УДК 681.326

ИССЛЕДОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ЧИСЛЕННОГО АЛГОРИТМА РЕШЕНИЯ ЖЕСТКИХ СИСТЕМ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ

ЧАДОВ С.Н., асп.

Исследуется вариант одного из методов численного решения систем жестких дифференциальных уравнений. Приводятся данные по относительному времени выполнения различных шагов алгоритма, на основании чего предлагаются варианты его усовершенствования.

Ключевые слова: система жестких дифференциальных уравнений, алгоритм, распараллеливание.

INVESTIGATION OF NUMERICAL SOLUTION ALGORITHM PRODUCTION OF STIFF DIFFERENTIAL EQUATION SYSTEMS

S.N. CHADOV, postgraduate

The author investigates the variant of one of numerical solution methods of stiff differential equation systems. There are some data on comparative time of algorithm different stages fulfillment, on the basis of which the author suggests variants of its improvement.

Key words: stiff differential equation system, algorithm, paralleling.

Введение. При решении многих практических задач возникают системы дифференциальных уравнений, обладающие высокой жесткостью и, как следствие, плохо поддающиеся решению традиционными методами (такими, как методы Рунге-Кутты, Адамса и т.д.). Для таких систем уравнений разработаны специальные методы. Наиболее распространенными из них являются методы Гира. Обладая сравнимой с «обычными» методами асимптотической сложностью, алгоритмы Гира на практике часто оказываются значительно медленнее. Соответственно, возникает задача анализа и оптимизации их производительности.

Алгоритм Гира. Приведем краткое описание рассматриваемого алгоритма.

Пусть дана задача Коши

$$y' = f(t,y), \quad y(t_0) = y_0, \qquad (1)$$

где каждая переменная представляет собой вектор размерности N.

Метод Гира основан на так называемых формулах дифференцирования назад (backward differentiation formulae, BDF) q-го порядка:

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} + h\beta_n f(t_n, y_n) = 0. \qquad (2)$$

Можно показать, что формулы порядков 1-6 обладают свойствами A(α) – устойчивости и жесткой устойчивости. Формулы дифференцирования назад порядков 1 и 2 являются также A-устойчивыми (определение A(α)-устойчивости и жесткой устойчивости см., например, в [1]).

Коэффициенты α и β для формул (2) приведены в табл. 1 [2].

Рассматриваемая реализация алгоритма в основном следует алгоритму, реализованному в библиотеке CVODE [3], и основана на использовании так называемого вектора Нордсика(Nordsieck vector) (подробное описание см. в [4]). Решение дифференциального уравнения y'=f(x,y) представляется как нахождение аппроксимирующего полинома, представленного вектором, содержащим q производных y:

$$Z = \left[y_n \quad hy'_n \quad \ldots \quad \frac{h^q y_n^{(q)}}{q!} \right]. \qquad (3)$$

Таблица 1. **Коэффициенты формул дифференцирования назад порядков 1-6**

q	α_6	α_5	α_4	α_3	α_2	α_1	α_0	β
1						1	–1	1
2					1	$-\frac{4}{3}$	$\frac{1}{3}$	$\frac{2}{3}$
3				1	$-\frac{18}{11}$	$\frac{9}{11}$	$-\frac{2}{11}$	$\frac{6}{11}$
4			1	$-\frac{48}{25}$	$\frac{36}{25}$	$-\frac{16}{25}$	$\frac{3}{25}$	$\frac{12}{25}$
5		1	$-\frac{300}{137}$	$\frac{300}{137}$	$-\frac{200}{137}$	$\frac{75}{137}$	$-\frac{12}{137}$	$\frac{60}{137}$
6	1	$-\frac{360}{147}$	$-\frac{72}{147}$	$-\frac{450}{147}$	$-\frac{400}{147}$	$-\frac{225}{147}$	$-\frac{10}{147}$	$\frac{60}{147}$

Этот подход имеет ряд преимуществ с точки зрения скорости вычислений и является стандартным при реализации алгоритмов Гира.

Как видно из формулы (2), формулы дифференцирования назад являются неявными. Поэтому для получения значения интегрируемой функции на очередном шаге требуется решение системы в общем случае нелинейных алгебраических уравнений. Наиболее очевидным способом этого решения будет использование итерационного метода Ньютона [1]. В свою очередь, метод Ньютона требует решения системы линейных алгебраических уравнений. Чаще всего для этого используется одна из вариаций метода Гаусса [1]. Однако практика показывает, что такое решение не является идеальным. Во многих реальных задачах (например, при моделировании электроэнергетических систем) системы уравнений имеют очень большую размерность, но при этом достаточно сильно разрежены. В этих условиях более выгодным представляется использование одного из итерационных методов [5]. Отличительной чертой этих методов является использование матрицы предобусловливания для ускорения сходимости, что позволяет (при правильном выборе матрицы предобусловливания и метода решения соответствующей системы уравнений) значительно повысить скорость сходимости (иногда в тысячи раз).

Для определенности, выберем для реализации стабилизированный метод бисопряженных градиентов (BiCG-Stab).

Анализ производительности. На самом верхнем уровне решение системы дифференциальных уравнений можно представить как последовательность итераций, в каждой из которых можно выделить следующие этапы:

а) выбор порядка и шага по времени на текущую итерацию;

б) вычисление предиктора;

в) вычисление корректора (решение системы нелинейных уравнений);

г) анализ результата и подготовка данных для следующей итерации.

Распределение времени вычислений (в относительных единицах) представлено в табл. 2.

Таблица 2. **Время выполнения основных этапов алгоритма**

Этап	Время выполнения
а	177
б	6572
в	45141
г	3
д	3986

Таким образом, большую часть времени выполнения процедуры занимает вычисление корректора, т. е. решение системы алгебраических уравнений. Наиболее ресурсоемкой частью алгоритма решения системы нелинейных уравнений является шаг решения системы линейных уравнений. Таким образом, производительность подпрограммы решения систем линейных уравнений оказывает определяющее влияние на производительность исследуемого алгоритма в целом.

Методы повышения производительности. Существует несколько основных способов увеличения производительности. Набольшие результаты, как правило, дают алгоритмическая оптимизация и распараллеливание используемых алгоритмов.

С точки зрения алгоритмической оптимизации наибольший эффект в данном случае, возможно, даст использование матрицы предобусловливания. Однако данный подход ограничен тем, что выбор матрицы предобусловливания часто является достаточно сложной творческой задачей, поскольку на настоящий момент неизвестно удобных на практике формальных алгоритмов для такого выбора (хотя существует несколько наиболее распространенных вариантов [5]). Более того, неправильный выбор предобусловливателя может существенно замедлить решение или даже превратить сходящееся решение в расходящееся.

Параллельная реализация в качестве решения для общего случая (т. е независимого от задачи и, следовательно, могущего быть реализованным в библиотечной функции) представляется более выгодной. Распараллеливание можно осуществлять на уровне задачи (геометрически), уровне алгоритма и уровне структур данных. Первый способ рассматривать не будем, так как возможность его эффективной реализации независимо от задачи сомнительна. Распараллеливание на уровне алгоритма (создание нового параллельного алгоритма, решающего ту же задачу) достаточно перспективно, однако зачастую требует серьезных исследований.

Рассмотрим вариант реализации алгоритма, использующий параллелизм на самом низком уровне. Воспользуемся следующим наблюдением: все операции алгоритма выполняются над векторами размерности, равной количеству уравнений в системе, причем многие операции могут выполняться над каждым элементом независимо (копирование, поэлементное сложение и т.д.). Таким образом, возможна реализация, в которой каждый процесс ответственен за обработку некоторой части каждого вектора. С точки зрения программной реализации операции над векторами могут быть вынесены в отдельный модуль, изолируя таким образом основную программу (т. е. непосредственно алгоритм метода Гира) от деталей реализации операций над векторами, таких как количество используемых процессоров, тип используемых межпроцессных коммуникаций (общая память, передача сообщений), тип используемого для межпроцессных взаимодействий API (OpenMP,MPI,TBB) и т.д. Такая реализация сильно повышает переносимость программного кода и делает его более легким для понимания и модификации. Учитывая современные тенденции развития векторных вычислительных систем, наиболее эффективной является реализация параллельного варианта алгоритма для SIMD-систем. Например, современные x86-совместимые процессоры поддерживают наборы специальных SIMD-инструкций (MMX,SSE,SSE2,SSE3 и др.). Таким образом, параллельная обработка возможна даже на компьютерах с одним физическим процессором. Более того, современные графические процессоры способны работать как векторные процессоры общего назначения с числом исполняющих устройств, значительно превышающим возможности центрального процессора (см., например, [7]).

Итак, для машин с разделенной памятью наиболее выгодной представляется следующая реализация: каждый из n процессов хранит в своем адресном пространстве свою часть каждого N-мерного вектора, состоящую из N/n элементов (будем считать, что N кратно n).

Каждый процесс исполняет один и тот же код, но над своей частью данных. Таким образом, достигается некоторая степень независимости процессов друг от друга. Межпроцессного обмена требуют лишь операции вычисления экстремальных значений, норм и скалярного произведения векторов. В качестве интерфейса для межпроцессных коммуникаций предлагается использовать MPI. Пример реализации такого подхода можно найти в библиотеке CVODE [3].

Для машин с общей памятью более выгодной может являться другая реализация. Можно разместить векторы целиком в общей памяти и сделать каждый процесс ответственным за обработку какого-либо конкретного участка. Преимуществом данного способа является отсутствие необходимости в межпроцессных коммуникациях. В данном случае более выгодным представляется использование не MPI, а OpenMP или TBB.

Приведем примеры реализации методов класса «параллельного» вектора для SMP-систем. В качестве программного интерфейса для распараллеливания будем использовать Intel TBB [8]. Первый пример – реализация метода fill, заполняющего все компоненты вектора константным значением. Легко видеть, что такая операция не требует межпроцессных коммуникаций и каждый процесс может заполнять свою часть массива вне зависимости от других:

```
struct Fill{
    realtype * a;
    realtype c;

    Fill(realtype* a_, realtype c_):a(a_),c(c_){}
    void operator ()(tbb::blocked_range<realtype>& r)const{
        for(int i=r.begin();i!=r.end();++i){
            a[i]=c;
        }
    }
};

void fill(realtype r){
    parallel_for(tbb::blocked_range<realtype>(0,data.size(),N/NTHREADS),
        Fill(&data[0],r)
        );
}
```

Второй пример – метод, вычисляющий скалярное произведение векторов. Этот метод, в отличие от предыдущего, требует осуществления обмена информацией между процессами:

```
struct Dot{
    const realtype* a;
    const realtype* b;
    realtype sum;

    Dot(const realtype* a_ ,const realtype* b_ ):sum(0),a(a_),b(b_){}
    Dot(Dot& right, tbb::split):sum(0){}

    void operator ()(tbb::blocked_range<realtype>& r){
        for(int i=r.begin();i!=r.end();++i){
            sum+=a[i]*b[i];
        }
    }
    void join(Dot& right){
        sum+=right.sum;
    }
};

realtype dot(const mvector& right) const
{
    Dot val(&data[0],&right[0]);
    tbb::parallel_reduce(tbb::blocked_range<realtype>(0, N, N/NTHREADS),
        val);
    return val.sum;
}
```

Аналогичный подход может быть использован для реализации операций над векторами на основе SIMD-инструкций современных x86 процессоров. В простейших случаях современные компиляторы (например, Intel C++ 9) способны произвести такую оптимизацию самостоятельно, однако в более сложных случаях приходится кодировать алгоритм вручную на языке ассемблера. Такого подхода, хотя он зачастую дает выигрыш в производительности, тем не менее, в большинстве случаев следует избегать, поскольку он жестко привязывает алгоритм к конкретной архитектуре процессора, типу данных (например, число с плавающей точкой одинарной точности) и т.д.

Предварительные результаты. На основе изложенного была реализована библиотека функций для решения жестких систем дифференциальных уравнений. Алгоритмически она в основном следует реализации CVODE, однако имеет ряд преимуществ:

а) реализована на современном C++ в объектно-ориентированном стиле, что упрощает сопровождение и улучшает читабельность кода, позволяет использовать современные программные средства;

б) в отличие от CVODE, реализующей параллелизм на основе MPI и, соответственно, нацеленной на системы с разделенной памятью, разработанная библиотека реализует параллельный режим также и для SMP-систем, для чего используется OpenMP и/или Intel TBB.

Было проведено тестирование параллельной версии на персональных компьютерах, оснащенных двухядерными процессорами(Intel P4D, Intel Core 2 Duo, AMD Athlon 64X2). В результате было выяснено, что реальный прирост производительности оказался ниже запланированного и составил ≈35%. Одной из основных причин такого результата предположительно являются высокие затраты на диспетчеризацию потоков, а также неоптимальная стратегия использования памяти.

Таким образом, возможны следующие пути дальнейшего улучшения производительности:

а) распараллеливание на уровне алгоритма, что приведет к уменьшению количества операций синхронизации и меньшим затратам на диспетчеризацию потоков;

б) реализация на специализированном устройстве (например, GPGPU [7]).

Список литературы

1. **Амосов А.А, Дубянский Ю.А., Копченов Н.В**. Вычислительные методы для инженеров: Учеб. пособие. – М.: Высш. шк., 1994.
2. **Bermejo R.** Stiff System. Dynamical Systems in Physiology and Medicine, Urbino (Italy) July 8–19, 2002.
3. **Hindmarsh A.C., Serban R.** User Documentation for cvode v2.4.0. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, March 24, 2006.
4. **Wanner G., Hairer E., Nørsett S.P.** Solving Ordinary Differential Equations. Springer Series In Computational Mathematics; Vol. 8. Springer-Verlag New York, Inc. New York, NY, USA, 1993.
5. **Richard Barrett, Michael Berry, Tony F. Chan**, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.
6. **Ouarraui C., Kaeli D**. Developing object-oriented parallel iterative methods.Department of Electrical and Computer Engineering,Northeastern University, Boston, MA 02115.
7. **NVIDIA CUDA Programming Guide**. NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050 www.nvidia.com
8. **Intel(R) Threading Building Blocks Reference Manual**. Document Number 315415-001US Revision: 1.6 World Wide Web: http://www.intel.com

Чадов Сергей Николаевич,
ГОУВПО «Ивановский государственный энергетический университет имени В.И. Ленина»,
аспирант кафедры высокопроизводительных вычислительных систем,
e-mail: sergei.chadov@gmail.com