Решение задачи анализа рыночной корзины на процессорах Cell*

К.С. Пан, М.Л. Цымблер

В работе рассматривается задача глубинного анализа данных — задача нахождения часто втречающихся наборов товаров. Предложен параллельные алгоритм, адаптированный для вычислительных систем на базе процессоров с архитектурой Cell Broadband Engine. Представлены результаты вычислительных экспериментов, показывающие эффективность предложенного алгоритма.

1. Введение

Задача *анализа рыночной корзины (market-basket problem)* заключается в нахождении всех наборов (множеств) товаров, которые часто приобретаются совместно [1]. Для формального описания задачи и алгоритма ее решения в работе используются следующие термины и обозначения.

Корзина (basket) — набор товаров, приобретенных совместно (в рамках одной покупки). Обозначим за B *(baskets)* множество анализируемых корзин, а за I *(items)* — множество всех товаров, т.е. $I = \bigcup_{b \in B} b$.

Опорное число (support) заданного набора товаров X — количество корзин во множестве B, каждая из которых содержит данный набор товаров X, т.е. $support(X, B) = card\{b \in B : X \subset b\}$.

Обозначим за s_{min} минимальное значение опорного числа, при котором набор товаров считается часто встречающимся. Обозначим за L *(large itemsets)* множество часто встречающихся наборов товаров, т.е. $L = \{l \subset I : support(l, B) \geqslant s_{min}\}$. Множество L содержит все наборы товаров, опорное число которых не меньше s_{min}.

В соответствии со введенными обозначениями задача анализа рыночной корзины формулируется следующим образом: для заданных множества B и числа s_{min} найти множество L. В табл. 1 приведены примеры множества L для различных значений B и s_{min}.

Таблица 1. Примеры входных и выходных данных задачи анализа рыночной корзины

B	s_{min}	L
$\{i_1\ i_2\ i_3\ i_4\}$, $\{i_1\ i_2\ i_4\}$, $\{i_1\ i_3\}$	2	$\{i_1\}$, $\{i_2\}$, $\{i_3\}$, $\{i_4\}$, $\{i_1\ i_2\}$, $\{i_1\ i_3\}$, $\{i_1\ i_4\}$, $\{i_2\ i_4\}$
	3	$\{i_1\}$
$\{i_1\ i_2\ i_3\ i_4\}$, $\{i_1\ i_2\ i_3\ i_4\}$, $\{i_1\ i_3\}$	3	$\{i_1\}$, $\{i_3\}$, $\{i_4\}$, $\{i_1\ i_3\}$, $\{i_1\ i_4\}$, $\{i_3\ i_4\}$, $\{i_1\ i_3\ i_4\}$

Далее мы рассмотрим последовательный алгоритм решения задачи анализа рыночной корзины, а затем на его основе построим параллельный алгоритм, адаптированный для вычислительных систем на базе процессоров с архитектурой Cell Broadband Engine.

*Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (проект 09-07-00241-а).

2. Последовательный алгоритм Apriori

Алгоритм анализа рыночной корзины на процессорах Cell построен на основе алгоритма Apriori, предложенного в [1]. Идея алгоритма Apriori состоит в использовании свойства *антимонотонности* опорного числа, которое заключается в следующем: опорное число множества товаров не превосходит опорного числа любого его подмножества, т.е.

$$\forall \gamma \subset c \;\; support(\gamma, B) \geqslant support(c, B).$$

В описании алгоритма Apriori используются следующие дополнительные термины и обозначения.

Кандидат c — набор товаров, для которого в ходе выполнения алгоритма выдвигается и проверяется гипотеза $c \in L$. На k-м шаге выполнения алгоритма вычисляются два множества: C_k и L_k.

C_k *(candidate k-itemsets)* — множество кандидатов длины k, где *длина кандидата* — это количество элементов в нем.

Каждый кандидат из C_k, опорное число которого не меньше s_{min}, попадает во множество L_k — множество часто встречающихся наборов товаров длины k, т.е.

$$L_k = \{c \in C_k : support(c, B) \geqslant s_{min}\}.$$

В ходе выполнения алгоритма вычисляются все множества L_k, объединение которых затем дает множество L, т.е. $L = \bigcup\limits_k L_k$.

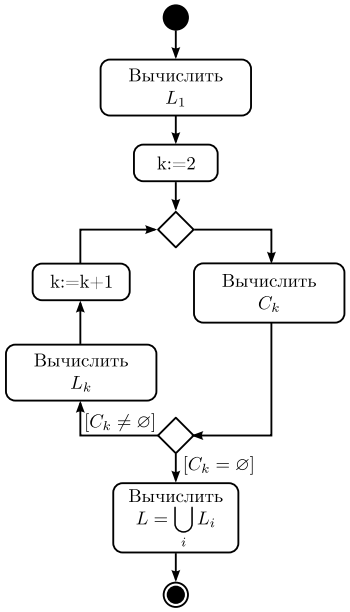


Рис. 1. Последовательный алгоритм Apriori

Последовательный алгоритм Apriori, представленный на рис. 1, кратко может быть описан следующим образом.

На первом шаге алгоритма формируется множество одноэлементных кандидатов C_1 ($k = 1$). Для этого производится перебор всех корзин B, в ходе которого каждый встреченный товар рассматривается в качестве одноэлементного кандидата и увеличивается его опорное число. Из всего множества полученных кандидатов выбираются те, опорное число которых не меньше s_{min}, и обозначаются как L_1.

На втором шаге алгоритма из часто встречающихся наборов товаров длины k (L_k) составляется множество кандидатов с большей на единицу длиной (C_{k+1}). Если на данном

шаге множество C_{k+1} получилось пустым, то выполнение алгоритма завершается. Вычисление C_{k+1} на основе L_k осуществляется с помощью суперпозиции операций *selfjoin* и *prune*.

Если представить множество L_k в виде реляционного отношения, имеющего атрибуты $item_1, item_2, \ldots, item_k$, то операция *selfjoin* представляет собой реляционную операцию Θ-соединения отношения L_k с самим собой: $selfjoin(L_k) = L_k \underset{\Theta}{\bowtie} L_k$, где Θ — это условие

$$(p.item_1 = q.item_1) \wedge (p.item_2 = q.item_2) \wedge \cdots \wedge (p.item_{k-1} = q.item_{k-1}) \wedge (p.item_k < q.item_k).$$

Операция *prune (отсечение)* заключается в сокращении множества кандидатов путем отбрасывания тех из них, у которых хотя бы одно подмножество не входит в L_k, т.е. $prune(C) = \{c \in C : \forall \hat{c} \subset c \quad \hat{c} \in L_k\}$.

Таким образом, $C_{k+1} = prune(selfjoin(L_k))$.

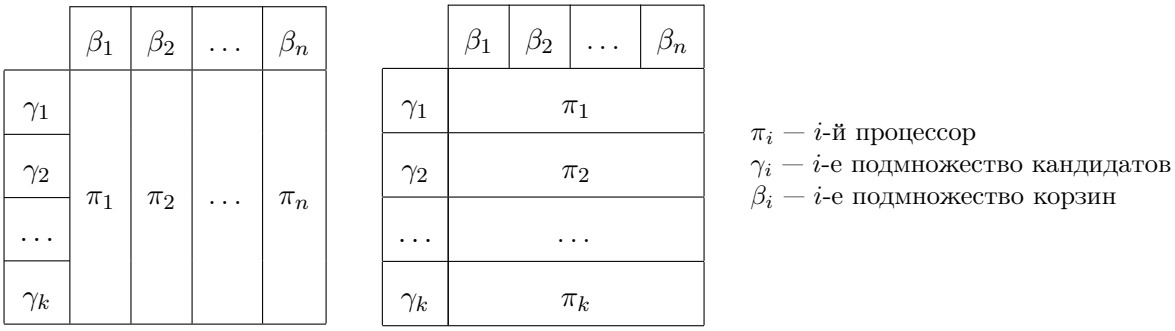
Третий шаг алгоритма заключается в том, что для новых кандидатов C_{k+1} вычисляются опорные числа. Для этого производится перебор всех корзин B, в ходе которого каждый кандидат из C_{k+1} проверяется на вхождение в каждую корзину. Выбираются все кандидаты с опорным числом не меньше s_{min} и обозначаются как множество L_{k+1}. После этого k увеличивается на единицу и выполнение алгоритма продолжается со второго шага.

3. Работы по тематике исследования

В соответствии с архитектурой Cell Broadband Engine (Cell BE) [2], процессор Cell представляет собой асимметричный многоядерный процессор, состоящий из одного управляющего ядра (Power Processing Element, PPE) и восьми вычислительных ядер (Synergistic Processing Element, SPE), которые поддерживают набор векторных инструкций и оперируют 128-битными векторами.

В работе [3] рассматривается построение параллельных алгоритмов Data Mining для решения задач кластеризации и классификации на процессорах Cell.

В настоящее время распараллеливание алгоритма Apriori осуществляется с помощью следующих основных подходов [4]: Count Distribution и Data Distribution. Распределение данных по вычислительным узлам, используемое в этих подходах, показано на рис. 2.



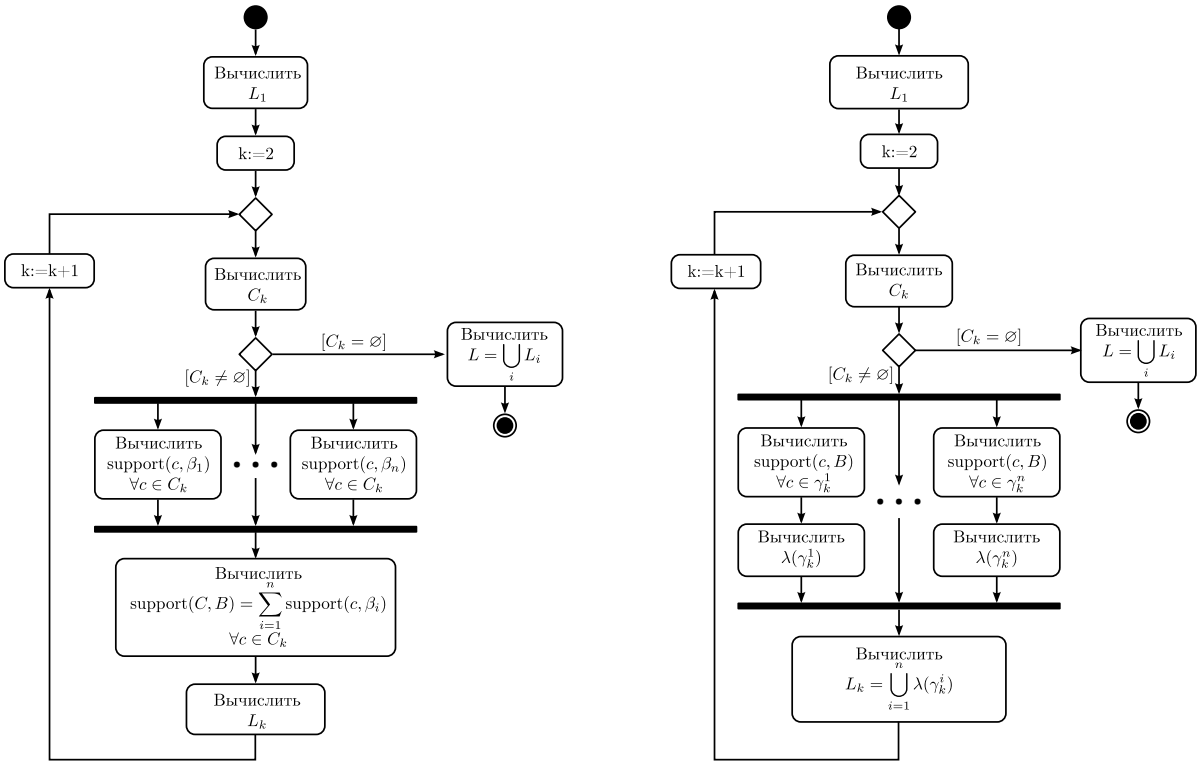
а) Подход Count Distribution б) Подход Data Distribution

Рис. 2. Подходы к распараллеливанию алгоритма Apriori

Подход *Count Distribution* [5] заключается в том, что множество корзин B разбивается на подмножества $\beta_1, \beta_2, \ldots, \beta_n$. Каждое множество β_i обрабатывается на своем процессоре π_i. Диаграмма деятельности, которая иллюстрирует подход Count Distribution, приведена на рис. 3а. Данный подход реализован для Cell в работе [6].

Подход *Data Distribution* [5] предполагает, что множество кандидатов C_k разбивается на подмножества $\gamma_k^1, \gamma_k^2, \ldots, \gamma_k^n$. Каждое множество γ_k^i обрабатывается своим процессором π_i. Диаграмма деятельности, которая иллюстрирует данный подход, представлена на рис. 3б. На данной диаграмме введена дополнительная функция $\lambda(\gamma)$, которая обозначает множе-

ство часто встречающихся наборов из подмножества кандидатов γ, т.е. $\lambda(\gamma) = \{c \in \gamma : support(c, B) \geqslant s_{min}\}$



а) Подход Count Distribution б) Подход Data Distribution

Рис. 3. Подходы к распараллеливанию алгоритма Apriori

В настоящее время подход Data Distribution, насколько нам известно, не реализован для архитектуры Cell BE.

4. Параллельный алгоритм для Cell

На основе подхода *Data Distribution* нами был разработан алгоритм *DDCapriori*, который реализует параллельный анализ рыночной корзины на процессорах Cell. В изложении алгоритма мы исходим из допущения, что анализируемое множество корзин может быть целиком размещено в оперативной памяти. Данное допущение не приводит к существенному ограничению общности, поскольку в настоящее время вычислительные системы на базе процессоров Cell комплектуют оперативной памятью с минимальным объемом 8 ГБ [2], где могут быть размещены данные о приблизительно $2 \cdot 10^9$ корзин, в каждой из которых от 1 до 16 товаров.

4.1. Проектирование

В алгоритме DDCapriori используется модель «мастер-рабочие». Нить-мастер запускается на управляющем ядре PPE и распределяет задания для рабочих. Нити-рабочие запускаются на вычислительных ядрах SPE и выполняют обработку данных, получаемых от мастера.

На рис. 4 представлены диаграммы деятельности мастера и рабочего, описывающие алгоритм DDCapriori. Для упрощения записи нами введены операции Send, Recv и MakeTask.

Процедура $Send(dst, msg)$ выполняет асинхронную отправку сообщения msg получателю dst.

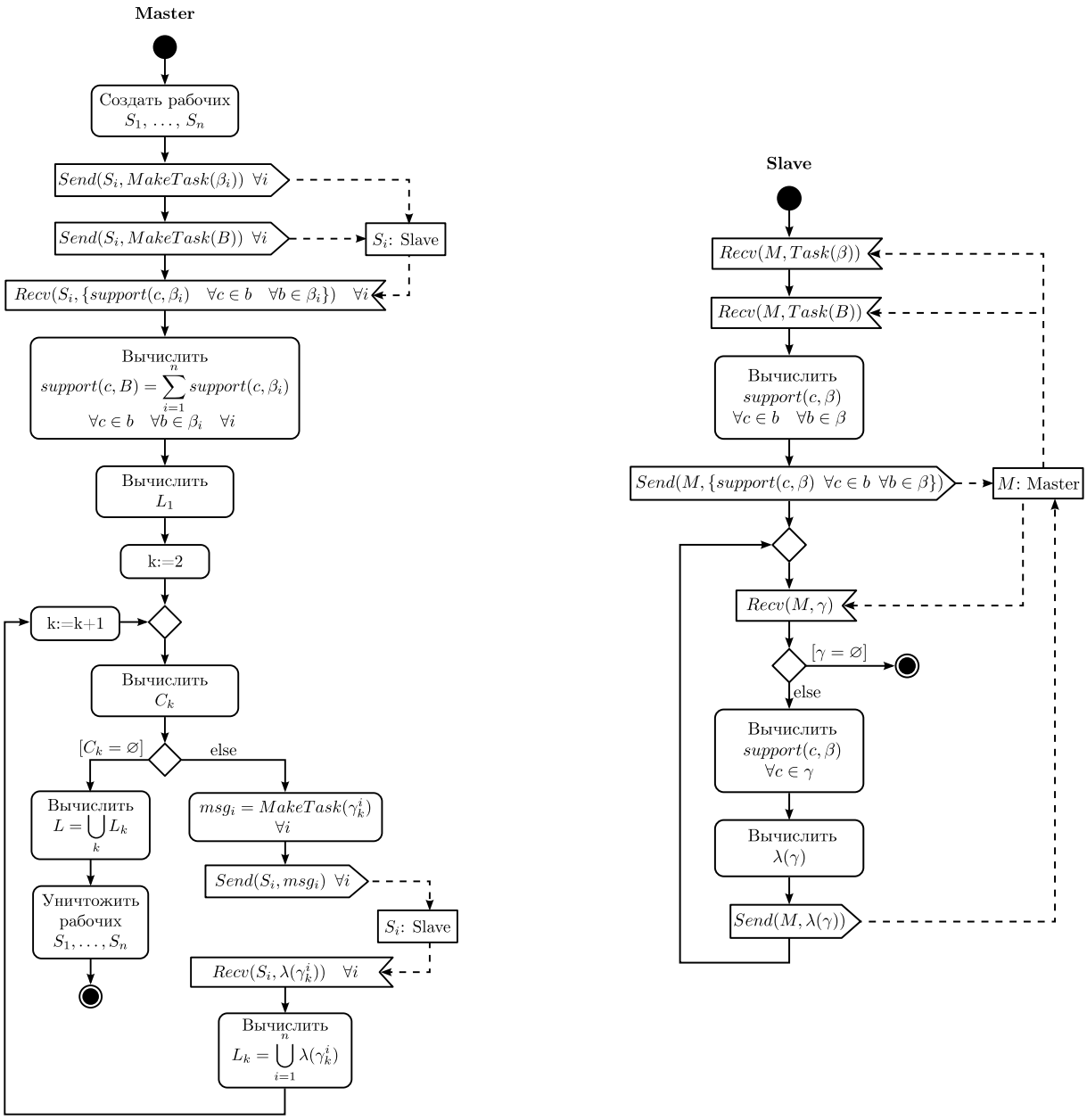


Рис. 4. Диаграммы деятельности мастера и рабочего

Процедура $Recv(src, msg)$ выполняет синхронное получение сообщения msg от отправителя src.

Функция $MakeTask(d)$ создает и возвращает задание на обработку данных d. Задание представляет собой совокупность адреса, по которому располагаются данные в оперативной памяти, и размера этих данных. В качестве данных может выступать множество корзин либо множество кандидатов.

Идея предлагаемого параллельного алгоритма заключается в том, чтобы возложить на мастера задачу формирования множеств C_k и L_k, а на рабочих — вычисление опорных чисел для кандидатов из C_k.

В отличие от подхода *Data Distribution*, при вычислении множества L_1 множество корзин B разбивается на подмножества, которые затем назначаются для обработки разным рабочим. Рабочий рассматривает каждый товар в своем подмножестве корзин как одноэлементный кандидат и увеличивает его опорное число всякий раз, как этот кандидат встре-

чается в корзинах. При вычислении множества L_k ($k > 1$) на рабочих распределяются кандидаты, а не корзины.

Деятельность мастера кратко может быть описана следующим образом. Сперва он создает рабочих, отправляет каждому из них задание и ожидает результаты. После получения агрегирует результаты во множество кандидатов единичной длины и, отсекая все редко встречающиеся кандидаты, формирует множество часто встречающихся наборов товаров единичной длины. Далее мастер в цикле формирует кандидаты большей длины ($k + 1$) из множества часто втречающихся товаров (длины k), распределяет сформированные кандидаты по рабочим, получает результаты подсчета опорных чисел и, отсекая редко встречающиеся кандидаты, формирует множество часто встречающихся наборов товаров большей длины ($k+1$). Цикл прерывается, если сформировать кандидаты большей длины не удается.

Деятельность рабочего кратко может быть описана следующим образом. Сначала рабочий получает задание, после чего формирует множество кандидатов единичной длины из своего подмножества корзин и отправляет результаты мастеру. Далее рабочий в цикле ожидает подмножество кандидатов, вычисляет для них опорные числа и отправляет результат мастеру. Цикл прерывается, если получено пустое множество кандидатов.

4.2. Реализация

Диаграмма классов, реализующих предложенный алгоритм, представлена на рис. 5.

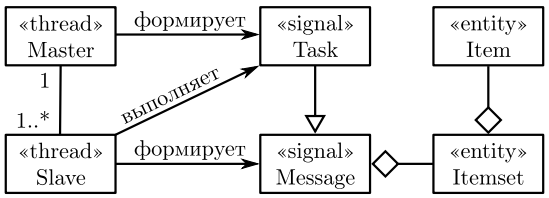


Рис. 5. Диаграмма классов, реализующих алгоритм DDCapriori

Класс *Master* реализует нить-мастер и выполняет следующие основные функции: управление рабочими и формирование множеств C_k и L_k. Экземпляр класса Master исполняется на управляющем ядре PPE.

Класс *Slave* реализует нить-рабочего и выполняет расчет опорных чисел для кандидатов из множества C_k. Экземпляры класса Slave создаются экземпляром класса Master на вычислительных ядрах SPE (по одному на каждом вычислительном ядре).

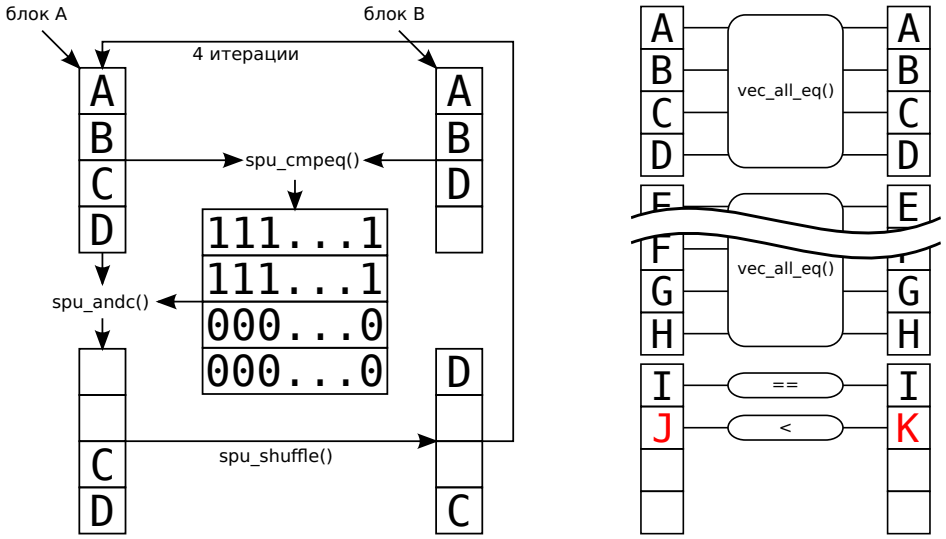
Класс *Task* служит для управления рабочими, выполняет роль сигнала и хранит входные данные для рабочего. Класс *Message* выполняет роль сигнала, отправляемого мастеру рабочим, и хранит результаты вычислений рабочего.

Вычисление опорных чисел кандидатов реализовано с помощью векторных операций процессора Cell.

Процессор Cell оперирует векторами длиной 16 байтов. В зависимости от длины идентификатора товара, в одном векторе может поместиться от 16 до 2 целочисленных идентификаторов. В нашей реализации используются 32-битные идентификаторы, то есть в одном векторе помещается 4 идентификатора.

Проверка вхождения кандидата в корзину ($c \subset b$) производится поблочно с помощью вспомогательной процедуры сравнения векторов, которая заменяет на нули в кандидате все элементы, содержащиеся в корзине. Таким образом, после применения этой процедуры ко всем парам векторов из кандидата и корзины в кандидате останутся только те элементы, которые не входят в корзину. Если их не осталось, значит кандидат полностью входит в корзину.

Сравнить каждый элемент одного вектора с каждым элементом другого вектора можно с помощью несколько векторных сравнений, выполняя на каждой итерации циклический



а) Операция проверки вхождения кандидата в корзину

б) Операция *selfjoin*

Рис. 6. Схема векторной реализации вычисления опорных чисел кандидатов

сдвиг одного из блоков (например, первого) на 1 элемент. Так как в одном векторе помещается 4 элемента, то сравнение двух векторов производится в 4 итерации. Данная процедура показана на рис. 6а.

Вычислительная сложность скалярной реализации проверки условия $c \subset b$, где b — корзина, а c — кандидат, составляет $|c| \cdot |b|$ операций, так как каждый товар в кандидате сравнивается с каждым товаром корзины.

Вычислительная сложность векторной реализации проверки условия $c \subset b$ составляет $3 \cdot 4 \cdot (\frac{|c|}{4} \cdot \frac{|b|}{4}) = \frac{3}{4} \cdot |c| \cdot |b|$. Важно отметить, что хотя такая проверка имеет вычислительную сложность всего в $\frac{4}{3}$ раза меньше, но векторные операции выполняются на ядрах SPE быстрее скалярных.

Генерация новых кандидатов C_{k+1} из множества L_k на шаге *selfjoin* реализуется с помощью векторных и скалярных операций. Они здесь применяются для проверки условия Θ-соединения. Проверка условия Θ производится по схеме, приведенной на рис. 6б. Здесь все соответствующие блоки двух корзин, кроме последнего, сравниваются с помощью векторной операции *vec_ all_ eq*. Однако в последнем блоке сравнение выполняется с помощью скалярных операций. В итоге за счет использования векторных операций вычислительная сложность понижается чуть менее чем в 4 раза относительно сложности скалярной реализации.

5. Вычислительные эксперименты

Для оценки эффективности разработанного алгоритма нами были проведены эксперименты с использованием вычислительной системы, технические характеристики которой приведены в табл. 2.

Для экспериментов нами были взяты реальные данные о посещении страниц web-сайта [9], которые также использовались для оценки эффективности алгоритмов Data Mining, предложенных в [10].

Множество B в тестовой задаче представляет собой записи о посещениях страниц сайта msnbc.com. Каждая запись содержит отметку о том, к какой семантической категории принадлежат посещенные за один сеанс страницы. В экспериментах осуществляется поиск наборов категорий страниц, часто посещаемых совместно (в течение одной сессии пользователя). Параметры экспериментов приведены в табл. 3.

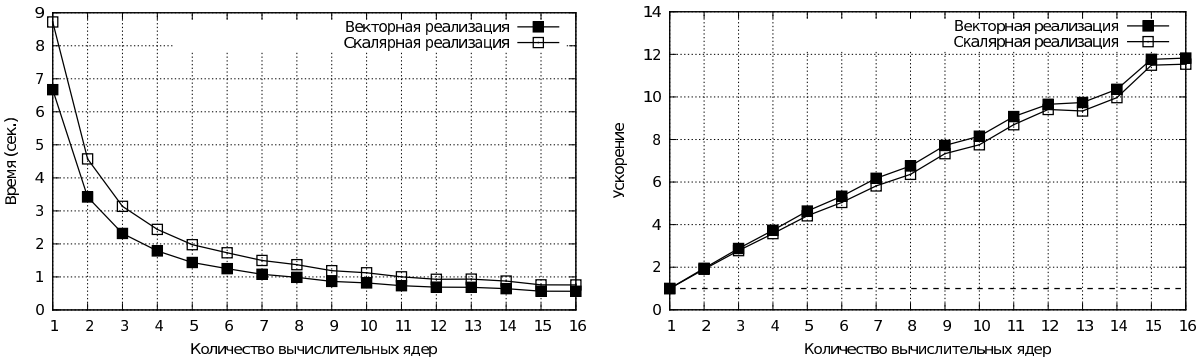
Таблица 2. Конфигурация вычислительной системы для экспериментов

Процессор	PowerXCell8i
Количество процессоров/вычислительных ядер	2/16
Тактовая частота процессора	3,2 ГГц
Пропускная способность шины EIB	200 ГБ/с

Таблица 3. Параметры экспериментов

Количество корзин ($\|B\|$)	10^6
Минимальное опорное число (s_{min})	20000
Количество рабочих (n)	от 1 до 16
Реализация проверки $c \subset b$	векторная, скалярная

Результаты экспериментов представлены на рис. 7 (на графике ускорения за единицу принята производительность при использовании одного вычислительного ядра). Эксперименты показывают, что алгоритм демонстрирует ускорение, близкое к линейному.



а) Время анализа б) Ускорение

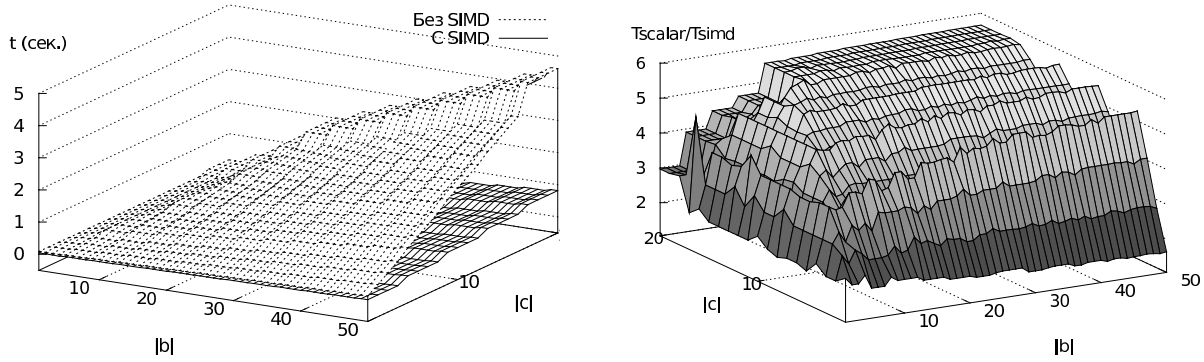
Рис. 7. Результаты экспериментов: время и ускорение

Также из результатов видно, что векторная реализация работает на тестовых исходных данных быстрее скалярной в 1,36 раза. Однако на других исходных данных это значение может сильно отличаться. Чтобы выяснить, как эта разница зависит от исходных данных, мы провели еще одну серию экспериментов, в которой замерялось время работы процедуры проверки условия $c \subset b$ в зависимости от длины кандидата c и корзины b. Результаты данной серии экспериментов показаны на рис. 8.

Как видно из результатов, чем больше дли́ны кандидата и корзины, тем более выгодно использовать векторную реализацию.

6. Заключение

В работе представлен параллельный алгоритм решения задачи анализа рыночной корзины, адаптированный для вычислительных систем на базе процессоров Cell. Параллелизм достигается путем разделения множества кандидатов на подмножества и распределения



а) Время проверки $c \subset b$ б) Выгода от SIMD при проверке $c \subset b$

Рис. 8. Результаты экспериментов: операция проверки $c \subset b$

этих подмножеств по вычислительным ядрам. При этом множество корзин передается целиком на каждое вычислительное ядро.

В реализации использована модель «мастер-рабочие». Нить-мастер запускается на управляющем ядре PPE и выполняет управление рабочими и формирование кандидатов и множеств часто встречающихся наборов на каждом шаге алгоритма. Нити-рабочие запускаются на вычислительных ядрах SPE и выполняют вычисление опорных чисел кандидатов.

Реализация выполнена на языке программирования C с использованием векторных функций из библиотек IBM Cell Broadband Engine SDK, которые позволяют эффективно реализовать операции подсчета опорного числа, отсечения и генерации кандидатов.

Литература

1. *Agrawal R., Imielinski T., Swami A.N.* Mining Association Rules between Sets of Items in Large Databases // Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. P. 207–216.

2. IBM Corporation. Cell Broadband Engine technology.
URL: http://www.alphaworks.ibm.com/topics/cell (дата обращения: 01.06.2009).

3. *Buehrer G., Parthasarathy S., Goyder M.* Data Mining on Cell Broadband Engine // Proceedings of the 22nd annual International Conference on Supercomputing. 2008. P. 26–35.

4. *Zaki M.J.* Parallel and Distributed Association Mining: A Survey // IEEE Concurrency. October 1999. Vol. 7. No. 4. P. 14–25.

5. *Zaki M.J., Ogihara M., Parthasarathy S., Li W.* Parallel data mining for association rules on shared-memory multi-processors // Proceedings of the 1996 ACM/IEEE conference on Supercomputing. 1996. Article No. 43.

6. *Duan R., Strey A.* Data Mining Algorithms on the Cell Broadband Engine // Proceedings of the 14th International Euro-Par Conference. 2008. P. 665–675.

7. *Han S., Karypis G., Kumar V.* Scalable Parallel Data Mining for Association Rules // IEEE Transactions on Knowledge and Data Engineering. Vol. 12. Issue 3. P. 337–352.

8. IBM Cell Broadband Engine SDK, Version 3.0 documentation.
URL: http://www-01.ibm.com/chips/techlib/techlib.nsf/products/IBM_SDK_for_Multicore_Acceleration (дата обращения: 01.06.2009).

9. msnbc.com anonymous web data.
URL: http://kdd.ics.uci.edu/databases/msnbc/msnbc.html (дата обращения: 13.12.2009).

10. *Cadez I., Heckerman D., Meek C., Smyth P., White S.* Visualization of Navigation Patterns on a Web Site Using Model Based Clustering. Technical Report MSR-TR-00-18. Microsoft Research. 2000.
URL: http://research.microsoft.com/pubs/69752/tr-2000-18.pdf (дата обращения: 13.12.2009).