The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

NUMERICAL METHODS FORSOLVING DIFFERENTIAL EQUATIONS

Practice 1. Numerical Solution of Ordinary Differential Equations as Illustrated by Brain Modeling Problem

Nizhni Novgorod

2014

OBJECTIVES

The purpose of this practice is the introduction to parallel numerical methods for solving systems of ordinary differential equations as illustrated by the brain neural activity modeling problem.

ABSTRACT

This practice gives a concise introduction into the problems of brain modeling, defines the respective mathematical model in a form of an ODE system and summarizes the Euler method for solving ODE systems. The practice describes a sequential and OpenMP-based parallel implementation, gives scalability analysis and features an implementation modified using the Intel MKL random number generator. It lists the experimental results and gives a conceptual interpretation of the model problem.

GUIDELINES

Successful applications of mathematical modeling and theory of nonlinear oscillations include neurodynamics and brain sciences.The human brain has about 10^{11} neurons - electrically excitable cells ensuring a number of complex cognitive functions as a result of their interaction. Neurons interact by means of synaptic connections that conduct nerve impulses - electrical signals resulting from sudden and strong changes in the voltage difference inside and outside the cell [1].

Currently, the most used neural activity model class in theoretical and computational neuroscience are models formulated as ordinary differential equations which describe the flow of ions through the cell membrane and change in the transmembrane potential. This class of models is called Hodgkin–Huxley models [3]. A complex model dynamics results from non-linear dependence of ion channel conductance on the transmembrane potential. The dynamics of a neuron network of *N* elements subject to a synaptic connection is described by a system of *5N* ordinary differential equations:

- Main equation of changes in the transmembrane potential V_i:

$$C_m \frac{dV_i}{dt} = I_l + I_{Na} + I_K + I_i^{ext} + I_i^{sin}, \qquad i = 1, \dots, N.$$

- 3 equations describing dependence of ion channel conductance on transmembrane potential. These correspond to time variation of the portal variables m_i, h_i, n_i ;
- the equation of synaptic connection activation resulting from changes in the membrane potential V_i (corresponds to changes in the dynamic variables r_i).

This practice proposes to use the Euler method for numerical ODE solution.

It includes a sequential implementation of brain modeling. The process of computation involves two embedded loops: the outer loop through time moments while the inner loop through neurons with one step of the Euler method corresponding to each neuron. For numerical solution of a system consisting of *5N* equations using the Euler method, one has to use two vectors sized *5N* that correspond to the required function value at the current and previous instants of time. This problem requires copies for the arrays **`V`** and **`r`** as at each iteration the values of their elements are used both as input data and operation results. Equations for m_i, h_i, n_i do not depend on the other system equations, thus the arrays **`m`**, **`h`**, **`n`** can be updated without allocating additional memory. To simulate random noise for I^{ext}, use the PRN (pseudorandom) generator **`rand()`**; the corresponding element of the array **`l_ext`** is initialized at each inner (neuron) loop iteration.

Then develop the first parallel version of the solution using OpenMP. For this purpose, use the directive **`pragma omp parallel for`** to distribute neuron loop iterations among threads. Results of this implementation are different from those of the sequential version and do not change from run to run. This is due to PRNG incorrect use by the parallel program. The function **`rand()`** is implemented so that each thread uses its own initializing value; this means that in a multithreaded program the number of generators will actually be the same to that of threads. As for the implementation in question, each new thread used the same value as the main thread for initialization, which means that for each thread the same number sequence was obtained.

Modify the parallel version: implement the function **`GenIExt()`** where all **`I_ext`** elements are initialized. The function is called before the neuron loop start, i. e. before the parallel region. If done this way, the parallel algorithm implementation will run correctly. The practice lists experimental results for this program version. For 8 threads, speed up does not reach 6. It should be noted that in case of such parallelization for each specific problem, speed up will depend on the relation of the number generation time to other computations. This is why the use of the sequential PRN generator may dramatically restrict the implementation scalability.

Now let us review a modified parallel version of the program. To generate pseudorandom numbers, this version uses the IntelMKL-MCG59 parallel generator [6, 6]. For this purpose use a PRN stream – a service variable of **`VSLStreamStatePtr`** type . To use the generator in the sequential mode, call the following functions:

1. **`vslNewStream()`** is the basic generator initialization. The function inputs are the pointer to the stream variable , basic generator type and initial value to initialize the generator;
2. **`vdRngUniform()`** is the uniform distribution generation. The function inputs are the invariable describing the method (**`VSL_BRNG_MCG59`** in our case), stream variable, number of random values to be generated (the generator enables obtaining several random numbers per call), buffer itself and distribution parameters (this time bounds of the distribution interval);
3. **`vslDeleteStream()`** is the release of service variables. The function input is the stream variable.

For the parallel version, to obtain random values from one and the same sequence starting from various numbers, **`call vslSkipAheadStream()`** before PRN generation. Its inputs are the PRN stream and number of random values to be skipped starting from the current PRN of the sequence.

In this implementation, at each time loop iteration **`N / NumThreads`** pseudorandom numbers are to be obtained for each thread subject to the number of threads. The parallel section is declared at the beginning of the main computational function before the time loop. This function creates individual PRN generators for each thread. In this case the second parameter for **`vslSkipAheadStream()`** equals **`N /NumThreads * num`** where **`num`** is the thread number . Upon completion of one iteration of the outer time loop, each thread must skip those numbers that were received by other threads. For this purpose, recall again **`vslSkipAheadStream()`** with its second parameter equal to **`N - N / NumThreads`**. Part of the array **`I_err`** will be used as a buffer for **`vdRngUniform()`** pseudorandom numbers.

Experimental results showed that the modified parallel version speed up reaches 7 times for 8 threads thanks to the use of more than one PRN generator.

The lecture also interprets the results of a numerical experiment that studied the dynamics of a network of 1000 neurons depending on the degree of synaptic connection between them. While the neurons were not interconnected, the network demonstrated asynchronous oscillations. Such

a behavior is due to the fact that each cell has its own oscillation frequency different from that of most other neurons due to inhomogeneous distribution of I_i^{ext} within the ensemble. When the connection is activated, the network dynamics show considerable changes. As a result of interactions, the synchronous oscillation mode set in the system with all system elements divided into three unequal groups: the first group contains neurons that do not oscillate, the second and the third one make two different clusters of elements that oscillate in a synchronous way. In this case, neurons from different clusters generate pulses in phase opposition. Therefore, despite initial inhomogeneity and different own oscillation frequency of neurons, synaptic connection led to partial synchronization of neurons which adopted the same oscillation rhythm.

RECOMMENDATIONS FOR STUDENTS

See [1 – 3] for the fundamentals of dynamic brain modeling. Methods for numerical solution of ODEs are described in [5].For tips on how to use MKL PRN generators, see [6, 6].

REFERENCES

1. Rabinovich M.I., VaronaP., Selverston A.I., Abarbanel H.D.I. Dynamical Principles in Neuroscience // Review of Modern Physics, vol. 78(4), 1213(53), 2006.
2. Nicholls J.G., Martin A.R., Fuchs P.A., Brown D.A., Diamond M.E., Weisblat D. From Neuron to Brain, 5th edition. Sunderland: Sinauer Associates Inc., 2011.
3. Izhikevich E.M. Dynamical systems in neuroscience: geometry of excitability and bursting. – MIT Press, 2006.
4. Butcher J.C. Numerical Methods for Ordinary Differential Equations. New York: John Wiley & Sons, 2003.
5. Hoffman J.D. Numerical Methods for Engineers and Scientists, 2nd Edition. New York: CRC Press, 2001.
6. Intel Math Kernel Library Reference Manual. [http://software.intel.com/en-us/mkl_11.2_ref_pdf].
7. Intel Vector Statistical Library Notes. [http://software.intel.com/en-us/mkl_11.1_vslnotes_pdf]

PRACTICE

1. Implement the uniform distribution generators MCG31 and MCG59 and the Skip-ahead technique of their use for parallel computation. Check the generators for correct results.
2. Develop a parallel implementation using Intel Cilk Plus. Perform computational experiments. Analyze results correctness and speed up.
3. Develop a parallel implementation using IntelTBB. Perform computational experiments. Analyze results correctness and speed up.
4. Use Intel Amplifier XE to identify the most time-consuming code sections. Think about possible optimization.

TEST

1. What do you call the variables describing dependence of ion channel conductance on transmembrane potential?
    a. + Portal
    b. Ion
    c. Channel

2. How many equations are there in the system that describes dynamics of a network of N neurons?
    a. N
    b. $3N$
    c. $+ 5N$
3. What type of parallelization is acceptable for the Euler method implementation to solve the considered problem?
    a. Outer time loop parallelization
    b. + Inner neuron loop parallelization
    c. Parallelization of the next approximation computation for an individual neuron
4. Why did the computational results of the first parallel version differ from those of the consecutive version?
    a. Data race took place
    b. Different PRN generators were used for different threads
    c. + One and the same PRN sequence was used for all threads
5. How does the Open MP directive **`omp single`** work?
    a. The structured code block will be executed by one thread only (zero thread). There is an implicit barrier at the end of the **`single`** block.
    b. + The struclured code block will be executed by one thread only (any thread). There is an implicit barrier at the end of the **`single`** block.
    c. The structured code block will be executed by one thread only (any thread).There is no barriers at the end of block, other threads will continue computations.
6. Why was it proposed to execute the updated value copying cycle in one thread?
    a. due to possible data races
    b. + due to low computational load on threads
    c. due to the necessity of thread synchronization
7. Where should one call **`omp_get_num_threads()`** to obtain the number of threads to execute the code block?
    a. + In the parallel section
    b. Before the parallel section
    c. After the parallel section
8. Which is NOT an input parameter for **`vslNewStream()`**?
    a. Initial value to initialize the generator
    b. + PRN stream variable
    c. Generator type
9. What is an input parameter of **`vslSkipAheadStream()?`**
    a. Number of values to be obtained by this thread
    b. +number of values to be skipped starting from the current PRN of the sequence.
    c. thread number
10. Why does correct operation of the parallel program require calling **`vslSkipAheadStream()`** at the end of time loop iteration?
    a. +Each thread has to skip part of the numbers from the PRN stream which was received by other threads by the end of the loop
    b. Change the number of values received for this thread
    c. Each thread has to generate PRNs again