Реализация менеджера памяти в work-stealing балансировщике[1]

Е. А. Барковский[2]
barkevgen@gmail.com

В работе описывается менеджер памяти, реализованный в экспериментальном work-stealing балансировщике задач. Менеджер разработан на основе математических моделей оптимального управления work-stealing деками. В менеджере реализованы и проанализированы следующие методы управления памятью: оптимальное деление; деление пополам. Также, работа экспериментального менеджера сравнивается с работой стандартного менеджера памяти.

Ключевые слова: оптимальное управление, work-stealing деки, менеджеры памяти.

1. Введение

Стратегии балансировки параллельных вычислений на многоядерной архитектуре разделяют на статические и динамические. Статическая балансировка используются когда известно практически все про выполняемые задачи. В этом случае можно заранее определить оптимальное расписание задач (например, минимизировать среднее время решения), но такие задачи считаются NP-полными и встречаются достаточно редко. При динамической балансировке балансировщик во время работы использует некую относительно простую стратегию, которая дает результат близкий к оптимальному. Одна из таких стратегий балансировки называется “work-stealing” – процессоры, ставшие пустыми, пытаются перехватить часть работы у других процессоров. Этот метод реализован в большом числе систем, например: Cilk [1]; TBB [2]; TPL [3]; X10 [4] и другие.

В этом методе балансировки нагрузки каждый процессор решает ряд задач, указатели на которые хранятся в деке этого процессора. Когда процессор создает новую задачу, он добавляет указатель

[1]Работа выполнена при поддержке РФФИ, проект №15-01-03404-а.
[2]©Е. А. Барковский, 2017

на нее в свой дек; когда процессору нужна задача, он берет указатель из вершины дека. Если процессор узнает, что его дек пуст, он перехватывает указатели на задачи у другого процессора. Первые две операции выполняются как в LIFO-стеке, а перехват происходит из основания дека – как в FIFO-очереди. Т.е. мы работаем (в терминологии Д. Кнута) с деком с ограниченным входом [5]. В [6] рассматривался перехват одного элемента, в [7] – половины элементов.

В предыдущих работах мы предлагали математические модели последовательного циклического представления work-stealing деков, где каждый дек расположен в отдельном участке памяти и на каждом шаге дискретного времени с ними происходят операции с заданными вероятностями [8,9]. В этой статье будет описана и проанализирована одна из реализаций вышеупомянутых моделей и методов, как менеджера памяти встроенного в work-stealing балансировщик задач.

На практике, для работы с деками в виде циклических массивов часто используются классические методы работы с кучей в трансляторах C/C++. Для каждого дека из кучи запрашивается массив. Когда он становится достаточно большой, то запрашивается новый массив, например, в два раза больший – туда копируются элементы дека, а старый массив отдается в список свободных блоков кучи. Если же наоборот, дек стал достаточно мал, то те же действия производятся с массивом, меньшим старого в два раза. Мы же в этой статье рассматриваем задачу поиска оптимальных альтернативных методов управления памятью для деков, в предположении, что известны некоторые их вероятностные характеристики.

2. Математическая модель

Приведем постановку задачи математической модели [9], на основе которой был разработан менеджер памяти.

Пусть в памяти размера m мы работаем с двумя последовательными циклическими деками, где все элементы имеют фиксированный размер в одну структурную единицу. Для последовательного представления деков выделим каждому некоторое количество единиц памяти из общего объема m. Пусть s – количество единиц па-

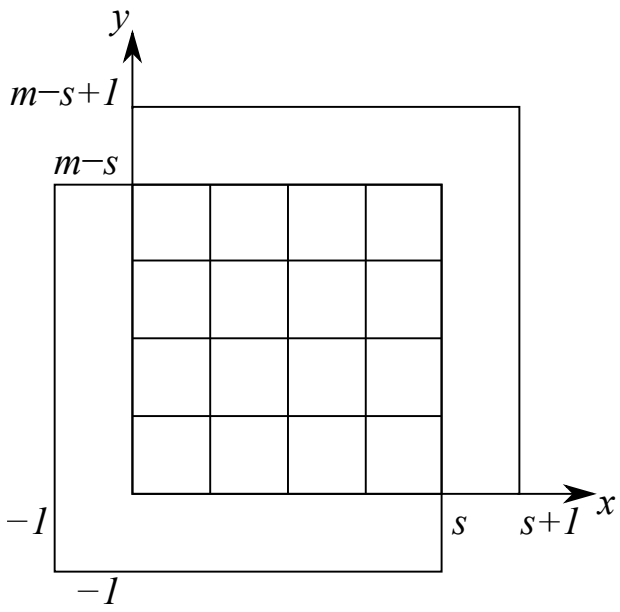

Рис. 1: Область блуждания

мяти, выделенное первому деку, тогда $m - s$ – количество памяти, выделенное второму деку.

Пусть x и y текущие длины первого и второго дека соответственно. В качестве математической модели мы рассматриваем случайное блуждание по двухмерной целочисленной решетке в области $-1 \leq x \leq s + 1$, $-1 \leq y \leq m - s + 1$ (Рис. 1).

Некоторые вероятностные характеристики операций, производимых над деками, заранее известны:

- включение элемента в первый дек с вероятностью p_1;
- включение элемента во второй дек с вероятностью p_2;
- включение элементов параллельно в оба дека – p_{12};
- исключение элемента из первого дека с вероятностью q_1;
- исключение элемента из второго дека с вероятностью q_2;

Таблица 1: Оптимальное разбиение памяти, перехват одного элемента

Входные данные	Среднее время до переполнения ($m = 100$)	
	оптимальное разделение	разделение пополам ($s = 50$)
Задача перемножения матриц		
$p_1 = 0.071$, $p_2 = 0.108$, $q_1 = 0.071$, $q_2 = 0.108$, $p_{12} = 0.014$, $q_{12} = 0.014$, $pq_{12} = 0.013$, $pq_{21} = 0.013$	11549 ($s = 46$)	11345

- исключение элементов параллельно из обоих деков – q_{12};
- включение в первый дек и исключение из второго – pq_{12};
- включение во второй дек и исключение из первого – pq_{21};
- деки не изменяют своей длины с вероятностью r (чтение или отсутствие операции).

$p_1 + p_2 + p_{12} + q_1 + q_2 + q_{12} + pq_{12} + pq_{21} + r = 1$. Как только дек становится пустым, он начинает перехватывать элементы у другого дека. Соответственно, исключение элемента из пустого дека не является аварийной ситуацией.

Блуждание начинается в начале координат. Прямые $x = s + 1$ и $y = m - s + 1$ образуют два поглощающих экрана – когда процесс попадает на эти экраны память переполняется и программа аварийно прекращается; Прямые $x = -1$, $y = -1$ образуют два отражающий экрана. Введение данных экранов позволяет установить процесс перехвата элемента. Формально деки попадают на экраны, но фактически – в область $1 \leq x < s$, $1 \leq y < m - s$. Для случая перехвата одного элемента: если процесс находится на точках $(x, -1)$ или $(-1, y)$, он перейдет на точки $(x - 1, 1)$ или $(1, y - 1)$.

Здесь в качестве критерия оптимальности рассматривается максимальное среднее время до переполнения памяти (т.е. до попадания на прямые $x = s + 1$ и $y = m - s + 1$).

В таблице 1 приведены некоторые результаты вычислений. Исходными данными являются оценки вероятностей (по частоте) работы системы при решении задачи перемножения матриц. Они были получены в результате экспериментов с разработанным work-stealing балансировщиком задач [10].

3. Реализация балансировщика задач

Кратко опишем принцип работы экспериментального work-stealing балансировщика [10], в который был встроен менеджер памяти.

Задачи, выполняемые балансировщиком, состоят из инструкций, которые должны быть выполнены последовательно, но во время своей работы они могут создавать подзадачи, которые можно выполнять параллельно с родительской. Таким образом, задачи можно представить в виде графа, в котором каждая задача указывает на создавшую ее родительскую задачу.

Например, задачу для вычисления чисел Фибоначчи можно записать (на псевдокоде) следующим образом:

```
1  function fib(n)
2  if n < 2 then
3  return n;
4  end if
5
6  x = spawn fib(n - 1);
7  y = spawn fib(n - 2);
8  sync;
9  return x + y;
10 end function
```

При выполнении $fib(n)$, задача создаст две подзадачи: $fib(n-1)$ и $fib(n-2)$. Созданные подзадачи будет помещены в очередь потока и могут быть украдены и выполнены другими потоками. Для того, чтобы дождаться завершения подзадач, вызывается функция *sync*, которая передает управление балансировщику, позволяя ему выполнять другие задачи, пока у родительской не завершатся все подзадачи. Задачи для выполнения поток сначала извлекает из своей очереди, а, когда они заканчиваются, начинает красть из очереди случайно выбранного потока.

Так как над очередью задач могут выполняться операции добавления и извлечения только ее владельцем, а операция перехвата другими потоками, то в реализациях балансировщиков используют очереди с двумя концами, т. н. деки (double ended queues).

В одной из последних и наиболее цитируемых работ [11] предложена реализация деков на основе циклического массива, с неблокирующими синхронизациями операций и с возможность динамического изменения размера. Относительно других реализаций, она нуждаются в меньших накладных расходах на выполнение операций и по использованию памяти. Работающие версии можно встретить в балансировщиках задач в библиотеке Intel TBB и в языке Rust.

На языке C++ 11 был реализован work-stealing балансировщик задач, работающий по описанному выше алгоритму и использующий реализацию деков из [11]. (Исходные коды и реализация на сайте `https://github.com/rkuchumov/staccato`.)

4. Реализация менеджера памяти

Экспериментальный менеджер памяти был встроен в work-stealing балансировщик задач. Для его реализации потребовалось переписать функции балансировщика, отвечающие за создание новых деков. В результате чего стало возможным контролировать размер каждого дека по отдельности.

Стоит отметить, что в исходной версии work-stealing балансировщика начальные размеры деков были одинаковы и равны степеням двойки.

В следующих таблицах приведены некоторые результаты экспериментов с менеджером памяти. В качестве теста использовалась задача перемножения матриц. Исходными данными являются: Размер перемножаемых квадратных матриц; Размер памяти, отведенный для хранения указателей на задачи; Размеры деков; Среднее время решения задач в секундах (для каждого размера матриц проводилось 30 экспериментов).

Все исследования проводились на процессоре Intel Pentium N3710 1.60GHz, 4 CPU, 32 L1, с Linux 4.4.0. Расчеты проводились на двух ядрах процессора.

Таблица 2: Среднее время решения задачи, оптимальное деление памяти

Размер матриц	Память	Дек 1	Дек 2	Время
256	104	50	54	5,699
512	118	56	62	45,599
1024	132	63	69	366,153

Таблица 3: Среднее время решения задачи, деление памяти пополам

Размер матриц	Начальные значения			После преполнения			Время
	память	дек 1	дек 2	память	дек 1	дек 2	
256	104	52	52	106	52	54	5,683
512	118	59	59	121	59	62	45,504
1024	132	66	66	135	66	69	366,106

В таблицах 2 и 3 приведены результаты работы системы с подключенным разработанным менеджером памяти, на основе разных методов управления памятью.

Таблица 2. При методе “оптимальное деление” каждому деку выделяется минимальное количество памяти, требуемое для решения задачи без переполнений. Если разделить размер первого дека на общий объем памяти, то можно получить процент памяти выделенный первому деку. Для всех рассмотренных размеров перемножаемых матриц этот процент примерно равен 48, что близко к теоретически полученному значению – 46 (таб. 1).

Таблица 3. При методе “деление пополам” каждому деку выставлялись одинаковые количества памяти, в результате чего один из деков переполнялся. Чтобы не прекращать работу системы, деку выделялось большее количество памяти. Соответственно, общий объем требуемой памяти также увеличивался.

Так, для размера матрицы 1024, начальные размеры деков были равны 66 (132/2). В процессе работы второй дек переполнился и его размер увеличился до 69, что привело к увеличению общей памяти до 135.

В таблице 4 среднее время работы сравнивается между двумя

Таблица 4: Среднее время решения задачи при различных менеджерах памяти

Размер матриц	Память	Дек 1	Дек 2	Время
Экспериментальная реализация				
256	104	50	54	5,699
512	118	56	62	45,599
1024	132	63	69	366,153
Стандартная реализация				
256	128	64	64	5,678
512	128	64	64	45,478
1024	192	64	128	363,058

реализациями менеджеров памяти:

В экспериментальной реализации (описанной в этой статье) декам выделяется оптимальное количество памяти. В результате чего переполнения не происходят.

“Стандартная реализация” – это вариант менеджера, использующийся в балансировщике по умолчанию. В нем декам выделяются одинаковые начальные размеры равные степеням двойки. При переполнении дек увеличивается в два раза.

В целом, во всех экспериментах, оптимальное деление памяти имеет наибольшее время решения задачи. В тоже время, с помощью этого метода можно решать задачи используя меньшее количество памяти, без переполнения деков.

Например, пусть мы решаем задачу перемножения квадратных матриц размером 1024:

Допустим, у нас имеется ограниченная память в 132 байта и размер указателя равен 1 байту. При переполнении дека работа прекращается. В таком случае, чтобы решить задачу нам требуется выделить первому деку 63 байта, а второму – 69. Если мы выделим обоим декам половину памяти – 66 байтов (таб. 3), то второй дек переполнится и задача не выполнится. То же произойдет, если мы выделим обоим декам 64 байта (таб. 4).

5. Заключение

Описанный в статье менеджер памяти имеет как преимущества, так и недостатки.

С одной стороны, использование менеджера (при использовании метода оптимального деления памяти) обеспечивает выполнение задач на меньших размерах памяти. С другой стороны, среднее время решения задач увеличивается.

Такой менеджер может быть полезен, например, в приложениях связанных с реальным временем, где небольшое увеличение времени работы программы может быть некритичным, но затраты на обработку переполнения памяти могут привести к аварийной остановке программы.

Стоит заметить, что на этом этапе разработки менеджер памяти не является самостоятельной библиотекой и реализован непосредственно в коде work-stealing балансировщика. В будущем планируется разработать версию менеджера, которую можно будет подключать к различным work-stealing балансировщикам задач.

Список литературы

[1] *Blumof R.D., Joerg C.F., Kuszmaul B.C., Leiserson C.E., Randall K.H., Zhou Y.* Cilk: an efficient multithreaded runtime system // Journal of Parallel and Distributed Computing. 1996. Vol. **37**. No. 1. P 55–69.

[2] *Marochko A.* TBB scheduler clandestine evolution. — Intel. 2010 [Online]. Available: https://software.intel.com/en-us/blogs/2010/12/09/tbb-scheduler-clandestine-evolution.

[3] *Leijen D., Schulte W., Burckhardt S.* The design of a task parallel library // ACM SIGPLAN Notices. 2009. Vol. **44**. No. 10. P 227–242.

[4] *Tardieu O., Wang H., Lin H.* A work-stealing scheduler for X10's task parallelism with suspension // ACM SIGPLAN Notices. 2012. Vol. **47**. No. 8. P 267–276.

[5] *Knuth D.* The art of computer programming, vol. 1. — Addison-Wesley. 2001.

[6] *Blumofe R.D., Leiserson C.E.* Scheduling multithreaded computations by work stealing // Journal of the ACM. 1999. No. 46. P 720–748.

[7] *Hendler D., Shavit N.* Non-blocking steal-half work queues // ACM Symposium on Principles of Distributed Computing. 2002. P 280–289.

[8] *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.

[9] *Барковский Е.А., Кучумов Р.И., Соколов А.В.* Оптимальное управление двумя work-stealing деками в общей памяти при различных стратегиях перехвата работы // Программные системы: Теория и приложения. 2017. № 1. С. 83–103.

[10] *Кучумов Р.И.* Реализация и анализ work-stealing планировщика задач // Стохастическая оптимизация в информатике. 2016. Т. 12. № 1. С. 20–39.

[11] *Chase D., Lev Y.* Dynamic circular work-stealing deque // ACM Symposium on Parallelism in Algorithms and Architectures. 2005. P 21–28.