Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

# Паралелизација Runge-Kutta метода 3. и 4. реда у архитектурама дељене и дистрибуиране меморије уз програмски језик Python

*Аутор:*
Милена Вујичић

*Индекс:*
E2 67/2020

20. фебруар 2021.

**Сажетак**

Runge-Kutta методе су често коришћен скуп метода за решавање обичних диференцијалних једначина. Имплементација алгоритма за решавање диференцијалних једначина методама Runge-Kutta, у зависности од жељене прецизности може заузети велику количину компјутерских ресурса. Паралелизацијом метода и њиховим извршавањем на већем броју језгара постиже се боља ефикасност у брзини извршавања алгоритма. Решење овог проблема имплементирано је уз помоћ mpi4py библиотеке за python програмски језик. Паралелизација је примењена на Runge-Kutta методама 3. и 4. реда и у оба случаја је постигнута знатно већа брзина извршавања алгоритма.

# Садржај

# Списак изворних кодова

# Списак табела

# 1 Увод

Предмет овог рада је паралелизација метода Runge-Kutta 3. и 4. реда у python програмском језику уз коришћење mpi4py библиотеке која имплементира MPI стандард за размену порука у паралелним рачунарским архитектурама.

# 2 Метода Runge-Kutta

Обичне диференцијалне једначине су диференцијалне једначине (ОДЈ) облика:

$$a_0(x)y + a_1(x)y' + a_2y'' + ... + a_n(x)y^{(n)} + b(x) = 0 \tag{1}$$

где су $a_0(x)...a_n(x)$ и $b(x)$ произвољне диференцијабилне функције, а $y'...y^{(n)}$ узастопни изводи непознате функције $y$ по промењљивој $x$ .

Највећи број ОДЈ се не може решити интеграљењем, па је потребно применити нумеричке методе. Нумеричке методе за решавање ОДЈ базирају се на развијању једначина у Тејлоров ред. Ове методе дају апроксимацију решења ОДЈ. Постоји велики број метода који се користе за апроксимацију решења ОДЈ. Најједноставнији нумерички метод за решавање ОДЈ је Ојлеров поступак.

## 2.1 Ојлеров поступак

Ојлеров постпуак се бави решавањем проблема облика:

$$\frac{dy}{dt} = f(t,\ y),\ y(t_0) = y_0 \tag{2}$$

Почетно решење $y_0$ је познато. Ојлеров поступак полази од функције $y(x+h)$ развијене у Тејлоров ред

$$y(x+h) \approx y(x) + hy'(x) + \frac{h^2y''(x)}{2!} + \frac{h^3y'''(x)}{3!} + ... \tag{3}$$

Ова функција се даље апроксимира са:

$$y(x+h) \approx y(x) + hf(x,\ y) \tag{4}$$

Апроксимацијом се постиже лакше рачунање, али се смањује прецизност добијеног резултата. Ојлеров поступак је итеративан, што значи да апроксимација за свако израчунато $y$ зависи од његове вредности добијене у претходном кораку:

$$y_{i+1} \approx y_i + hf(x_i,\ y_i) \tag{5}$$

Решење је прецизније што је $h$ мање.

## 2.2 Runge-Kutta 3. и 4. реда

Методе Runge-Kutta представљају скуп алгоритама за нумеричко решавање обичних диференцијалних једначина. Ове методе су изведене уопштавањем Ојелровог поступка и увођењем више чланова који боље апроксимирају вредност $y$ . У наставку ће бити приказане методе Runge-Kutta 3. и 4. реда, без извођења.

### 2.2.1 Метода Runge-Kutta 3. реда

Полазимо од једначина:

$$\begin{aligned} y_{i+1} &= y_i + c_1k_1 + c_2k_2 + c_3k_3 \\ k_1 &= f(y_i, t_i)h \\ k_2 &= f(y_i + a_2k_1, t_i + a_2h)h \\ k_3 &= f(y_i + b_3k_1 + (a_3 - b_3)k_2, t_i + a_3h)h \end{aligned} \tag{6}$$

из којих следи:

$$\begin{aligned} c_1 + c_2 + c_3 &= 1 \\ c_2a_2 + c_3a_2 &= \frac{1}{2} \\ c_2a_2^2 + c_3a_3^2 &= \frac{1}{3} \\ c_3(a_3 - b_3)a_2 &= \frac{1}{6} \end{aligned} \tag{7}$$

Решавањем система (7) добијају се коефицијенти $c_1, c_2, c_3$, као и, $a_2, a_3, b_3$ при чему не постоји једнозначно решење.

### 2.2.2 Метода Runge-Kutta 4. реда

Runge-Kutta 4. реда израчунава $y_{i+1}$ коришћењем коефицијената добијених из следећег система:

$$\begin{aligned} y_{i+1} &= y_i + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 \\ k_1 &= f(y_i, t_i)h \\ k_2 &= f(y_i + b_{2,1}k_1, t_i + a_2h)h \\ k_3 &= f(y_i + b_{3,1}k_1 + b_{3,2}k_2, t_i + a_3h)h \\ k_4 &= f(y_i + b_{4,1}k_1 + b_{4,2}k_2 + b_{4,3}k_3, t_i + a_4h)h \end{aligned} \tag{8}$$

Изједначавањем система (8) са одговарајућим члановима Тејлоровог реда добија се нови систем:

$$
\begin{gathered}
c_1 + c_2 + c_3 + c_4 = 1 \\
c_1 a_2 + c_2 a_3 + c_3 a_4 = \frac{1}{2} \\
c_1 a_2^2 + c_2 a_3^2 + c_3 a_4^2 = \frac{1}{3} \\
c_1 a_2^3 + c_2 a_3^3 + c_3 a_4^3 = \frac{1}{4} \\
c_3 a_2 b_{3,2} + c_4 (a_2 b_{4,2} + a_3 b_{4,3}) = \frac{1}{6} \\
c_3 a_3 a_2 b_{3,2} + c_4 a_4 (a_2 b_{4,2} + a a_3 b_{4,3}) = \frac{1}{8} \\
c_3 a_2^2 b_{3,2} + c_4 (a_2^2 b_{4,2} + a_3^2 b4, 3) = \frac{1}{12} \\
c_4 a_2 b_{3,2} b_{4,3} = \frac{1}{24} \\
b_{2,1} = a_2 \\
b_{3,1} + b_{3,2} = a_3 \\
b_{4,1} + b_{4,2} + b_{4,3} = a_4
\end{gathered}
\tag{9}
$$

Овај систем има 11 једначина и 13 непознатих па због тога не постоји једнозначно решење. Да би се систем решио, потребно је произвољно изабрати две непознате.

# 3 MPI

*Message Passing Interface* (*MPI*) је спецификација стандарда за размену порука намењена употреби у паралелном рачунарству. *MPI* није библиотека, већ опис шта библиотека која имплементира овај стандард треба да садржи. Постоје многе имплементације MPI стандарда за различите програмске језике.

*MPI* описује начин прослеђивања порука између адресних простора различитих процеса. *MPI* се може користити и у дистрибуираним архитектурама, архитектурама са дељеном меморијом као и хибридним архитектурама.

Неке од важнијих имплементација MPI стандарда су: *OpenMPI*, *MVAPICH*, *Intel MPI*. У даљем тексту овог поглавља детаљније ће бити описана *mpi4py* имплементација *MPI* стандарда јер је она коришћена за израду програма.

## 3.1 Библиотека mpi4py

Библиотека *mpi4py* је имплементација *MPI-2* стандарда за python програмски језик. Ова библиотека преводи синтаксу већ постојаће *MPI* имплементације за *C*/*C++* у *python* код.

Метод серијализације и десеријализације података на ком се базира рад библиотеке је *pickling*. *Pickling* процес претвара *python* објекте у бајт ток података, док *unpickling* враћа објекте из бајт тока података *python* објекат. Још један назив за *pickling* је и серијализација. *mpi4py* библиотека користи *pickling* за бинарну репрезентацију података током размене порука.

Основна класа за комуникацију је MPI.Comm. Све друге класе за комуникацију наслеђују ову класу. MPI.Comm садржи информацију о рангу процеса, као и о броју процеса који су покренути.

*Point to Point* комуникација омогућава да један процес шаље поруку другом процесу који је чита. Функција за слање порука је MPI.Comm.send() и она као параметре узима податак који се шаље (који може бити било који *python* објекат), ранг процеса којем се шаље порука и *tag* којим је порука означена. Примање и читање поруке врши функција MPI.Comm.recv(). Параметри ове функције су ранг процеса од ког се добија порука као и *tag* којим је порука означена. Повратна вредност ове функције је *python* објекат који је послат. [1]

Библиотека такође омогућава и групну комуникацију. Оваква комуникација је омогућена функцијама MPI.Comm.bcast(), MPI.Comm.scatter() и MPI.Comm.gather().

MPI.Comm.bcast() функција шаље поруку из једног коренског процеса свим другим процесима. MPI.Comm.scatter() ради на сличном принципу као и функција MPI.Comm.bcast(), али MPI.Comm.scatter() може да пошаље више различитих делова информације из коренског процеса. MPI.Comm.gather() прикупља поруке послате од стране различитих процеса. [1]

```
def calculate_new_y(f, x, h, y):

    k1 = f(x, y)
    k2 = f(x+h/2, y+h*k1/2)
    k3 = f(x+h, y-h*k1+2*k2*h)
    y = y + h*(k1+4*k2+k3)/6
    return y
```

Изворни код 1: Рачунање следећег члана Runge-Kutta методом 3. реда

```
    def calculate_new_y(f, x, h, y):

    k1 = f(x, y)
    k2 = f(x+h/2, y+k1*h/2)
    k3 = f(x+h/2, y+k2*h/2)
    k4 = f(x+h, y+h*k3)
    y = y + h*(k1+2*k2+2*k3+k4)/6

    return y
```

Изворни код 2: Рачунање следећег члана Runge-Kutta методом 4. реда

# 4 Имплементација решења

Решење је имплементирано у *python* програмском језику коришћењем *mpi4py* библиотеке.

Обе методе паралелизоване су на исти начин. Једина разлика је начин на који се рачуна следећи члан.

Код Runge-Kutta методе 3. реда, наредна вредност порменљиве $y$ рачуна се на начин приказан у изворном коду 1 .

Код Runge-Kutta методе 4. реда, наредна вредност порменљиве $y$ рачуна се на начин приказан у изворном коду 2

У даљем тексту биће дат пример паралелизације Runge-Kutta методе 4. реда. Runge-Kutta методе 3. реда паралелизована је на исти начин осим што се код ће позива метод из изворног кода 1 уместо метода из изворног кода 2

Процес ранга 0 је коренски процес који координира рад осталих процеса. Коренски процес дели низ вредности променљиве $x$ на приближно једнаке делове и шаље их осталим процесима на обраду уз помоћ функције MPI.Comm.bcast(). Процеси вишег ранга апроксимирају вредности функције позивом методе из изворног кода 2. Сваки процес врши апроксимацију за вредности променљиве $x$ коју је добио од коренског процеса. Сваки процес као повратну вредност враћа свој део низа вредности променљиве $x$, као и низ израчунатих вредности за $y$. Поступак је приказан изворним кодом 3.

Након што процеси врате израчунате повратне вредности, потребно је преузети их одговарајућим редоследом. Сваки процес шаље своју повратну вредност процесу ранга већег за 1. Последњи процес прима све податке и исцртава график на основу добијених података. Такође, он исписује време потребно да се обаве прорачуни пре обједињења података. Поступак је приказан изворним кодом 4.

## 4.1 Пример извршавања кода

Код је извршен на функцији датој у изворном коду 5.

Табелом 1 илустрована је зависност времена потребног за извршење прорачуна од броја коришћених језгара. Програм је извршаван за вредност променљиве $x$ у интервалу од 0 до 10000 са кораком 0.001.

| број језгара | време за rk3 (s) | време за rk4 (s) |
|---|---|---|
| 2 | 26.70 | 33.24 |
| 4 | 9.86 | 11.51 |
| 8 | 5.22 | 5.15 |

Табела 1: Табела са временима иршавања

```python
def rk4_parallelized(f, xb, yb, h, xe, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()
    x_vals = []

    if rank == 0:
        x=xb
        while x < xe:
            x_vals.append(x)
            x +=h
        comm.bcast(x_vals, root=0)

        return [],[]
    else:
        x_recv = []
        x_recv = comm.bcast(x_vals, root=0)
        step = int(len(x_recv) / (size-1))
        y_temp = []
        if rank != (size-1):
            b_idx = (rank-1)*step
            e_idx = rank*step
            for x in x_recv[b_idx:e_idx]:
                y = calculate_new_y(f, x, yb, h)
                y_temp.append(y)
            return x_recv[b_idx: e_idx], y_temp

        else:
            b_idx = (rank-1)*step
            e_idx = len(x_recv)
            for x in x_recv[b_idx:e_idx]:
                y = calculate_new_y(f, x, yb, h)
                y_temp.append(y)

            return x_recv[b_idx: e_idx], y_temp
```

Изворни код 3: Рачунање вредности променљиве $y$ за Runge-Kutta методу 4. реда

```

    for i in range(size):
        if rank == i:
            start4 = time.time()
            x_rest, y_rest = rk4.rk4_parallelized(f.func, x4, y4, h4, x_
            end4 = time.time()

            if rank == 0:
                pack = []
                pack.append((rank, x_rest, y_rest))

                comm.send(pack, dest=rank+1, tag=0)
                print(Rank, rank, sending)
            elif rank != size - 1:
                pack = comm.recv(source=rank-1, tag=0)
                pack.append((rank, x_rest, y_rest))
                comm.send(pack, dest=rank+1, tag=0)
                print(Rank, rank, sending)

            else:
                pack = comm.recv(source=rank-1, tag=0)
                pack.append((rank, x_rest, y_rest))
                print(pack[1][0])
                for i in range(1, size):
                    plt.title(RK4)
                    plt.plot(np.array(pack[i][1]), np.array(pack[i][2])
                print(nTotal time is: , end4-start4)

    plt.show()
```

Изворни код 4: Сакупљање израчунатих података и њихово графичко представљање за Runge-Kutta методу 4. реда

```
import numpy as np
def func(x, y):
        return np.sin(x)-y
```

Изворни код 5: Пример функције над којом је извршен код

## 5 Закључак

Паралелизацијом Runge-Kutta метода постиже се много већа брзина извршавања прорачуна. С обзиром да методе Runge-Kutta заузимају велику количину ресурса приликом прорачуна, повећање брзине извршавања је од велике важности у системима који баратају великом количином података. Методом Runge-Kutta 4. реда постиже се већа тачност апроксимације, а паралелизацијом и време прорачуна овом методом се значајно смањује и чини њену примену веома ефикасном.

# Библиографија

[1] mpi4py documentation. `https://mpi4py.readthedocs.io/en/stable/`. Accessed: 2021-02-19.