УДК 510.57 DOI: 10.14529/mmp200305

# РАСПАРАЛЛЕЛИВАНИЕ РЕКУРРЕНТНЫХ ЦИКЛОВ С ПРЕДВАРИТЕЛЬНЫМ ВЫЧИСЛЕНИЕМ СУПЕРПОЗИЦИЙ

***О.Б. Штейнберг,*** Южный федеральный университет, г. Ростов-на-Дону, Российская Федерация

Как правило, именно циклы являются участками кода, вычисление которых занимает много времени. Поэтому, именно на них направляются особые усилия при ускорении программ, в частности, через распараллеливание.

В статье описывается алгоритм распараллеливания циклов, вычисляющих элементы рекуррентно заданной последовательности. Рекуррентные циклы, рассматриваемые в статье, непосредственно распараллелены быть не могут. С помощью вспомогательных преобразований иногда их можно привести к циклам, допускающим параллельное выполнение. Ранее автором статьи был опубликован другой алгоритм распараллеливания циклов, вычисляющих элементы рекурсивно заданной последовательности. В современных процессорах время выполнения арифметических операций оказывается на порядок меньше, чем считывание аргументов этих операций из оперативной памяти. В данной статье приводятся оценки сложности по обращению к памяти, для описываемого алгоритма. Представленный в статье параллельный алгоритм оказывается более эффективным по обращениям к памяти, чем алгоритм, описанный автором ранее.

*Ключевые слова: рекуррентные циклы; численные методы; параллельные вычисления; преобразования программ; рекуррентные последовательности.*

## Введение

Данная статья относится к теме преобразования программ [1–3]. Она посвящена новому преобразованию, способствующему распараллеливанию рекуррентных циклов. Некоторые работы, посвященные этой теме, имеют дело с циклами, вычисляющими сумму последовательности. Также встречаются работы посвященные распараллеливанию рекуррентных циклов с условными операторами [4]. Данная статья рассматривает новый алгоритм распараллеливания рекуррентных циклов, вычисляющих сами элементы рекуррентной последовательности.

Ранее автором был предложен другой алгоритм распараллеливания рекуррентных циклов [5–7], который является развитием алгоритма описанного в работе [8]. В работе [6] для него были найдены условия устойчивости. Описываемый в данной работе алгоритм отличается от предыдущего и по вычислительной сложности, и по обращениям к памяти, и по работе с памятью в процессе параллельного вычисления. В частности, при применении предыдущего алгоритма на архитектуре современных CPU возникали синхронизации при обращении к кэшу, замедляющие вычисления.

Циклы, как правило, являются самыми долго считаемыми участками кода в программах. Ввиду этого, именно на них направляются особые усилия для ускорения, в частности, через распараллеливание. Рекуррентные циклы непосредственно распараллелены быть не могут. Иногда, с помощью вспомогательных преобразований их можно привести к циклам, допускающим параллельное выполнение. Теме распараллеливающих преобразований программ посвящены работы [1, 2]. Алгоритм, опи-

санный в данной работе, применим, в частности, к циклам с аффинной, дробно-аффинной или матричной рекуррентной зависимостью. Последовательности с аффинными рекуррентными зависимостями приводятся в [9].

Следует отметить, что к выполнению рекуррентных циклов сводится метод прогонки решения системы аффинных алгебраических уравнений с трехдиагональной матрицей, который используется при численных методах решения дифференциальных уравнений [10].

В современных процессорах скорость выполнения арифметических операций оказывается на порядок быстрее, чем скорость считывания аргументов этих операций из оперативной памяти [11]. Это явление объясняет, например, почему известный алгоритм Штрассена перемножения матриц оказывается неэффективным на современных процессорах. В данной статье приводятся оценки сложности алгоритма по обращениям к памяти. По этому показателю представленный в данной статье алгоритм превосходит описанный ранее алгоритм [5–7] и алгоритм на котором он был основан [8].

Описываемое преобразование может быть реализовано, для автоматического выполнения, в оптимизирующих распараллеливающих компиляторах (например, [12–14]).

## 1. Алгоритм распараллеливания рекуррентных циклов с предварительным вычислением суперпозиций

Обозначим через $P$ количество вычислительных устройств (ВУ). Задача состоит в том, чтобы за как можно меньшее время вычислить последовательность $X_i(i = (1, 2, \ldots, N))$ через отображения $G_i$ по рекуррентной формуле $X_i = G_i(X_{i-1})$, при условии, что в наличии имеется $P$ ВУ.

Таким образом, в качестве распараллеливаемого будет рассматриваться программный цикл вида:

```
for(i = 1; i <= N; i++)
  X[i] = G[i](X[i-1]).
```

Далее будем считать, что $P > 1$. Исходя из постановки задачи, $X_i$ нельзя вычислить, не зная $X_{i-1}$, а значит, несколько ВУ не смогут одновременно вычислять различные элементы искомого массива, если не выполнить каких-нибудь дополнительных действий. Как раз такими действиями может являться вычисление суперпозиций $G_i \circ G_{i+1}$.

Рассмотрим пример, демонстрирующий то, каким образом предварительно вычисленные суперпозиции способны помочь распараллеливанию.

**Пример 1.** Цикл, содержащий аффинную рекуррентную зависимость:

```
for(i = 1; i <= N; i++)
  X[i] = A[i]*X[i-1]+B[i].
```

В этом цикле никакие две итерации нельзя вычислить одновременно.

Заметим что $X_1 = G_1(X_0) = A_1 * X_0 + B_1$, а $X_2 = G_2(X_1) = A_2 * X_1 + B_2$. Следовательно, $X_2 = G_2(G_1(X_0)) = A_2 * (A_1 * X_0 + B_1) + B_2 = A_2 * A_1 * X_0 + A_2 * B_1 + B_2$.

Таким образом, если предварительно вычислить $AA_1 = A_2 * A_1$ и $BB_1 = A_2 * B_1$, то можно будет одновременно получить $X_1 = A_1 * X_0 + B_1$ и $X_2 = AA_1 * X_0 + BB_1$.

### Идея алгоритма

Идея алгоритма состоит в том, чтобы параллельно вычислять все элементы рекуррентной последовательности, предварительно вычислив несколько ее элементов с помощью заранее полученных суперпозиций.

Рассмотрим идею алгоритма на простом примере. Предположим, что необходимо вычислить 100 элементов массива $X$ (значит N=100) и для вычисления используется 2 ВУ ($P = 2$). Изначально дано $X_0$. Исходя из постановки задачи, на первом шаге можно вычислить $X_1 = G_1(X_0)$, вторым шагом можно вычислить $X_2 = G_2(X_1)$ и т.д. Теперь, предположим, что изначально имелось бы не только $X_0$, но и $X_{50}$. В таком случае, на первом шаге вторым ВУ можно было бы вычислять $X_{51}$ одновременно с $X_1$, на втором шаге $X_{52}$ параллельно с $X_2$ и т.д. Идея алгоритма состоит в том, чтобы предварительно вычислить элемент $X_{50}$, при помощи суперпозиции отображений $G_1, G_2, \ldots, G_{50}$.

### Описание алгоритма

Алгоритм состоит из трех этапов:

1. Вычисление $(P-1)$-ого отображения $Q_i = G_{j-1+N/P} \circ \cdots \circ G_{j+1} \circ G_j$ (где $i = 1, 2, \ldots, P-1; j = ((i-1)*N/P)+1$) через суперпозиции $N/P$ исходных отображений (рис. 1).

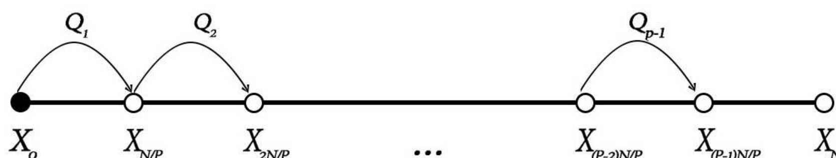


**Рис. 1**. Иллюстрация первого этапа алгоритма

Псевдокод, соответствующий первому этапу алгоритма:

```
for(i = 0; i < P-1; i++) {
  k = ((i+1)*N)/P;
  Q[i] = G[k];
  for(j = 1; j < N/P; j++)
    Q[i] = Superposition(Q[i], G[k-j])}.
```

2. Вычисление элементов $X_{N/P}, X_{2*N/P}, \ldots, X_{(P-1)*N/P}$ посредством суперпозиций, вычисленных на первом этапе (рис. 2).

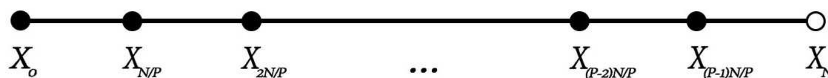


**Рис. 2**. Иллюстрация второго этапа алгоритма

Псевдокод, соответствующий второму этапу алгоритма:

```
for(i = 0; i < P-1; i++)
  X[((i+1)*N)/P] = Q[i](X[(i*N)/P]).
```

3. Вычисление элементов $X_i$ всеми $P$ ВУ начиная с $X_0$ и элементов вычисленных на втором этапе алгоритма (рис. 3).

Псевдокод, соответствующий третьему этапу алгоритма:

```
for(i = 0; i < P; i++)
for(j = 0; j < N/P - 1; j++)
  X[(i*N)/P + j+1] = G[j](X[(i*N)/P + j]).
```

Заметим, что внешний цикл допускает параллельное выполнение.

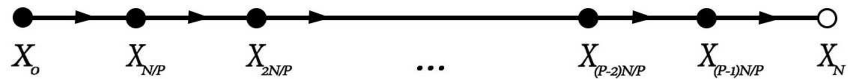


**Рис. 3**. Иллюстрация третьего этапа алгоритма

Рассмотрим применение алгоритма к циклу с аффинной рекуррентной зависимостью при условии, что распараллеливание ведется на 2 ВУ.

**Пример 2.** Применим алгоритм к циклу с аффинной рекуррентной зависимостью (см. пример 1). На первом этапе будет получена суперпозиция, позволяющая вычислить элемент $X_{N/2}$ через $X_0$:

```
AA = A[N/2]*A[N/2-1];
BB = A[N/2]*B[N/2-1]+B[N/2];
for(j = N/2-2; 0 < j; j--) {
  BB = AA*B[j]+BB;
  AA = AA*A[j]}.
```

Вторым этапом, с помощью суперпозиции, полученной на первом этапе, вычисляется элемент $X_{N/2}$:

```
X[N/2] = AA*X[0]+BB.
```

На третьем этапе оба ВУ ведут вычисления своей половины элементов $X_i$:

```
for(j = 1; j < N/2; j++)//вычисления для первого ВУ
  X[j] = A[j]*X[j-1]+B[j];
for(j = N/2+1; j <= N; j++)//вычисления для второго ВУ
  X[j] = A[j]*X[j-1]+B[j].
```

Стоит отметить, что вычислять суперпозиции на первом этапе, можно используя принцип сдваивания [4].

**Пример 3.** Рассмотрим два варианта получения суперпозиции, позволяющей вычислить $X_8$ через $X_0$:

1) $G_8 \circ (G_7 \circ (G_6 \circ (G_5 \circ (G_4 \circ (G_3 \circ (G_2 \circ G_1))))))$,

2) $(((G_8 \circ G_7) \circ (G_6 \circ G_5)) \circ ((G_4 \circ G_3) \circ (G_2 \circ G_1)))$.

В первом случае суперпозиция вычисляется *последовательно*, а во втором с использованием *принципа сдваивания.* В обоих случаях для вычисления суперпозиции требуется 7 операций, но при использовании принципа сдваивания эти операции можно распараллелить. Минусом использования принципа сдваиивания является необходимость хранения данных, описывающих промежуточные суперпозиции.

## 2. Анализ сложности алгоритма

Проведем анализ сложности алгоритма для нескольких частных случаев отображений $G_i$.

**Пример 4.** В цикле примера 1 отображением $G_i$ является $G_i(X_{i-1}) = A_i * X_{i-1} + B_i$.

**Пример 5.** Цикл, содержащий дробно-аффинную рекуррентную зависимость:

```
for(i = 1; i <= N; i++)
  X[i] = (A[i]*X[i-1]+B[i]) / (C[i]*X[i-1]+D[i]).
```

В данном примере отображением $G_i$ является $G_i(X_{i-1}) = \dfrac{A_i * X_{i-1} + B_i}{C_i * X_{i-1} + D_i}$.

Заметим, что суперпозиция аффинных (дробно-аффинных) отображений будет аффинным (дробно-аффинным) отображением. Введем обозначения: $T_{superposition}$ – время, вычисления суперпозиции; $T_{function}$ – время затрачиваемое на вычисление элемента последовательности с помощью отображения; $T_{read}$ – время, затрачиваемое на чтение из одной ячейки памяти; $T_{write}$ – время, затрачиваемое на запись в одну ячейку памяти; $Data$ – количество данных, используемых для описания отображения. Заметим что, отображение описывающее аффинную рекуррентную зависимость примера 1 состоит из двух коэффициентов $A$ и $B$, что дает $Data = 2$, а дробно-аффинное примера 5 дает $Data = 4$.

Сложность алгоритма рассматривается с точки зрения вычислений и с точки зрения обращений к памяти.

### 2.1. Последовательное выполнение рекуррентного цикла

При последовательном выполнении цикла вычисляется $N$ элементов массива $X$ с использованием отображений $G_i$. Итого, вычислительная сложность алгоритма составляет $N * T_{function}$. Сложность по обращениям к памяти последовательного алгоритма состоит из записи $N$ элементов $X$ и чтения, используемых при вычислениях, данных. Итого, сложность алгоритма по обращениям к памяти равна $N * T_{write} + N * Data * T_{read}$. Исходя из вышесказанного общая сложность алгоритма равна $N * T_{function} + N * T_{write} + N * Data * T_{read}$.

### 2.2. Выполнение рекуррентного цикла с использованием рассматриваемого алгоритма

На первом этапе находится $P - 1$ отображение $Q_i$ (где $i = 1, 2, \ldots, P - 1$). При вычислении каждого отображения применяется $N/P - 1$ суперпозиция. То есть $((N/P - 1) * (P - 1)) * T_{superposition}$. Все суперпозиции можно вычислять на $P - 1$ ВУ параллельно. Итого, время выполнения получается равным $(N/P - 1) * T_{superposition}$.

На втором этапе вычисляется всего $P - 1$ элемент массива $X$. Эти элементы вычисляются за время $(P-1)*T_{function}$. Отметим что, вычисление этих же элементов последовательным алгоритмом занимает такое же время.

Третий этап алгоритма состоит в параллельном вычислении оставшихся элементов массива $X$. Это занимает $((N-(P-1))/P)*T_{function}$. Итого, вычислительная сложность составляет: $(N/P-1)*T_{superposition}+(P-1)*T_{function}+((N-(P-1))/P)*T_{function}$. Рассмотрим сложность алгоритма по обращениям к памяти.

На первом этапе алгоритма, считываются данные, необходимые для вычисления суперпозиций. Это занимает $(P-1)*(N/P)*Data*T_{read}$. Если суперпозиции вычислять последовательно, то данные относящиеся к промежуточным суперпозициям (или промежуточные суперпозиции) могут быть перезаписаны. Откуда получается, что в итоге первого этапа потребуется записать лишь данные, соответствующие $(P-1)$-ому отображению $Q_i$ (где $i = 1, 2, \ldots, P-1$), т.е. $(P-1)*Data*T_{write}$. На втором и третьем этапе производятся вычисления аналогичные последовательному вычислению цикла, соответственно, сложность по обращениям к памяти совпадает со сложностью при последовательном выполнении $N*T_{write}+N*Data*T_{read}$. Итого, время выполнения алгоритма равно $(P-1)*(N/P)*Data*T_{read}+(P-1)*Data*T_{write}+N*T_{write}+N*Data*Tread$.

Заметим, что оценка времени написана для общего случая и должна уточняться исходя из конкретной вычислительной архитектуры. Так, например, если вычисления производятся на архитектуре с кэш-памятью, то данные, считываемые для вычисления суперпозиций на первом этапе, будут храниться в кэш-памяти и не будут повторно считываться из медленной оперативной памяти на третьем этапе. Кроме того, элементы могут считываться не по одному, а целыми кэш-линейками, да притом еще и несколькими каналами чтения.

## Заключение

В данной работе представлен новый алгоритм параллельного выполнения рекуррентных циклов. Приведена его общая оценка вычислительной сложности и сложности по обращениям к памяти, для случаев определенного типа рекуррентных зависимостей. Сложность по обращениям к памяти лучше известного параллельного алгоритма [8], в котором много промежуточных суперпозиций должно храниться в памяти.

Проводилось тестирование быстродействия алгоритма на двух компьютерах:

1) процессор Intel CoreTM i3-3227U 1.9GHz, оперативная память 4GB DDR3,

2) процессор Intel Core i7-3820 3.60GHz, оперативная память 8GB DDR3.

Алгоритм применялся к циклу с аффинной рекуррентной зависимостью, вычисляющему 67108864 элемента. Распараллеливания проводилось с использованием OpenMP 4.0. Результаты тестирования приведены в таблице.

**Таблица**

Результаты тестирования быстродействия алгоритма в случае аффинной рекуррентной зависимости

| Процессор | Последовательное выполнение (сек) | Выполнение 2-мя потоками (сек) | Выполнение 4-мя потоками (сек) |
|---|---|---|---|
| Intel i3 | 0,484124 | 0,310807 | 0,262585 |
| Intel i7 | 0,383944 | 0,255872 | 0,210152 |

*Работа поддержана грантом Правительства РФ № 075-15-2019-1928.*

## Литература


1. Allen, R. Optimizing Compilers for Modern Architectures / R. Allen, K. Kennedy. – San Francisco; San Diego; New York; Boston; London; Sidney; Tokyo: Morgan Kaufmann Publishers, 2002.

2. Wolfe, M. High Performance Compilers for Parallel Computing / M. Wolfe. – Redwood City: Addison-Wesley Publishing Company, 1996.

3. Steinberg, O.B. Circular Shift of Loop Body-Programme Transformation, Promoting Parallelism / O.B. Steinberg // Вестник ЮУрГУ. Серия: Математическое моделирование и программирование. – 2017. – Т. 10, № 3. – P. 120–132.

4. Штейнберг, Б.Я. Распараллеливание рекуррентных циклов с условными операторами / Б.Я. Штейнберг // Автоматика и телемеханика. – 1995. – № 6. – С. 176–184.

5. Штейнберг, О.Б. Распараллеливание рекуррентных циклов с нерегулярным вычислением суперпозиций / О.Б. Штейнберг // Известия ВУЗов. Северокавказский регион. Естественные науки. – 2009. – № 2. – С. 18–21.

6. Штейнберг, О.Б. Автоматическое распараллеливание рекуррентных циклов с проверкой устойчивости / О.Б. Штейнберг, С.Е. Суховерхов // Информационные технологии. – 2010. – № 1. – С. 40–45.

7. Штейнберг, О.Б. Распараллеливание циклов, допускающих рекуррентные зависимости: дис. ... канд. физ.-мат. наук / О.Б. Штейнберг. – М., 2014.

8. Штейнберг, Б.Я. Математические методы распараллеливания рекуррентных программных циклов на суперкомпьютеры с параллельной памятью / Б.Я. Штейнберг. – Ростов-на-Дону: Изд-во Ростовского ун-та, 2004.

9. Graham, R.L. Concrete Mathematics: a Foundation for Computer Science / R.L. Graham, D.E. Knuth, O. Patashnik. – New York: Addison-Welsey, 1994.

10. Самарский, А.А. Введение в численные методы / А.А. Самарский. – М.: Наука, 1997.

11. Graham, S.L. Getting Up to Speed: the Future of Supercomputing / S.L. Graham, M. Snir, C.A. Patterson. – Washington: National Academies Press, 2005.

12. Оптимизирующая распараллеливающая система. – URL: http://ops.rsu.ru (дата обращения 1.07.2020).

13. Оптимизирующий компилятор Pluto. – URL: http://pluto-compiler.sourcefofge.net (дата обращения 1.07.2020).

14. Компилятор ROSE. – URL: http://rosecompiler.org/ (дата обращения 1.07.2020).


Олег Борисович Штейнберг, кандидат физико-математических наук, старший научный сотрудник, лаборатория «Вычислительная механика», Институт математики, механики и компьютерных наук им. И.И. Воровича, Южный федеральный университет (г. Ростов-на-Дону, Российская Федерация), olegsteinb@gmail.com.

*Поступила в редакцию 7 марта 2020 г.*

MSC 68W10 DOI: 10.14529/mmp200305

**PARALLELIZATION OF RECURRENT LOOPS DUE TO THE PRELIMINARY COMPUTATION OF SUPERPOSITIONS**

***O.B. Steinberg,*** South Federal University, Rostov-on-Don, Russian Federation, olegsteinb@gmail.com

As a rule, sections of code that take a lot of time to compute are loops. Therefore, it is precisely on them that special efforts are directed when accelerating programs, in particular, through parallelization.

The article describes the parallelization algorithm for loops calculating the elements of a recursively given sequence. The recurrent loops considered in the article cannot be directly parallelized. With the help of auxiliary transformations, they can sometimes be reduced to loops that allow parallel execution. Earlier, the author of the article published another algorithm for parallelizing loops that calculate the elements of a recursively given sequence.

In modern processors, the execution time of arithmetic operations is an order of magnitude faster than reading the arguments of these operations from RAM. This article provides estimates of the complexity of accessing memory for the described algorithm. The parallel algorithm presented in the article is more efficient in accessing memory than the algorithm described by the author earlier.

*Keywords: recurrent loops; numerical methods; parallel computation; program transformations; recurrent sequences.*

## References

1. Allen R., Kennedy K. *Optimizing Compilers for Modern Architectures.* San Francisco, San Diego, New York, Boston, London, Sidney, Tokyo, Morgan Kaufmann Publishers, 2002.
2. Wolfe M. *High Performance Compilers for Parallel Computing.* Redwood City, Addison-Wesley Publishing Company, 1996.
3. Steinberg O.B. Circular Shift of Loop Body-Programme Transformation, Promoting Parallelism. *Bulletin of the South Ural State University. Series: Mathematical Modelling, Programming and Computer Software*, 2017, vol. 10, no. 3, pp. 120–132. DOI: 10.14529/mmp170310
4. Steinberg B.Y. [Parallelizing Recurrence Loops with Conditional Statements]. *Automation and Telemechanics*, 1995, no. 6, pp. 176–184. (in Russian)
5. Steinberg O.B. [Parallelizing Recurrent Program Loops with Irregular Superposition Computation]. *University News. North-Caucasian Region. Natural Sciences Series*, 2009, no. 2, pp. 18–21. (in Russian)
6. Steinberg O.B., Sukhoverkhov S.E. [Recurrent Program Loops with Stability Check]. *Information Technologies*, 2010, no. 1, pp. 40–45. (in Russian)
7. Steinberg O.B. *Parallelizing Loops Allowing Recurrent Dependences. PhD Thesis.* Moscow, Institute for System Programming of the Russian Academy of Sciences, 2014. (in Russian)
8. Steinberg B.J. *Matematicheskie metody rasparallelivaniya rekurrentnykh programnykh tsiklov na superkompyutery s parallel'noy pamyatyu* [Parallelizing Recurrent Program Loops with Irregular Superposition Computation]. Rostov-on-Don, Rostov University Publishing House, 2004. (in Russian)
9. Graham R.L., Knuth D.E., Patashnik O. *Concrete Mathematics: a Foundation for Computer Science.* N.Y., Addison-Welsey, 1994.

10. Samarskiy A.A. *Vvedenie v chislennye metody* [Introduction to Numerical Methods]. Moscow, Nauka, 1997. (in Russian)
11. Graham S.L., Snir M., Patterson C.A. *Getting up to Speed: the Future of Supercomputing.* Washington, National Academies Press, 2005.
12. *Optimizing Parallelizing System.* Available at: http://ops.rsu.ru (accessed 1.07.2020).
13. *Optimizing Compiler Pluto.* Available at: http://pluto-compiler.sourcefofge.net (accessed 1.07.2020).
14. *ROSE Compiler.* Available at: http://rosecompiler.org/ (accessed 1.07.2020).

*Received March 7, 2020*