM[10], а проведенные измерения показали обоснованность выбранных решений. Так, коэффициент ускорения на 4-х ядерной машине при тестировании реальных приложений варьируется от 1.76 до 2.57.

Несмотря на то, что прототип продемонстрировал хорошие результаты на графах объектов реальных приложений, решены ещё не все проблемы с производительностью на некоторых синтетических тестах. Также не проведено непосредственное сравнение производительности описанного алгоритма с алгоритмами из других работ [7, 14], что требует их реализации в той же управляемой среде.

Некоторые допущения при разработке алгоритма основаны на опыте или результатах экспериментов. Они могут быть слишком категоричными и затрагивать только частные случаи, поэтому необходимо исследовать и другие возможные подходы к параллельной разметке.

Наконец, задача параллельной разметки может и должна рассматриваться в контексте всей задачи сборки мусора. Например, фаза построения корневого множества для разметки графа, которое меняется в ходе работы программы, также является вычислительной задачей и может быть выполнена параллельно. Кроме того, стратегия распределения корневого множества между работающими потоками обладает рядом степеней свободы, и выбор правильной схемы может быть обусловлен анализом корневого множества с опорой на семантику среды исполнения и программы. Перечисленные задачи составляют предмет дальнейших исследований.

Список литературы

1. Appel A. W. Simple generational garbage collection and fast allocation // SoftwareвЂ"Practice & Experience. 1989. Vol. 19, 2. P. 171–183.
2. Blackburn S. M., Garner R., Hoffmann C. et al. The DaCapo benchmarks: java benchmarking development and analysis // Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. New York, NY, USA: ACM, 2006. P. 169–190.
3. Bolosky W. J., Scott M. L. False Sharing and its Effect on Shared Memory Performance // Proceedings of the USENIX SEDMS IV Conference. 1993.
4. Burton F. W., Sleep M. R. Executing functional programs on a virtual tree of processors // Proceedings of the 1981 conference on Functional programming languages and computer architecture. 1981.
5. Click C., Tene G., Wolf M. The Pauseless GC Algorithm // Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. VEE '05. New York, NY, USA: ACM, 2005. P. 46–56. URL: `http://doi.acm.org/10.1145/1064979.1064988`.
6. Drepper U. What Every Programmer Should Know About Memory. `http://people.redhat.com/drepper/cpumemory.pdf`.
7. Flood C. H., Detlefs D., Shavit N., Zhang X. Parallel garbage collection for shared memory multiprocessors // Proceedings of the 2001 Symposium on Java^{TM}Virtual Machine Research and Technology Symposium - Volume 1. JVM'01. Berkeley, CA, USA: USENIX Association, 2001. P. 21–21.
8. Halstead R. H. Implementation of multilisp: Lisp on a multiprocessor // Proceedings of the 1984 ACM Symposium on LISP and functional programming. 1984.
9. Jones R., Lins R. Garbage collection: algorithms for automatic dynamic memory management. New York, NY, USA: John Wiley & Sons, Inc., 1996.
10. Mikheev V., Lipsky N., Gurchenkov D. et al. Overview of excelsior JET, a high performance alternative to java virtual machines // Proceedings of the 3rd international workshop on Software and performance. New York, NY, USA: ACM, 2002. P. 104–113.
11. SPEC releases free SPECjvm2008 benchmark. Press release. URL: `http://www.spec.org/jvm2008/press/release.html`.
12. SPECjbb2000, A Java Business Benchmark. White paper. URL: `http://www.spec.org/`

`jbb2000/docs/whitepaper.html`.

13. Ungar D. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm // Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments. New York, NY, USA: ACM, 1984. P. 157–167.
14. Wu M., Li X.-F. Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations // IEEE International Parallel and Distributed Processing Symposium. 2007.