О распараллеливании метода динамического программирования для задачи о ранце

М.А. Посыпкин[1], Си Ту Тант Син[2]

***Аннотация*—Работа посвящена практическому и теоретическом исследованию параллельных вариантов метода динамического программирования для задачи о сумме подмножеств. Рассматривается стандартный табличный вариант метода, а также его модификации. Проводится теоретическое и экспериментальное сравнение эффективности предложенных алгоритмов. Для экспериментального сравнение используется процессор Intel Xeon Phi с 61 вычислительным ядром.**

***Ключевые слова*—параллельные вычисления, задачи булева программирования, динамическое программирование, дискретная оптимизация**

I. ВВЕДЕНИЕ

Задачи булева программирования являются важным классом задач дискретной оптимизации. Их исследованию посвящены многочисленные статьи и монографии [1], [2], [3], [4], [5].

В статье рассматривается наиболее просто формулируемый частный случай задачи о ранце — задача о сумме подмножеств:

$$\begin{array}{l} f(x) = \sum_{i=1}^{n} w_i x_i \to \max, \\ g(x) = \sum_{i=1}^{n} w_i x_i \le C, \\ x_i \in \{0,1\}, i \in \overline{1,n}. \end{array} \quad (1)$$

Несмотря на простоту формулировки, данная задача относится к классу NP-полных задач [6]. На практике решение данной задачи требует больших вычислительных затрат, что обосновывает целесообразность применения параллельных вычислений. Примеры реализаций стандартных вариантов метода динамического программирования можно найти в статьях [7], [8]. В работах [9], [10] изучалась теоретически максимальная сложность решения задачи о сумме подмножеств методом ветвей и границ.

Классическая табличная реализация метода динамического программирования для задачи о ранце хорошо поддается распараллеливанию, но требует хранения большой по объему таблицы, а также не учитывает того, что не все целые числа в диапазоне $1, C$ могут быть значением целевой функции. Мы предлагаем два новых алгоритма на базе вектора, первый из которых требует существенно меньших затрат оперативной памяти по сравнению с табличным вариантом. Второй алгоритм делает меньше итараций за счет того, что обрабатывает только те числа в диапазоне $1, C$, которые на самом деле являются значением целевой функции в некоторой подзадаче. Оба алгоритма распараллелены. Проводится теоретическое и экспериментальное сравнение эффективности предложенных алгоритмов. Для экспериментального сравнения используется процессор Intel Xeon Phi с 61 вычислительным ядром.

*Работа выполнена при поддержке РФФИ, проект 15-07-03102

[1]М.А. Посыпкин — заведующий отделом прикладных проблем оптимизации Вычислительного центра им. А.А. Дородницына Федерального исследовательского центра «Информатика и управление» Российской академии наук. mposypkin@gmail.com

[2]Си Ту Тант Син — аспирант кафедры Вычислительной техники Национального исследовательского университета «МИЭТ». sithuthantsin86@gmail.com

II. ПОСЛЕДОВАТЕЛЬНЫЕ ВАРИАНТЫ МЕТОДА ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Методы динамического программирования основаны на принципе оптимальности Беллмана. Для задачи о сумме подмножеств данный принцип может быть сформулирован следующим образом:

$$f^*(m,c) = \max(f^*(m-1,c), w_m + f^*(m-1, c-w_m)),$$
$$f^*(m,c) = 0 \text{ если } m = 0 \text{ или если } c \le 0,$$

где $f^*(m,c)$ при $1 \le m \le n$ и $1 \le c \le C$ — оптимальное решение подзадачи о сумме подмножеств вида

$$\begin{array}{l} f(x) = \sum_{i=1}^{m} w_i x_i \to \max, \\ g(x) = \sum_{i=1}^{m} w_i x_i \le c, \\ x_i \in \{0,1\}, i \in \overline{1,m}. \end{array} \quad (2)$$

Широко известен базовый табличный вариант динамического программирования для задачи о ранце. В этом варианте строится таблица, в которой строки соответствуют переменным, а столбцы — возможным целочисленным значениям целевой функции. Добавляются еще вспомогательные столбец для нулевого значения целевой функции $c = 0$, и строка для нулевой переменной $i = 0$, не имеющие содержательного смысла, но используемые в вычислениях. Таким образом, в таблице $n + 1$ строк и $C+1$ столбов. На первой фазе табличный алгоритм динамического программирования (TabDP) последовательно

заполняет строки таблицы в направлении слева-направо:

```
1  for c = 0 to C do
2  |  g(0, c) := 0
3  end
4  for i = 0 to n do
5  |  g(i, 0) := 0
6  end
7  for i = 1 to n do
8  |  for c = 0 to C do
9  |  |  if c < w_i then
10 |  |  |  g(i, c) := g(i − 1, c)
11 |  |  else
12 |  |  |  g(i, c) := max(g(i−1, c), g(i−1, c−w_i)+w_i)
13 |  |  end
14 |  end
15 end
```

Algorithm 1: TabDP

После завершения алгоритма оптимальное значение целевой функции находится в элементе таблицы $g(n, C)$. Вторая фаза метода динамического программирования — алгоритм BackTabDP, восстанавливает вектор x^* по таблице g.

```
1  c := C
2  for i = n downto 1 do
3  |  if g(i, c) = g(i − 1, c) then
4  |  |  x*_i := 0
5  |  else
6  |  |  x*_i := 1
7  |  |  c := c − w_i
8  |  end
9  end
```

Algorithm 2: BackTabDP

Алгоритм обратного хода имеет линейную сложность $O(n)$. Основные вычислительные затраты приходятся на алгоритм TabDP. Несложно показать, что число итераций алгоритма TabDP имеет порядок $O(nC)$, а объем памяти, требуемый для хранения информации также пропорционален $O(nC)$. В случае, если C достаточно велико указанные величины могут превысить доступные ресурсы. Распараллеливание может снизить время выполнения, но при этом большой объем памяти, занимаемый таблицей по-прежнему может препятствовать эффективному распараллеливанию.

Так как циклы в строках 1-6 и 8-14 алгоритма TabDP не содержат зависимостей по данным между итерациями, то они могут быть подвергнуты распараллеливанию:

```
1  par forall c ∈ 1, C do
2  |  g(0, c) := 0
3  end
4  par forall i ∈ 1, n do
5  |  g(i, 0) := 0
6  end
7  for i = 1 to n do
8  |  par forall c ∈ 1, C do
9  |  |  if c < w_i then
10 |  |  |  g(i, c) := g(i − 1, c)
11 |  |  else
12 |  |  |  g(i, c) := max(g(i−1, c), g(i−1, c−w_i)+w_i)
13 |  |  end
14 |  end
15 end
```

Algorithm 3: ParTabDP

Рассмотренный табличный вариант метода динамического программирования может быть реализован более эффективно. Для хранения промежуточных результатов достаточно вектора h длиной $C + 1$. Изначально все элементы вектора h равны 0. На первой фазе алгоритм последовательно перебирает индексы переменных от 1 до n. На последующих шагах элементу вектора h с индексом c присваивается i, если в результате присваивания переменной x_i значения 1 получено значение C.

```
1  h(0) := 1
2  for c = 1 to C do
3  |  h(c) := 0
4  end
5  i := 1
6  while i ≤ n and h(C) = 0 do
7  |  for c = C downto w_i do
8  |  |  if h(c) = 0 and h(c − w_i) ≠ 0 then
9  |  |  |  h(c) := i
10 |  |  end
11 |  end
12 |  i := i + 1
13 end
```

Algorithm 4: VectorDP

Алгоритм $VectorDP$ завершает свою работу, как только перебраны все переменные и значение i становится равным n или выполнено условие $h(C) \neq 0$. Максимальным значением целевой функции является $f^* = \max\{c : h(c) \neq 0\}$. Для восстановления оптимального решения x^* применим алгоритм $BackVectorDP$.

```
1  for i := 1 to n do
2  |  x*(c) := 0
3  end
4  c := f*
5  while c ≠ 0 do
6  |  x*_{h(c)} := 1
7  |  c := c − w_{h(c)}
8  end
```

Algorithm 5: BackVectorDP

Вычислительная сложность алгоритма VectorDP составляет $O(nC)$, что совпадает с аналогичным показателем для TabDP. При этом требования к ресурсам памяти сократились с $O(nC)$ требуется до $O(C)$. В отличие от табличного варианта, на различных итерациях цикла в строках 7-13 алгоритма VectorDP читаются и записываются элементы одного и того же массива h: на итерации c читается элемент $h(c - w_i)$, который записывается на итерации $c - w_i$. Поэтому в параллельном варианте

потребуется дополнительный массив h':

```
1  h(0) := 1
2  for c = 1 to C do
3  |  h(c) := 0
4  end
5  i := 1
6  while i ≤ n and h(C) = 0 do
7  |  par forall c ∈ 1,C do
8  |  |  if c ≥ w_i and h(c) = 0 and h(c − w_i) ≠ 0 then
9  |  |  |  h'(c) := i
10 |  |  else
11 |  |  |  h'(c) := h(c)
12 |  |  end
13 |  end
14 |  h := h'
15 |  i := i + 1
16 end
```

Algorithm 6: ParVectorDP

В представленном варианте не возникает конфликта из-за одновременного чтения и записи одних и тех же адресов памяти, т.к читаются элементы одного (h), а записываются элементы другого массива (h'). После параллельного цикла производится копирование массива h' в массив h. В языках, поддерживающих указатели (например C ++), данная операция может быть эффективно без копирования элементов массивов с помощью обмена указателями h и h'.

Наконец рассмотрим третий возможный вариант реализации метода динамического программирования, основанный на наблюдении, что фактически в порождении новых элементов участвуют только ненулевые элементы вектора h, поэтому именно их требуется хранить и обрабатывать. Для этого дополнительно будем хранить массив ν индексов ненулевых элементов в массиве h.

```
1  ν(1) := 0
2  b := 1
3  h(0) := 1
4  for c = 1 to C do
5  |  h(c) := 0
6  end
7  i := 1
8  while i ≤ n and h(C) = 0 do
9  |  d := b
10 |  for j = 1 to b do
11 |  |  c := ν(j)
12 |  |  c' := c + w_i
13 |  |  if c' ≤ C and h(c') = 0 then
14 |  |  |  h(c') := i
15 |  |  |  d := d + 1
16 |  |  |  ν(d) := c'
17 |  |  end
18 |  end
19 |  b := d
20 |  i := i + 1
21 end
```

Algorithm 7: SmartDP

В алгоритме SmartDP ненулевые элементы массива h хранятся в массиве ν, а их число в переменной b. При добавлении нового элемента в массив ν в строках 14-16 используется вспомогательная переменная d.

Восстановление решения в алгоритме SmartDP ничем не отличается от такового у VectorDP. Верхняя оценка вычислительной сложности остается той же самой и составляет $O(nC)$. Но при решении конкретных задач фактическое число выполненных операций может быть ниже, т.к. просматривается не весь массив h, а только элементы с индексами, хранящимися в списке ν.

Распараллеливание алгоритма SmartDP сложнее, чем TabDP и VectorDP по двум причинам. Во-первых, число итераций этого цикла меняется и при малых i может быть весьма невелико. В такой ситуации затраты на организацию параллельного выполнения могут превысить выигрыш от распараллеливания. Во-вторых, в цикле в строках 10-17 происходит обновление списка ν, что нельзя делать одновременно в нескольких параллельных потоках. Поэтому этой действие необходимо сделать критической секцией, т.е. участком кода, который не может выполняться одновременно несколькими потоками:

```
1  NZ = {0}
2  h(0) := 1
3  for c = 1 to C do
4  |  h(c) := 0
5  end
6  i := 1
7  while i ≤ n and h(C) = 0 do
8  |  d := b
9  |  par forall j ∈ 1,b do
10 |  |  c := ν(j)
11 |  |  c' := c + w_i
12 |  |  if c' ≤ C and h(c') = 0 then
13 |  |  |  h(c') := i
14 |  |  |  critical
15 |  |  |  |  d := d + 1
16 |  |  |  |  ν(d) := c'
17 |  |  |  end
18 |  |  end
19 |  end
20 |  b := d
21 |  i := i + 1
22 end
```

Algorithm 8: ParSmartDP

III. ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ

Для оценки эффективности предложенных алгоритмов была проведена серия вычислительных экспериментов. Были сгенерированы приведенные ниже серии тестовых задач [4]. Запись $w_i \in \rho(a,b)$ означает, что w_i — последовательность равномерно-распределенных псевдо-случайных чисел в интервале $[a,b]$.

1. **pthree**: $w_i \in \rho(1,10^3)$, $C = \lfloor n\frac{10^3}{4} \rfloor$;
2. **psix**: $w_i \in \rho(1,10^6)$, $C = \lfloor n\frac{10^6}{4} \rfloor$;
3. **evenodd**: $w_i \in \rho(1,10^3)$, w_i — четное, $C = 2\lfloor n\frac{10^3}{8} \rfloor + 1$;
4. **avis**: $w_i = n(n+1)+j$, $C = n(n+1)\lfloor (n-1)/2 \rfloor + n(n-1)/2$;
5. **somatoth**: $w_1 \in \rho(1,n), w_2 \in \rho(1,n)$, w_1 и w_2 — взаимнопросты, $c/w_1 \le n/2$, $c/w_2 \le n/2$, $w_i = w_1\lceil \frac{i}{2} \rceil$ для четных i, $w_i = w_2\lceil \frac{i}{2} \rceil$ для нечетных i, $3 \le i \le n$.

Первая серия проводилась на обычном персональном компьютере Intel Core i5, 2.7 GHz с целью анализа эффективности последовательных вариантов алгоритмов VectorDP и SmartDP. Сравнивалось суммарное число итераций внутреннего цикла прямого хода. Как следует

Таблица I: Сравнение числа итераций алгоритмов VectorDP и SmartDP

задача	VectorDP	SmartDP	отношение
pthree	124000290	60515483	2.04
psix	130499447272	64471378820	2.02
evenodd	249498230	94050604	2.65
avis	498997000500	28573302375	17.46
somatoth	5780984	10978055	0.53

Таблица II: Время работы алгоритмов VectorDP и ParVectorDP в миллесекундах.

задача	время посл.	время паралл.	ускорение
pthree	2301	287	8
psix	2409179	15163	158.9
evenodd	5153	308	16.7
avis	10004448	55171	181.3
somatoth	894	295	3

из таблицы I, в большинстве случаев число итераций SmartDP меньше, чем у VectorDP. Для серий pthree, psix и evenodd число итераций SmartDP меньше числа итераций VectorDP в среднем в 2-2.5 раза. Для серии avis выигрыш составляет примерно 17 раз. В то же время, поскольку цикл алгоритма VectorDP проще и эффективнее, то даже при меньшем числе итераций время работы SmartDP нередко больше, чем VectorDP. В серии somatoth число итераций цикла SmartDP больше, чем VectoDP, что объясняется тем, что диапазон изменения итератора внутреннего цикла алгоритма VectorDP от w_i до C, что при достаточно больших w_i существенно меньше, чем от 1 до C.

Вторая серия экспериментов была направлена на оценку эффективности распараллеливания. Эксперименты проводились на платформе Intel Xeon Phi 7120P с 16GB оперативной памяти и 61 ядром с тактовой частотой 1.238 GHz. Каждое ядро может выполнять до четырех потоков в режим гипертрейдинга, создавая тем самым потенциальную возможность для ускорения в 240 раз. В таблице II приведены значения времен работы алгоритмов VectorDP и ParVectorDP в миллисекундах. Аналогичные данные для алгоритмов SmartDP и ParSmartDP даны в таблице III. Можно видеть, что алгоритм VectorDP существенно эффективнее распараллеливается по сравнению со SmartDP. Поэтому для серии avis алгоритм ParSmartDP проигрывает ParVectorDP, несмотря на то, что последовательный вариант SmartDP существенно эффективнее.

В таблице IV представлены сравнительные времена выполнения последовательного и параллельного вариантов табличного метода динамического программирования. Прочерки в строках psix и avis означают, что для работы программы не хватило доступного объема оперативной памяти, что обусловлено необходимостью хранения полной таблицы промежуточных значений. Для

Таблица III: Время работы алгоритмов SmartDP и ParSmartDP

задача	время посл.	время паралл.	ускорение
pthree	1258	657	1.9
psix	10976928	378786	28.9
evenodd	1929	501	3.8
avis	2713134	100012	27.1
somatoth	1438	494	2.9

Таблица IV: Время работы алгоритмов TabDP и ParTabDP в миллесекундах.

задача	время посл.	время паралл.	ускорение
pthree	10798	2465	4.38
psix	-	-	-
evenodd	10787	2419	4.46
avis	-	-	-
somatoth	3831	1121	3.42

других примеров можно отметить относительно небольшое ускорение. Таким образом можно заключить, что алгоритм VectorDP существенно эффективнее алгоритма TabDP как в стандартном последовательном варианте, так и при распараллеливании.

IV. ЗАКЛЮЧЕНИЕ

В работе рассмотрены различные варианты реализации метода динамического программирования для задачи о сумме подмножеств: стандартный табличный метод (TabDP), метод, основанный на одномерном векторе (VectorDP) и метод, учитывающий реальное число различных значений суммы элементов (SmartDP). Для указанных алгоритмов разработаны параллельные варианты, произведено их экспериментальное сравнение на современном многоядерном процессоре Intel Xeon Phi. Результаты экспериментов показали, что наиболее эффективным является параллельный вариант метода, основанного на одномерном векторе (ParVectorDP).

Список литературы

[1] Lazarev Alexander A, Werner Frank. A graphical realization of the dynamic programming method for solving np-hard combinatorial problems // Computers & Mathematics with Applications. — 2009. — Vol. 58, no. 4. — P. 619–631.

[2] Kolpakov Roman Maksimovich, Posypkin Mikhail Anatol'evich. Upper and lower bounds for the complexity of the branch and bound method for the knapsack problem // Discrete Mathematics and Applications. — 2010. — Vol. 20, no. 1. — P. 95–112.

[3] Kolpakov RM, Posypkin MA, Sin Si Tu Tant. Complexity of solving the subset sum problem with the branch-and-bound method with domination and cardinality filtering // Automation and Remote Control. — 2017. — Vol. 78, no. 3. — P. 463–474.

[4] Kellerer Hans, Pferschy Ulrich, Pisinger David. Knapsack problems. 2004.

[5] Martello Silvano, Toth Paolo. Knapsack problems: algorithms and computer implementations. — John Wiley & Sons, Inc., 1990.

[6] Gary Michael R, Johnson David S. Computers and intractability: A guide to the theory of np-completeness. — 1979.

[7] El Baz Didier, Elkihel Moussa. Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0–1 knapsack problem // Journal of Parallel and Distributed Computing. — 2005. — Vol. 65, no. 1. — P. 74–84.

[8] Tan Guangming, Sun Ninghui, Gao Guang R. A parallel dynamic programming algorithm on a multi-core architecture // Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures / ACM. — 2007. — P. 135–144.

[9] Kolpakov Roman Maksimovich, Posypkin Mikhail Anatolevich, Sigal I Kh. On a lower bound on the computational complexity of a parallel implementation of the branch-and-bound method // Automation and Remote Control. — 2010. — Vol. 71, no. 10. — P. 2152–2161.

[10] Kolpakov Roman, Posypkin Mikhail. The lower bound on complexity of parallel branch-and-bound algorithm for subset sum problem // AIP Conference Proceedings / AIP Publishing. — Vol. 1776. — 2016. — P. 050008.

On the parallelization of dynamic programming method for knapsack problem

Mikhail A. Posypkin, Si Thu Thant Sin

***Abstract*—The work is dedicated to the practical and theoretical study of parallel variants of the dynamic programming method for the subset sum problem. The standard table based variant of the method is considered as well as its modifications. We perform theoretical and experimental comparison of the effectiveness of the proposed algorithms. For experimental comparison we use an Intel Xeon Phi with 61 computational kernel.**

***Keywords*—parallel computing, boolean programming problem, dynamic programming, discrete optimization**