МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМ. М.В. ЛОМОНОСОВА

Механико-математический факультет

Курсовая работа

Студент 3 курса: Нагорных Я.В.
Научный руководитель: Богачев К.Ю.

Москва
2017

Содержание

Введение

Печать большив массивов чисел всегда занимает много времени. Кроме того, у печати данных мало ресурсов для ускорения.

Печать чисел с плавающей запятой также является проблемой, так как само значение числа и его экспоненту нельзя обрабатывать независимо.

Стандартный подход недостаточно точен и в некоторых случаях дает неверные результаты. Кроме того использование функций стандартных библиотек (`printf`, `sprintf`) достаточно затратно по времени.

Цели работы:

1. Ускорить печать больших массивов;

2. Использовать быстрые алгоритмы печати целых чисел и чисел с плавающей точкой.

1 Проблемы и способы их решения

Как уже было сказано, у печати массивов мало ресурсов для ускорения. Также проблемой является и то, что печать данных файл должна быть строго последовательной, поэтому нельзя "простым" образом использовать распараллеливание.

Однако, известно что большую часть времени занимает преобразование типа `int` или `double` в буффер типа `const char *` непосредственно для печати. Именно это можно и распараллелить, используя многопоточное программирование. Непосредственно печать в сам файл упирается в возможности диска. Ее ускорить нельзя.

Кроме того, можно заменить стандартный алгоритм преобразования числа в строку, на более быстрые. Мы будем использовать алгоритм Grisu2 для печати вещественных чисел и SSE2 для печати целых чисел, о которых будет рассказано позже.

2 Описание алгоритма

2.1 Используемые структуры и классы

Структура `reduce_chunk`. В ней находится строковый буффер (готовый для печати) и его порядковый номер (`chunk_id`). Кроме того, хранится флаг, является ли этот `reduce_chunk` последним.

Структура `map_chunk`. Этот тип состоит из лямбда-функии, которая должна обработать определенный фрагмент массива чисел, и элемента типа `reduce_chunk`, возвращаемый функцией.

Класс mutex_wait_queue**.** Это реализация *блокирующей очереди*, или *мьютексной очереди*. Под ней понимается очередь со следующим свойством: когда поток пытается прочитать что-то из пустой очереди, то он блокируется, до тех пор, пока какой-нибудь другой поток не положит в нее элемент. У этой очереди есть следующие методы:

- dequeue – достает верхний элемент из очереди, если очередь непустая. Иначе, поток, вызвавший этот метод блокируется. Также можно передать время блокировки, по истечении которого, поток разблокируется и вернется ни с чем;

- dequeue_all – аналогично dequeue, но достает все элементы, находящиеся в очереди, и складывает в указатель вектор из них;

- enqueue – складывает элемент в конец очереди.

Класс task**.**

Класс worker_thread**.** Это главный управляющий поток. Он хранит две блокирующие очереди m_map_queue и m_reduce_queue, состоящие из map_chunk и reduce_chunk соответственно. Зачем нужны такие очереди будет сказано позже.

2.2 Распределение задач

Управляющий (главный) поток worker_thread вызывает функцию create_mappers (), которая создает несколько потоков-обработчиков чисел, и функцию create_reducer (), которая создает печатающий поток. Сам управляюиий поток будет складывать элементы типа map_chunk в очередь m_map_queue. Потоки-обработчики будут доставать из этой очереди map_chunk-и на обработку. Они должны конвертировать числа в буфферы, готовые для печати. Эти готовые буфферы reduce_chunk они складывают в другую очередь m_reduce_queue. Печатающий поток должен забирать готовые буфферы из этой очереди и печатать их в правильном порядке в файл.

Схематично работа потоков показана на Рисунке 1.

2.3 Преобразование чисел в строковый тип

2.3.1 Используемые обозначения.

Как известно, числа с 5 на конце, могут округляться по-разному. Используем следующие обозначения:

- $[x]^{\uparrow}$ – округление вверх

- $[x]^{\square}$ – округление до ближайшего четного (то есть число $0,5$ округляется до 0, а число $1,5$ до 2)

- $[x]^{\star}$ – когда неважно, как именно округлять.

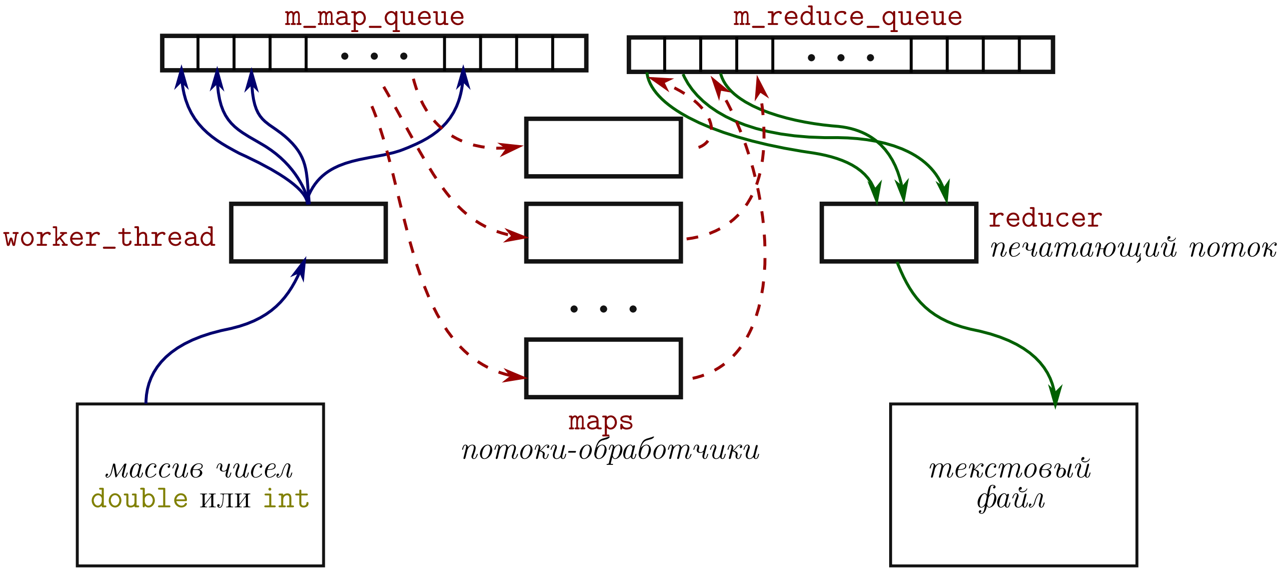

Рисунок 1: Работа потоков.

- $\tilde{x} = [x]_p^s$ – округленное число до p знаков после запятой, а s – один из вышеизложенных способов округления.

Ошибка: $x = f \times b^e$ должно быть округлено до ближайшего $|\tilde{x} - x| \leqslant 0,5 \times b^e$, другими словами, до половины `ulp` (*unit in the last place*).

$v = f_v \times b^{e_v}$ – положительное число. v^- – предыдущее число для v, хранящееся в памяти. Аналогично v^+ – следующее число за v. Если v наименьшее, то $v^- = 0$. Если v наибольшее, то $v^+ = v + (v - v^-)$.

Определим `diy_fp` для x как беззнаковое целое число f_x, состоящее из q битов, и знакового целого числа e_x неограниченного диапазона. Значение x можно вычислить как $x = f_x \times 2^{e_x}$. Очевидно произведение двух таких типов отличатся от обычного. Вычислять и обозначать его будем следующим образом:

$$x \otimes y := \left[\frac{f_x \times f_y}{2^q}\right]^{\uparrow} \times 2^{e_x + e_y + q}$$

В статье [1] описан алгоритм Grisu и его улучшения, также доказана их точность. Опишем кратко эти алгоритмы.

2.3.2 Grisu

Идея алгоритма. Предполагается, не умаляя общности, что у числа с плавающей точкой v отрицательный показатель. Тогда это число можно выразить как $v = \dfrac{f_v}{2^{-e_v}}$, где f_v – мантисса, а e_v – экспонента. Десятичные цифры v могут быть вычислены путем нахождения десятичного показателя t, для которого $1 \leqslant \dfrac{f_v \times 10^t}{2^{-e_v}} < 10$.

Первая цифра является целой частью этой дроби. Последующие цифры вычисляются путем повторного использования оставшейся дроби: нужно умножить числитель на 10 и взять целую часть от вновь полученной дроби.

Идея Grisu состоит в том, чтобы кешировать приблизительные значения $\frac{10^t}{2^{e_t}}$. Дорогих операций с большими числами не будет: они заменяются операциями с целыми числами фиксированного размера.

Кэш для всевозможных значений t и e_t может быть дорогостоящим. Из-за этого требования к кеш-памяти в Grisu упрощены. Кэш хранит только нормированные приближения с плавающей точкой всех соответствующих степеней десяти: $\tilde{c_k} := \left[10^k\right]_q^\star$, где q – точность кэшированных чисел. Кэшированные числа сокращают большую часть экспоненты v, так что остается только небольшой показатель.

Процесс генерации цифр использует степени десяти с экспонентой $e_{\tilde{c_t}}$, близкой к e_v. Разница между двумя показателями будет небольшой.

Фактически, Grisu выбирает степени десяти так, что разница лежит в определенном диапазоне.

Реализация. Алгоритм Grisu:

- *Вход:* положительное число с плавающей точкой v точности p.
- *Условие:* точность `diy_fp` удовлетворяет $q \geqslant p + 2$, а кеш степеней десяти состоит из предварительно вычисленных нормированных округленных `diy_fp` значений $\tilde{c_k} := \left[10^k\right]_q^\star$
- *Вывод:* строковое представление в основании 10 для V такое, что $[V]_p^\square = v$. То есть V должен быть округлен до v при чтении вновь.

  Шаги алгоритма:

1. *Преобразование:* определим нормированный `diy_fp` w такой, что $w = v$.
2. *Кэширование степеней:* находим с заданной точностью $\tilde{c}_{-k} = f_c \times 2^{e_c}$ такое, что $\alpha \leqslant e_c + e_w + q \leqslant \gamma$.
3. *Произведение:* пусть $\tilde{D} = f_D \times 2^{e_D} := w \otimes \tilde{c}_{-k}$.
4. *Выход:* определим искомое $V := \tilde{D} \times 10^k$. Вычислим десятичное представление $\tilde{D}$, за которым следует строка `e` и десятичное представление k.

Поскольку значение `diy_fp` больше, чем значение входного числа, преобразование шага 1 дает точный результат. По определению `diy_fp`-ы имеют бесконечный диапазон экспоненциальности и показатель степени w, следовательно, достаточно велик для нормирования. Заметим, что показатель e_w удовлетворяет $e_w \leqslant e_v - (q - p)$.

Легко показать, что $\forall i, 0 < \tilde{e}_{c_i} - \tilde{e}_{c_{i-1}} \leqslant 4$, и поскольку кеш неограничен, требуемый $\tilde{c}_{-k}$ должен находиться в кеше. Это является причиной первоначального требования $\gamma \geqslant \alpha + 3$.

Разумеется, бесконечный кеш не нужен. k зависит только от типа номера с плавающей точкой ввода (его диапазон экспоненты), точности `diy_fp` и пары α и γ.

Результатом Grisu является строка, содержащая десятичное представление $\tilde{D}$, за которым следуют символ `e` и k знаков. Таким образом, он представляет собой число $V := \tilde{D} \times 10^k$. Утверждается, что V дает v при округлении до числа с плавающей точкой с точностью p.

2.3.3 Grisu2

Но у Grisu есть недостаток: так число 1 будет напечатано в виде `10000000000000000000e-19`. Поэтому будем использовать Grisu2. Этот алгоритм является усовершенствованием предыдущего и не записывает лишние нули в конец числа. Так если целочисленный тип `diy_fp` содержит более двух дополнительных битов, то эти биты могут использоваться для сокращения выходной строки. В отличие от Grisu, Grisu2 не генерирует полное десятичное представление, а просто вовзвращает цифры (`123`) и соответствующий показатель (`-2`). Затем процедура форматирования объединяет эти данные для получения представления в требуемом формате.

Идея алгоритма. Как описано выше, Grisu2 использует дополнительные биты для создания более короткой выходной строчки. Также Grisu2 не будет работать с точными числами, а вместо этого будет вычислять аппроксимации m^- и m^+. Чтобы избежать ошибочных результатов, которые не удовлетворяют требованиям, добавляется "безопасное пространство" (*safety margin*) вокруг приблизительных границ. Как следствие, Grisu2 иногда может вернуть не самое оптимальное представление, которое может лежать вне нужных изначально границ. Также это safety-margin требует от нас изменить предварительное условие. Действительно, используя только 2 дополнительных бита, вычисление настолько неточно, что Grisu2 может закончиться ничем. Чтобы таких проблем избежать добавляется третий дополнительный бит: $q \geqslant p + 3$.

Реализация. Алгоритм Grisu2:

- *Вход:* положительное число с плавающей точкой v точности p.
- *Условие:* точность `diy_fp` удовлетворяет $q \geqslant p + 3$, а кеш степеней десяти состоит из предварительно вычисленных нормированных округленных `diy_fp` значений $\tilde{c_k} := \left[10^k\right]^{\star}_{q}$
- *Вывод:* десятичные знаки d_i, где $0 \leqslant i \leqslant n$ и целочисленное K, такое что $V := d_0 \ldots d_n \times 10^K$ удовлетворяет $[V]^{\square}_{p} = v$.

Шаги алгоритма:

1. *Границы:* вычисляем границы для v: m^- и m^+.

2. *Преобразование:* определим `diy_fp` w^+ так ,что $w^+ = m^+$. Определим также `diy_fp` w^- так, что $w^- = m^-$ и $e_w^- = e_w^+$.

3. *Кэширование степеней:* находим с заданной точностью $\tilde{c}_{-k} = f_c \times 2^{e_c}$ такое, что $\alpha \leqslant e_c + e_w + q \leqslant \gamma$.

4. *Произведение:* вычисляем $\tilde{M}^- := w^- \otimes \tilde{c}_{-k}$, $\tilde{M}^+ := w^+ \otimes \tilde{c}_{-k}$, и пусть $M_\uparrow^- := \tilde{M}^- + 1\mathtt{ulp}$, $M_\downarrow^+ := \tilde{M}^+ - 1\mathtt{ulp}$, $\delta := M_\downarrow^+ - M_\uparrow^-$.

5. *Количество разрядов:* находим наибольшее κ такое, что $M_\downarrow^+ \mod 10^\kappa \leqslant \delta$ и определим $P := \left\lfloor \dfrac{M_\downarrow^+}{10^\kappa} \right\rfloor$.

6. *Выход:* определим $V := P \times 10^{k+\kappa}$. Десятичные знаки d_i и n получены путем вычисления десятичного представления P. Положим $K := k + \kappa$ и возвращаем его с n знаками d_i.

Grisu2 не дает никаких гарантий относительно краткости результата. Его результатом является кратчайшее возможное число в интервале от $M_\uparrow^- \times 10^k$ до $M_\downarrow^+ \times 10^k$ включительно, где $M_\uparrow^- \times 10^k$ и $M_\downarrow^+ \times 10^k$ зависят от точности q для `diy_fp`. Чем больше q, тем ближе $M_\uparrow^- \times 10^k$ и $M_\downarrow^+ \times 10^k$ к фактическим границам m^- и m^+.

2.3.4 Еще что-то

Будем использовать следующее улучшение. Если в массиве есть n подряд идущих одинаковых чисел x, то будем записывать их как `n*x`. Таким образом, если в нашем массиве много повторяющихся чисел, то выходная строчка будет гораздо короче, а значит, можно сэкономить память и время работы программы.

Для преобразования целых чисел используется алгоритм SSE2, о котором подробнее написано в статье [2]. Суть алгоритма заключается в быстром логарифмировании числа по основанию 10.

3 Результаты работы и ускорение

Время работы в секундах для массива с разными случайными числами представлено в следующей таблице:

Размер массива	Число потоков				Стандартная печать
	16	12	4	1	
10^7	0.609	0.550	0.880	3.196	4.256
	0.567	0.500	0.841	3.239	4.176
	0.506	0.473	0.802	3.052	4.188
$5 \cdot 10^7$	2.420	2.528	4.044	15.377	22.476
	2.522	2.446	4.273	16.309	21.116
	2.587	2.339	4.179	15.327	20.893
10^8	5.025	4.665	8.276	32.461	41.712
	4.787	4.630	7.970	30.571	41.785
	4.844	4.544	8.078	30.757	41.961
$5 \cdot 10^8$	21.199	20.074	37.515	148.548	201.941
	21.312	20.297	37.627	148.829	202.333
	21.231	20.171	37.686	149.217	201.692

Размер массива	Число потоков				Размер файла
	16	12	4	1	
10^7	7.50	8.20	5.00	1.33	245 Mb
$5 \cdot 10^7$	8.57	8.80	5.16	1.37	1.2 Gb
10^8	10.61	11.23	6.39	1.66	2.4 Gb
$5 \cdot 10^8$	9.51	10.01	5.37	1.36	12 Gb

Время работы на массиве с множеством повторяющихся чисел:

Размер массива	Число потоков				Стандартная печать
	16	12	4	1	
10^7	0.318	0.249	0.188	0.652	3.645
	0.334	0.256	0.190	0.629	3.622
	0.307	0.251	0.192	0.661	3.620
$5 \cdot 10^7$	1.657	1.274	0.884	3.183	18.412
	1.505	1.247	0.891	3.167	18.306
	1.522	1.262	0.894	3.175	18.261
10^8	3.105	2.441	1.726	6.306	36.194
	2.983	2.453	1.820	6.329	36.388
	3.246	2.505	1.759	6.339	36.419
$5 \cdot 10^8$	16.194	12.575	8.681	31.505	181.221
	16.414	12.564	8.787	31.291	181.406
	15.815	12.555	8.764	31.690	182.524

Размер массива	Число потоков				Размер файла
	16	12	4	1	
10^7	11.35	14.40	19.10	5.61	24 Mb / 187 Mb
$5 \cdot 10^7$	11.74	14.53	20.60	5.77	123 Mb / 936 Mb
10^8	11.68	14.73	20.55	5.74	245 Mb / 1.8 Gb
$5 \cdot 10^8$	11.26	14.46	20.78	5.77	1.2 Gb / 9.4 Gb

4 Заключение

Список литературы

[1] FLORIAN LOITSCH. Printing Floating-Point Numbers Quickly and Accurately with Integers, 2004.

[2] WOJCIECH MULA. SSE: conversion integers to decimal representation, 2011.