Параллельные алгоритмы сложения чисел с плавающей запятой

Майкал Т. Гудрич
(Michael T. Goodrich)
Отдел информационных технологий, Калифорнийский Университет, Ирвин
Ирвин, Калифорния, США, 92697
goodrich@acm.org

Ахмед Елдави
(Ahmed Eldawy)
Отдел информационных технологий, Калифорнийский Университет, Риверсайд
Риверсайд, Калифорния, США, 92521
eldawy@cs.ucr.edu

Содержание

Содержание 1
Резюме 2
1. Введение 2
1.1 Ранее полученные результаты 4
1.2 Полученные результаты 6
2. Используемое представление чисел 6
3. Заключение 11
4. Словарь 11

Резюме

Проблема точности вычисления суммы n чисел с плавающей запятой является фундаментальной и встречается в различных приложения работающих в области крупномасштабного моделирования и вычислительной геометрии. К сожалению, в следствии накапливающейся ошибки округления при использовании стандартных операциях над числами с плавающей запятой, данная проблема становится еще более сложной задачей. Кроме того, все существующие реализации полагаются на последовательные алгоритмы, которе не могут оперировать большими объемами данных.

В этой статье мы опишем несколько эффективных параллельных алгоритмов для сложения n чисел с плавающей запятой которые получают на выходе точно округленное представление суммы чисел с плавающей запятой. Мы рассмотрим алгоритмы для параллельных машин с произвольным доступом (PRAM), алгоритмы использующие внешнюю память и алгоритмы работающие с MapReduce моделью. А так же приведем экспериментальный анализ простого и, на практике эффективного, алгоритма, использующего модель MapReduce.

1. Введение

Числа с плавающей запятой это тип данных общего назначения. Переменные этого типа могут иметь сильно отличающиеся друг от друга значения. Представление[1] числа с плавающей запятой (по основанию два) *x*, включает себя знак мантиссы *b* и пару целочисленную последовательность *(b, M, E)*, такую что:

$$x = (-1)^b \times (1 + 2^{-t}M) \times 2^{E-2^{l-1}-1}$$

где t представляет число битов, выделенное на хранение M и l представляет число бит, выделенное на хранение E. Значение M называют **мантиссой** или знаковым и значение E называют экспонентой. Для получения более подробной инфорации о числах с плавающей запятой, пожалуйста, обратитесь к другим источникам. например, к превосходному обзору, выполненному Голдбергом [15].

В представлении чисел с **фиксированной точностью**, значения переменных t и l устанавливаются исходя из особенностей машинной архитектуры иил согласно определенному стандарту. Например, в стандарте IEEE 754, числа с плавающей запятой одинарной точности используют $t = 23$ и $l = 8$, числа двойной точности используют $t = 52$ и $l = 11$, и числа четверной точности используют $t = 112$ и $l = 15$.

В представлении чисел с **произвольной точностью**, количество битов, отведенных для мантиссы t может быть установлено до сколько угодно больших размеров, вплоть до объема оперативной памят, либо до произвольного значения, указанного пользователем. В данном представлении чисел, количество бит, отведенное на мантиссу l, как правило установлено в величину машинного слова, по той причине, что данного объема обычно хватает для зранения любого адреса в памяти. Примерами систем работающих с числами с произвольной точностью могут послужить: Apfloat for Java [15], the GNU Multiple Precision (GMP) Arithmetic Library [14], the bigfloat type in LEDA [4], и the GNU Multiple-Precision Floating-Point (MPFR) Library [12, 19].

В данной работе, мы не делаем предположения о том, что числа с определенной точностью, как и числа с произвольной точностью нормализованны, то есть их старший значащий бит мантиссы установлен в значение 1.

Заданно множество чисел с плавающей запятой $X = \{x_1, x_2, \ldots, x_n\}$, для которого нас интересует разработка и анализ параллельного алгоритма вычисления такого числа с плавающей запятой S_n^*, которое лучше всего выражается суммой:

$$S_n = \sum_{i=1}^{n} x_i$$

Кроме того, нас интересуют методы не ограниченные определенным представлением чисел с плавающей запятой, например таких как IEEE 754 чисел двойной точности. В частности, нас интересует вопрос вычисления такого числа с плпваюзей запятой S_n^* которое является **корректно округленным** представлением S_n, где мы считаем значение S_n^* корректно округленным до тех пор пока S_n^*, либо наибольшее число с плавающей запятой меньшее или равное S_n, либо наименьшее число с плавающей запятой большее или равное S_n. Для получения информации о других методах округления числе с плавающей запятой, пожалуйста, обратитесь к стандарту IEEE 754.

Несмотря на то, что проблема точного сложения n чисел с плавающей запятой может на первый взгляд показаться простой, она не по обыкновению сложна в следствии ошибок округления свойственным арифметике с плавающей запятой. Стандартная операция сложения чисел с плавающей запятой, которое мы обозначис через "$\oplus$" имеет неточность, такую что для двух чисел с плавающей запятой x и y, мы имеем:

$$x \oplus y = (x + y)(1 + x_{x,y}); \left|\epsilon_{x,y}\right| \leqslant \left|\epsilon\right|$$

где $\varepsilon_{x,y}$ специфическая для слагаемого ошибка округления и ε относящаяся к машины худшая величина ошибки [23, 25].

Как известно, стандартный способ сложения n чисел из данного множества X, заключается в создании из чисел множества X двоичного дерево суммирования T, в котором каждый внутренний узел v связан с суммой обоих значений дочерних узлов v, - в данном случае используя операцию $\oplus$. К сожалению, если набор слагаемых X содержит положительные и отрицательные числа, то задача построения дерева суммирования T становится NP-полной, и она использует операцию $\oplus$ и минимизирует получение худшего случая нахождения суммы [23]. Даже в том случае, когда все числа множества X положительны, отптимаьлным деревом суммирования, основанным на операции $\oplus$, служит терево Хаффмана [23], которое не достаточно эффективно в параллельном использовании. Таким образом, разработка эффективного параллельного алгоритма точного словежния n чисел с плавающей запятой является сложной проблемой.

В следствии данной сложности и неопределенности, возникающей благодаря ошибкам округления, многие современные алгоритмы сложения чисел с плавающей запятой основаны на точном вычислении суммы двух числе с плавающей запятой x и y, а так же точного значения ошибки округления, используя следующую функцию:

$$AddTwo(x, y) \rightarrow (x, e_s)$$

где, $s = x \oplus y$ и $x + y = s + e_s$, причем s и e_s числа с плавающей запятой тойже точности, что и x и y. Пример реазлицазии функции AddTwo включает использование алгоритмов Деккера [8] и Кнута [25], обе из которых используют константное число операций с плавающей запятой. Как и в случае последних подходов, в данной статье мы будет рассматривать сложение чисел множества X, как точное, так и приближенное, с округление точной суммы до соответствующего представления с плавающей запятой.

Как упоминалось ранее, в связи с тем, что мы хотим построить, независимый от машиной точности, алгоритм, который сможет функционировать с числами, как в представлении с фиксированной, так и произвольной точностью, мы, в этом документе, не берем за основу какое либо конкретное представление чисел с плавающей запятой, такой как, например, IEE 754 двойной точности. Как это часто бывает для алгоритмов независящих от машинной точности, мы предполагаем представление чисел с плавающей запятой достаточно точным чтобы число n могло быть представлено в виде суммы постоянного числа чисел с плавающей запятой. Аналогичное предположение сделано для алгоритмов[2] RAM для целых чисел и PRAM, что не создает каких-либо ограничений в практическом использовании, так как адрес памяти всегда может быть сохранен в $O(1)$ машинных слов. Например, даже если вся компьютерная память на Земле будет выделена для IEEE 754 чисел одинарной точности, которые будут сложенны между собой, то мы могли бы представить размер входных раддын через n, как (точную) сумму не более четырех числе с плавающей запятой одинарной точности.

В качестве способа для описания различных возникающих сложностей, приводящих к существенному объему ошибок, введем для множества X **условное число** $C(X)$ [34, 39, 40], такое что:

$$C(X) = \frac{\sum_{i=1}^{n} x_i}{|\sum_{i=1}^{n} x_i|}$$

которое, разумеется, определенно только для ненулевых сум[4]. На практике, проблемные случаи с больгим значением условного числа сильно более сложны, чем те, когда условное число близко к единице.

Наш подход к разработке эффективных параллельных алгоритмов сложения n чисел с плавающей запятой, даже для сложных проблем с большим значением условного числа, использует подход, которые несколько напоминает классический алгоритм Быстрого разложения Фурье [3, 18]. Говоря более точно, для вычисления суммы n чисел с плавающей запятой, мы преобразуем числа в альтернативное представление, вычисляем сумму чисел в этом представлении и затем преобразуем результат обратно в (корректно округленное) число с плавающей запятой. В нашем случае, важной особенностью альтернативного представления является то, что оно позволяет вычислить промежуточную сумму без распространения битов переноса, что дает возможность параллелизации.

1.1 Ранее полученные результаты

Проблема сложения чисел с плавающей запятой бросает интересные вызовы для параллельного вычисления, поскольку большинство существующих алгоритмов точного сложения по своей сути последовательны. Например, нам не известны какие

либо параллельные алгоритмы, решающие данную задачу, которые выполняются в худшем случае за полилогарифмическое время.

Нил [30] описывает последовательные алгоритмы, использующие представление чисел, которое он сам называет **супер-аккумуляторным**, и вычисляет с его помощью точную сумму n чисел с плавающей точкой, которое он затем преобразует в корректно округленное число с плавающей запятой. К сожалению, в то время как супер-аккумуляторное представление Нила уменьшает распространение битов переноса, оно не устраняет его, как это требуется для высоко-параллельных алгоритмов. Аналогичная идея была использована в библиотеке с открытым исходным кодом ExBLAS [6], созданной для вычислений с числами с плавающей запятой. Шевчук [33] описывает альтернативное представление для точного представления промежуточных результатов арифметики с плавающей точкой, но его метод также не уменьшает распространение битов переноса в сложении; следовательно этот метод так же не может быть использован в эффективных параллельных алгоритмах. В дополнении к этим методам, существует целый ряд, по своей сути, последовательных методов точного сложения n чисел с плавающей запятой использующих различные структуры данных для представления промежуточных результатов, включая ExBLAS [6], алгоритмы Чжу и Хаеса [39, 40], Деммеля и Хида [9, 10], Рампа [34], Преста [32] и Малколма [29]. Хотя метод Деммеля и Хида [10] может быть лишен переносов для ограниченного числа слагаемых, ни один их этих последовательных методов не использует лишенное переносов промежуточное представление, подходящее для эффективного распараллеливания. Тем не менее, Рамп [34] предлагает интересный анализ, в котором время выполнения последовательного метода зависит от n, логарифма условного числа a и ряда факторов. Деммел и Хид [10] показали, что сложение чисел в порядке убывания показателя степени дает точный результат, который, не может быть назван корректно округленным.

Имеется сильно ограниченное число опубликованных работ по разработке параллельных алгоритмов сложения чисел с плавающей запятой. Лойпрехт и Оберайгнер [28], к примеру, описывают параллельные алгоритмы сложения чисел с плавающей запятой, но из метод лишь расспараллеливает конвеерные данные, и не используют эффективные параллельные методы с полилогарифмическим временем выполнения, алгоритмы работающие с внешней памятью и модель MapReduce. Действительно, их методы могут проводить до $O(n)$ проходов над данными. Кадрик [22] предложил параллельный конвейерный метод использующий аналогичный Лойпрехту и Оберайгнеру подход, при одновременном повышении его сходимости на практике, но, тем не менее, этот метод зависит на, по свокй сути, последовательном конвейере и итеративных операциях уточнения. Сравнительно недавно, Деммел и Нгуен [11] представили метод параллельного сложения чисел с плавающей запятой используя супер-аккумулятор, но, как и в предыдущем последовательном супер-аккумуляторном методе, описанном выше, их метод не использует лишенное переноса промежуточное представление; следовательно, он по своей сути последовательный шаг с распространением переноса в части его “внутреннего цикла” вычислений.

Насколько нам известно, ни один из существующих алгоритмов сложения чисел с плавающей запятой не использует лишенное переноса промежуточное представление, хотя такие представления известны для целочисленной арифметикм (смотрите [31]). Наиболее известным из которых является избыточное бинарное представление (RBR), которое является ничем иным как позиционным бинарным представлением где каждая позиция имеет одно из значений множества {-1, 0, 1}.

1.2 Полученные результаты

В этой статье мы покажем, что можно вычислить корректно округленную сумму S_n^*, n чисел с плавающей запятой со следующими показателями производительности:

- S_n^* может быть вычислена за время $O(\log n)$ используя n процессоров в модели EREW PRAM. Как мы далее покажем, это первый PRAM алгоритм с таким временем выполнения для худшего случая.
- S_n^* может быть вычислена за время $O(\log^2 n \log\log\log C(X))$ используя $O(n \log C(X))$ работы в моделе EREW PRAM, где $C(X)$ условное число от X. Это первый параллельный алгоритм сложения, время выполнения которого зависит от условного числа.
- S_n^* может быть вычислена во внешней память за время $O(sort(n))$, где $sort(n))$ сложность операций ввода-ввыода сортировки.
- S_n^* может быть вычислена во внешней памяти за вермя $O(scan(n))$, где $scan(n))$ сложность операций ввода-выода сканирования, при условии, что размер внутренней памяти M равен $\Omega(\sigma(n))$ и $\sigma(n)$ равна объему памяти, затрачиваемому на внутреннее представление суммы n чисел с плавающей точкой. Указанные ограничения производительности операций ввода-выода, разумеется оптимальны для данного случая. Как правило, $\sigma(n)$ и $O(\log n)$ на практике равны, что обусловленно распределением значением экспоненты в реальных данных (смотрите [20, 27]), или по той причине, что числа с плавающей запятой имеют относительно малое зачение l (количество битов, используемых для значений экспоненты) по сравнению с n.
- S_n^* может быть вычислена одно-проходным алгоритмом MapReduce за время выполнения $O(\sigma(n/p)p+n/p)$ и работу $O(n)$ используя p процессоров с высокой вероятностью принятия p значению $O(n)$.

Кроме того, в следствии просторы нашей реализации алгоритма MapReduce, мы обеспечим экспериментальный анализ ряда версий данного алгоритма. Мы покажем, что наш алгоритм MapReduce может достигать ускорения в 80 раз превосходящее классические последовательные алгоритмы. Что позволяет достичь линейной масштабируемости для исходных данных и числа ядер в кластере.

2. Используемое представление чисел

Существуют многочисленные представления чисел с плавающей запятой, пригодных для хранения суммы n чисел. Прежде всего, вспомните, что число с плавающей запятой (с основание 2) x имеет в своем представлении значащий бит b и пару целых чисел (b, M, E), такую, что:

$$x = (-1)^b \times (1 + 2^{-t}M) \times 2^{E-2^{l-1}-1}$$

где, t - количество битов выделенных для хранения M и l - число бытов выделенное для хранения E. Например, для чисел с плавающей запятой двойной точности, стандартом IEEE 754 определенны следующие значения - t = 52 и l = 11.

Таким образом, в решении задачи сложения n чисел с плавающей запятой из множества X, мы имеем возможность раздельно представить каждое число с

плавающей запятой, включая каждую пару сум, как число двоичное с фиксированной точкой, состоящее из значащего бита, $t+2^{l-1}+[log\ n]$ битов для части числа, предшествующей двоичному разделителю, и $t+2^{l-1}$ битов для части числа, стоящей после двоичного разделителя. Использую данное представление, мы не встретим ошибок в сложении чисел множества X, предполагая отсутствие ненужных переполнений. Другими словами, мы может рассматривать данное фиксированное представление как представление целово двоичного числа с большим значением, имеющего смещение. Разумеется, данное представление требуем существенных затрат памяти, но, тем не менее, будем рассматривать данную его для улучшения других представлений.

Несмотря на то, что данное представления чисел с фиксированной точкой расточительно по памяти, оно не на столько плохо для представления частичных сум слагаемых одинарной точности. Например, в стандарте IEEE 754, 32-битные числа с плавающей запятой одинарной точности могут быть представлены как 256-битные числа с фиксированной запятой. Таким образом, память, необходимая для безошибочного представления чисел с фиксированной запятой одинарной точности заняла бы тот же объем памяти, что и восемь чисел одинарной точности. Тем не менее, существует один недостаток данного представления - в худшем случае возможно большое число распределений разрядного бита, которое сопутствует операциям сложения и негативно сказывает на производительность параллельного алгоритма.

В **супер-аккумуляторном** представлении, использованном в ряде ранее описанных алгоритмов суммирования (смотрите [26, 30, 32, 33, 40]), мы представляем числа с фиксированной запятой (или плавающей запятой) как вектор Y чисел с плавающей запятой $(y_k, y_{k-1},\ldots,y_0)$ так, что число y представлено через Y как:

$$y = \sum_{i=o}^{k} y_i$$

где числа в Y имеют строго возрастающую экспоненту и y_0 младшее значащее число (и суммирование основанно на истинном сложении без ошибок округления).

Как упоминалось ранее, ряд опубликованных в последнее время работ по точному сложению чисел с плавающей запятой предлагают решения на основе супер-аккумуляторной архитектуре, упуская вопрос распараллеливания. Так, например, Чжу и Хаес [40] рассматривают, по сути, последовательный алгоритм, который скоадывает числа из множества X и супер-аккумулятор, инициализированный нулевым значением, затем обрабатывает переносы для получения корректно-округленной суммы из старшей значащей записи. Нил [30[предлагает аналогичный алгоритм, но так же рассматривает альтернативный подход, основанный на вычислении параллельного дерева сложения на основе супер-аккумуляторной модели. Однако, его параллельный алгоритм, создающий супер-аккумуляторы для каждого из листов дерева, не эффективно справляется с этой задачей, к тому же, аккумуляторы перекрывают наполовину свои биты и не являются лишенными переносов.

В нашем случае, мы позволяем каждому y_i иметь положительное и отрицательное значения. Мы так же используем дополнительные условия, делающие супер-аккумуляторное сложение лишенным переносов. Нам удается достичь данного результата путем расширения обобщенного знаковой цифры (ОЗЦ), в целочисленном представлении [31], до чисел с плавающей запятой. Что говорит об избыточном

представлении, так как существует огромное количество способов представления одного и того же числа с плавающей запятой.

Для упрощения нашего обсуждения, договоримся сдвигать двоичный разделитель для y, таким образом, что сделает его целым числом. Данный сдвиг, благодаря тому, что мы можем сдвигать двоичный разделитель назад, произведен без потери общности, и означает, что мы можем вычислить точное представление суммы чисел из множества X.

Далее, следую ОЗЦ подходу [31], мы может сказать, что супер-аккумулятор (α, β)-упорядочен если

$$y_i = Y_i \times R^i$$

для данного основания R, и каждая мантисса Y_i это целое числа в диапазоне $[-\alpha, \beta]$, при $\alpha, \beta \geq 2$. В частности, в нашей реалищации алгоритма, для всех фиксированных значений t, мы выбираем R кратное двум ($2^{t-1} > 2$), так, что, каждое y_i может быть представлено значением экспоненты (числа с плавающей запятой), хранящей множитель от $t-1$ (как и ранее, в данной работе, мы используем основание двойки для представления чисел в плавающей запятой). Для чисел с плавающей запятой произвольной точности, мы, аналогичным образом выбираем $R = 2^{t_0^1} > 2$ с константным значением $t0 \geq 2$, представляющим разумный количество битов, необходимых для мантиссы (пропорционально выбранному размеру машинного слова). В любом случае, мы считаем:

$$\alpha = \beta = R - 1$$

Наш выбор значений параметров α и β сделан так, что мы можем достичь переноса компонент сложения, без распространения, что будет показано далее.

Наш параллельный алгоритм сложения двух супер-аккумуляторов y и x имеет следующую последовательность действий. Сперва, мы вычисляем покомпонентную сумму мантис $P_i = Y_i + Z_i$. Затем уменьшаем полученную сумму на временную сумму мантисс $W_i = P_i - C_{i+1}R$, где C_{i+1} знаковый бит переноса, то есть $C_{i+1} \in \{-1, 0, 1\}$, что гарантирует диапазон значений W_i в пределах $|-(\alpha - 1), \beta - 1|$. (Ниже мы покажем, что это всегда возможно.) После чего, вычисленная сумма мантис становится равной выражению $Si = Wi + Ci$, таким образом, что полученный набор s_i компонент (α, β)-упорядочен и осовождает от необходимости в распространении бита переноса. Как покажет следующая лемма, данный подход позволяет нам избежать распространения бита переноса через супер-аккумулятор после каждого сложения в вычислении суммы, точно представляя каждую частную сумму.

Лемма 1. Достаточно выбрать такие $\alpha = \beta = R - 1$ для $R > 2$, что сумма y и z будет (α, β)-упорядочена, то есть каждое значение S_i будет в диапазоне $[-\alpha, \beta]$.

Доказательство. Учитывая, что y и z (α, β)-упорядочены, значение мантиссы P_i будет в диапазоне $[-2\alpha, 2\beta]$. Мы хотим показать, что результат s = y + z будет так же (α, β)-упорядочен, то есть, что каждая мантисса S_i имеет значение в диапазоне $[-\alpha, \beta]$. Обратите внимание, если $-R + 1 < Pi < R - 1$, тогда мы можем гарантировать S_i быть в диапазоне для любого C_i из {-1, 0, 1} устанавливая $C_{i+1} = 0$. Таким образом мы будем рассматривать случаи, когда значение P_i, либо сильно отрицательно, либо сильно положительно.

Случай 1. $P_i \geq R - 1$. Обратите внимание

$$P_i \leq 2\beta = 2R - 2$$

Следовательно, мы может выбрать $C_{i+1} = 1$, так, что

$$W_i = P_i - C_{i+1}R = P_i - R \leq R - 2 = \beta - 1$$

Кроме того, $W_i \geq -1 \geq -(\alpha - 1)$, в данном случае справедливо, так как $\alpha \geq 2$.

Таким образом,

$$S_i = W_i + C_i \leq R - 2 + 1 \leq R - 1 = \beta$$

и $S_i \geq -\alpha$.

Случай 2. $P_i \leq R + 1$. Обратите внимание

$$P_i \geq 2\alpha = 2R - 2$$

Следовательно, мы может выбрать $C_{i+1} = -1$, так, что

$$W_i = P_i - C_{i+1}R = P_i + R \geq -R + 2$$

Кроме того, $W_i \leq 1 \leq -(\beta - 1)$, в данном случае справедливо, так как $\beta \geq 2$.

Таким образом,

$$S_i = W_i + C_i \leq -R + 2 - 1 \geq -(R - 1) = -\alpha$$

и $S_i \leq \beta$. □

Это подразумевает что, выбирая $\alpha = \beta = R - 1$, мы гарантируем, что сумма двух (α, β)-упорядоченых супер-аккумулятров может быть найдена с использованием параллельного алгоритма за константное время, при том, что каждый перенос производится не более чем на крайний компонент. Разумеется, есть шанс, что сумма двух супер-аккумуляторов, пронумерованных от 0 до k, иожет привести к супер-аккумулятору с индексами от 0 до $k+1$. Однако, для нашего решения это не вызывает дополнительных проблем, так как любая мантисса может хранить $\Omega(\log n)$ битов; следовательно, один дополнительный супер-аккумуляторный компонент достаточен для хранения всех смежных компонентов переноса во время сложения суммы n чисел с плавающей запятой.

Дополнительное наблюдение состоит в том, что R - в нашем случае, степень по основанию двойки, и мы может вычислить каждые W_i и C_i используя базовый операции, включающие сложение и вычитание числа со степенью двойки из данного P_i. Кроме того, так как мы зарезервировали дополнительный бит в каждом супер-аккумуляторе, вычисление P_i может быть выполнено без переполнения для стандартного представления компонент чисел с плавающей запятой.

Интуитивно, каждая пара супер-аккумуляторных компонент y_i и y_{i+1} «перекрыты» одним битов и каждая компонента имеет дополнительный знаковый бит. Как альтернатива представлению каждого y_i числом с плавающей запятой, мы можем использовать для каждого y_i целочисленное представление, с явным перекрытием одного бита между каждой последовательной парой чисел y_i и y_{i+1}. Это позволит нам сохранить некоторый объем памяти от избыточных экспонент используемых для представления y_i. Однако, ради простоты, в нашей работе мы приняли

предположение, что каждый y_i само по себе число с плавающей запятой, и наш алгоритм может быть легко изменен для случая когда Y имеет целочисленное представление. В любом случае, перекрытие между последовательными числами из Y позволяет нам применять ленивую стратегию для битов переноса в аккумуляторе, без полного распространения, что преодолевает недостаток предыдущих представлений.

Дадим важное пояснение. В данной работе, в анализе, мы не делаем предположения, что число битов l, выделенных для экспоненты числе с плавающей запятой, константно. Следовательно, наш аналих не предполагает, что, используемые нами числа с плавающей запятой одного размера с эквивалентным представлением числе с плавающей запятой, а так же, что, супер-аккумулятор для этих чисел константен.

Таким образом, для небольшого количества чисел с плавающей запятой произвольной точности, возможжно, что наши (α, β)-упорядоченые супер-аккумуляторы могут впустую расходовать ресурс памяти. С целью избежания этой проблемы, мы можем представить числа используя формат, который мы названи “разряженный супер-аккумулятором”. Данный супер-аккумулятор

$$Y = (y_k, y_{k-1},\ldots,y_0)$$

представляет разряженный супер-аккумулятор Y для вектора

$$Y' = (y_k, y_{k-1},\ldots,y_0)$$

состоящий из всех активных элементов Y. для $i_0 < i_1 < y_j$.

Этим мы хотим показать, что супер-аккумуляторный элемент i активен если y_i, на данный момент, не равен нулю или, что лучше, не был нулевым ранее (при просмотре супер-аккумулятора в качестве упорядоченной структуры данных).

Одна из возможных реализаций алгоритма сложения двух разряженных супер-аккумуляторов $Y = (y_{kj1}, y_{k-1},\ldots,y_{i0})$ и $Z = (z_{ij2}, z_{i-1},\ldots,z_{i0})$ выполнена следующим образом. На начальном шаге происходит объединение активных индексов Y' и Z', после чего складываются соответствующие элементы (при необходимости, с переносом в смежных компонентах). Обратите внимание, мы не прибегаем к распространению переносов при использовании (α, β)-упорядоченых супер-аккумуляторов. Таким образом, индекс в сумме активен, если он был активным в и Y' и Z', или стал ненулевым в конечной сумме. Данное определение несколько связано с адаптивным представлением чисел с плавающей запятой, предложенного Шевчуком [33], которвый представил разреженность и адаптивность, но только для векторов, не имеюзим перекрываемых чисел с плавающей запятой с произвольной экспонентой, нежеди с экспонентами являющимися степенью по основаних R, как в представленном нами представлении разряженного (α, β)-упорядоченого супер-аккумулятора.

Кроме того, для данного рязряженного супер-аккумулятора Y' и параметра Υ, мы определяем Υ-усеченный разряженный супер-аккумулятор, так что для Y' он будет вектором Y'', состоящий из первого (старше-значащего) Υ елемента Y'.

3. Заключение

В этой работе мы дали описание многих реализаций эффективных параллельных алгоритмов сложения корректно округленных представлений суммы чисел с плавающей запятой. Наши алгоритмы разработаны для различных параллельных моделей, таких как PRAM, внешняя память и модель MapReduce. Основная парадигма разработки, использовавшаяся в наших моделях основана на преобразовании входных значений в промежуточное представление, называемое рязряженным супер-аккумулятором, и затем преобразование полученной точной суммы в корректно округленное представление чисел с плавающей запятой. Наша экспериментальная оценка показывает, что наш MapReduce алгоритм позволяет достигнуть ускорения в 80 раз, по сравнению в современными последовательными алгоритмами. Алгоритм MapReduce представляет средства линейно масштабируемости для входных данных и узлов кластера.

4. Словарь

Термин	Перевод значения
Arbitrary-precision	Произвольная точность
Binary point	Двоичный разделитель
Carry-bit	Бит переноса
Conditional number	Условное число
Double-precision	Двоайная точность
Exponent	Экспонента
Faithfully-rounded	Корректно округленный
Fixed-point number	Число с фиксированной запятой
Fixed-precision	Фиксированная точность
Floating-point number	Число с плавающей запятой
Mantissa	Мантисса
PRAM	Параллельные машины с произвольным доступом к памяти
Quad-precision	Четверная точность
Single-precision	Одинарная точность
Sparce superaccumulator	Разряженный супер-аккумулятор
Superaccumulator	Супер-аккумулятор