International Journal of Software Engineering and Computer Systems (IJSECS)
ISSN: 2289-8522, Volume 4 Issue 1, pp. 84-94, February 2018

https://doi.org/10.15282/ijsecs.4.1.2018.7.0040

PARALLEL INTEGRATION ALGORITHM AND ITS USAGE FOR A PRACTICAL SIMULATION OF SPACECRAFT ATTITUDE MOTION

Ravil' Kudermetov

Computer Systems and Networks Dept., Zaporizhzhya National Technical University,
69063, Zhukovsky str., 64, Zaporizhzhya, Ukraine.
E-mail: kudermetov@gmail.com

ABSTRACT

Nowadays multi-core processors are installed almost in each modern workstation, but the question of these computational resources effective utilization is still a topical one. In this paper the four-point block one-step integration method is considered, the parallel algorithm of this method is proposed and the Java programmatic implementation of this algorithm is discussed. The effectiveness of the proposed algorithm is demonstrated by way of spacecraft attitude motion simulation. The results of this work can be used for practical simulation of dynamic systems that are described by ordinary differential equations. The results are also applicable to the development and debugging of computer programs that integrate the dynamic and kinematic equations of the angular motion of a rigid body.

Keywords: parallel computing, multi-core processors, multi-threading, hyper-threading, speedup, spacecraft attitude motion

INTRODUCTION

Today, most general purpose computers contain two- and four-core processors and there is a tendency of further growth of the number of cores in the processors. At the same time there is a considerable lag in the development of software that would effectively use the resources of the multi-core processors, this is related, presumably, with long-term experience of a successive programming (Cooper, 2014). The ordinary successive programs at the worst case can load only one core of the processor, leaving the other cores in an inactive state. So, the usage of traditional, successive methods of programming for modern applications working on the multi-core processors can result in the considerable loss of their productivity. This issue also applies to existing simulation tools that are used to simulation complex dynamic systems, for example, cyber-physical systems. Most modern simulation environments still execute on a single thread, which does not take advantage of the processing power available on modern multi-core CPUs (Carl & Biswas, 2016). The usage of a bundle of multiple parallel processes or threads that are not consistent with an architecture of the multi-core processors also can be a cause of an inefficient exploitation of the multi-core processors and even can lead to a loss of operating speed compared to non multi-core processors.

Simulation models of continuous dynamic systems and the search for solutions of real-world problems require the solution of ordinary differential equations (ODEs). But many of these simulation models and problems do not have exact analytical solutions and can only be obtained by numerical approximation methods, the reliability of which has been proven by many researchers (Waeleh & Majid, 2016).

In the practical application programming, including a numerical solution of ordinary differential equations (ODE), the sequential methods of integration are dominated. Many methods of parallel integration ODE often remains a subject of scientific research without the widespread introduction to practical applications. The development of parallel software applications for simulation of dynamic systems on the base of multi-core processors can provide significant performance benefits. In the context of implementing the parallel integration of ODEs the block integration methods take an important place, since they have a natural parallelism. Block methods for ODEs were developed and investigated by many authors; the references to researches and surveys of block methods can be found in (Majid, Mehrkanoon & Othman, 2010; Ishak & Suleiman, 2012). Various types of block integration methods ODEs are developed, their systematization can be seen in (Waeleh & Majid, 2017).

There are many technologies for parallel computations, but on the practice the Message Passing Interface (MPI) for multi-processor computers, OpenMP and multi-threaded programming on C/C++ and Java languages for symmetric multi-processing (SMP) and multi-core CPUs, CUDA for graphical processing units (GPUs) are the most common. Parallel implementations of block integration methods using some of their technologies are known. The parallel block method for solving of higher order ODEs, its algorithm and implementation in C language for SMP computer have been developed in (Majid & Suleiman, 2009). Majid, Mehrkanoon and Othman (2010) employed MPI technology for their parallel block method for solving large systems of ODEs. In (Carl & Biswas, 2016) the multi-threaded implementation in C++ has developed for parallel solution of ODEs on a multi-core CPUs. Approaches to parallel solution of the ODEs on the GPUs are given in (Al-Omari, Arnold, Taha & Schuttler, 2013; Ahnert, Demidov, & Mulansky, 2014).

In this paper the parallel realization of the four-point block one-step method of integration ODEs is proposed. This realization is based on multi-threading mechanism of Java language. The possibility to perform iterations in four points of block independently allows to parallelize an integration process on four programmatic threads. If there are four cores in processor then each thread may be executed on a single core, whereby it is theoretically possible to achieve a four-fold speedup of ODEs integration. Significant speedup can also be obtained in the case with a smaller number of cores on the processor that have an embedded hyper-threading technology. However, the data exchange between the threads and their synchronization require an additional time, so a practical speedup is always less than a theoretical speedup. It is shown that the effectiveness of multi-threaded realization of the block method of integration with blocking queues for data exchanges and synchronization of threads essentially depends on the amount of computations for the right-hand side (RHS) of ODEs. The correctness of the proposed parallel algorithm is verified on solving of ODEs for simulation of spacecraft attitude motion. The discussed results can be used for estimation of the necessity and possibility of usage the parallel methods of integration and their multi-threaded realizations.

FOUR-POINT BLOCK ONE-STEP INTEGRATION METHOD AND PARALLEL ALGORITHM

Consider the ODE of the form

$$\frac{dy}{dt} = f(t, y),\ y(t_0) = y_0 \tag{1}$$

on the interval $[t_0, T]$ that divided by the mesh-points $t_n = t_0 + nh$, $n = 1, 2, ..., N$, where $h = (T - t_0)/N$ is the step size. Suppose that for each n we seek a numerical approximation y_n to $y(t_n)$.

In accordance with introduced notations the k-point block one-step method of a numerical integration can be represented by the following computation scheme

$$\frac{y_{n,i} - y_{n,0}}{ih} = b_i F_{n,0} + \sum_{j=1}^{k} a_{i,j} F_{n,j}, \tag{2}$$

where $i = \overline{1,k}$ is a number of the point in the block; $n = 1, 2, \text{K}$ – a number of the block, and $F_{n,j} = f(t_n + jh, y(t_n + jh))$.

The k-point block one-step method has order of approximation p if the following conditions are satisfied [3]:

$$b_i + \sum_{j=1}^{k} a_{i,j} = 1, \ \sum_{j=1}^{k} j^{m-1} a_{i,j} = \frac{i^{m-1}}{m}, \ i = \overline{1,k}, \ m = \overline{2,p}. \tag{3}$$

If $p = k + 1 = 5$, then the solving of Eq.(3) gives all coefficients b_i and $a_{i,j}$ of the four-point block one-step method. These coefficients are conveniently written in the vector-matrix form:

$$B = \left(\frac{251}{720} \quad \frac{29}{180} \quad \frac{9}{80} \quad \frac{7}{90} \right), \ A = \begin{pmatrix} \frac{323}{360} & -\frac{11}{30} & \frac{53}{360} & -\frac{19}{720} \\ \frac{31}{45} & \frac{2}{15} & \frac{1}{45} & -\frac{1}{180} \\ \frac{17}{40} & \frac{3}{10} & \frac{7}{40} & -\frac{1}{80} \\ \frac{16}{45} & \frac{2}{15} & \frac{16}{45} & \frac{7}{90} \end{pmatrix}. \tag{4}$$

The highest order of approximation of this method is $k+1$, i.e. $O(h^5)$, and the errors at the points of a block are defined by the formula (Feldman & Dmitrieva, 2001)

$$\varepsilon_{n,i} = \frac{h^{k+1}}{(k+1)!} y_{n,0}^{(k+2)} \left(\sum_{j=1}^{k} j^{k+1} a_{i,j} - \frac{i^{k+1}}{k+2} \right) + O(h^{k+2}). \tag{5}$$

Thus, the four-point block one-step method has order of error $O(h^6)$. The integration step size h must be selected according to the following condition:

$$h < \frac{1}{kL\|A\|}, \tag{6}$$

where L is the Lipschitz constant, and $\|A\| = 1{,}4375$ is the matrix norm of A Eq.(5).

For calculating the approximate solutions $y_{n,k}$ of the unknown y of the Cauchy problem (1) at the points $t_n + kh$ for each n-th block it is necessary to iterate recurrence formulae that derived by substituting the coefficients b_i and $a_{i,j}$ (Eq.(4)) into Eq.(2):

$$y_{n+k,0} = y_n + khF_n, \ k = \overline{1,4}, \tag{7}$$

$$y_{n+1,s+1} = y_n + \frac{h}{720}(251F_n + 646F_{n+1,s} - 256F_{n+2,s} + 106F_{n+3,s} - 19F_{n+4,s}), \tag{8}$$

$$y_{n+2,s+1} = y_n + \frac{h}{90}(29F_n + 124F_{n+1,s} + 2F_{n+2,s} + 4F_{n+3,s} - F_{n+4,s}), \tag{9}$$

$$y_{n+3,\,s+1} = y_n + \frac{3h}{80}\left(9F_n + 34F_{n+1,s} + 24F_{n+2,s} + 14F_{n+3,s} - F_{n+4,s}\right), \tag{10}$$

$$y_{n+4,\,s+1} = y_n + \frac{2h}{45}\left(7F_n + 32F_{n+1,s} + 12F_{n+2,s} + 32F_{n+3,s} + 7F_{n+4,s}\right), \tag{11}$$

where $s = \overline{0, S}$ is the iteration number, $y_{n+k,\,s}$ is an approximation of the exact solution at k-th point on the s-th iteration, $F_n = f\left(t_n, y_n\right)$ is a value of $f(t, y)$ at the first point of n-th block, and $F_{n+k,\,s+1} = f\left(t_n + kh, y_{n+k,\,s}\right)$ is a value of $f(t, y)$ at the k-th point of the block on the iteration number $s+1$. The result of the last iteration is the approximate solution of Eq.(1) at fourth point of the block. This value is used as an initial value for calculating in the next block, i.e. in the block number $n+1$.

An implementation of the four-point block one-step method is organized as the Java multi-threaded application. For calculating approximate solutions at each point the separate thread is created, so that program has four threads for integration. These threads for convenience are called Thread-1, Thread-2, Thread-3 and Thread-4. Thread-1 calculates F_n, first approximation y_{n+1} to $y\left(t_n + h\right)$ according to Eq.(7) and next approximations $y_{n+1,1}, \mathrm{K}, y_{n+1,s+1}$ according to Eq.(8). Thread-2 calculates first approximation y_{n+2} to $y\left(t_n + 2h\right)$ according to Eq.(7) and next approximations $y_{n+2,1}, \mathrm{K}, y_{n+2,s+1}$ according to Eq.(9). Thread-3 and Thread-4 execute similar calculations in compliance with Eq.(7), Eq.(10), Eq.(11). These threads can work in parallel because the calculations at the points of the block are independent from each other. At each iteration the threads exchange between themselves by the results of calculations.

The algorithms of calculations for these threads are shown in Figures 1-3. Note, that after the last iteration in the Thread-4 the integration time is increased by the value $4h$.

```
1    t1 ← t0
2    y1 ← y0
3    while t1 < T do //T – time of integration
4      F0 ← f(t1, y1)
5      for i ← 2 to 4 do
6        F0,i ← F0
7        put(F0,i)
8      end for
9      u1 ← y1 + hF0
10     for j ← 0 to S do
11       F1 ← f(t1 + h, u1)
12       for i ← 2 to 4 do
13         F1,i ← F
14         put(F1,i)
15         take(Fi,1)
16       end for
```

```
17      u_1 ← y_1 + h/720 (251F_0 + 646F_1 − 256F_{2,1} + 106F_{3,1} − 19F_{4,1})
18    end for
19    take(t_1)
20    take(y_1)
21  end while
```

Figure 1. Algorithm of the calculation in the Thread-1

```
1     r ← 2
1'    r ← 3
2     t_r ← t_0
3     y_r ← y_0
4     while t_r < T do
5       take(F_{0,r})
6       u_r ← y_r + rhF_{0,r}
7       for j ← 0 to S do
8         F_r ← f(t_r + rh, u_r)
9         for i ← 1 to k do
10          if i ≠ r do
11            F_{r,i} ← F_r
12            put(F_{r,i})
13            take(F_{i,r})
14          end if
15        end for
16        u_r ← y_r + h/90 (29F_{0,r} + 124F_{1,r} + 2F_{2,r} + 4F_{3,r} − F_{4,r})
16'       u_r ← y_r + 3h/80 (9F_{0,r} + 34F_{1,r} + 24F_{2,r} + 14F_{3,r} − F_{4,r})
17      end for
18      take(t_r)
19      take(y_r)
20    end while
```

Figure 2. Algorithms of the calculation in the Thread-2 and Thread-3. Steps 1 and 16 are taken only by the Thread-2, Steps 1' and 16' are taken only by the Thread-3

```
1     t_4 ← t_0
2     y_4 ← y_0
3     while t_4 < T do
4       take(F_{0,4})
5       u_4 ← y_4 + 4hF_{0,4}
6       for j ← 0 to S do
```

```
7        F_4 ← f(t_4 + 4h, u_4)
8        for i ←1 to 3 do
9           F_{4,i} ← F_4
10          put(F_{4,i})
11          take(F_{i,4})
12       end for
13       u_4 ← y_4 + (2h/45)(7F_{0,4} + 32F_{1,4} +12F_{2,4} + 32F_{3,4} + 7F_4)
14    end for
15    t_4 ←t_4 + 4h
16    y_4 ←u_4
17    for i ←1 to 3 do
18       t_i ← t_4
19       y_i ← u_4
20       put(t_i)
21       put(y_i)
22    end for
23 end while
```

Figure 3. Algorithm of the calculation in the Thread-4

To exchange the results of calculations between the threads the blocking queue class **`LinkedBlockingQueue`** of the **`java.util.concurrent`** package and its methods **`put()`** and **`take()`** are used. The **`put()`** method inserts the specified element into a queue, waiting for a free slot if necessary. The **`take()`** method waits for a head element of a queue and removes it. If the queue is empty, it blocks and waits for an element to become available. So the data received by threads are always relevant. However, the usage of synchronization always causes downtimes of threads associated with the expectations of the inserting and removing of the queue data. For multi-threaded implementation of the integration method uses multiple blocking queues; they denoted $F_{i,j}$, y_j, t_j, where $i = \overline{0,4}$ is the index of functions, $j = \overline{1,4}$ is the thread number. The use these queues and methods **`put()`** and **`take()`** is clear of the algorithms presented here, for instance in the line 7 (Figure 1) the $\text{put}(F_{0,i})$ method inserts the value of $F_{0,i}$ to queue $F_{0,i}$ and in the line 15 of this algorithm the $\text{take}(F_{i,1})$ method removes the value of $F_{1,j}$ from queue $F_{1,j}$.

SIMULATION MODEL OF SPACECRAFT ATTITUDE MOTION

The Euler's equation describing the dynamics of the attitude motion of a spacecraft about the centre of mass has the form

$$\boldsymbol{J}\dot{\overline{\omega}} + \overline{\omega} \times \boldsymbol{J}\overline{\omega} = \boldsymbol{M}, \tag{12}$$

where the matrix $\boldsymbol{J} = \|J_{n,m}\|$ is the inertia matrix and $n, m = x, y, z$; the vector $\overline{\omega} = (\omega_x(t), \omega_y(t), \omega_z(t))$ is the angular velocity vector expressed in the body-fixed

frame, and the vector $\boldsymbol{M} = (M_x(t), M_y(t), M_z(t))$ is the applied torques. A kinematics equation gives the dependency of the time derivative of its relative orientation in space from the angular velocity (Branets & Shmyglevskii, 1973)

$$\dot{\mathbf{Q}} = \frac{1}{2}\mathbf{Q} \circ \bar{\omega}, \tag{13}$$

where $\mathbf{Q} = q_0(t) + q_1(t)\bar{i}_1 + q_2(t)\bar{i}_2 + q_3(t)\bar{i}_3$ or $\mathbf{Q} = (q_0(t), q_1(t), q_2(t), q_3(t))$ is the rotation quaternion of a rigid body and the symbol $\circ$ denote quaternion multiplication.

In the general case the exact solution of the kinematics equation (Eq.(13)) for any motion cannot be expressed in terms of elementary functions. Therefore, in practice we have to use approximate methods, computer modelling and numerical integration.

For the numerical simulation of the spacecraft angular motion Eq.(12) and Eq.(13) are represented as a system of first-order differential equations:

$$\begin{aligned}
\dot{\omega}_x &= \frac{J_z - J_y}{J_x}\omega_y(t)\omega_z(t) + \frac{M_x(t)}{J_x},\\
\dot{\omega}_y &= \frac{J_x - J_z}{J_y}\omega_x(t)\omega_z(t) + \frac{M_y(t)}{J_y},\\
\dot{\omega}_z &= \frac{J_y - J_z}{J_z}\omega_x(t)\omega_y(t) + \frac{M_z(t)}{J_z},\\
\dot{q}_0 &= 0{,}5\left(q_1(t)\omega_x(t) + q_2(t)\omega_y(t) + q_3(t)\omega_z(t)\right),\\
\dot{q}_1 &= 0{,}5\left(q_0(t)\omega_x(t) + q_2(t)\omega_z(t) - q_3(t)\omega_y(t)\right),\\
\dot{q}_2 &= 0{,}5\left(q_0(t)\omega_y(t) - q_1(t)\omega_z(t) + q_3(t)\omega_x(t)\right),\\
\dot{q}_3 &= 0{,}5\left(q_0(t)\omega_z(t) + q_1(t)\omega_y(t) - q_2(t)\omega_x(t)\right)
\end{aligned} \tag{14}$$

and vector $(\omega_x(t), \omega_y(t), \omega_z(t), q_0(t), q_1(t), q_2(t), q_3(t))$ is the variable $y(t)$ for Eq.(1).

The numerical integration methods are the approximate methods for solving differential equations and they have the methodical errors. In addition to these errors the computer realizations of numerical integration methods have the rounding errors. So, after each integration step a quaternion will contain the methodical and rounding errors. Denote this errors by $\delta\mathbf{\Lambda}$, then the quaternion $\mathbf{Q}$ can be represented as

$$\mathbf{Q} = \mathbf{\Lambda} \circ \delta\mathbf{\Lambda}, \tag{15}$$

where the $\mathbf{\Lambda} = \lambda_0(t) + \lambda_1(t)\bar{i}_1 + \lambda_2(t)\bar{i}_2 + \lambda_3(t)\bar{i}_3$ is the exact quaternion and $\delta\mathbf{\Lambda} = 1 + \delta\lambda_0(t) + \delta\lambda_1(t)\bar{i}_1 + \delta\lambda_2(t)\bar{i}_2 + \delta\lambda_3(t)\bar{i}$. To reduce the accumulation of the integration errors it is recommended to update quaternion at each time step as follows (Andrle & Crassidis, 2013)

$$\mathbf{Q}_n \leftarrow \mathbf{Q}_n / |\mathbf{Q}_n|, \tag{16}$$

where $\mathbf{Q}_n$ is the result of integration of Eq.(13) at the n-th block (see Figure 3) end $|\mathbf{Q}_n| = \sqrt{q_0^2(t) + q_1^2(t) + q_2^2(t) + q_3^2(t)}$ is its quaternion norm.

To verify a programmatic realization of the algorithm and to evaluate the accuracy of a method of integration it is expedient to use a comparison of the numerical computer solution with the analytical solution. As noted above kinematics equation Eq.(13) does not have an analytical solution in general case, but there are solutions for some specific cases. Here an analytical solution for conical motion of spacecraft around of a fixed axis was used and it is assumed that a spacecraft is a symmetrical perfectly rigid body and

there are no external torques acting on this body. The spacecraft parameters and the initial conditions for simulation and the formulae of analytical solution are presented below without a detailed derivation.

The principal moments of inertia are $J_x = 2{,}0\,\mathrm{kg}\cdot\mathrm{m}^2$, $J_y = J_z = 100\,\mathrm{kg}\cdot\mathrm{m}^2$, the rest moments of matrix inertia are zero. The external torques $\boldsymbol{M}$ are zero. The initial values of components of angular velocity are $\omega_x(0) = \omega_y(0) = \omega_z(0) = 0{,}01\,\mathrm{rad}\cdot\mathrm{s}^{-1}$. $\mathbf{Q}_0 = (0.9238564387, 0.38267387, -0.0065323249, 0.0027057776)$ is the initial quaternion. Then the components of angular velocity are calculated according to the formulas:

$$\omega_x(t) = \omega_x(0),\ \omega_y(t) = \Omega\sin(\nu t + \nu_0),\ \omega_z(t) = \Omega\cos(\nu t + \nu_0), \tag{17}$$

where $\Omega = \sqrt{\omega_y^2(0) + \omega_z^2(0)}$, $\nu_0 = \arctan\dfrac{\omega_y(0)}{\omega_z(0)}$, $\nu = \dfrac{J_y - J_x}{J_y}\omega_x(0)$.

A rotation quaternion can be the product of three quaternions:

$$\boldsymbol{\Lambda} = \boldsymbol{\Lambda}_0 \circ \mathbf{M} \circ \mathbf{P}, \tag{18}$$

where $\boldsymbol{\Lambda}_0 = \mathbf{Q}_0$,

$$\mathbf{M} = \mu_0(t) + \mu_1(t)\bar{i}_1 + \mu_2(t)\bar{i}_2 + \mu_3(t)\bar{i}_3,$$

$$\mathbf{P} = \rho_0(t) + \rho_1(t)\bar{i}_1 + \rho_2(t)\bar{i}_2 + \rho_3(t)\bar{i}_3,$$

$$\mu_0(t) = \cos\frac{\sqrt{\theta}}{2}t,\ \mu_1(t) = \frac{\omega_x(0) - \nu}{\sqrt{\theta}}\sin\frac{\sqrt{\theta}}{2}t,$$

$$\mu_2(t) = \frac{\omega_y(0)}{\sqrt{\theta}}\sin\frac{\sqrt{\theta}}{2}t,\ \mu_3(t) = \frac{\omega_z(0)}{\sqrt{\theta}}\sin\frac{\sqrt{\theta}}{2}t,$$

$$\rho_0(t) = \cos\frac{\nu}{2}t,\ \rho_1(t) = \sin\frac{\nu}{2}t,\ \rho_2(t) = \rho_3(t) = 0,$$

$$\theta = (\omega_x(0) - \nu)^2 + \omega_y^2(0) + \omega_z^2(0).$$

The error estimation $\delta\boldsymbol{\Lambda}$ between the analytical solution and the numerical solution is calculated by multiplying the conjugate of the $\boldsymbol{\Lambda}$, denoted as $\tilde{\boldsymbol{\Lambda}}$, with $\mathbf{Q}$:

$$\delta\boldsymbol{\Lambda} = \tilde{\boldsymbol{\Lambda}} \circ \mathbf{Q}, \tag{19}$$

where $\tilde{\boldsymbol{\Lambda}} = \lambda_0(t) - \lambda_1(t)\bar{i}_1 - \lambda_2(t)\bar{i}_2 - \lambda_3(t)\bar{i}_3$. In the case of a small difference between quaternions $\boldsymbol{\Lambda}$ and $\mathbf{Q}$ an error quaternion can be expressed using Euler's angles:

$$\delta\boldsymbol{\Lambda} = 1 + \frac{\varepsilon_1(t)}{2}\bar{i}_1 + \frac{\varepsilon_2(t)}{2}\bar{i}_2 + \frac{\varepsilon_3(t)}{2}\bar{i}_3, \tag{20}$$

where $\varepsilon_1 = \varphi - \varphi'$, $\varepsilon_2 = \psi - \psi'$, $\varepsilon_1 = \vartheta - \vartheta'$ are small differences between respective angles of analytical and calculated spacecraft attitude. From this, in practice, convenient to use the following formula to calculate the error

$$\varepsilon = \sqrt{\varepsilon_1^2(t) + \varepsilon_2^2(t) + \varepsilon_3^2(t)}. \tag{21}$$

DISCUSSION OF RESULTS

Table 1 shows some results of error evaluation of the proposed algorithm obtained by comparing the numerical solution of Eq.(14) with the analytical solution (Eq.(18)) using the formula Eq.(21).

Table 1. The maximum errors of numerical solution Eq.(14) for the time simulation of spacecraft attitude motion $T = 1000\,s$

S	$\varepsilon(h=1{,}0)$	$\varepsilon(h=0{,}5)$	$\varepsilon(h=0{,}1)$
3	$9{,}72\cdot10^{-7}$	$5{,}94\cdot10^{-8}$	$9{,}25\cdot10^{-11}$
4	$8{,}69\cdot10^{-9}$	$2{,}75\cdot10^{-10}$	$7{,}63\cdot10^{-13}$
5	$4{,}34\cdot10^{-11}$	$6{,}69\cdot10^{-13}$	$6{,}85\cdot10^{-13}$
6	$4{,}43\cdot10^{-13}$	$2{,}22\cdot10^{-14}$	$3{,}35\cdot10^{-13}$

Any parallel computations have some waste of time to data exchange between multiple processors, processes or threads, to their synchronization (Eyerman & Eeckhout, 2010) and other overhead. Parallel algorithm can be effective if the time required for calculations is significantly exceed the time required to synchronize threads. From the formulae of calculation scheme (Eq.(7) - Eq.(11)) it is clear that the bulk of the calculations relates to calculating of the RHS of Eq.(1). Indeed, it is necessary to calculate $F(t)$ in each block $4S+1$ times and if the calculations are distributed among the four threads then according to the algorithms (Figure 1 - Figure 3) the Thread-2, Thread-3, Thread-4 calculate the function $F(t)$ the S times, and Thread-1 calculate this function $S+1$ times. Thus, if each thread is executed on separate core of multi-core processor, then theoretical speedup S_P of the parallel implementation of the algorithm may be

$$S_P = (4S+1)/(S+1). \tag{22}$$

For example, if $S = 6$ then theoretical speedup is 3,7 on quad-core processor. The similar estimates of speedup can be obtained by using known Amdahl's Law.

Figure 4 shows the speedup estimates of the proposed algorithm obtained on quad-core processor AMD A6-3650 APU 2,60 GHz. As can be seen from the graph of this figure the speedup approaches the estimate by Eq.(22) with an increase in the proportion of time spent on calculating $F(t)$ of the total time calculating.

If the number of threads is less than the number of processor cores and the processor utilizes Hyper-Threading (HT) technology then theoretical speedup can be estimated by modified Amdahl's Law (Akhter & Roberts, 2006):

$$S_{HT} = \frac{1}{\alpha + 0{,}67\dfrac{1-\alpha}{n} + H(n)}, \tag{23}$$

where α is the time spent executing the serial portion of parallel algorithm, i.e. on one core; n is the number of logical cores and $H(n)$ is the overhead. Figure 5 shows the speedup estimates obtained on dual-core processor that utilizes HT technology Intel® Core™ i3-2310M @ CPU 2.10 GHz. In this case number of logical cores $n = 4$, $\alpha = 1/25$ because the function $F(t)$ is calculated 25 times for $S = 6$ according to the algorithm in Figure 1 and if to ignore overhead then according to Eq.(23) $S_{HT} \approx 2{,}5$.

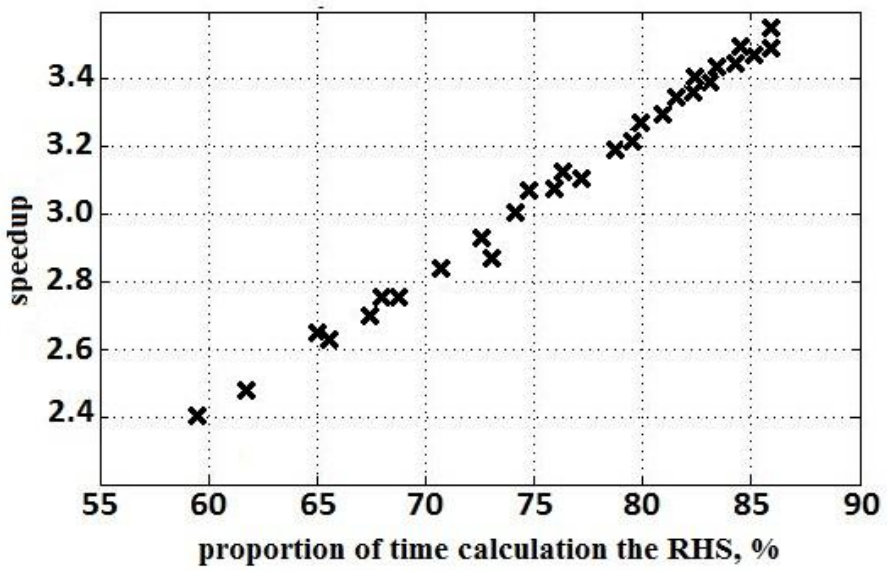

Figure 4. Speedup of parallel algorithm in quad-core processor

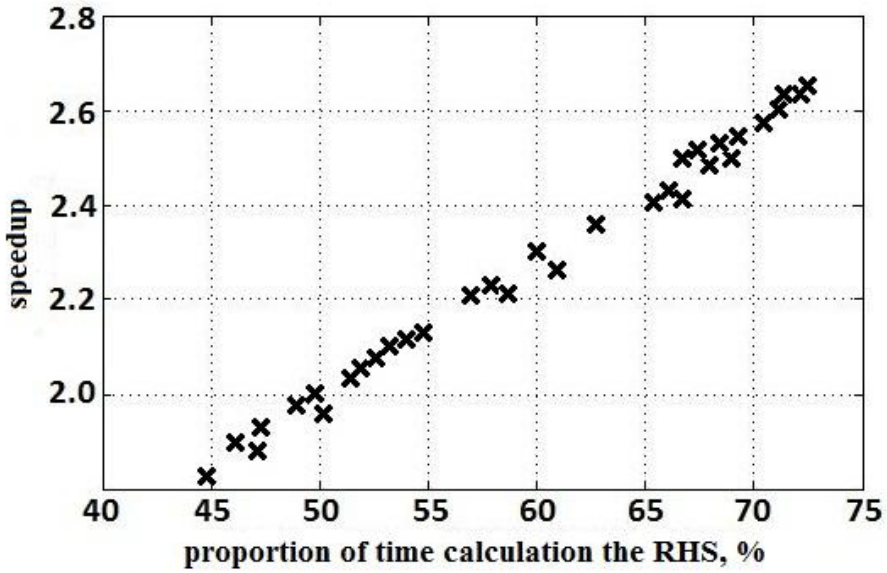

Figure 5. Speedup of parallel algorithm in dual-core processor with HT technology

CONCLUSION

The multi-threaded parallel implementation of the four-point block one-step integration method has been proposed. For this purpose the blocking queues for the data exchange synchronization between threads have been used. The effectiveness of proposed algorithm has been proved for the case, when ODE's RHS-related calculation time exceeds the time required for synchronization. Algorithm verification has been conducted by way of spacecraft attitude motion simulation. The model of spacecraft attitude motion and the analytical solution used for the purpose of proposed algorithm verification can be applicable for numerical integration methods study and for appropriate practical programmatic implementations debugging.

REFERENCES

Ahnert, K., Demidov, D. & Mulansky, M. (2014). Solving Ordinary Differential Equations on GPUs. In V. Kindratenko (Ed.), *Numerical Computations with GPUs*. Springer International Publishing, Cham. 125-157. doi: 10.1007/978-3-319-06548-9_7

Akhter, S. & Roberts, J. (2006). *Multi-core Programming: Increasing Performance Through Software Multi-threading*. Intel Press

Al-Omari, A., Arnold, J., Taha, T. & Schüttler, H.-B. (2013). Solving large nonlinear systems of first-order ordinary differential equations with hierarchical structure using multi-GPGPUs and an Adaptive Runge Kutta ODE solver. *Access IEEE*, 1, 770–777. doi: 10.1109/ACCESS.2013.2290623

Andrle, M. S. & Crassidis, J. L. (2013). Geometric Integration of Quaternions. *Journal of Guidance, Control, and Dynamics*, *36*(6), 1762-1767. doi: 10.2514/1.58558

Branets, V. N. & Shmyglevskii, I. P. (1973). *Application of Quaternions in Problems of Orientation of a Rigid Body.* Nauka, Moscow. [in Russian]

Carl, J.D. & Biswas, G. (2016). An approach to parallel simulation of ordinary differential equations. *Journal of Software Engineering and Applications, 9*, 250–290. doi: 10.4236/jsea.2016.95019

Cooper, K. D. (2014). Making effective use of multicore systems: A software perspective: the multicore transformation (Ubiquity symposium). Ubiquity 2014, 1–8. doi:10.1145/2618407

Eyerman, S. & Eeckhout, L. (2010). Modeling critical sections in Amdahl's law and its implications for multicore design. *SIGARCH Comput. Aschit. News, 38(3)*, 362-370. doi: 10.1145/1815961.1816011

Feldman, L. P. & Dmitrieva O. A. (2001). Effective methods of multisequencing of Caushi problem's numeral decision for ordinary differential equations. *Matem. Mod., 37(7)*, 66–72. [in Russian]

Ishak, F. & Suleiman, M. B. (2012). Parallel method using MPI for solving large systems of delay differential equations. *CHUSER 2012 – 2012 IEEE Colloquium on Humanities, Science and Engineering Research*, 255–259. doi: 10.1109/CHUSER.2012.6504320

Majid, Z.A., Mehrkanoon, S.M., & Othman, K.I. (2010). Parallel block method for solving large systems of ODEs using MPI. In *Proceedings of the 4th international conference on Applied mathematics, simulation, modelling*. World Scientific and Engineering Academy and Society (WSEAS), 34-38

Majid, Z.A. & Suleiman, M. (2009). Parallel direct integration variable step block method for solving large system of higher order ordinary differential equations. In *Advanced Technologies*. InTech, 47–56. doi: 10.5772/8201

Waeleh, N. & Majid, Z.A. (2016). A 4-point block method for solving higher order ordinary differential equations directly. *International Journal of Mathematics and Mathematical Sciences, vol. 2016*, 8 pages. doi: 10.1155/2016/9823147

Waeleh, N. & Majid, Z.A. (2017). Numerical algorithm of block method for general second order ODEs using variable step size. *Sains Malaysiana, 46(5)*, 817–824. doi: 10.17576/jsm-2017-4605-16