ПРОГРАММИРОВАНИЕ, 2021, № 2, с. 15–27

КОМПЬЮТЕРНАЯ АЛГЕБРА

УДК 004.272.2,512.62,519.688,519.816

ФАКТОРИЗАЦИЯ БУЛЕВЫХ ПОЛИНОМОВ: ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ И ЭКСПЕРИМЕНТАЛЬНАЯ ОЦЕНКА

© 2021 г. П. Г. Емельянов[a],*, М. Кришна[c],, В. Кулкарни[b],***, С. К. Нанди[b],****, Д. К. Пономарев [a],*****, С. Раха[b],********

[a] *Институт систем информатики им. А.П. Ершова СО РАН; Новосибирский государственный университет, Новосибирск, Россия*

[b] *Department of Computational and Data Sciences, Indian Institute of Science, Бангалор, Индия*

[c] *Morphing Machines Pvt Ltd, Бангалор, Индия*

**E-mail: emelyanov@iis.nsk.su*

***E-mail: madhava@iisc.ac.in*

****E-mail: vadirajk@iisc.ac.in*

*****E-mail: nandy@iisc.ac.in;*

******E-mail: ponom@iis.nsk.su*

*******E-mail: raha@iisc.ac.in*

Поступила в редакцию 02.04.2020 г.
После доработки 06.05.2020 г.
Принята к публикации 06.05.2020 г.

Факторизация полиномов – классическая алгоритмическая проблема алгебры, имеющая широкий спектр приложений. Особый интерес представляет факторизация над конечными полями, среди которых поле порядка два является, вероятно, наиболее важным в связи с представлением булевых функций полиномами Жегалкина. В частности, факторизация булевых полиномов соответствует конъюнктивной декомпозиции булевых функций, заданных в алгебраической нормальной форме. Кроме того, факторизация дает решение проблемы декомпозиции функций, заданных в СДНФ и позитивных ДНФ, а также декартовой декомпозиции реляционых данных. Эти приложения демонстрируют важность разработки быстрых алгоритмов факторизации. В статье мы рассматриваем некоторые недавно предложенные алгоритмы факторизации полиномиальной сложности и описываем параллельную MIMD-реализацию, которая использует как параллелизм уровня задачи, так и параллелизм уровня данных. Мы представляем эксперименты, выполненные на бенчмарках логического синтеза и на синтетических (случайных) полиномах, которые показывают значительное ускорение факторизации. В заключение представлены результаты тестирования параллельной реализации алгоритма на массивнопараллельной многоядерной архитектуре (Redefine).

DOI: 10.31857/S0132347421020059

1. ВВЕДЕНИЕ

Факторизация полиномов – классическая алгоритмическая проблема алгебры [1], которая имеет широкий спектр приложений. Один вариант проблемы, привлекающий особое внимание — факторизация булевых полиномов, т.е. мультилинейных полиномов над конечным полем порядка 2. Булев многочлен – это одно из широко известных представлений булевых функций, также известных в математической логике как полиномы Жегалкина [2] или каноническая форма Рида–Маллера [3] в синтезе логических схем и теории помехоустойчивого кодирования. В последнее время это представление становится все более популярным в связи со следующими преимуществами. Во-первых, это естественное и компактное представление некоторых важных классов булевых функций (например, арифметические функции, кодеры, и т.д.). Во-вторых, оно допускает естественное отображение в некоторых схемных технологиях (например, основанных на FPGA или в наноструктурной электронике), а также имеет хорошие свойства для тестирования.

Факторизация булевых полиномов является частным случаем конъюнктивной декомпозиции булевых функций (декомпозиция в конъюнкцию функций с непересекающимися множествами переменных или AND-декомпозиция). Действительно, для булева полинома, в котором всякая переменная имеет степень 1, факторизация дает

факторы с непересекающимися наборами переменных: $F(X,Y) = F_1(X) \times F_2(Y)$, $X \cap Y = \varnothing$.

Недавно было показано [4, 5], что факторизация булевых полиномов решает задачу конъюнктивной декомпозиции функций, заданных в СДНФ и позитивных ДНФ. Кроме того, это дает метод декартовой декомпозиции таблиц реляционных баз данных [6, 7], т.е. отыскание таблиц таких, что их неупорядоченное декартово произведение совпадает с исходной таблицей (обращение оператора CROSS JOIN). Несколько примеров, иллюстрирующих указанные применения, приведены ниже.

Рассмотрим следующую ДНФ

$$\varphi = (x \wedge u) \vee (x \wedge v) \vee (y \wedge u) \vee \vee (y \wedge v) \vee (x \wedge u \wedge v).$$

Она эквивалентна

$$\psi = (x \wedge u) \vee (x \wedge v) \vee (y \wedge u) \vee (y \wedge v),$$

так как последняя конъюнкция избыточна. Можно видеть, что

$$\psi \equiv (x \vee y) \wedge (u \vee v)$$

и компоненты декомпозиции $x \vee y$ и $u \vee v$ могут быть восстановлены из факторов полинома

$$F_\psi = xu + xv + yu + yv = (x + y) \times (u + v),$$

построенного для ψ.

Следующая СДНФ

$$\varphi = (x \wedge \neg y \wedge u \wedge \neg v) \vee (x \wedge \neg y \wedge \neg u \wedge v) \vee \vee (\neg x \wedge y \wedge u \wedge \neg v) \vee (\neg x \wedge y \wedge \neg u \wedge v)$$

эквивалентна

$$(x \wedge \neg y) \vee (\neg x \wedge y) \vee (u \wedge \neg v) \vee (\neg u \wedge v)$$

и компоненты декомпозиции φ могут быть восстановлены из факторов полинома

$$F_\varphi = x\bar{y}u\bar{v} + x\bar{y}\bar{u}v + \bar{x}yu\bar{v} + \bar{x}y\bar{u}v = (x\bar{y} + \bar{x}y) \times (u\bar{v} + \bar{u}v), \quad (1.1)$$

построенного для φ.

Наконец, декартова декомпозиция следующей таблицы

B	E	D	A	C
z	q	u	x	y
y	q	u	x	y
y	r	v	x	z
z	r	v	x	z
y	p	u	x	x
z	p	u	x	x

$=$

A	B
x	y
x	z

$\times$

C	D	E
x	u	p
y	u	q
z	v	r

может быть найдена факторизацией полинома

$$z_B \times q \times u \times x_A \times y_C + y_B \times q \times u \times x_A \times y_C + + y_B \times r \times v \times x_A \times z_C + z_B \times r \times v \times x_A \times z_C + + y_B \times p \times u \times x_A \times x_C + z_B \times p \times u \times x_A \times x_C = = (x_A \times y_B + x_A \times z_B) \times \times (q \times u \times y_C + r \times v \times z_C + p \times u \times x_C),$$

построенного на основе содержимого таблицы.

Декомпозиция позволяет находить более компактное представление булевых функций и таблиц данных, которые применяются в области синтеза логических схем, самоорганизующихся баз данных и поиска зависимостей в данных. Так как большие объемы входов – типичная ситуация в перечисленных задачах, чрезвычайно важно иметь эффективные на практике алгоритмы факторизации булевых полиномов.

В [8], изучая проблему факторизации полиномов над произвольными конечными полями, представленных в виде арифметических схем, Шпилька и Волкович показали связь между факторизацией полиномов и проверкой тождества полиномов. В частности, из их результата следует, что булев полином может быть факторизован за время $O(l^3)$, где l – размер полинома заданного как символьная последовательность. Подход использует умножение полиномов, полученных из исходного, и эта операция является дорогостоящей для больших входов. В [4] был предложен альтернативный подход к факторизации, где явного умножения можно избежать. В дальнейшем этот подход обсуждался в [9].

В данной статье мы рассматриваем параллельную версию алгоритма декомпозиции из [4, 9] и сравниваем его с параллельными реализациями аналогичных алгоритмов. В разделе 2 мы сначала приводим обзор последовательного алгоритма факторизации и его вариантов. В разделе 3 описывается параллельная MIMD реализация алгоритма и далее в разделе 4 представлен количественный анализ параллельного и последовательного алгоритмов. Наконец, в разделе 5 приведены результаты тестирования нашего алгоритма на массивно-параллельной многоядерной архитектуре (Redefine).

2. ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

Введем базовые определения и обозначения и рассмотрим последовательный алгоритм факторизации из [4, 9].

Полином $F \in \mathbb{F}_2[x_1, \ldots, x_n]$ называется *факторизуемым* если $F = F_1 \times \ldots \times F_k$, где $k \geq 2$ и $F_1, \ldots, F_k$ неконстантные полиномы. В противном случае F называется *неприводимым*. Полиномы $F_1, \ldots, F_k$ на-

зываются *факторами F*. Важно отметить, что так как рассматриваются мультилинейные полиномы (каждая переменная может входить только в степени не более 1), факторы являются полиномами *над непересекающимися множествами переменных*. В дальнейшем мы полагаем, что полином F не имеет *тривиальных факторов* (*делителей*), т.е., ни x, ни $x+1$ не делят F. Очевидно, тривиальные делители легко распознаваемы.

Пусть F – полином. Для переменной x из множества переменных $Var(F)$ полинома F и значения $a \in \{0, 1\}$, обозначим через $F_{x=a}$ полином, полученный из F посредством подстановки для x значения a. $\dfrac{\partial F}{\partial x}$ обозначает *формальную производную F* по переменной x. Если дана переменная z, то $z \mid F$ означает, что z делит F, т.е., z входит в каждый моном F (заметим, что это эквивалентно условию $\dfrac{\partial F}{\partial z} = F_{z=1}$). Если дано множество переменных Σ и моном m, то определим *проекцию m* на Σ следующим образом. Она равна 1, если m не содержит переменных из Σ, и в противном случае равна моному, полученному из m удалением всех переменных, которые не входят в Σ. *Проекция* полинома F на Σ, обозначаемая $F\big|_{\Sigma}$ – есть полином, полученный как сумма проекций мономов F на Σ, в которой мономы-дубликаты удалены.

2.1. Алгоритм факторизации

Алгоритм 1 описывает последовательную версию процедуры факторизации. Как уже было упомянуто, факторы булева полинома имеют непересекающиеся множества переменных. Это свойство используется в алгоритме, который пытается вычислить разбиение переменных по компонентам. Если разбиение построено, то соответствующие факторы могут быть найдены с помощью проекций исходного полинома на компоненты полученного разбиения. Далее мы полагаем, что входной полином F имеет, по крайней мере, две переменные и не содержит пары одинаковых мономов.

Algorithm 1. Последовательный алгоритм факторизации

Input Факторизуемый булев полином F

Output F_{same} и F_{other}, которые являются факторами входного полинома F

1: Выбрать произвольную переменную x, входящую в F

2: Пусть $A = \dfrac{\partial F}{\partial x}$, $B = F_{x=0}$

3: Пусть $\Sigma_{same} = x$, $\Sigma_{other} = \varnothing$, $F_{same} = 0$, $F_{other} = 0$

4: **for each** $y \in var(F) \backslash \{x\}$ **do**

5: Пусть $C = \dfrac{\partial A}{\partial y}$, $D = \dfrac{\partial B}{\partial y}$

6: **if** IsEqual(A, D, B, C) **then**

7: $\Sigma_{other} = \Sigma_{other} \cup \{y\}$

8: **else**

9: $\Sigma_{same} = \Sigma_{same} \cup \{y\}$

10: **end if**

11: **end for**

12: Если $\Sigma_{other} = \emptyset$, тогда F неприводим.

13: Вернуть полиномы F_{same} и F_{other} как проекции F на Σ_{same} и Σ_{other}, соответственно.

Таким образом, алгоритм выбирает произвольным образом переменную x из F и разбивает множество переменных F на два множества относительно x:

• первое множество Σ_{same} содержит выбранную переменную x и соответствует фактору, представляющему неприводимый полином;

• второе множество Σ_{other} соответствует второму фактору, который допускает дальнейшую факторизацию.

Факторы F_{same} и F_{other} получаются проекциями исходного полинома на Σ_{same} и Σ_{other}, соответственно.

В строках 1–3 выбирается произвольная x из F и вычисляются полиномы A и B. A – это производная F по переменной x, а B – полином, полученный из F означиванием x нулем. Строки 4–11 занимают цикл над множеством переменных F, откуда исключена переменная x. Для переменной на очередной итерации вычисляются полиномы C и D, где C – производная A и D – производная B. Затем проверяется равенство $AD = BC$. Для проверки этого равенства вызывается процедура **IsEqual** (строка 6). Она описывается в следующем разделе.

2.2. Процедура IsEqual

В Алгоритме 2 представлена последовательная версия процедуры IsEqual.

• Процедура принимает на вход полиномы A, B, C, D и проверяет равенство $AD = BC$ посредством рекурсии.

• Строки 1–16 реализуют базовые случаи, когда равенство $AD = BC$ может быть определено простым образом.

• В строках 3–6 проверяется, делит ли переменная z полиномы A, B, C, D так, что условие в строке 4 верно. Если это не верно, то переменная z из A, B, C, D удаляется (строка 7) и выполняется проверка на равенство произведений полученных полиномов.

• В строках 17–26 процедура **IsEqual** вызывается рекурсивно на полиномах меньших размеров.

2.3. Возможность параллелизма

Суть Алгоритма 1 заключена в цикле, представленном строками 4–11. Легко заметить, что различные итерации цикла независимы. Следовательно цикл обеспечивает параллелизм уровня потока, который может быть использован для улучшения производительности. Условный блок внутри цикла в строках 6–10 может быть использован для организации параллелизма уровня задачи между многими потоками.

Algorithm 2. Последовательная процедура IsEqual

Input Булевы полиномы A, B, C, D

Output TRUE если AD равно BC и FALSE иначе.

1: If A = 0 or D = 0 then return (B = 0 or C = 0)

2: If B = 0 or C = 0 then return FALSE

3: **for each** z, встречающуюся, по крайней мере, в одном из A, B, C, D **do**

4: **if** $z \mid A$ or $z \mid D$ x or $z \mid B$ or $z \mid C$ **then**

5: return FALSE

6: **end if**

7: Заменить каждый $X \in \{A, B, C, D\}$ на $\frac{\partial X}{\partial z}$ при условии $z \mid X$

8: **end for**

9: **if** A = 1 and D = 1 **then** return (B = 1 and C = 1)

10: **end if**

11: **if** B = 1 and C = 1 **then** return FALSE

12: **end if**

13: **if** A = 1 and B = 1 **then** return (D = C)

14: **end if**

15: **if** D = 1 and C = 1 **then** return (A = B)

16: **end if**

17: Выбрать переменную z

18: **if** not(IsEqual($A_{z=0}, D_{z=0}, B_{z=0}, C_{z=0}$)) **then** return FALSE

19: **end if**

20: **if** not(IsEqual$\left(\frac{\partial A}{\partial z}, \frac{\partial D}{\partial z}, \frac{\partial B}{\partial z}, \frac{\partial C}{\partial z}\right)$) **then** return FALSE

21: **end if**

22: **if** IsEqual$\left(\frac{\partial A}{\partial z}, B_{z=0}, A_{z=0}, \frac{\partial B}{\partial z}\right)$ **then** return TRUE

23: **end if**

24: **if** IsEqual$\left(\frac{\partial A}{\partial z}, C_{z=0}, A_{z=0}, \frac{\partial C}{\partial z}\right)$ **then** return TRUE

25: **else** return FALSE

26: **end if**

Множество частей Алгоритма 2 поддаются параллелизму. Проверка делимости полиномов A, B, C, D в строках 3–6 процедуры **IsEqual** может быть выполнено независимо. В строках 17–26 рекурсивные вызовы **IsEqual** независимы и представляют параллелизм уровня задачи. В следующем разделе рассматривается параллельный алгоритм, использующий эти наблюдения.

3. ПРЕДЛАГАЕМЫЙ ПОДХОД

3.1. Параллельный алгоритм факторизации

Алгоритм 3 описывает параллельную версию алгоритма факторизации. В строках 1–3 выбирается произвольная перменная x из множества переменных полинома F и вычисляются полиномы A и B. В строках 4–11 порождением нескольких потоков несколько итераций цикла выполняются параллельно. Каждый поток возвращает два множества Σ_{same}^{tid} и Σ_{other}^{tid}, специфичные для потока и обозначаемые идентификатором потока tid. В строках 12–14, множества переменных Σ_{same} и Σ_{other} вычисляются как объединение данных соответствующих разным потокам. Отметим синхронизационный барьер для всех параллельных потоков.

Algorithm 3. Параллельный алгоритм факторизации

Input Факторизуемый булев полином F

Output F_{same} и F_{other}, которые являются факторами входного полинома F

1: Выбрать произвольную переменную x из F

2: Пусть $A = \frac{\partial F}{\partial z}$, $B = F_{z=0}$

3: Пусть $\Sigma_{same} = x$, $\Sigma_{other} = \emptyset$, $F_{same} = 0$, $F_{other} = 0$

4: **for each** $y \in var(F) \backslash \{x\}$ **do in parallel**

5: Пусть $C = \frac{\partial A}{\partial y}$, $D = \frac{\partial B}{\partial y}$

6: **if** IsEqual(A, D, B, C) **then**

7: $\Sigma_{other}^{tid} = \Sigma_{other}^{tid} \cup \{y\}$

8: **else**

9: $\Sigma_{same}^{tid} = \Sigma_{same}^{tid} \cup \{y\}$

10: **end if**

11: **end for**

12: **Ждать пока все параллельные потоки не закончатся**

13: $\Sigma_{other} = \bigcup_{tid} \Sigma_{other}^{tid}$

14: $\Sigma_{same} = \bigcup_{tid} \Sigma_{same}^{tid}$

15: Если $\Sigma_{other} = \emptyset$, тогда F неприводим; остановиться.

16: Вернуть полиномы F_{same} и F_{other}, полученные как проекции F на Σ_{same} и Σ_{other}, соответственно.

3.2. Параллельная версия функции IsEqual

Алгоритм 4 описывает параллельную версию процедуры IsEqual. Алгоритм принимает на вход четыре полинома A, D, B, C и проверяет равенство $AD = BC$. Строки 1–21 описывают случаи, когда определение равенства $AD = BC$ тривиально.

Algorithm 4. Параллельная версия функции IsEqual

Input Булевы полиномы A, B, C, D

Output TRUE если AD = BC и FALSE, иначе.

1: If A = 0 or D = 0 then return (B = 0 or C = 0)

2: If B = 0 or C = 0 then return FALSE

3: **for each** z, входящая в хотя бы один из A, B, C, D **do in parallel**

4: Положить $flag^{tid}$ = True

5: **if** $z \mid A$ or $z \mid D$ x or $z \mid B$ or $z \mid C$ **then**

6: Положить $flag^{tid}$ = FALSE

7: **end if**

8: Заменить каждый $X^{tid} \in \{A, B, C, D\}$ на $\dfrac{\partial X^{tid}}{\partial z}$, при условии $z \mid X^{tid}$

9: **end for**

10: **Ждать пока все параллельные потоки не закончатся**

11: if $\bigwedge_{tid} flag^{tid} = FALSE$ **then** return FALSE

12: **end if**

13: $X = \bigcap_{tid} X^{tid}$, for $X \in \{A, B, C, D\}$

14: **if** A = 1 and D = 1 **then** return (B = 1 and C = 1)

15: **end if**

16: **if** B = 1 and C = 1 **then** return FALSE

17: **end if**

18: **if** A = 1 and B = 1 **then** return (D = C)

19: **end if**

20: **if** D = 1 and C = 1 return (A = B)

21: **end if**

22: Выбрать переменную z

23: **Выполнять следующие 4 строки параллельно**

24: x = not(IsEqual($A_{z=0}, D_{z=0}, B_{z=0}, C_{z=0}$))

25: y = not(IsEqual$\left(\dfrac{\partial A}{\partial z}, \dfrac{\partial D}{\partial z}, \dfrac{\partial B}{\partial z}, \dfrac{\partial C}{\partial z}\right)$)

26: z = IsEqual$\left(\dfrac{\partial A}{\partial z}, B_{z=0}, A_{z=0}, \dfrac{\partial B}{\partial z}\right)$

27: w = IsEqual$\left(\dfrac{\partial A}{\partial z}, C_{z=0}, A_{z=0}, \dfrac{\partial C}{\partial z}\right)$

28: **Ждать пока все параллельные потоки не закончатся**

29: **if** not(x) **then** return FALSE

30: **end if**

31: **if** not(y) **then** return FALSE

32: **end if**

33: **if** z **then** return TRUE

34: **end if**

35: **if** w **then** return TRUE

36: **else** return FALSE

37: **end if**

В строках 3–9 проверяется, делит ли переменная z входные полиномы A, D, B, C так, что условие в строке 5 выполнено. Если это не так, все они делятся на z для того, чтобы получить приведенные полиномы (не содержащие тривиальных делителей). Вышеупомянутые операции выполняются для каждой переменной независимо в параллельно работающих потоках. В строке 8 в каждом потоке проверяется, является ли переменная z^{tid} (tid – идентификатор потока) делителем какого-то полинома A, B, C, D. Если z^{tid} делит какой-то из полиномов A, B, C, D, то вычисляются соответствующие приведенные полиномы A^{tid}, D^{tid}, B^{tid}, C^{tid}. В строке 13 берется попарное пересечение соответствующих мономов полиномов из потоков A^{tid}, D^{tid}, B^{tid}, C^{tid} для того, чтобы сформировать полиномы, свободные от тривиальных делителей. Пересечение двух мономов m_1, m_2 равно 1, если m_1, m_2 не имеют общих переменных, в противном случае это моном, состоящий из переменных, которые входят в оба монома m_1 и m_2. В строках 23–27 выполняются четыре независимых рекурсивных вызова функции IsEqual в параллельных потоках. В строках 28–37 ожидается завершение всех потоков и сравнение их результатов для формирования окончательного результата. Отметим

в строках 10 и 28 синхронизирующий барьер для всех параллельных потоков.

3.3. Оценка ускорения параллельного алгоритма, основанного на распределении переменных по параллельным потокам исполнения

Реализация параллелизма в Алгоритмах 3 и 4 исходит из предположения о неограниченных возможностях использования ресурсов — параллельно запускаются все достаточно объемные секции кода и коммуникация между ними не представляет никаких сложностей. Это интересно с теоретической точки зрения для исследования глубины параллелизма. Однако, с точки зрения оценки глубины "практического" параллелизма, следует рассмотреть некоторые ограничения. В частности, фиксированное количество процессоров/ядер, доступных для исполнения программы. В этой связи можно рассмотреть вариант Алгоритма 3, в котором распараллеливание в цикле по всем переменным $y \in var(F)\backslash\{x\}$ заменяется на параллелизм по блокам разбиения множества переменных $var(F)\backslash\{x\}$. Количество блоков определяется количеством доступных процессоров/ядер. Следует отметить, что параллельные потоки исполнения могут не взаимодействовать друг с другом (иметь изолированную память) что, например, может быть привлекательным при реализации алгоритма факторизации на входящих в широкую практику NUMA-архитектурах. Вопрос повышения эффективности вычислений за счет доступа к переиспользуемым общим данным на данный момент остается открытым.

Изучая поведение алгоритма факторизации полиномов, представленных в виде списка мономов (ESOP), было обнаружено, что IsEqual работает в разы больше времени для переменных из компоненты, которая противоположна компоненте, содержащей выбранную на первом шаге переменную: доказательство, что равенство не выполняется, заканчивается быстрее, чем доказательство равенства, когда нужно довести разбиение полиномов до "мельчайшего", чтобы проверка равенства была тривиальной. Один из выводов, который из этого следует, – неприводимые полиномы будут обрабатываться быстрее, чем разложимые.

Оценим возможное ускорение в этом случае. Рассмотрим следующие параметры:

- N – количество переменных в полиноме;
- $\mu \in (0\ldots1)$ – доля переменных из Σ_{same} и $1-\mu$ – доля переменных из Σ_{other} среди всех N переменных;
- t – среднее время обработки переменной из Σ_{same};
- $t\nu, \nu > 1$, – среднее время обработки переменной из Σ_{other};
- P – количество доступных в параллельной версии процессоров (отметим, задачи, запущенные на них, никак не синхронизируются).

Время решения задачи факторизации (той части, где ищется разбиение на Σ_{same} и Σ_{other}) на одном процессоре

$$T_1 = \mu Nt + (1-\mu)Nt\nu = Nt(\mu + (1-\mu)\nu).$$

Легко видеть, что ускорение вычислений зависит от того, насколько "удачно" распределились по потокам переменные (разумным предположением является, что каждый поток обрабатывает примерно одинаковое количество переменных). В данном случае самый удачный вариант – это когда в каждом потоке воспроизводится исходная пропорция переменных из Σ_{same} и Σ_{other}:

$$T_P^{best} = \frac{T_1}{P}.$$

Таким образом, в идеале возможно ускорение в P раз (это максимально возможное ускорение).

Худших варианта два:

1. Существует поток, который обрабатывает переменные только из Σ_{other}. Тогда

$$T_P^{worst} = \frac{N}{P}t\nu.$$

$$\frac{T_1}{T_P^{worst}} = P\frac{(\mu + (1-\mu)\nu)}{\nu} = P\left(1 - \mu\frac{\nu-1}{\nu}\right).$$

Аналитическая зависимость для ν пока не установлена, но на достаточно больших полиномах стабильно проявляется, что $\nu \gg 1$, а значит ускорение составляет примерно $P - \mu P$.

2. Все потоки кроме одного обрабатывают переменные только из Σ_{same}, а этот выделенный поток обрабатывает как переменные из Σ_{same}, т.е. $\mu > \frac{P-1}{P}$, так и все из Σ_{other}.

$$T_P^{worst} = \left(\mu - \frac{P-1}{P}\right)Nt + (1-\mu)Nt\nu$$

$$\frac{T_1}{T_P^{worst}} = \frac{\mu + (1-\mu)\nu}{\mu + (1-\mu)\nu - \frac{P-1}{P}} \le 1 + \frac{\mu}{(1-\mu)\nu}.$$

Таблица 1. Статистика об ускорении при 2, 3 и 4 потоках

Потоков	Мин.	Макс.	Сред.	Ст. откл.	Медиана
2	1.528	1.947	1.715	0.062	1.704
3	1.836	2.700	2.198	0.093	2.196
4	2.150	3.673	2.630	0.155	2.613

Таблица 2. Результаты запуска на процессоре Xeon 2.8 GHz с 4-мя потоками

Тест	Послед.	Многопот.	Ускорен.
ITC'99	4324(s)	1441(s)	3.01
Iscas'85	7181(s)	2633(s)	2.73
EPFL Adder	1381(s)	374(s)	3.69

Видно, что при таком распределении переменных по потокам и при $\nu \gg 1$ рост количества процессоров не будет оказывать значительное влияние на ускорение.

После анализа ускорения при таком распараллеливании в Maple-реализации алгоритма было сделано следующее. Перед разбиением множества переменных по k потокам – разрезания массива переменных на k частей – выполняется несколько случайных перестановок этого массива с целью балансировки распределения переменных в разных частях.

4. ЭКСПЕРИМЕНТЫ ПО РАСПАРАЛЛЕЛИВАНИЮ ОБРАБОТКИ ПЕРЕМЕННЫХ ПО ПОТОКАМ

Для экспериментальной оценки последовательного и параллельного алгоритмов, основанного на распределении проверки переменных по потокам, использовались известные бенчмарки для логического синтеза и искусственно сгенерированные (случайные) булевы полиномы.

4.1. Реализация полиномов с использованием Zero-suppressed Decision Diagrams

Для представления булевых полиномов используется ZDD из библиотеки CUDD версии 3.0.0 [10]. Его можно трактовать как представление в виде множества (сам полином) подмножеств (его мономы) некоторого конечного множества (переменные полинома). Реализация использует стандартный механизм потоков в C++. Для тестирования использовались 500 случайных полиномов со 100 переменными и 10000 мономами (с различными пропорциями переменных и мономов в би-компонентах разложения). В табл. 1 приведены результаты тестирования ускорения алгоритма с распараллеливанием обработки переменных на 2, 3 и 4 потока. На рис. 1 основная статистическая информация приведена в графическом виде.

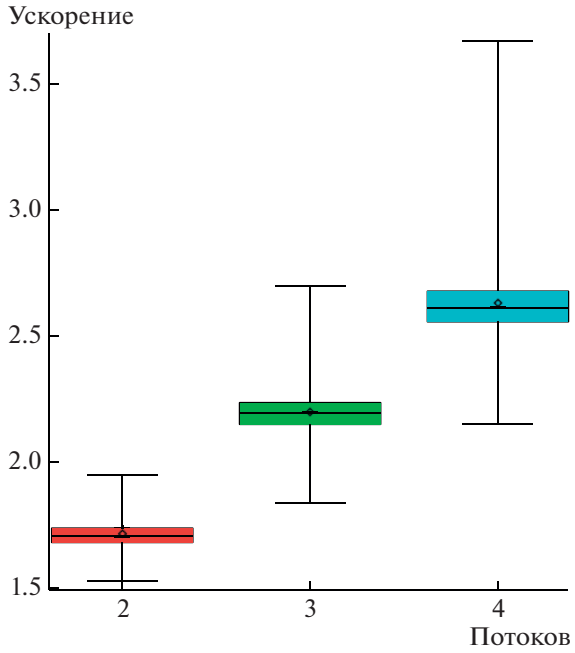


Рис. 1. Ускорение при распараллеливании обработки переменных.

4.2. Реализация полиномов в виде списков мономов

Мы использовали бенчмарки логического синтеза ITC'99 [11], Iscas'85 [12] и бенчмарк n-битного

Таблица 3. Время факторизации случайных полиномов на процессоре Xeon 2.8 GHz с 4-мя потоками

Мономов	Послед.	Многопот.	Ускорен.
10	0.023(s)	0.0074(s)	3.12
50	16.29(s)	5.07(s)	3.21
100	103.5(s)	30.44(s)	3.4
500	483.6(s)	178.1(s)	2.7
1000	1165(s)	520.9(s)	2.2
5000	1430(s)	735.11(s)	1.91
10000	12614(s)	8034(s)	1.57

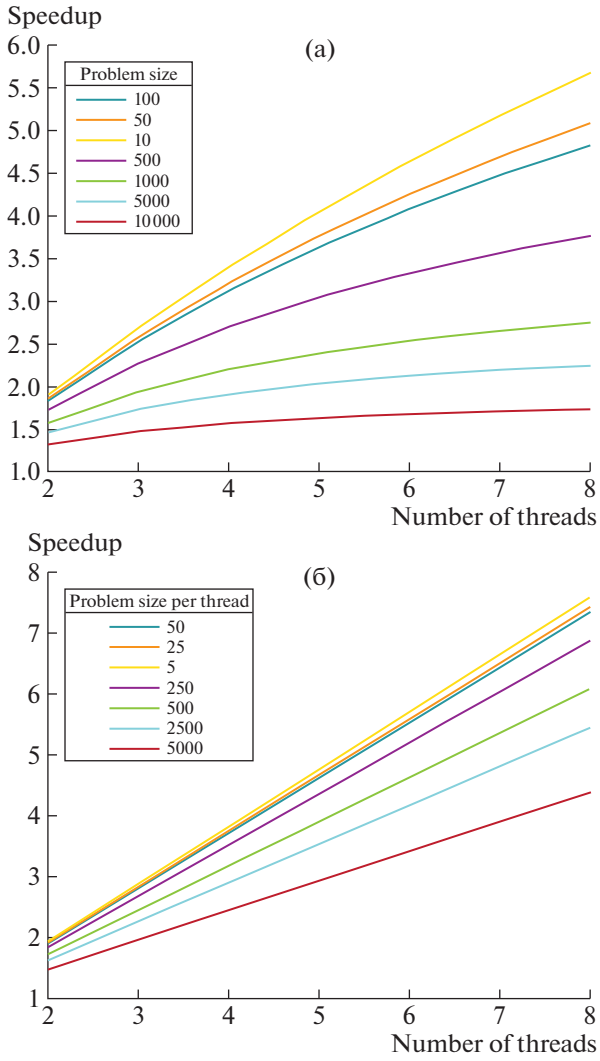
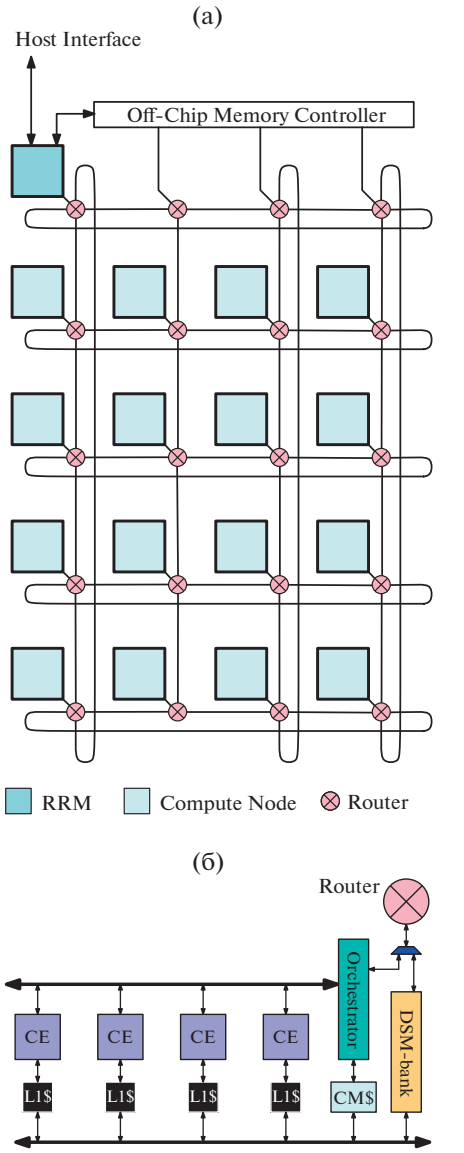


Рис. 2. (а) Ускорение за счет параллелизации по отн. к числу потоков для одного входа. (б) Ускорение за счет параллелизации по отн. к числу потоков и фикс. входа для каждого потока.



L1$: Private L1-cache for global memory address space
CM$: Cache for context memory address space
DSM-bank: Distributed Shared Memory bank, hosts a region of global memory and context memory

Рис. 3. (а) Архитектура REDEFINE, состоящая из 16 выч. узлов, расположенных в сеть 4 × 4 тороидальной конфигурации, включающая маршрутизаторы и менеджер ресурсов (RRM) для взаимодействия с хост-контроллером. (б) Составляющие вычислительного узла.

сумматора с переносом [13]. Описания схем были конвертированы из формата Verilog в СДНФ, чтобы получить соответствующие булевы полиномы. Последовательный и параллельный алгоритмы тестировались на полученных полиномах. В табл. 2 показано усредненное по 5 запускам время работы последовательного и параллельного алгоритмов на процессоре Xeon 2.8 Ghz с четыремя потоками.

Для последовательного и параллельного алгоритмов также было проведено тестирование на случайных полиномах различной сложности (экспериментах число переменных полинома варьируется от десятков до пары сотен). В табл. 3 показано усредненное по 5 запускам время работы последовательного и параллельного алгоритмов на процессоре Xeon 2.8 Ghz с четыремя потоками.

График 2а показывает ускорение, достигаемое параллельным алгоритмом по отношению к числу потоков. Размер входа фиксирован для того, чтобы

проанализировать свойства сильной масштабируемости параллельного алгоритма. Можно заметить, что ускорение снижается в увеличением сложности входного полинома. С увеличением размера входа увеличивается число вызовов к узкому месту алгоритма, каковым является упрощение полиномов, что, в свою очередь, влечет снижение быстродействия.

Для результатов экспериментов, показанных на рис. 2б, размер входа фиксирован для каждого из потоков, чтобы проанализировать свойства слабой масштабируемости параллельного алгоритма. Прирост скорости с увеличением числа потоков оказывается меньше максимально возможного линейного ускорения. Это обусловлено узкими местами алгоритма, требующими последовательных вычислений (упрощение полиномов), а также затратами на коммуникацию между потоками.

5. РЕАЛИЗАЦИЯ НА REDEFINE

В основе архитектуры REDEFINE [14] лежит набор вычислительных модулей, соединенных в сеть на чипе (см. иллюстрацию 3а).

REDEFINE – это ускоритель, который может настраиваться на выполнение конкретной вычислительной задачи посредством реконфигурации. Настройка конфигурации REDEFINE возможна на двух уровнях: посредством агрегации узлов в вычислительные ядра для выполнения высокоуровневых операций с множественными входами и выходами, а также на уровне специализированных функциональных элементов, представленных на слое аппаратных абстракций в виде расширенных наборов инструкций. В отличие от традиционных архитектур, расширенные наборы инструкций могут быть определены на пост-производственном этапе. Эта специфика выделяет REDEFINE на фоне широко распространенных многоядерных решений, предоставляя возможность настройки на приложения различных типов.

REDEFINE реализует исполнение в духе модели макропотоков данных. В ней исполнение представлено в виде иерархического графа потока данных (рис. 4), в котором вершинами являются гипероперации, а дуги представляют явное направление данных или порядок выполнения связанных гипероперацией. Гипероперация представляет собой макрооперацию с множественными входами и выходами. Гипероперация подлежит исполнению как только доступны все ее операнды, и все завимости порядка исполнения или синхронизации выполнены. Помимо арифметических, связанных с чтением/записью в память и управляющих инструкций, модель исполнения REDEFINE включает примитивы для явного обмена данными и синхронизации между гипероперациями, а также примитивы для добавления новых узлов (гипероперацией) и дуг графа в процессе исполнения. Таким образом, модель исполнения поддерживает динамический (зависимый от данных) параллелизм. В модели реализовано невытесняющее планирование гипероперацией, поэтому циклические зависимости между ними запрещены. Управляющий модуль планирует готовые гипероперации к выполнению на вычислительных узлах. Каждый узел содержит четыре вычислительных элемента, каждый из которых может выполнить ту или иную гипероперацию (рис. 2б). Обмен данными между гипероперациями является однонаправленным, т.е. только одна и та же гипероперация может инициировать и завершить обмен. Таким образом, в условиях достаточного параллелизма обмен данными может накладываться на вычисления. По сравнению с другими гибридными моделями потока данных и управления, модель REDEFINE упрощает управление ресурсами и моделью памяти, необходимыми для поддержки параллелизма любого типа.

5.1. Алгоритм факторизации на основе гипероперацией

Algorithm 5. Алгоритм декомпозиции на основе гипероперацией

Input Факторизуемый булев полином F

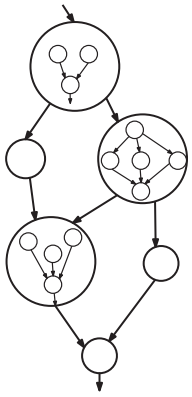


Рис. 4. Модель исполнения на уровне макропотоков данных. Исполнение представлено в виде иерархического графа потока данных, в котором вершинами являются гипероперации, а дуги представляют явное направление данных или порядок выполнения связанных гипероперацией.

Таблица 4. Параллельная факторизация случайных булевских полиномов на эмуляторе REDEFINE с использованием процессора Intel Xeon 2.8 GHz

Кол-во мономов	Последов. циклов ЦП	Многопот. циклов ЦП	Redefine циклов ЦП
30	17192×10^3	7896×10^3	6837×10^3
50	45612×10^3	14196×10^3	12320×10^3

Output F_{same} и F_{other}, которые являются факторами входного полинома F **Глобальные переменные** $\Sigma_{same}, \Sigma_{other}$

1: **Begin HyperOp**
2: Входы: A, B, переменная y
3: Вычислить $C = \frac{\partial A}{\partial y}$, $D = \frac{\partial B}{\partial y}$
4: **if** IsEqual(A,D,B,C) **then**
5: $\Sigma_{other} = \Sigma_{other} \cup \{y\}$
6: **else**
7: $\Sigma_{same} = \Sigma_{same} \cup \{y\}$
8: **end if**
9: Call Sync HyperOp
10: **End HyperOp**
11: Выбрать произвольную переменную x из F
12: Let $A = \frac{\partial F}{\partial z}$, $B = F_{z=0}$
13: Let $\Sigma_{same} = x$, $\Sigma_{other} = \emptyset$, $F_{same} = 0$, $F_{other} = 0$
14: **for each** $y \in var(F)\backslash\{x\}$ **do in parallel**
15: Породить HyperOp со входами A, B, y
16: **end for**
17: Ждать исполнения Sync HyperOp
18: Если $\Sigma_{other} = \emptyset$ то F неприводим; остановиться.
19: Вернуть полиномы F_{same} и F_{other}, являющиеся проекциями F на Σ_{same} и Σ_{other}, соответственно.

В Алгоритме 5 приведен псевдокод алгоритма факторизации в стиле реализации на языке C с гипероперациями. Фрагмент кода, соответствующий Алгоритму 5, представлен в распечатке ниже. В ней выражения вида `__CMAddr`, `__Sync`, `__kernel`, `__WriteCM`, `CMADDR` являются аннотациями, используемыми для REDEFINE. Строки 2–8 и 12–13 Алгоритма 5 соответствуют строкам 5–10 и 1–3 Алгоритма 1, соответственно.

В строках 14–16 Алгоритма 5 для каждой переменной y из F (отличной от x) параллельно порождаются гипероперации, которые устанавливают, какому из множеств Σ_{same} или Σ_{other} принадлежит y. В строках 1–10 дано определение гипероперации, которая берет на вход булевы полиномы A, B и переменную y и добавляет y в Σ_{same} или Σ_{other}. В строках 17–19 указано ожидание завершения всех гипероперации и выдача полиномов F_{same} и F_{other}.

В ПРИЛОЖЕНИИ 1 приведена реализация алгоритма факторизации на языке C с гипероперациями. Алгоритм был протестирован на эмуляторе REDEFINE, запущенном на процессоре Intel Xeon. В табл. 4 приведено время работы алгоритма факторизации на эмуляторе REDEFINE для случайных полиномов. Реализация на REDEFINE требует наименьшее число циклов ЦП.

6. ЗАКЛЮЧЕНИЕ И ДАЛЬНЕЙШАЯ РАБОТА

В статье рассмотрена проблема факторизации булевых полиномов, лежащая в основе декомпозиции булевых функций в ДНФ и реляционных данных. В связи с этой проблемой важным является построение эффективных алгоритмов факторизации. Мы рассмотрели алгоритм факторизации булевых полиномов, предложенный в [4], и предложили параллельную MIMD реализацию этого алгоритма, который использует параллелизм уровня задачи и параллелизм уровня данных для повышения быстродействия. Эксперименты с последовательным и параллельным алгоритмами факторизации на бенчмарках логического синтеза и случайных полиномах показали существенное ускорение вследствие параллелизации. Реализация параллельного алгоритма на эмуляторе REDEFINE продемонстрировало преимущество в быстродействии благодаря массивнопараллельной многоядерной архитектуре. Модель исполнения, реализуемая архитектурой REDEFINE, основана на принципах потока данных, что позволяет избежать необходимости барьерной синхронизации. Это дает повышение скорости MIMD задач (в частности, факторизации) на архитектуре REDEFINE. Указанные реализации

будут применены для вычисления недизъюнктной декомпозиции ДНФ [15] и табличных данных [6], которая опирается на решение массовой задачи факторизации булевых полиномов.

7. ПРИЛОЖЕНИЕ

Листинг1: Snippet of decomposition algorithm using Hyperops

```
__hyperOp__ void
decomp (__CMAddr selfId ,
          __Op32 a , __Op32 b ,
          __Op32 p_s ,__Op32 p_o ,
          __Op32 m, __Op32 n ,
          __Op32 i , __Op32 consumFrId )
{
 int *A=a.ptr ;
 int *B=b.ptr ;
 int * partition_same =p_s.ptr ;
 int * partition_o the r=p_o.ptr ;
 int I=i.i32;
 int n=n.i32;
 int m=m.i32;
 int J=0;
 int *C, *D;
  __CMAddr c onf r Id=consumFrId.cmAddr ;
 for ( I=0; I<n ; I++) {
   *( partition_same+J )=0;
   *( partition_o the r+J )=0;
 }
 for ( I=0; I<m; I++) {
   for ( J=0; J<n ; J++) {
     *(C+I * columns+J )=0;
     *(D+I * columns+J )=0;
   }
 }
 derivative (A,B,C, i ) ;
 derviative (A,B,D, i ) ;
 if ( I sEqual (A,D,B,C) )
    *( partition_o the r+i ) =1;
 else
   *( partition_same+i ) =1;
 __Sync( confrId , =1);
}
__kernel int
decomp_start ( int *A, int *B,
                  int * partition_same ,
```

```
                    int * partition_othe r ,
                    int N)
{
  int i =0;
  static int counter=0;
  __CMAddr decompFr ;
  __CMAddr syncFr=__CreateInst(&smd_Sync ) ;
  __WriteCM(CMADDR( syncFr , 1 5 ) ,N=1);


 for ( i = 1 ; i<N; i++) {
   decompFr=__CreateInst(&smd_decomp ) ;
   __WriteCM(CMADDR( decompFr , 0 ) ,
             ( void * ) (A) ) ;
   __WriteCM(CMADDR( decompFr , 1 ) ,
             ( void * ) (B) ) ;
   __WriteCM(CMADDR( decompFr , 2 ) ,
             ( void * ) ( partition_same ) ) ;
   __WriteCM(CMADDR( decompFr , 3 ) ,
             ( void * ) ( partition_o the r ) ) ;
   __WriteCM(CMADDR( decompFr , 4 ) , M) ;
   __WriteCM(CMADDR( decompFr , 5 ) , N) ;
   __WriteCM(CMADDR( decompFr , 6 ) , i ) ;
   __WriteCM(CMADDR( decompFr , 7 ) ,
CMADDR( syncFr , 1 5 ) ) ;
 }
 return 0 ;
}
```

СПИСОК ЛИТЕРАТУРЫ

1. *von zur Gathen J., Gerhard J.* Modern Computer Algebra. Third edition. New York, NY, USA: Cambridge University Press, 2013. P. 812.
2. *Жегалкин И.И.* Арифметизация символической логики // Математический сборник. 1928. Т. 35. № 1. С. 311–377.
3. *Muller D.E.* Application of Boolean algebra to switching circuit design and to error detection // IRE Transactions on Electronic Computers. 1954. V. EC-3. P. 6–12.
4. *Емельянов П.Г., Пономарев Д.К.* Алгоритмические вопросы конъюнктивной декомпозиции булевых формул // Программирование. 2015. Т. 41. № 3. С. 162–169.
5. *Emelyanov P., Ponomaryov D.* On tractability of disjoint AND–decomposition of boolean formulas // Proceedings of the PSI 2014: 9th Ershov Informatics Conference. V. 8974 of *Lecture Notes in Computer Science*. Springer, 2015. P. 92–101.
6. *Emelyanov P.* On two kinds of dataset decomposition // Proceedings of the 18th International Conference on Computational Science (ICCS 2018), Part II. V. 10861 of *Lecture Notes in Computer Science*. Springer, 2018. P. 171–183.
7. *Emelyanov P., Ponomaryov D.* Cartesian decomposition in data analysis // Proceedings of the Siberian Symposium on Data Science and Engineering (SSDSE 2017). 2017. P. 55–60.
8. *Shpilka A., Volkovich I.* On the relation between polynomial identity testing and finding variable disjoint factors // Proceedings of the 37th International Colloquium on Automata, Languages and Programming. Part 1 (ICALP 2010). V. 6198 of *Lecture Notes in Computer Science*. Springer, 2010. P. 408–419.
9. *Emelyanov P., Ponomaryov D.* On a polytime factorization algorithm for multilinear polynomials over F_2 // Proceedings of the 20th International Workshop on Computer Algebra in Scientific Computing (CASC

2018). V. 11077 of *Lecture Notes in Computer Science*. Springer, 2018. P. 164–176.

10. *Somenzi F.* CUDD: CU Decision Diagram package // https://github.com/ivmai/cudd. Accessed 2019-12-11.
11. *Corno F., Reorda M. S., Squillero G.* RT-level ITC'99 benchmarks and first ATPG results // IEEE Design & Test of Computers. 2000. V. 17. № 3. P. 44–53.
12. *Hansen M.C., Yalcin H., Hayes J.P.* Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering // IEEE Design Test of Computers. 1999. V. 16. № 3. P. 72–80.
13. *Fišer P., Schmidt J.* A prudent approach to benchmark collection // Proceedings of the 12th International Workshop on Boolean Problems (IWSBP'2016). 2016. P. 129–136.
14. Redefine – reconfigurable silicon core description // http://morphing.in/redefine. Accessed 2018-12-07.
15. *Ponomaryov D.* A polynomial time delta-decomposition algorithm for positive DNFs // Proceedings of the 14th International Computer Science Symposium in Russia (CSR'2019). V. 11532 of *Lecture Notes in Computer Science*. Springer, 2019. P. 325–336.