AL-FARABI KAZAKH NATIONAL UNIVERSITY

A. S. Kussainov

SELECTED PROBLEMS IN PARALLEL COMPUTING AND MULTITHREADING IN NUCLEAR PHYSICS

Educational-methodical manual

Almaty
«Qazaq University»
2017

UDC
LBC
K

*Recommended for publication by the decision
of the Academic Council of the Faculty of Physics and Technology,
Editorial and Publishing Council
of Al-Farabi Kazakh National University
(Protocol №3 dated 07.12.2017)*

Reviewer
Doctor PhD ***M.M. Muratov***

Kussainov A.S.

Selected problems in parallel computing and multithreading in nuclear physics: educational-methodical manual / A.S. Kussainov. – Almaty: Qazaq university, 2017. – 98 p.

ISBN 978-601-04-3128-7

A wide range of different problems of nuclear physics, parallelized in the Linux and windows operating environment, ready for studying by students within the framework of special courses on programming, numerical methods and computer modeling, is considered at the modern level. Questions and independent exercises for modification of the proposed program code are given. The text of the programs is also available in electronic form for downloading and further independent work.

This publication was supported by grant №3824/ГФ4 provided by the Science Committee at the Ministry of Science and Education of Republic of Kazakhstan to the principal investigator at the National Nanotechnology Laboratory of Open Type, Physics and Technology Department, al-Farabi Kazakh National University.

Published in authorial release.

UDC
LBC

ISBN 978-601-04-3128-7

INTRODUCTION

The main and the only way to improve the performance of processors by increasing the CPU internal frequency, was augmented and replaced at some point by increase in the number of cores in CPU. Roughly speaking, processor manufacturers have learned how to put several processors on a single chip. Now almost any computer is equipped with a multi-core processor. Even the entry-level desktop systems have two cores. Normally the regular desktop systems have four-and eight-core CPUs. If Moore's Law to keeps its rule over the industry, then within next five years the average computer will have 16, or even 32 cores on the chip.

The problem is that the software industry does not yet have time to make up for the available hardware, and only small part of the applications can effectively use the resources of the multi-core processors. Each program has one main execution thread – a set of instructions that are executed sequentially one after another. Naturally, in this case, one core of the processor is involved. The programmer must take care of loading the rest of the cores with some other work, in other words, he must make sure that some instructions are executed not sequentially, but simultaneously – in parallel mode.

It should be noted that performance does not increase linearly with the number of cores. That is, the use of four cores does not guarantee a fourfold increase in productivity. Nevertheless, there is an increase, and every year it will be biggeras there will be more programs optimized for the multi-core processors.

How can programmers manage threads to use the full power of the processor? How to make the application work as fast as possible, scaled with the increase in the number of cores, and to write such an application was not a nightmare programmer? One option is to manually create threads in the code, give them tasks to perform, then delete them. But in this case you need to take care of one very important thing – synchronization. If one task requires data that is being counted by another task, the situation becomes more complicated. It is difficult to understand what happens when different threads at the same time try to change the values of common variables. And if one does not want to manually create threads and delegate tasks to them then the various libraries and standards for parallel programming come to the rescue. Let's consider more in detail the most widespread standard for parallelization of programs in languages C, C ++, Fortran and OpenMP.

In the first four chapters we will be talking about the OpenMPI implementation of the parallel computing on Linux. The last two chapters are about OpenMP and Windows applications.

1

DEPLOYMENT AND TESTING OF THE PARALLEL ENVIRONMENT FOR THE QUANTUM VARIATIONAL MONTE CARLO METHOD

Below, a brief course of action is given to guide a beginner programmer towards the basics of parallel and multithread programming. First it is explained how to quickly deploy parallel programming environment, compile and run your own parallel program and analyze the data and performance. The *OpenMPI* implementation of the Message Passing Interface (*MPI*) protocol has been used as an example of multithreading and a tool to increase performance of the code on the multicore systems. Variational Monte Carlo Method to solve Schrodinger equation for the quantum harmonic oscillator is implemented numerically. The transition between the single thread standalone *C++* program to the more efficient and faster parallel program is given with explanation.

Introduction

High performance personal desktops or workstations, as well as clusters and high performance computing systems which could be accessed remotely, became widely available nowadays. Many of these are state-of-the-art, expensive machines maintained by the numerous staff and capable to address the fundamental pure and applied science problems of our days [1-2]. In case of the later ones, the researcher could directly proceed to the coding of the parallel program, delegating the tedious task of network and parallel environment configuration to a network administrator. This type of IT professionals, though possessing the vast knowledge and experience in networking protocols, most likely will make you to work with the closed box solution, even though the creation and configuration of your own computational cluster is relatively easy task for today's scientist.

To fill this gap in the professional IT training the multiple studies and help resources have been printed and circulated among the interested scientists and researches [3, 4]. Research universities around the world including Kazakhstan are in possession or include in their strategic development plans supercomputers and HPC (high-performance computing) systems [5].

There is another important moment to address. Many beginner programmers though have in their possession the state-of-the-art multicore computing system are able to utilize only a small portion of its computational power. All modern computers have the multicore central processing units (CPUs). Unless you have a sophisticated compiler which could parse your, let us say *C++*, code to run simultaneously on all the cores, regularly you will have only one of eight cores, for octet CPU, to handle your task. You need to multithread your application by yourself because each thread of execution can only saturate one core [6].

These two moments are addressed at once by using the *OpenMPI* library [7] which can handle multithread coding and feed it to a multicore CPU or distribute the tasks across the network of computers connected into a computational cluster. Unlike the similar, *OpenMP*, development of the message parsing protocols [8] it is mainly and extensively documented in electronic resources and much easier to deploy for the beginner.

Additionally, one could make himself comfortable with parallel programing before attempting to learn and configure the complex networks and investing in buying equipment or machine time on high-performance computing systems.

We have selected variational Monte Carlo method for the Schrodinger equation [9] of the quantum harmonic oscillator to implement numerically as an excellent example of multithreading and performance optimization in scientific computing [10].

Methods

In our work the *Debian* distribution of the *Linux* class operational systems was made an operational system of choice mainly because of its flexibility, freeware nature, and more than modest requirements for an existing hardware. You will have multiple versions available for download free of charge at [11]. To successfully run a fully functional version with graphical desktop, parallel computation library, ssh client/server *etc* we need the modest 256 megabytes of RAM and several gigabytes of hard disk space. The *Windows OS* users may install an *Oracle virtualbox* software [12] and populate it with any *Linux* installation of their choice or install a second OS with ability to select it at the boot time.

One should take caution to install software and OS of the same register size, that is 32-bit or 64-bit (*amd64* or *i386* packages), across your complex computational system and virtual environment.

Please be aware, that if your computer is old enough you may came across the problem of it not supporting 64-bit software and virtualization technology. From now on, we assume that your computer's CPU has more than one core.

Using *OpenMPI* is the easiest and quickest way to learn parallel programming and maximize the performance of your desktop system. This package is always included in full installation DVDs, CDs or online depositories and may be installed by the following command:

```
$ apt-get install openmpi-bin openmpi-common libopenmpi    (1)
                    1.6 libopenmpi-dev
```

Depending on the state of your system, some of these components may already be installed or unavailable and you will be offered with an alternative.

For a numerical problem to calculate we chose the variational Monte Carlo method to solve the Schrodinger equation for the harmonic oscillator. We start with the following, one dimensional, time independent Schrodinger equation

$$-\frac{\hbar^2}{2m}\frac{d^2\varphi(x)}{dx^2}+\frac{1}{2}mk^2x^2\varphi(x)=E\varphi(x) \tag{2}$$

where k is the force constant. If we know the complete set of N eigenfunctions in the following form

$$\Phi(x) = \sum_{i=0}^{N-1} c_i \varphi_i(x) \tag{3}$$

the average value of energy $<E>$ will be given by the these expressions

$$\langle E \rangle = \frac{\int_{-\infty}^{+\infty} dx \Phi(x)^* H \Phi(x)}{\int_{-\infty}^{+\infty} dx \Phi(x)^* \Phi(x)} = E_0 + \frac{\sum_{i=0}^{N-1} |c_i|^2 (E_i - E_0)}{\sum_{i=0}^{N-1} |c_i|^2} \tag{4}$$

here H is the Hamiltonian given by the left part of equation (2) and E_0 is the ground state energy. Variational Monte Carlo method uses equation (4) as a starting point. Replacing the unknown set $\Phi(x)$ of eigenfunctionsby the trial wavefunction $\Phi_{T,a}(x)$ we are then varying he parameter a, see equation (5). If we are lucky, the calculated average energy at the local minimum will give as a ground state value and corresponding characteristic wave function.

$$\langle E \rangle = \int_{-\infty}^{+\infty} dx \omega(x) E_L(x), \text{ where } \omega(x) = \frac{|\Phi_{T,a}(x)|^2}{\int_{-\infty}^{+\infty} dx' |\Phi_{T,a}(x')|^2} \text{ and } E_L(x) = \frac{H\Phi_{T,a}(x)}{\Phi_{T,a}(x)} \tag{5}$$

One of the main features of this method is that we do not sample the whole configuration space from minus to plus infinity indiscriminately rather than traversing it in the manner described as Metropolis-Hastings algorithm. Target wave function’s tails go to zero pretty fast even not far away from the origin and its overall shape is similar to the normal distribution. The Metropolis-Hastings algorithm [13], which samples the space according to the weight function $\omega(x)$ in equation (5), is a Markov chain Monte Carlo (MCMC) algorithm.

We place n of the so called *walkers* randomly and uniformly across the selected region at coordinates $(x_n)_t$ (index t stands for the current state of walkers’ coordinates). The next set of coordinates $(x_n)_{t+1}$ of walkers, who are sampling the integral in equation (4) independently, is determined by the following set of rules:

$$x_{t+1} = x_t + \delta \,,$$
$$\text{if } \frac{\omega(x_{t+1})}{\omega(x_t)} \geq 1 \text{ the step is accepted },$$
$$\text{if } \frac{\omega(x_{t+1})}{\omega(x_t)} < 1 \text{ accept it only if it's larger than random number on } (0,1) \tag{6}$$

where δ is the size of step which is determined based on the problem's conditions. If this step is accepted the local energy $E_L(x)$ calculated at this walker's coordinate contributes to the integral in equation (5).

If the trial function chosen to be $\Phi_{T,a}(x) = exp(-ax^2)$, the value of $<E>$ is than calculated according to the formula

$$\langle E \rangle = \frac{1}{counts} \sum_{i=1}^{counts} E_L(x_i) \text{ where } E_L(x) = a + x^2(0.5 - 2a^2) \tag{7}$$

where *counts* are the number of accepted steps across all walkers' trajectories.

Here we used expression for E_L derived from equation (5). We also assume that in Hamiltonian, see equation (2), we choose the values of $\hbar=m=k=1$. It is very neat numerical problem to implement in a single thread and in parallel.

Results and Discussions

The problem itself is pretty straightforward to formulate using *C++* programming language, see Table 1. We have structured the code in such a way that if one decides to comment out the code lines printed in boldface font he will end up with the regular single thread program. The parallel version is compiled and run in 8 threads through the following set of commands

```
$ mpicxx \-o Jan06_vmc_paral Jan06_vmc_paral.cpp
$ mpirun -np 8 Jan06_vmc_paral
```
(8)

while the single thread version is compiled and run in a more conventional fashion

```
$ g++ -o arman_vmc arman_vmc.cpp
$./arman_vmc                                  (9)
```

If you are running a parallel program on a cluster of individual machines the second line in equation (8) is replaced with

```
$ mpirun --hostfile my_hostfile -np 8 Jan06_vmc_paral        (10)
```

where the text file *my_hostfile* specifies the configuration of your network and the number of cores per individual CPU.

The configuration of our computer is listed as follows: Intel Core i7 4790K, 4.0GHz/LGA-1150/22nm/Haswell/8Mb L3 Cache/IntelHD4600/EM64T, DDR-3 DIMM 16Gb/1866MHz PC14900 Kingston HyperX Fury Black, 2x8Gb Kit, CL10.

The main features of our code are the following: we split, see the lines 41-53, the total number of walkers uniformly between the claimed number of threads, which is eight, see equation (8); then we used the Box-Muller transform [14] to generate the pairs of independent, standard, normally distributed with zero expectation and unit variance random numbers, given a source of uniformly distributed random numbers from standard *C++* rand() generator, see the lines 32-33; the *MPI_Scatter* and *MPI_Gather* were the *MPI* functions to facilitated our data exchange between the threads, see the lines 52 and 62. We have not made an additional effort to optimize the step's size or the standard random generator's quality.

Table 1

C++ code used to calculate the range of ground state energy values as a function of parameter a. In bold font are given the pieces of code which convert a single thread program to a multithread.

```
1   //header file to provide parallel programing environment
2   #include <mpi.h>

3   #include <cstdlib>
4   #include <iostream>
5   #include <cmath>
6   using namespace std;
```

```
double a,e,sum_e,*x;
int c_ounter;

//walkers’ number and Monte Carlo steps values
int recv_data[2]={2400, 20000}, *send_data;

// arrays to collect energy and accepted steps data from individual parallel process
double senback_data[2], *collect_data;

int id, ntasks, len;
double w(double xt, double x) {

//the ratio of the weight function computed for consecutive x values
return exp(-2*a*(xt*xt - x*x));}

double e_nergy(double x) {

// local energy function
return a+x*x*(0.5-2*a*a);}

//function to initialize starting point for all walkers
void Assign_Positions(){
srand(time(NULL)*id+1);
x = new double [recv_data[0]];
for (int i = 0; i < recv_data[0]; i++){

//walkers uniformly distributed within (-0.5:+0.5) range
x[i] = rand()/(RAND_MAX + 1.0) - 0.5;}}

//function to move all walkers according to Metropolis algorithm
void Stir_All_Walkers(){
for(int j=0;j < recv_data[0]; j++){

// Box-Muller transform to generate normal distribution
double xt=x[j]+pow(-2.0*log(rand()/(RAND_MAX+1.0)),0.5)*
cos(2.0*3.141592*rand()/(RAND_MAX+1.0));

if(w(xt, x[j])>1){
x[j] = xt;++c_ounter;e = e_nergy(x[j]);sum_e += e;}
else{
if (w(xt,x[j])>rand()/(RAND_MAX+1.0)){
x[j] = xt;++c_ounter; e= e_nergy(x[j]);sum_e += e;}}
}}
int main(int argc, char *argv[]){
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
MPI_Comm_rank(MPI_COMM_WORLD, &id);

//subroutine to evenly distribute walkers between the claimed number of processes
if(id==0){
int WperTask = floor(recv_data[0]/ntasks);
send_data = new int [2*ntasks];
for (int i = 0; i < ntasks; i++)
{send_data[2*i]=WperTask;
if(i<=recv_data[0]%ntasks-1){send_data[2*i]++;}
send_data[2*i+1]=recv_data[1];}
collect_data = new double [2*ntasks];}

//sending assigned number of walkers and individual number of step to each process
MPI_Scatter(send_data,2,MPI_INT,recv_data,2,MPI_INT,0,MPI_COMM_WORLD);

Assign_Positions();
for(a=0.1;a<=1.5;a+=0.05){
for(int k=0;k<=floor(0.2*recv_data[1]);k++){Stir_All_Walkers();}
double avg_e=sum_e=0;c_ounter=0;
for(int k=0;k<=recv_data[1];k++){
Stir_All_Walkers();}
avg_e = sum_e/c_ounter;
senback_data[0]=sum_e;senback_data[1]=c_ounter;

//collecting data back from all processes and calculating alpha and average energy values
MPI_Gather(senback_data,2,MPI_DOUBLE,
collect_data,2,MPI_DOUBLE,0,MPI_COMM_WORLD);
if(id==0){double total_e=0;double total_count=0;
for(int i=0;i<ntasks;i++){
total_e+=collect_data[2*i];
total_count+=collect_data[2*i+1];}
cout<<a<<"\t"<<total_e/total_count<<"\n";}
c_ounter=0;}

MPI_Finalize();
exit(0);}
```

The data obtained from our simulations are plotted on Figure 1.

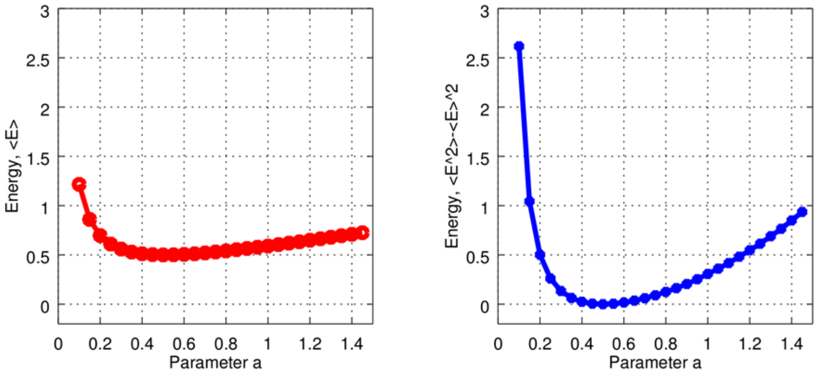


Figure 1. Numerical simulation data for the quantum harmonic oscillator. The average energy value <*E*> on the left and the quantity $<E^2>-<E>^2$ on the right are both plotted as the functions of parameter *a*

The left part represents the plot of the average energy versus parameter *a*. As expected, we can clearly see the minimum at *a*=0.5 where the local energy function has no dependence on *a*. This value of *a* corresponds to the zero variance value on the right plot which pinpoints the ground state in our problem.

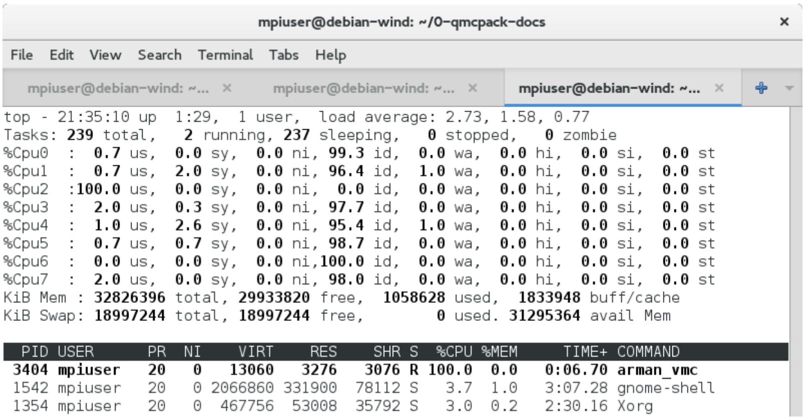
In order to test the performance improvement we have run the code listed in Table 1 with the following parameters: 2400 walkers and 20000 iterations plus the thermalization stage in 4000 steps. As we can see from Table 2, the same computational problem experienced more than fourfold increase in performance being implemented in the parallel algorithm compared to the single threaded one.

Table 2

Performance comparison between the parallel and single threaded implementations of the variational Monte Carlo algorithm

	Wall time	CPU time
Parallel algorithm timing (averaged between 8 threads). Process name is Jan06_vmc_paral	54.8723	53.709
Single threaded one. Process name is arman_vmc	275.519	275.361

For the reader's info, *CPU time* is the time which is actually spent by CPU to work on the process, while the *Wall time* additionally includes the time spent by the process in the line awaiting to be handled plus some other delays. How the work load is now uniformly spread across the processes is clearly seen from the Figures 2 and 3. The single instance of *arman_vmc* process is managed singlehandedly by one core #2, see Figure 2. While in case of the parallel program all eight cores of our CPU are loaded with their own fraction of work, running 8 instances of the *Jan06_vmc_parall* simultaneously each with their own parameter, see Figure 3.


mpiuser@debian-wind: ~/0-qmcpack-docs
File Edit View Search Terminal Tabs Help
mpiuser@debian-wind: ~...
mpiuser@debian-wind: ~...
mpiuser@debian-wind: ~...

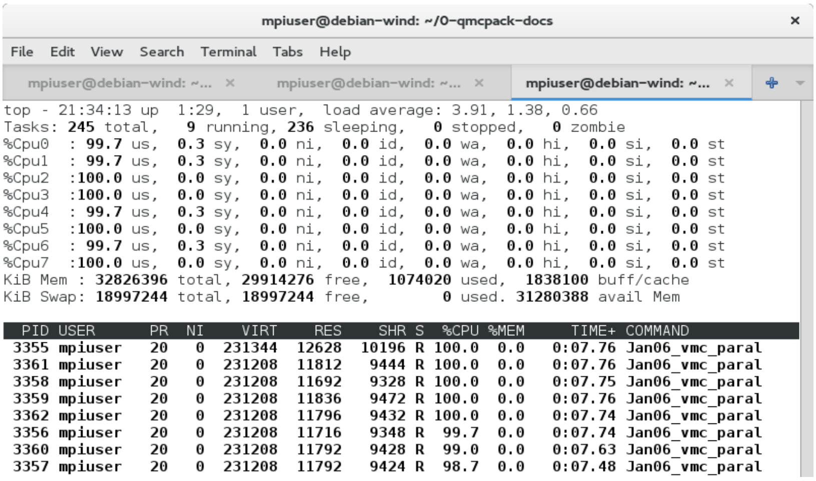
```
top - 21:35:10 up  1:29,  1 user,  load average: 2.73, 1.58, 0.77
Tasks: 239 total,   2 running, 237 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.7 us,  0.0 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.7 us,  2.0 sy,  0.0 ni, 96.4 id,  1.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  2.0 us,  0.3 sy,  0.0 ni, 97.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  1.0 us,  2.6 sy,  0.0 ni, 95.4 id,  1.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.7 us,  0.7 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  2.0 us,  0.0 sy,  0.0 ni, 98.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 32826396 total, 29933820 free,  1058628 used,  1833948 buff/cache
KiB Swap: 18997244 total, 18997244 free,        0 used. 31295364 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3404 mpiuser   20   0   13060   3276   3076 R 100.0  0.0   0:06.70 arman_vmc
 1542 mpiuser   20   0 2066860 331900  78112 S   3.7  1.0   3:07.28 gnome-shell
 1354 mpiuser   20   0  467756  53008  35792 S   3.0  0.2   2:30.16 Xorg
```

Figure 2. Output of the *top* command displaying info about the state of the individual cores of our CPU for a single thread *C++* program of variational Monte Carlo simulation

As we can see, the relatively straightforward modification of our system allows us to run multiple parallel algorithms and increase productivity of our code in many times. The coding and experience gain in such an exercise is a good start in transition to the distributed and high performance computations. Implemented variational Monte Carlo method is a key tool for many computationally extensive and effective numerical methods in science.

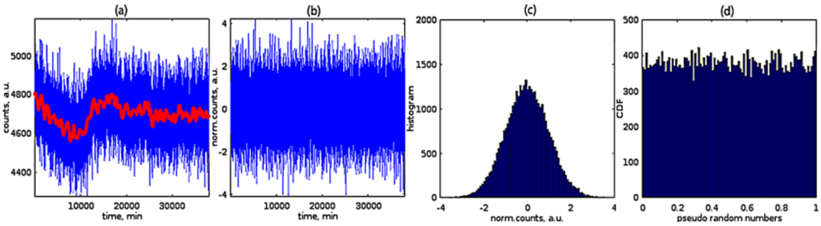
As we said before, we need two individual random numbers distributions to move a singlewalker around. The firrst one is the normal distribution and another one is a uniform randomnumber distribution used to sample the regions with low-density probability, see equations (4)-(5).The splinefit function, implemented in Octave package on Debian, has been used to detect characteristic variations in the intensities of the cosmic rays and to subtract them as a baseline from the raw data thus producing the Gaussian white noise, see Figure 4 (a)-(c).


mpiuser@debian-wind: ~/0-qmcpack-docs
File Edit View Search Terminal Tabs Help
mpiuser@debian-wind: ~...
mpiuser@debian-wind: ~...
mpiuser@debian-wind: ~...

```
top - 21:34:13 up  1:29,  1 user,  load average: 3.91, 1.38, 0.66
Tasks: 245 total,   9 running, 236 sleeping,   0 stopped,   0 zombie
%Cpu0  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 32826396 total, 29914276 free,  1074020 used,  1838100 buff/cache
KiB Swap: 18997244 total, 18997244 free,        0 used. 31280388 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3355 mpiuser   20   0  231344  12628  10196 R 100.0  0.0   0:07.76 Jan06_vmc_paral
 3361 mpiuser   20   0  231208  11812   9444 R 100.0  0.0   0:07.76 Jan06_vmc_paral
 3358 mpiuser   20   0  231208  11692   9328 R 100.0  0.0   0:07.75 Jan06_vmc_paral
 3359 mpiuser   20   0  231208  11836   9472 R 100.0  0.0   0:07.76 Jan06_vmc_paral
 3362 mpiuser   20   0  231208  11796   9432 R 100.0  0.0   0:07.74 Jan06_vmc_paral
 3356 mpiuser   20   0  231208  11716   9348 R  99.7  0.0   0:07.74 Jan06_vmc_paral
 3360 mpiuser   20   0  231208  11792   9428 R  99.0  0.0   0:07.63 Jan06_vmc_paral
 3357 mpiuser   20   0  231208  11792   9424 R  98.7  0.0   0:07.48 Jan06_vmc_paral
```

Figure 3. Output of the *top* command displaying info about the state of the individual cores of our CPU for the multithreaded C++ program of variational Monte Carlo simulation



(a) Original data approximated using the function splinefit, (b) white noise obtained as a result of mathematical processing, (c) normal and (d) uniform distribution of random numbers obtained from white noise

Figure 4. Modeling of the random number generator

The standardization procedure, see equation (11) further transforms the filtered noise component to a new one with zero mean and unit variance, which is our final normal distribution, see Figure 4 (b)-(c)

$$x=(x-\mu)/\sigma \tag{9}$$

where σ and μ are the standard deviation and mean values of the extracted noise before standardization procedure.

It is known from the probability integral transform that data values that are generated from any given continuous distribution, normal in our case, can be transformed to random deviates with a uniform distribution on the interval [0,1] as $U=F(x)$, where $F(x)$ is the standard normal cumulative distribution function. We used this to produce a uniform random distribution plotted on Figure 4 (d).

As a reference simulations we have used a pseudo-random generator provided by the *g++* compiler on *Debian* Linux OS. In this reference case, Box-Muller transform was used to go from a uniform random number distribution to a normal random distribution. In order to compare these data with our experimentally generated random numbers distributions we need the number of walkers to be the same in both cases.

If the walkers are placed not far away from the origin, we need no more than 1000 steps, including thermalization stage, to reach and amply sample the area of maximum probability.

All channels are highly correlated in general. The sources of extracted noise in each individual channel may be different from just a local background radiation and other independent processes in the registration hardware and could correlate as well. Fitting the data with splines may be not good enough to get rid of the possible statistical imprints of the common registered events.

Nevertheless, we obtained a nearly perfect match, see Figure 5, when we pulled the new values of the random normal and random uniform deviates from all available channels, in a successive order, one by one from each channel. We have got this match from as many as 17 walkers.

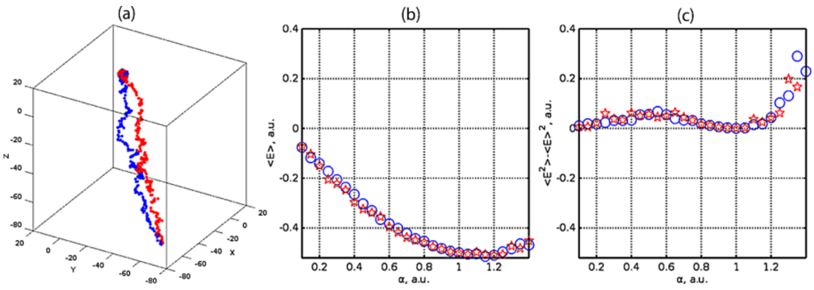
As one can see, the quality of experimental data, after few simple transformations, is reasonably good to bring the walker even from the remote location to the maximum probability point, see Figure 5 (a). The data were good enough not to introduce a special procedure to treat the outliers in the raw data.

The next two plots to the right from trajectories' plot, represent the average energy and its variance versus variation parameter α=0. As expected, we can clearly see the minimum at α=0.5 because the E_L loses its α dependence at this point, see equations (6) and (7). This value of α corresponds to the zero variance value on the right plot which pinpoints the ground state in our problem.

The slight differences observed on the picture are related to the quality of the random numbers used in our simulations.

Supporting our MCMC simulations, the one-sample Kolmogorov-Smirnov test *kstest*(x) returns a positive test decision for the null hypothesis that our data indeed come from a standard normal distribution.

The formal application of the full range of normality tests, as well as the tests of global and local randomness, has been left for a further study.



The 3D trajectories (a) of individual walkers driven to the maximum probability region by a regular pseudorandom distributions (red), and by a random numbers from our experimental data (blue).Computed average energy values (b) and computed values of the variance (c), where blue open circles stand for a regular pseudorandom distributions and red pentagrams stand for an experimental data.

Figure 5. Data comparison for the different random number generators

Conclusions

We have successfully used the neutron monitor data as a source for the normal and uniform random number distributions to drive a MCMC method. The desired stochastic component was extracted by fitting the data with splines and subtracting this fit from the raw data. Further scaling of our data to zero mean and variance of one was sufficient to obtain a stable standard normal random variate. Cumulative distribution function was used as a source of the uniform random numbers.

We are able to obtained the exact values of the variational parameter $\alpha$and local energy value E_L required to pinpoint the ground state of the three dimensional quantum harmonic oscillator. The negligible difference in the calculated values of the local energy for different α values as compared to a reference case may be attributed to the multiple factors related not only to the quality of the noise extraction but to a thorough analysis of this noise sources in different channels. For some problems, as ours, this quality is good enough or may be improved by additional treatment of the raw data. Distributions under consideration pass additional normality test.

Acknowledgments

This research was supported by grant №3824/ГФ4 provided by the Science Committee at the Ministry of Science and Education of Republic of Kazakhstan to the principal investigator at the National Nanotechnology Laboratory of Open Type, Physics and Technology Department, al-Farabi Kazakh National University.

References:

1. Top 500 Supercomputer Sites [web data base]. – The list of the 500 most powerful computer systems. – http://www.top500.org/lists/2015/11/
2. Sadaf R., Alam et al. An Evaluation of the Oak Ridge National Laboratory Cray XT3 // International Journal of High Performance Computing Applications. – 2008. – V. 22. – N. 1. – P.52-80.
3. Gergel' V.P. Vysokoproizvoditel'nye vychisleniya dl mnogoyadernyh mnogoprocessornyh sistem. Uchebnoe posobie. – Nizhnij Novgorod: Izd-vo NNGU im. N.I. Lobachevskogo, 2010. – 421 s.
4. Scott L.R., Clark T. and Bagheri B. Scientific Parallel Computing. – Princeton, NJ, USA: Princeton University Press, 2005. – 392 p.
5. Abdrahmanov R. Superkomp'yuter ot partnyora: superkomp'yuter grantu pravitel'stva KNR // Vechernij Almaty. – 2015. – 12 sentyabrya.
6. Akhter S., Roberts J. Multi-core Programming: Increasing Performance Through Software Multi-threading. – Intel Press, 2006. – 336 p.

7. OpenMPI project [web data base]. – Information and resources. – http://www.open-mpi.org/
8. Chapman B. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). – The MIT Press: Scientific and Engineering Computation edition, 2007. – 384 p.
9. Kashurnikov V.A., Krasavina V. Vychislitel'nye metody v kvantovoj fizike: Uchebnoe posobie. – M.: MIFI, 2005. – 412 s.
10. Voevodin V.V., Voevodin Vl.V. Parallel'nye vychisleniya. – SPb.: BHV-Peterburg, 2002. – 608 s.
11. Debian OS [web data base]. – Information and resources. – https://www.debian.org/
12. Oracle software and applications [web data base]. – Free access information and resources. – https://www.virtualbox.org/
13. Metropolis N., Rosenbluth A.W., Rosenbluth M.N., Teller A.H., Teller E. Equations of State Calculations by Fast Computing Machines // Journal of Chemical Physics. – 1953. – 21 (6). – P. 1087-1092.
14. Box G.E.P., Muller M.E. A Note on the Generation of Random Normal Deviates // The Annals of Mathematical Statistics. – 1958. – Vol. 29. – N. 2. – P. 610-611.

Questions:

1. What is message passing interface is used for?
2. Describe the main principle behind the basic variational principle.
3. What is the difference between multiprocessing on the clusters and on the individual multicore CPUs?
4. Describe the basic features of the Monte Carlo methods.
5. How the quantum oscillator is different from the classic oscillator?
6. What are the minimum requirements to install and run the Linux operational system on PC?
7. What is an alternative for full installation of the operational system on PC?
8. What is the difference between 32 and 64-bit register size operational system?
9. What package is used on Linux software to install the missing software?
10. Describe the time independent Schrodinger equation for the quantum oscillator in 1D.
11. What are the eigenfunctions?
12. How to calculate the average value of the observable?
13. Describe Metropolis-Hasting algorithm.
14. What is the biased sampling?
15. What types of trial function in variational method could be used?
16. What is the compiler directive to compile an openmpi program?
17. How to specify multiple threads when you run a multithread program?
18. What is the walker?
19. Describe the syntax and usage of the *MPI_Scatter* and *MPI_Gather* directives.
20. How to create the array with dynamically allocated size?

21. Describe the syntax and outcome of the *MPI_Finalize* directive.
22. How to define the stop point for the variational Monte Carlo method?
23. What is the Wall and CPU times? Which was is bigger?
24. How to visualize the CPU load for each core on the Linux system?
25. Explain the information provided by each column in the output of the *top* command.

2

HURST EXPONENT ESTIMATION, VERIFICATION, PORTABILITY AND PARALLELIZATION

We present multiple software programs for the Hurst exponent calculations for a sample time series collected by a neutron monitor detectors array. The first application is carried out by the finite differences approach, using a spreadsheet-type application for a single one hour long data series; the second is a complete, one and a half week long, mathematical and graphical analysis of six acquisition channels in Matlab; the third and the fourth are the data file parser and analyzer in C/C++ compiler on Windows platform, and its modified Linux version for simultaneous, parallel computing on a virtual cluster of three machines. All applications produce the same results proving the codes' validity and portability across the operational systems and software packages.

Introduction

The applications of the Hurst exponent are ranging from stock market analysis [1] to electron gas modeling [2] and addressing data statistics and system's fractal properties. Originally, it was introduced in hydrology [3] with the purpose to construct an optimal irrigation system. Since then, multiple studies have been done including the studies of cosmic rays variations. Hurst exponent estimates are strongly dependent on the length of data sample. For example Sankar N.P. et al [4] analyzed 36 years long data series on cosmic rays density covering almost three solar cycles and came to conclusion that «the present data is anti-persistent in behavior and the process is a short memory process» with the H value of 0.15. Flynn M.N. and Pereira W. on the contrary, studied extra short, hundred points and less, data sequences [5] and extracted vital information from a data sample on population dynamics.

Our primary goal in this work is effective parsing of raw data, reliability of results of the Hurst exponent calculations and cross platform compatibility software.

Methods

Conventional algorithm for the Hurst exponent calculation is as follows:

Original time series of length N is divided into the sets of shorter series with length $n = N$, $N/2$, $N/4$, …, 4, 3,and 2 points. The upper, $n = N$, and lower, $n = 2$, cutoff limits are different from study to study and depend on data availability and the phenomena, targeted for analysis.

For each set with particular n value, and for every partial series $\{X_i\}$ within this set, the following intermediate values have been calculated:

The mean value of each partial series

$$m = \frac{1}{n}\sum_{i=1}^{n} X_i \tag{1}$$

The mean-adjusted series derived from each $\{X_i\}$

$$Y_t = X_t - m \text{ for } t = 1, 2, \dots, n. \tag{2}$$

The standard deviation S

$$S(n) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - m)^2} \tag{3}$$

and the rescaled range R

$$R(n) = \max(Z_1, Z_1, \dots Z_n) - \min(Z_1, Z_1, \dots Z_n) \quad (4)$$

where the cumulative deviate series Z_n are given by the following expression

$$Z_i = \sum_{1=1}^{t} Y_i \text{ for } t = 1, 2, \dots, n. \quad (5)$$

The procedure is repeated for all possible values of n. Based on these $E(n)$ and n values we have tabulated the following function

$$E\left[\frac{R(n)}{S(n)}\right] = Cn^H \quad (6)$$

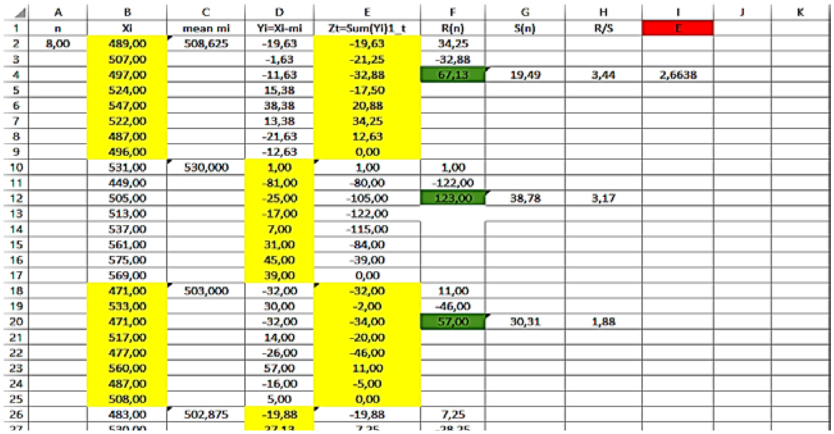
The value of H then could be calculated from fitting the tabular data into a polynomial (see Matlab's *polyfit* data on Fig.3), or calculating the slope of the straight line *log(E)=log(C)+H*log(n)* (see Matlab's *lsqcurvefit* model plotted on Fig.2). For more elaborate and mathematically sound calculations one may choose to work with the generalized Hurst exponent which is directly related to fractal dimension [6].

Fig.1 shows our H estimates for an hour-long observation, calculated with the algorithm above. The complete 6 channels, 10 days long data analysis is shown on Fig.2-3. The results of the code adaptation to a C/C++ programming environment and parallel computation code are listed on the last Fig.4.

Results and Discussion

The original data were retrieved from the Nikolay Pushkov‘s Institute of Earth Magnetism, Ionosphere and Radiowaves Propagation of the Russian Academy of Sciences (IZMIRAN) mobile 6NM64 supermonitor database [7]. Neutron counts were acquired at one minute interval from June the 31st, 2014 till August the 8th, 2014 in the vicinity of Moscow city.

All the steps described above in Eq.1-6 are shown in our first example in Fig.1. Here, the initial 64 data points (see column *B*) where divided into *n*=8 series each 8 points long. The following parameters have been calculated for each series: the mean values (see column *C*), cumulative deviate series Z_i (see column *D*), minimum and maximum values of this deviate series, and the range *R* (see column *F*). The last three columns *G-I* are the standard deviations *S(n)*, *R/S* for each subseries and a single value of *E* for *n*=8 which is an average over all values of *R(n)/S(n)*.



	A	B	C	D	E	F	G	H	I	J	K
1	n	Xi	mean mi	Yi=Xi-mi	Zt=Sum(Yi)1_t	R(n)	S(n)	R/S	E		
2	8,00	489,00	508,625	-19,63	-19,63	34,25					
3		507,00		-1,63	-21,25	-32,88					
4		497,00		-11,63	-32,88	67,13	19,49	3,44	2,6638		
5		524,00		15,38	-17,50						
6		547,00		38,38	20,88						
7		522,00		13,38	34,25						
8		487,00		-21,63	12,63						
9		496,00		-12,63	0,00						
10		531,00	530,000	1,00	1,00	1,00					
11		449,00		-81,00	-80,00	-122,00					
12		505,00		-25,00	-105,00	123,00	38,78	3,17			
13		513,00		-17,00	-122,00						
14		537,00		7,00	-115,00						
15		561,00		31,00	-84,00						
16		575,00		45,00	-39,00						
17		569,00		39,00	0,00						
18		471,00	503,000	-32,00	-32,00	11,00					
19		533,00		30,00	-2,00	-46,00					
20		471,00		-32,00	-34,00	57,00	30,31	1,88			
21		517,00		14,00	-20,00						
22		477,00		-26,00	-46,00						
23		560,00		57,00	11,00						
24		487,00		-16,00	-5,00						
25		508,00		5,00	0,00						
26		483,00	502,875	-19,88	-19,88	7,25					
27		530,00		27,13	7,25	-28,25					

Figure 1. Microsoft Excel spreadsheet calculation of a single *E* value for *n*=8 for the first acquisition channel in the data file

Next, using the range of the matrix tools and loop structures available in Matlab, we have calculated multiple values of *E* as a function of *n* for all 6 channels in the data file and fit them with the straight line for *log(n)* vs *log(E)* representation (see Fig.2), and with polynomial function similar to Eq.6 (see Fig.4).

Both linear and polynomial fitting models closely follow the original data points in the selected range of *n*.

Coefficients *C*, *log(C)* and *H* values are shown above each subplot, with 5 significant figures after the decimal point, though we use such precision mainly to control the algorithm performance across the different channels. No more than 2 significant figures are usually taken into consideration for data analysis.

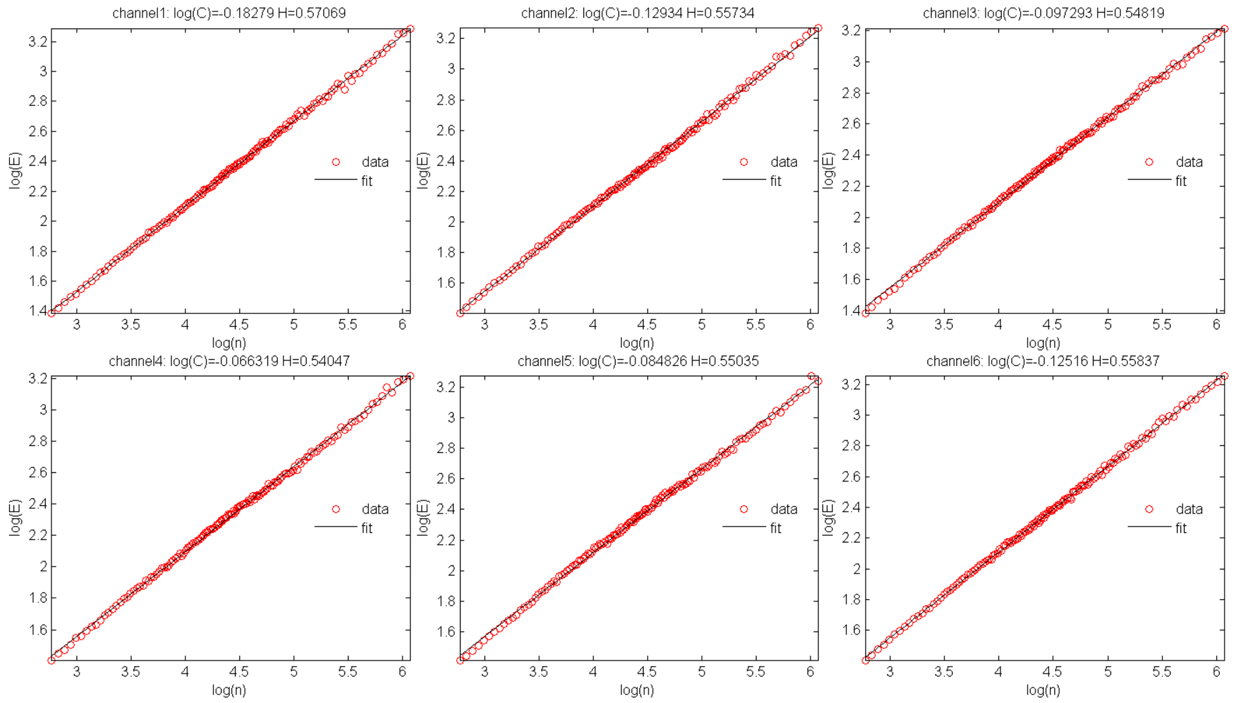


Figure 2. Matlab implementation of the 6 channels data analysis in log vs log representation and data fit by a straight line using *polyfit* function. Channel’s number, values of *log(C)* and *H* are printed above each subplot

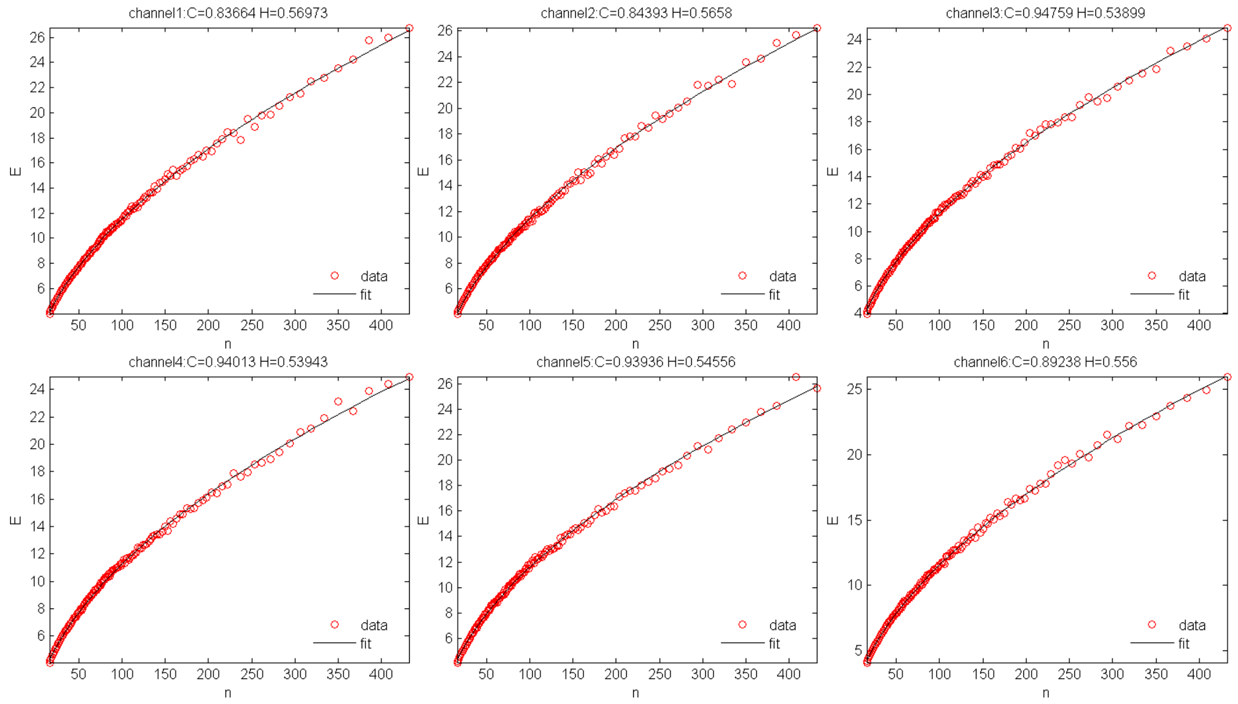
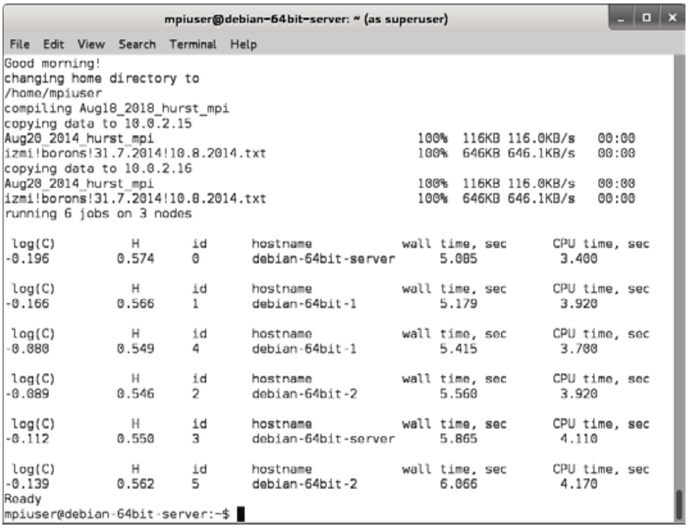


Figure 3. Matlab implementation of the 6 channels data analysis fitting with *lsqcurvefit* function. Channel’s number, values of *log(C)* and *H* are printed above each subplot

Microsoft Excel was used for the spreadsheet calculations, and Matlab 8.2.0.701 (R2013b) for Fig.2-3 results.

Next, we implemented our algorithm on C/C++ language with Bloodshed Dev-C++ compiler (the data is not shown for the brevity sake). Then, to address a persistent need for the effective parallel algorithms in data processing we designed the basic adaptation of C/C++ code to the Message Passing Interface (*MPI*, and *OpenMPI* in our case) parallel computations environment. We have used channel-by-channel workload distribution between the threads, as shown in Fig.4. The coefficients *log(C)* and *H* with corresponding thread (process) for each individual channel, are also shown.

For the purpose of parallel computing we have configured a virtual cluster on the Oracle VM VirtualBox. The host is 64 bit Windows 8.1 operating system running on Intel Core i3-3220 CPU with 4 Gb of RAM. The guest operating systems are the three Linux machines with 64 bit Debian GNU/Linux 7.4 (wheezy) with 512 Mb of RAM per each server and two nodes. Message Passing Interface is provided by *OpenMPI* v.1.4.5 bundled with Debian distribution.


mpiuser@debian-64bit-server: ~ (as superuser)
File Edit View Search Terminal Help

```
Good morning!
changing home directory to
/home/mpiuser
compiling Aug18_2018_hurst_mpi
copying data to 10.0.2.15
Aug20_2014_hurst_mpi                                  100%  116KB 116.0KB/s   00:00
izmi!borons!31.7.2014!10.8.2014.txt                   100%  646KB 646.1KB/s   00:00
copying data to 10.0.2.16
Aug20_2014_hurst_mpi                                  100%  116KB 116.0KB/s   00:00
izmi!borons!31.7.2014!10.8.2014.txt                   100%  646KB 646.1KB/s   00:00
running 6 jobs on 3 nodes

 log(C)         H        id      hostname             wall time, sec      CPU time, sec
-0.196        0.574      0       debian-64bit-server       5.085            3.400

 log(C)         H        id      hostname             wall time, sec      CPU time, sec
-0.166        0.566      1       debian-64bit-1            5.179            3.920

 log(C)         H        id      hostname             wall time, sec      CPU time, sec
-0.080        0.549      4       debian-64bit-1            5.415            3.700

 log(C)         H        id      hostname             wall time, sec      CPU time, sec
-0.089        0.546      2       debian-64bit-2            5.560            3.920

 log(C)         H        id      hostname             wall time, sec      CPU time, sec
-0.112        0.550      3       debian-64bit-server       5.865            4.110

 log(C)         H        id      hostname             wall time, sec      CPU time, sec
-0.139        0.562      5       debian-64bit-2            6.066            4.170
Ready
mpiuser@debian-64bit-server:~$
```

Figure 4. Calculated data displayed in the terminal window of the master process in the virtual cluster. Calculated values of *log(C)* and *H*, process *id*, *hostname* and execution times for each process are given

The Hurst exponent estimates in our study are matching the majority of the previously obtained results for geomagnetic indices [8] where *H* value is above *0.5*. Variations in our computed values of the *C* and *log(C)* (see the Fig.2-3 and Fig.4) are caused by the slightly different fitting models used for these estimates.

In addition, we have tested the code with a generated *sine* wave of the same duration as the longest neutron data sequence and with close to diurnal variations frequency. As we have anticipated, the obtained results of *H*~0.15 reflect no short memory in the series, supporting the validity of the coding.

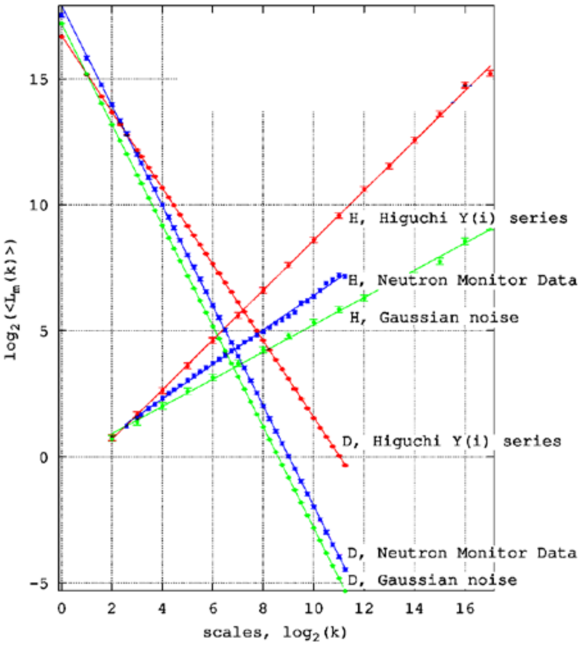


Figure 5. Higuchi time series (red lines), neutron monitor data (blue lines) and Gaussian noise (green lines) calculations of *H* and *D* values

To estimate the coding performance the «wall time» and «cpu time» have been streamed to the screen and file output for each process. To avoid possible interference between the processes the individual copies of the data files were supplied to each process/node at the beginning of the execution time by physically copying the data to the specified location, as shown in Fig.4.

Next figure 5 shows our H estimate for the first out of six registration channels of the nuclear monitor, see the blue line with the negative slope. Each data time series contains 10 days long data, taken with one minute resolution. Operator makes the choice of registration channel within the program.

Fractal dimension *D* is closely related to the Hurst exponent and could be calculated, according to Higuchi cornerstone paper [3], by constructing the following sets of subseries X_k^m

$$\begin{gathered} X(m), X(m+k), X(m+2k), \dots, X\left(m + \left[\frac{N-m}{k}\right]k\right); \\ m = 1,2,\dots,k;\ k = 1,2,\dots,\left[2^{(j-1)/4}\right];\ j = 11,12,13,\dots \end{gathered} \tag{7}$$

where the square brackets are used to denote the closest integer after rounding the fraction to zero.

Next, we calculate the normalized lengths *L_m(k)* of the constructed subseries

$$L_m(k) = \left\{ \frac{N-1}{\left[\frac{N-m}{k}\right]k} \sum_{i=1}^{\left[\frac{N-m}{k}\right]} |X(m+ik) - X(m+(i-1)k)| \right\} k^{-1} \tag{8}$$

These are expected to follow the power law in the form of $\langle L_m(k)\rangle \propto k^{-D}$ after averaging within all sets of *k* values for different *m*=1,2, … , *k*.

To test the validity of the program we reproduced the exact same time series *Y(i)* which were used by Higuchi in his derivations [3] and plotted the results on the same figure, see Figure 1 and the red line with the negative slope.

Here $(i) = \sum_{j=1}^{1000+i} Z(j)$, where *Z(j)* is a Gaussian noise with mean zero and standard deviation equals to 1. The set of $L_m(k)$ subseries is again expected to produce data following the power law $\langle L(k)\rangle \propto k^{-D}$ and the value of *D* could be extracted following the same procedure as described above for the Hurst exponent. The same code can be applied to this procedure. We get *D* value equals precisely -1.5341 for a Higuchi's time series, see Table 1. Separate test run of the code for the Gaussian time series *Z(j)* produced another set of values *H* and *D* equal -1.9999 and 0.54000 correspondingly, as expected for the highly uncorrelated times series, see Table I. The *Y(j)* data are plotted with the green color on Figure 5.

The *n* length of subseries,in case of the Hurst exponent calculations, and *k* value for the fractal dimension calculation have been treated equally in our code, that is as a single variable *k=n* in one of the outer parent loops. Two separate arrays of size *k* were allocated for both types of calculations. In case of the Hurst exponent calculations the array's values have been filled sequentially by the data file readout. For fractal dimension calculations, each array's element contained the value $<L_m(k)>$ assembled through the data readout according to the scheme described in Eq.6. The number of processes was equal to the number of virtual machines and kept equal to three.

The original neutron monitor data were retrieved from the Nikolay Pushkov's Institute of Earth Magnetism, Ionosphere and Radiowaves Propagation of the Russian Academy of Sciences (IZMIRAN) mobile 6NM64 supermonitor database [7]. Neutron counts were acquired at one minute interval from June the 31st, 2014 till August the 8th, 2014 in the vicinity of Moscow city. Neutron monitor time series data additionally underwent simple exponential smoothing filtration procedure as described in [9].

To address a persistent need for the effective parallel algorithms in data processing we designed our piece of C/C++ code compatible with available on *Debian* distribution *Open Message Passing Interface* (OpenMPI) parallel computations environment. We have configured our virtual cluster using the *Oracle VM VirtualBox*. The host is 64 bit *Windows 8.1* operating system running on *Intel Core i3-3220 CPU* with *4 Gb* of *RAM*. The guest operating systems are the three *Linux* machines with 64 bit *Debian GNU/Linux 7.4* (*Wheezy*) with 512 Mb of *RAM* per each server and two nodes.

Calculated slopes for all three sets of data are given in the Table 1. These are the Higuchi time series, Gaussian noise and neutron monitor data *H* and *D* values.

The $\pm\Delta y_i$ error bars were plotted on the figure around the experimental data points (x_i, y_i) using *polyfit*, *polyval* and *errobar* functions available in *Matlab/Octave* in such a way that then $y_i \pm \Delta y_i$

contains at least 50% of the predictions of future observations at x_i, see *Matalb/Octave* help notes on the selected functions.

In both *H* and *D* calculations, different scales evaluations were distributed between three processes. The number of processes in our case is equal to the number of machines used for calculations.

Table 1

Curve's slope estimates in *D* and *H* calculations

Time series name	Number of points in time series, *N*	Fractal dimension, *D*	Hurst exponent, *H*
Higuchi time series	2^{17}	-1.5143	0.98758
Gaussian noise	$1000+2^{17}$	-1.9999	0.54000
NM data, 1st channel	14703	-1.9973	0.68353
2nd channel	14703	-1.9984	0.66275
3rd channel	14703	-1.9985	0.63786
4th channel	14703	-1.9982	0.67716
5th channel	14703	-1.9994	0.64220
6th channel	14703	-1.9973	0.65107

H and *D* values for Higuchi time series in our calculations were found to be equal to the *D* and *H* values in [3], where *H* value close to 1 (see Table 1) indicates long term positive autocorrelation, as expected from the *Y(i)* series. For the neutron monitor series, Hurst exponent estimates are matching the majority of the previously obtained results for geomagnetic indices [10] where *H* value is above *0.5*. In our previous studies [11] of the same data with the same algorithm, the *H* value was slightly bigger than 0.6. The single reason for this change in the first decimal place is the change in the spectrum of *k* values used for our calculation. These slight changes easily rotate the curve in *log(E) vs. log(k)* plane. Nethertheless, all values stay well above 0.5. Between all six channels, the *H* values are in 0.64-0.68 range, suggesting that at the chosen series duration, our time series do have some scalable order. However, possibility of a long-term positive autocorrelation requires further studies.

The Gaussian noise data *Z(i)* in its turn produces values of *D*=-1.9999 and *H*=0.54000 as expected from the highly uncorrelated data.

As we can see, for all sets of data the processes are not completely self-affine in the sense of $D+H=n+1$ relationship, and the dependence between D and H is not of linear nature.

It is also interesting to observe how the slope changed according to the changes in data structure. We can see this transition when we come from Higuchi time series down to the chaos in the Gaussian noise. Rotations happen to be around (2:0) point for the Hurst exponent and (2:14) points for Hurst exponent and fractal dimension data calculations. This may be due to the loss of differences between time series of different origin at the small scales.

Conclusion

We have demonstrated various methods of the Hurst exponent calculation on different OS and software. Basic parallel algorithm allocating the data as one channel per one thread fashion has been demonstrated as well. Further studies involve parallel algorithm optimization and interpretation in terms of quantum algorithms.

Our values for H range from 0.55 to 0.58 across all 6 channels, suggesting that at the chosen series duration, without prior noise filtering, we are pretty close to a stochastic signal. However, possibility of a long-term positive autocorrelation requires further studies.

The code is compiled to run independently on different computers and could be used as a tool to study time series of different nature and origin. Timing and optimization in this study are the subjects of further studies. Using *C++* programming language we implemented an algorithm for the simultaneous parallel calculations of Hurst exponent H and fractal dimension D over specific time series. Parallel programming environment was provided by an *OpenMPI* package installed on three machines networked in the virtual cluster and operated by a *64* bit *Debian Wheeze 7.4* operating systems. We used our program to perform a comparative analysis of the week and a half long, one minute resolution, six channels data file from neutron monitor. To verify the functionality of the written code, we compared these results with a similar data analysis of the random Gaussian noise signal and time series with manually introduced self-affinity features and known values of *H* and *D*. All results are in good correlation with each other and are supported by the modern theories on signal processing, thus confirming the validity of the implemented algorithms.

Besides the straightforward workload distribution between the parallel processes, performed by splitting the data on the channel or on the calculated scale basis, we also identified common features in the calculations of two variables of interest, and tracked them in a single common outer loop within a single thread, providing additional optimization for the code. Our data and algorithms have multiple applications such as quick data self-affinity [12] test and have great potentials for the future development. The code is compiled to run independently on different networked computers and could be used as a tool to study time series of different nature and origin. Timing and optimization in this study are the subjects of further studies.

Acknowledgements

This research was supported by grant №2532/ГФ3 provided by the Science Committee at the Ministry of Science and Education of Republic of Kazakhstan to the principal investigator at the National Nanotechnology Open Laboratory.

We also would like to thank Dr. Yelena White, from East Georgia State College, Swainsboro, GA, USA, for editing the manuscript.

References:

1. Zhou J., Gu G.-F., Jiang Z.-Q., et al. Microscopic determinants of the weak-form efficiency of an artificial order-driven stock market. arXiv: 1404. 1051. – 2014. – P. 1-11.
2. Hurst J., Morandi O., Manfredi G., Hervieux P.-A. Semiclassical Vlasov and fluid models for an electron gas with spin effects. The European Physical Journal D. – 2014. – Vol. 68. – Issue 6. – P. 176-179.
3. Higuchi T. 1988. Approach to an irregular time series on the basis of the fractal theory. Phys. D. – 1988. – V. 31(2). – P. 277-283.
4. Sankar N.P. et al. Scaling and Fractal Dimension Analysis of Daily Forbush Decrease Data // International Journal of Electronic Engineering Research, – 2011, – V. 3(2), – P. 237-246.

5. Flynn M, Pereira W. Ecological studies from biotic data by Hurst exponent and the R/S analysis adaptation to short time series. – Biomatematica, 2013. – Vol. 23. – P. 1-14.
6. Mandelbrot B.B., Passoja D.E., and Paullay A.J. Nature. – 1984. – V. 308. – P. 721-724.
7. The Shepetov's database in IZMIRAN at http://cr29.izmiran.ru/vardbaccess/frames-vari.html accessed on August 10, 2014.
8. Pesnell W.D. Solar Cycle Predictions (Invited Review). Sol. Phys, 2012. – Vol. 281. – P. 507-532.
9. Brown R.G. Exponential smoothing for predicting demand. – Cambridge, Massachusetts: Arthur D. Little Inc, 1956. – 15 p.
10. Pesnell W.D. Solar Cycle Predictions (Invited Review). Sol. Phys, 2012. – Vol. 281. – P. 507-532.
11. Kussainov A.S., Kussainov S.G. Hurst exponent estimation, verification, portability and parallelization. Vestnik KazNU (Physics Edition), 2014, 4(47), in press.
12. Tilmann Gneiting and Martin Schlather. Stochastic Models That Separate Fractal Dimension and the Hurst Effect, SIAM Review, 2004. – 46(2). – P. 269-282.

Questions:
1. When was the first time the Hurst exponent was used for scientific data analysis?
2. What are the characteristic time scales in the cosmic ray data analysis?
3. How long/short should be the data time series to provide a reliable information about the process under consideration?
4. What is the raw data parsing?
5. Describe the main steps in data formatting procedures for the Hurst's exponent calculations.
6. Describe and sketch up a simple code explaining how to use *Matlab*'s *polyfit* and *lsqcurvefit* functions for data analysis.
7. How one could benefit from the explicitly chartering Hurst's exponent calculation procedure in *Microsoft Excel* spreadsheets?
8. Estimate the performance in data analysis when switching consequently from *Microsoft Excel* up to the serial type *C++* code and further to the parallel multithreaded implementation of the algorithm.

Program code
```
// References to the functions' description
/* http://www.cplusplus.com/reference/clibrary/cstdio/fgets/ */
/* http://www.cplusplus.com/reference/clibrary/cstdio/feof/ */
/* http://www.cplusplus.com/reference/clibrary/cstring/strstr/ */
/* http://www.cplusplus.com/reference/cstdio/fgetc/ */
#include <stdio.h>
#include <string.h>
#include <string>
```

```
#include <stdlib.h>
#include <sstream>
#include <iostream>
#include <fstream>
#include <cmath>
#include <new>
#include <math.h>
#include <mpi.h>

// library to organize time counters
#include <sys/time.h>

using namespace std;

// functions to provide timers
double get_wall_time(){
struct timeval time;
if (gettimeofday(&time,NULL)){return 0;}
return (double)time.tv_sec + (double)time.tv_usec * .000001;}
double get_cpu_time(){
return (double)clock() / CLOCKS_PER_SEC;}

int main(int argc, char *argv[]){
double wall0 = get_wall_time();
double cpu0 = get_cpu_time();

// pointers to the data files
FILE * pFile;
FILE * mFile;

int row_count;
char mystring [200];
char mean_value [200];
int chnls_nmbr=6;
string m;
string read_out;

// data from channel in numerical format
double ch;
int s_um=0.0;
double m_ax, m_in;
double Z=0.0;
int E_count;
double m_ean,s_td,E;
int c_ount,point_count=0;
double xy=0.0, x=0.0, y=0.0, x2=0.0, a,b;
```

```
int id, ntasks, len;
char single_channel[]="single_channel_";
char mean_data[]="mean_data_";
char E_data[]="E_data_";

char current_id [33];
char hostname[MPI_MAX_PROCESSOR_NAME];

// basic parallel environment variables
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Get_processor_name(hostname, &len);

sprintf(current_id,"%d",id);
strncat(strncat(single_channel, current_id,1),".txt",4);
strncat(strncat(mean_data, current_id,1),".txt",4);
strncat(strncat(E_data, current_id,1),".txt",4);

// selecting a single data channel for processing
int clmn=id+1;

// reading the data base
pFile = fopen ("izmi!borons!31.7.2014!10.8.2014.txt" , "r");
fstream SingleChannel;
SingleChannel.open(single_channel);
if (pFile == NULL) perror ("Error opening data file");
else{while(!feof(pFile)){
fgets (mystring ,200, pFile);
if (strlen(mystring)>=4){ read_out=string(mystring);
row_count++;
read_out.erase(0,20);
m.assign(read_out,(clmn-1)*4,3);
SingleChannel<<atof(m.c_str())<<"\n";}}
c_ount=row_count;}
fclose (pFile);SingleChannel.close();

point_count=0;
ofstream Efile; Efile.open(E_data);
for(int i=4;i<=900;i++){
point_count++;
// how points are spreading out
int n=i*2;
for(int pass=1;pass<=2;pass++){
/* Calculate and save to a separate file mean vales*/
if(pass==1){row_count=0;
```

```
pFile = fopen (single_channel, "r");
ofstream outFile; outFile.open(mean_data);
if (pFile == NULL) perror ("Error opening data file");
else{s_um=0.0;while(!feof(pFile)){

fgets (mystring ,200, pFile);

read_out=string(mystring);
row_count++;

// removing time stamp from the data field
m.assign(read_out,0,3);
s_um=s_um+atof(m.c_str());

if(row_count%n==0 && row_count<=c_ount){
outFile<<double(s_um)/double(n)<<"\n";s_um=0;}
}}
fclose (pFile);outFile.close();
}

else{E=0.0;E_count=0;row_count=0;
pFile = fopen (single_channel, "r");
mFile = fopen (mean_data, "r");

if (mFile == NULL) perror ("Error opening data file");
else{while(!feof(pFile)){

row_count++;
if (row_count%n==1 && row_count<=c_ount){fgets (mean_value ,200, mFile);
read_out=string(mean_value);
// 8 because I need more decimal points
m.assign(read_out,0,8);
 m_ean=atof(m.c_str());}

fgets (mystring ,200, pFile);
read_out=string(mystring);
m.assign(read_out,0,3);
ch=double(atof(m.c_str()));

if(row_count%n==1&&row_count<=c_ount){
m_ax=ch-m_ean; m_in=ch-m_ean;s_td=0.0;}
Z=Z+ch-m_ean;
s_td=s_td +(ch-m_ean)*(ch-m_ean);
if (Z>m_ax){m_ax=Z;};if(Z<m_in){m_in=Z;};
if (row_count%n==0 && row_count<=c_ount){E_count++;
E=E+(m_ax-m_in)/(sqrt(s_td/double(n)));
s_td=0;Z=0;}}}

fclose (pFile);fclose(mFile);}}

Efile<<n<<"\t"<<E/double(E_count)<<"\n";

// if E=C*n^H then log(E)=log(C)+H*log(n)
// y = a + b*x
// b=[<xy>-<x><y>]/[<x^2>-<x>^2]
// a=<y>-b*<x>;
xy+=log(n)*log(E/double(E_count));
x+=log(n);
y+=log(E/double(E_count));
x2+=log(n)*log(n);}

b=(xy-x*y/double(point_count))/(x2-x*x/double(point_count));
a=(y-b*x)/double(point_count);

// Stop timers
double wall1 = get_wall_time();
double cpu1 = get_cpu_time();

printf("\n %s \t %s \t %s \t %s \t %s \t %s \n","log(C)","H","id", "hostname", " wall time, sec", "CPU time, sec");
printf("%6.3f \t %s %6.3f \t %d \t %s \t %6.3f \t %6.3f \n",a," ",b,id,hostname, wall1-wall0,cpu1-cpu0);

Efile<<a<<"\t"<<b<<"\n";
Efile.close();

MPI_Finalize();       /* Terminate MPI */
if (id==0) printf("Ready\n");
exit(0);}
```

3

QUICK AND EASY PARALLELIZATION TECHNIQUE FOR THE ISING 2D MODEL IN OPEN MPI

Below, we have demonstrated a quick and easy compartmentalization method of the 2D Ising model and studied its efficiency and data produced. To optimize optional distributed computations, the intercompartmental communication was kept at minimum level. Boundary data between the compartments were updated only with each Monte Carlo step, that is, only after annealing has taken place within each individual compartment with number of steps bigger than the number of sites in compartment. *Open MPI* package implementing Message Passing Interface (MPI) for Debian Linux operational system was chosen to provide parallelization environment. The suggested method is straightforward, easy to use, produces correct results and is seamlessly scaled. Simulated ferromagnetic domains beyond the compartment size are clearly observed, and freely develop across the simulation grid. Exact results from the existing publications have been reproduced with greater efficiency. Significant speedup on the octacore desktop computer has been demonstrated.

Introduction

Powerful personal desktops or workstations, as well as clusters and high performance computing systems with remote access, are widely available nowadays. These are state-of-the-art and expensive machines maintained by numerous staff and capable of addressing the fundamental pure and applied science problems of today. Still, enormous amount of numerical simulations are done in a single thread. This fact seriously deteriorates the efficiency of computations and reduces the naturally parallel tasks to the bottleneck of a single thread application. Many scientists, though having in their possession the state-of-the-art multicore computing systems, are able to utilize only a small portion of their computational power. Multiple studies and helpful resources are published and circulated within the interested scientific and research communities to help them get familiar with parallel programming [1, 2].

Physicists have made many major advances in computational and mathematical physics. They clearly see the underlining physical processes and could tailor mathematics and programming algorithms to their specific tasks. For example, quantum computing is essentially a parallel multitasking at all levels. Blind use of an automated parsing software is unacceptable. Programming that uses the basic physics principles is required.

We address these and many other points by using the *Open MPI* library [3] that can handle multithread coding and feed it to a multicore CPU (central processing unit) or distribute the tasks across a network of computers connected in the computational cluster. Unlike its counterpart, the *Open MP* development of the message parsing protocols, [4] *Open MPI* is extensively documented in electronic resources and much easier to deploy.

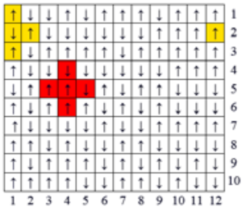
We have implemented Monte Carlo method with importance sampling in 2D spin glasses for the Ising model [5] as an example of multithreading and performance optimization in scientific computing. Conventionally, performance could be gain by compartmentalization and deployment of custom-made communicator between compartments for the transient phenomena. The same results could be achieved by understanding the simple topology and physics of the problem, reducing communication time and instances to a minimum.

Methods

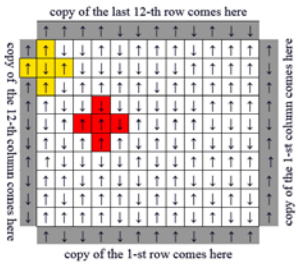
The Ising model we chose is described by the two-dimensional *m* x *n* grid of spins [6]. Each spin can take only two values, *up* or *down*, $s_i=\{+1,-1\}$. Assuming that magnetic interaction strength is dropping as fast as $1/r^3$ with the distance, we will consider interactions only between the closest neighbors. These neighbors are forming a cross pattern, (see, for example, five spins' sites shaded in red in Fig.1(a)). Immediately, we introduce periodic boundary conditions. That is, if the left neighbor in the left corner of interaction pattern is missing, (see yellow-shaded areas), we assume that its place is taken by the spin across the whole grid on the right boundary. The same technique is valid for the right, upper and bottom boundary sites.

One could directly index such sites and use them as a precursor to a custom-made communicator between the processes in the parallel version of the program. Our choice is to increase the simulation grid by copying the right boundary column to the left and the left boundary column to the right, as well as the top row to the bottom and bottom row to the top, (see the gray-shaded columns and rows in Fig. 1 (b)). In Figure 1(c) this procedure will insure the communication between the compartments in parallel algorithm. However, in this case, the boundary will be provided by a neighboring compartment.

(a)



(b)


copy of the last 12-th row comes here
copy of the 12-th column comes here
copy of the 1-st column comes here
copy of the 1-st row comes here

(c)

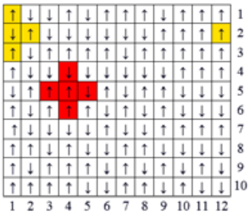


Figure 1. Compartmentalization scheme for three processes

Mathematical description of the model is presented below. The total interaction energy associated with all possible closest neighbor spins is given by

$$E = -J \sum_{ij} s_i s_j - H \sum_{i=1}^{m\times n} s_i, \tag{1}$$

where J is the spin-spin interaction strength and H is the external magnetic field strength. If J>0 the system is ferromagnetic, otherwise, if J<0, it is paramagnetic. Indices i and j sample all the available pairs of neighboring spins on the grid, excluding the double count of ij and ji pairs. The upper bound for the first sum is determined by the range of interaction and number of spins within this range. If the spins are allowed to have $Q \geq 2$ states, this will be the generalized Q-state Potts model [7]. Total magnetization value at the certain spins configurations is calculated as a sum

$$M = \sum_{i=1}^{m\times n} s_i \tag{2}$$

If the external field strength H is set to zero then there are two distinct states at low and high temperatures. These are ferromagnetic and paramagnetic phases separated by the transition region around Currie temperature T_c.

According to Metropolis algorithm [8] instead of trying to calculate quantum-mechanical observables over all possible combinations of states

$$M = \frac{\sum_{conf\ .space} M e^{E/kT}}{\sum_{conf\ .space} e^{E/kT}}, \tag{3}$$

Temperature T is given in the units of [E/k], where k is the Boltzmann factor. E is dimensionless but in general has the units provided by expression (1). We should include only those configurations which are sampled according to the Boltzman factor, see algorithm below. For a sequence of N such states, magnetization for the particular temperature will be given by the formula

$$\langle M \rangle = 1/N \sum_{j=1}^{N} \sum_{i=1}^{m\times n} s_i \tag{4}$$

The following steps should be taken repeatedly to achieve a desired distribution of states at the certain temperature T:

1. Choose at random any spin s_i and flip its sign, $s_i' = -s_i$.

2. Calculate change in the total energy according to formula

$$\Delta E = E_{s_i'} - E_{s_i} \tag{5}$$

3. If $e^{\Delta E/kT} > X$, where $X \sim U([0,1])$ is a random variable uniformly distributed on [0,1], the change in sign is accepted.

4. Repeat the previous steps to achieve an equilibrium magnetization value for a given simulation grid at the temperature selected for the system.

These four steps, repeated multiple times but usually not less than the number of sites in simulation grid or compartment, represent one Monte Carlo step. To calculate any observable value for a given temperature, MC steps have to be repeated several times. Other properties, including density of states, could be calculated via similar algorithms [9, 10].

Convergence to the equilibrium values of observable parameters with one spin flip is slow. Changes introduced by a single local spin flip propagate diffusively. Various algorithms have been proposed to speed up the process by flipping the whole clusters of spins at once. Swendsen-Wang [11] and Wolff [12] Monte Carlo methods are among them. The Wolff algorithm is an improvement over the Swendsen–Wang algorithm since it has a larger probability of flipping bigger clusters. Alternatively, we could partition the simulation volume for the parallel processing [13, 14] as we did in this paper, see Figure 1.

One may also avoid numerous exponentiation steps by noticing, that for a two dimensional grid, ΔE takes a limited number of values depending on orientation of the four neighboring spins. We can easily show that this number is equal to five and it doubles if the external magnetic field H is switched on.

Results and Discussions

Configuration of our eight core desktop computer is listed as follows: Intel Core i7 4790K, 4.0GHz/LGA-1150/22nm/Haswell/8Mb L3 Cache, DDR-3 DIMM 16Gb/1866MHz PC14900, 2x8Gb Kit, CL10.

As we said before, we perform the boundary conditions exchange every time the equilibrium within each compartment is reached. That is the boundary conditions are renewed for all compartments every time after each Monte Carlo step in accordance with equilibrium spins configuration in the neighboring compartments. One should take care not to replicate the simple periodic boundary conditions for each compartment. This means that we have to make sure that compartments are communicating with each other and supplying each other with information about the boundary conditions along the interface line.

Fig. 2(a) shows the spins orientation distribution computed in one thread without any compartmentalization.

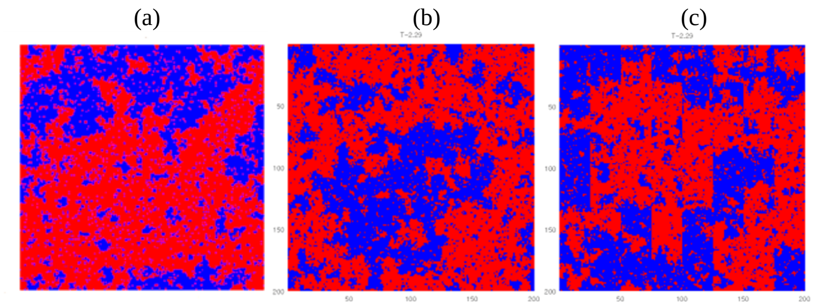


Figure 2. Sample magnetization distribution under different boundary conditions

Simple periodic boundary conditions on four boundaries were used. In Fig. 2(b), we plotted the case of eight communicating compartments when data in the compartments are formatted according to our algorithm. The case when the closed toroidal boundary conditions were implemented for each compartment without the proper boundary information exchange is shown in Fig. 2(c). Fig. 2(c) shows the obvious signs of an erroneous dynamics, such as band structure, indicating that the compartments are not able to communicate with each other.

The next Fig.3 gives a diagram for the simulated annealing of our system. Normalized magnetization M/M_0, where M_0 is the sum of all spins, changes its value from 1 at T=0, to 0 as the temperature rises beyond the critical value. Temperature of the phase transition T_c is about 2.3.

As one can see, the data from a single thread experiment with no compartmentalization, (represented by a green line and triangle markers), and data with proper compartmentalization, (blue line and diamond markers, are almost identical. The red line with pentagram markers stands for the incomplete implementation of the boundary conditions and exhibits a deviant behavior at phase transition. Nevertheless, all give about the same value of T_c.

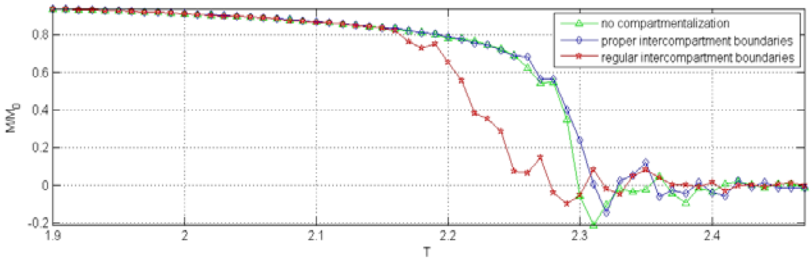


Figure 3. Simulated annealing for the 200 x 200 grid. Green line and triangle markers – no compartmentalization; Blue line and diamond markers – custom made compartmentalization and data exchange technique; Red line and pentagram markers – data produced with unadjusted boundary conditions between compartments

Next, Fig. 4 shows the timing data from our simulations when the number of threads goes up from 1 to 12, see the Y axis. Two sets of curves are plotted. The one on the left is for 200 by 200 data grid another one is for 400 by 400 grid. The obvious benefits of compartmentalization are visible. For the 200 by 200 grid, five thousands Monte Carlo steps and only one temperature value, CPU time is cut in half if we split the simulation volume between eight cores of a single processor. Normally, we simulate annealing for the range of T values and number of Monte Carlo steps is much higher than the number we used. Thus, the time savings are enormous.

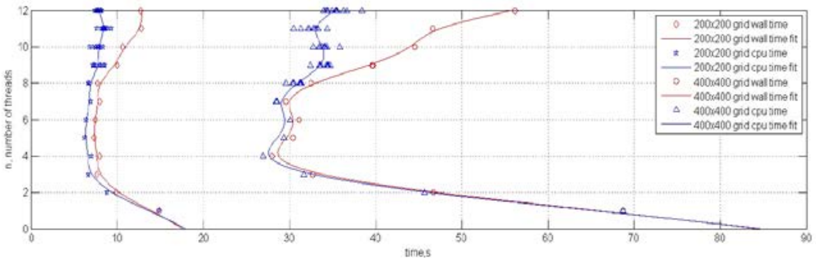


Figure 4. Parallelization algorithm performance for 200 x 200 and 400 x 400 grids. CPU, blue lines, and wall, red lines, times are given as a function of number of threads on the octacore processor

Multiple parallel platforms and custom made algorithms make it difficult to compare performance for each case. For the formal performance evaluation and further optimization of the arbitrary code the *LogP* model of Culler is available [15].

Conclusions

We reported our studies of implementation and efficiency of the quick and robust method of compartmentalization for the 2D Ising model. *Open MPI* package has provided the parallel computation environment. Algorithm structure is optimized for optional distributed computations by updating the boundary data between compartments with each Monte Carlo step. Meanwhile annealing, within each Monte Carlo step and for each individual compartment, takes the same number of steps as the number of sites in a compartment.

This boundary information update is enough to couple the statistical processes within each individual compartment to its neighbors. Thus construction of the complicated blocking type communications through a message passing interface is avoided. In general, it allows us to keep the basic simple model of the spin glass intact and evenly distribute intensive computations between the available threads. These two facts about our method contribute to the clarity of the model and its data interpretation as well as to increased speed of calculations.

Our proposed method is straightforward, easy to use, and produces correct results identical to the previously published data. Simulated ferromagnetic clusters of spins, freely developing between individual compartments, are clearly observed. Significant speedup on the octacore desktop computer has been demonstrated.

The model and its development represent the computational basis for the whole generation of the quantum algorithms for approximating partition.

Acknowledgments

This research was supported by grant №3824/ГФ4 provided by the Science Committee at the Ministry of Science and Education of Republic of Kazakhstan to the principal investigator at the National Nanotechnology Laboratory of Open Type, Physics and Technology Department, al-Farabi Kazakh National University.

References

1. Gergel' V.P. Vysokoproizvoditel'nye vychisleniya dl mnogoyadernyh mnogoprocessornyh sistem. Uchebnoe posobie. – Nizhnij Novgorod: Izd-vo NNGU im. N.I. Lobachevskogo, 2010. – 421 s.
2. Akhter S., Roberts J. Multi-core Programming: Increasing Performance Through Software Multi-threading. – Intel Press, 2006. – 336 p.
3. Open MPI project [web data base].-Information and resources. – http://www.open-mpi.org/
4. Chapman B. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). – The MIT Press: Scientific and Engineering Computation edition, 2007. – 384 p.
5. Ising E. Beitrag zur Theorie des Ferromagnetismus // Zeitschrift für Physik. – 1925. – 31. – Pp. 253-258.
6. Domb C., Green M.S. Ising Model. Phase Transitions and Critical Phenomena, Academic Press, 1974. – V3. – Pp. 357-484.
7. Potts R.B. Some Generalized Order-Disorder Transformations // Proceedings of the Cambridge Philosophy Society. – 1952. – 48(1). – Pp. 106-109.
8. Metropolis N., Rosenbluth A.W. Equation of State Calculations by Fast Computing Machines // Journal of Chemical Physics. – 1953. – 21. – Pp. 1087-1092.
9. Wang F., Landau D.P. Efficient, multiple-range random walk algorithm to calculate the density of states // Phys. Rev. Lett. – 2011. – 86. – Pp. 2050-2053.
10. Zhang C., Ma J. Simulation via direct computation of partition functions // Phys. Rev. E. – 2007. – 76. – Pp. 036708-15.

11. Wang J.-S., Swendsen R.H. Cluster Monte Carlo algorithms // Physica. – A: Statistical Mechanics and its Applications. – 1990. – 167(3). – Pp. 565-579.
12. Wolff U. Collective Monte Carlo Updating for Spin Systems // Physical Review Letters. – 1989. – 62(4). – Pp. 361-364.
13. Altevogt P., Linke A. Parallelization of the two-dimensional Ising model on a cluster of IBM RISC system / 6000 workstations // Parallel Computing. – 1993. – 19(9). – Pp. 1041-1052.
14. Heermann D.W., Burkitt A.N. Parallel Algorithms in Computational Science. Springer-Verlag New York, Inc. – New York, NY, USA, 1991.
15. Culler D., Karp R., Patterson D., Sahay A., Schauser K.E., Santos E., Subramonian R. and Thorsten von Eicken. LogP: towards a realistic model of parallel computation // Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '93). – ACM, New York, NY, USA, pp. 1-12.

Questions:

1. Describe the basic elements for the Ising and other spin glasses numerical models.
2. How dimensionality of the model related to the phase transition phenomena?
3. Describe Hamiltonian for the basic Ising spin grid model.
4. How big is the range of interaction in the Ising model?
5. What are the boundary conditions for the basic spin glass model?
6. Why we give a name «toroidal» boundary condition for the special type of the boundary conditions in the Ising model?
7. Describe the possible way of compartmentalization for the Ising model.
8. How different compartments in the Ising model could and should interact with each other?
9. How to calculate magnetization *M* for the spin grid?
10. How to calculate different observables for the Ising model?
11. Give a description of the Metropolis biased sampling scheme.
12. What are the equilibrium values?
13. What is the critical temperature T_c?
14. How the performance of the algorithm relates to the 2D grid size?
15. What are the factors preventing the linear increase in the algorithm performance with the increase of the parallel threads number?
16. What is the regular number of Monte Carlo steps required for the system's thermalization process?

Program code

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <mpi.h>
```

```
#include <sys/time.h>
using namespace std;
const double pi = 3.14159;
const int qq=2;

// timing functions
double get_wall_time(){
struct timeval time;
if (gettimeofday(&time,NULL)){ return 0;}
return (double)time.tv_sec + (double)time.tv_usec*.000001;}

double get_cpu_time(){return (double)clock() / CLOCKS_PER_SEC;}

int main(int argc, char *argv[]){

// Start Timers
double wall0 = get_wall_time();
double cpu0 = get_cpu_time();

double J=+1,H=0,T,k=1;
int m=200, n=200;
int id, ntasks;
int total_steps=5100; int therm_steps=int(0.2*total_steps);
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
MPI_Comm_rank(MPI_COMM_WORLD, &id);

// Memory and variable allocation stage
// this will clean the previous data stored in these files
if(id==0){
ofstream file("ising.txt");file.close();
ofstream mfile("magnetization.txt");mfile.close();}

// changing value of n to make sure that each process recives the same ammount of data
// the array will be expanded to provide threads with equal amount of data
int RowsPerTask = ceil(double(n)/double(ntasks));
n=RowsPerTask*ntasks;

// now we need to simulate periodic boundary by additional data columns for each
// process/thread. Each data piece should get two additional rows
int N=n+ntasks*2; // two extra column for each data band

// 1D array to collect magnetization data in root process
double* collect_data = new double [ntasks];
```

```
// 1D spin orientation data recieved by single array
double* s_recv = new double[(m+2)*(RowsPerTask+2)];
// 2D array of spins for indiviudal process for its own chank of data
double* spin_s = new double[(m+2)*(RowsPerTask+2)];
double** s_pins = new double*[m+2];
// initialize it
for(int i = 0; i < m+2; ++i)
s_pins[i] = spin_s + (RowsPerTask+2)*i;

// create initial collective 2D array of spins
double* s_data = new double[m*n];
double** s = new double*[m];
// initialize it
for(int i = 0; i < m; i++)
s[i] = s_data + n*i;
// set element values = 1
// for(int j=0;j<n;j++){for(int i=0;i<m;i++){s[i][j] = +1;}}
for(int j=0;j<n;j++){
for(int i=0;i<m;i++){
int nn=rand()%qq;
s[i][j] = cos(2*pi*double(nn)/double(qq));}}

// create linear array of spins to distribute data between the processes
// OpenMPI could only sends linear data array
double* s_send = new double[(m+2)*N];

// Array will be holding all data including the boundaries above and between the clusters
double* S_data = new double[(m+2)*N];
double** S = new double*[(m+2)];
for (int i = 0; i < (m+2); ++i)
S[i] = S_data + N*i;
for(int i=0;i<(m+2);i++){for(int j=0;j<N;j++){S[i][j] = 0;}}

//************************
double dT=0.01; srand ((double(id)+1)*time(NULL));
for(double T=2.2;T<=2.5;T+=dT){
double M=0;
for(int c_ount=1;c_ount<=total_steps;c_ount++){

if(id==0){
// Active part of DATA reshafling

int cou_nt=0;
for(int p_os=1;p_os<=1+(ntasks-1)*(RowsPerTask+2);
p_os+=(RowsPerTask+2)){
```

```
for(int j=0+p_os;j<RowsPerTask+p_os;j++){
for(int i=1;i<(m+2)-1;i++){
S[i][j] = s[i-1][j-1-2*cou_nt];}}cou_nt++;}

// global top and bottom boundaries
for(int j=0;j<N;j++){S[0][j]=S[(m+2)-2][j];S[(m+2)-1][j]=S[1][j];}
// global left and right boundaries
for(int i=0;i<(m+2);i++){S[i][0]=S[i][N-2];S[i][N-1]=S[i][1];}
// internal boundaries
for(int j=0;j<ntasks-1;j++){
for(int i=0;i<(m+2);i++){
// innner boundaries between the processes
S[i][(j+1)*(RowsPerTask+1)+j]=S[i][(j+1)*(RowsPerTask+1)+j+2];
S[i][(j+1)*(RowsPerTask+1)+j+1]=S[i][(j+1)*(RowsPerTask+1)+j-1];}}

// reassigning 2D data to this 1D array for further distribution
// COLUMN by COLUMN
for (int t = 0; t < N; t++){
for (int q = 0; q < (m+2); q++){
s_send[t * (m+2) + q] = S[q][t];}}
}
// distribute data
// individual process gets its data beyond this point
MPI_Scatter(s_send,(m+2)*(RowsPerTask+2),MPI_DOUBLE,
s_recv,(m+2)*(RowsPerTask+2),MPI_DOUBLE,0,MPI_COMM_WORLD);

// reconstruct from linear to 2D
for (int t = 0; t < RowsPerTask+2; t++){
for (int q = 0; q < m+2; q++){
s_pins[q][t] = s_recv[t * (m+2) + q];}}

// we assume that we should randomly access pretty much all spins in the array
for (int ii=0;ii<8*m*RowsPerTask;ii++){
int i = rand()%m+1; int j = rand()%RowsPerTask+1;

// the indices should fall within the boundaries
double dE=2*J*s_pins[i][j]*(s_pins[i+1][j] + s_pins[i-1][j] +
s_pins[i][j-1] + s_pins[i][j+1]);
if (exp(-dE/(k*T))>(rand()/(RAND_MAX + 1.0))){s_pins[i][j] = -s_pins[i][j];}}

double M_0=0;
for(int i=0;i<m;i++){for(int j=0;j<RowsPerTask;j++)
{M_0+=s_pins[i][j];}}
// collect magnetization data only after thermalization stage
```

```
if (c_ount>=therm_steps){M+=M_0;}
// convert from 2D to linear to send back
for(int t=0; t<(RowsPerTask+2); t++){
for(int q = 0; q<(m+2); q++){
s_recv[t*(m+2)+q] = s_pins[q][t]; }}

// send it back to root process
MPI_Gather(s_recv,(m+2)*(RowsPerTask+2),MPI_DOUBLE,
s_send,(m+2)*(RowsPerTask+2),MPI_DOUBLE,0,MPI_COMM_WORLD);

if(id==0){

// reconstruction of the collected data from 1D array
for(int t=0;t<N;t++){
for(int q=0;q<(m+2);q++){
S[q][t]= s_send[t*(m+2)+q];}}

// back reconstruction of the expanded array to the original ones,
// that is stripping from boundaries
int cou_nt=0;
for(int p_os=1;p_os<=1+(ntasks-1)*(RowsPerTask+2);
p_os+=(RowsPerTask+2)){
for(int j=0+p_os;j<RowsPerTask+p_os;j++){
for(int i=1;i<(m+2)-1;i++){
s[i-1][j-1-2*cou_nt]=S[i][j];
}}cou_nt++;}
}
// loop over id==0
}
// loop "for(int c_ount=1;c_ount<=total_steps;c_ount++)" over monte carlo steps is
// over,that includes thermalization and magnetization stage

double M_local=M/double(m*RowsPerTask)/double(0.8*total_steps);
MPI_Gather(&M_local,1,MPI_DOUBLE,
collect_data,1,MPI_DOUBLE,0,MPI_COMM_WORLD);

if(id==0){
cout<<id<<"\t"<<T<<"\t"<<M/double(m*RowsPerTask)/double(0.8*total_steps)
<<"\n";
ofstream file("ising.txt",std::ofstream::out | std::ofstream::app);
double M_total=0;
for(int i=0;i<ntasks;i++)
M_total+=collect_data[i];
file<<T<<"\t"<<M_total/ntasks<<"\n";file.close();
```

```
// to save magnetization configuration data
ofstream mfile("magnetization.txt",std::ofstream::out | std::ofstream::app);
for(int q = 0; q < m; q++){
for(int t = 0; t < n; t++){
mfile<<s[q][t]<<"\t";}mfile<<"\n";}mfile.close();}

} // loop over T values of temperature
// dynamic arrays clean up
//************************
delete[] spin_s;
delete[] s_pins;
delete[] s_recv;
delete[] s_data;
delete[] s;
delete[] s_send;
delete[] S_data;
delete[] S;

delete[] collect_data;

MPI_Finalize();
// Stop timers
double wall1 = get_wall_time();
double cpu1 = get_cpu_time();
cout<<wall1-wall0<<"\t"<<cpu1-cpu0<<"\n";
return 0;}
```

4

BUILDING THE DENSITY OF STATES FROM THE WANG-LANDAU PARALLEL ALGORITHM FOR THE SIMPLE SPIN GRID SYSTEMS

Below we have described a universal algorithm to build a resultant density of states from the multiple data pieces provided by the parallel implementation of the Wang Landau sampling Monte Carlo algorithm. Vector and standard Pott model as well as the Ising model were used as an example of two dimensional spin lattices. Several factors of an immediate importance for the seamless stitching procedure were considered. These include but not limited to the speed and universality of the original parallel algorithm implementation as well as data post processing technique to produce a final density of states. Additional efforts were taken to include steps implementing the latest development of the algorithm as the replica exchange scheme.

Introduction

Wang-Landau algorithm became a tool of choice to study complex energy landscapes of the multidimensional systems with multiple degrees of freedom especially those who exhibit the phase transitions and complex behavior around the critical points [1]. Its simplicity and universality allowed application to the diverse range of phenomena. Its power came from the fact that it is able to include in its fabric almost every effective modern algorithm modeling the behavior of the spin glasses. These systems play the crucial role in the efficient complex optimization, graph theory etc. Wang-Landau algorithm is based on the Monte Carlo method and Metropolis algorithm and includes multiple methods to sample complex systems and to overcome critical slowdown around the transition points.

Methods

We reproduced a unique variation of the Monte Carlo method with Metropolis algorithm type importance sampling [2]. Applied to the density of states it is known as the Wang-Landau algorithm. Systems of interest existing nowadays are usually bigger than 64^3 points and exhibit critical slow down not only around the transition points but also during the regular thermalization stage. The single spin flip algorithm is generally inefficient. Convergence to the equilibrium values of the observable parameters with one spin flip is slow. Changes introduced by a single local spin flip propagate diffusively. One approach is to deploy a spin cluster flip algorithm. Various algorithms have been proposed to speed up the process by flipping the clusters of spins at once. Swendsen-Wang [3] and Wolff [4] Monte Carlo methods are among them. The Wolff algorithm is an improvement over the Swendsen–Wang algorithm since it has a larger probability of flipping the bigger clusters.

Another approach is to parallelize the tasks between the multiple walkers [5-6] or split the energy range into pieces. In all cases the unique and custom made procedure to unify the results from a multiple sources is required.

We used the *Open MPI* library [7] that can manage the multithread coding and feed it to the multicore CPU (central processing unit) or distribute the tasks across a network of computers connected in the computational cluster. *OpenMPI* is extensively documented in electronic resources and much easier to deploy on the Linux systems. Among the available alternatives the most efficient one is *OpenMP* project though it works at its full power on the paid platform and software [8]. In our project, Linux based *GCC* 6.3 compiler and the latest update for *Octave* package were used.

The model we used is originally credited to Domb [9] and represents a two-dimensional Euclidian *m* x *n* lattice hosting a single spin in every node of its discreet structure. The spin is taking *q* possible values distributed about the circle with the following steps

$$\theta_n = {}^{2\pi n}/_{q}\,, \tag{1}$$

where *n* goes from 1 to *q* and interaction Hamiltonian is defined as

$$\mathcal{H} = J \sum\nolimits_{i,j} \cos\left(\theta_{s_i} - \theta_{s_j}\right). \tag{2}$$

This model is known as vector Pott’s model or clock model [8]. Here J is some spin-spin coupling constant and θ_{si} and θ_{si} are the spins’ positions around the clock dial. The simpler standard Potts model has the Hamiltonian

$$\mathcal{H} = -J \sum\nolimits_{i,j} \delta(\boldsymbol{s_i}, \boldsymbol{s_j}), \tag{3}$$

where δ is the Kroneker’s delta. This delta could be replaced by a simple scalar product of two spin-vectors, where each spin can take only two values, *up* or *down*, s_i={+1,-1}. In this case it is known as the Ising model [11]. As one can see this scalar product gives twice as big energy range compared to the standard Pott’s model.

Assuming that magnetic interaction strength is dropping as fast as $1/r^3$ with the distance, we need to consider interactions only between the closest neighbors. For the rectangular grid these immediate neighbors are forming a straight cross pattern. Periodic boundary conditions are used. That is, if the left neighbor in the left corner of the interaction pattern is missing, we assume that its place is taken by the spin across the whole grid on the right boundary. The same technique is used for the right, upper and bottom boundary sites.

If the external magnetic field $\boldsymbol{B}\neq 0$ is superimposed, the total interaction energy associated with the presence of all closest neighbor spins is given by

$$E = -J \sum\nolimits_{ij} \boldsymbol{s_i s_j} - \boldsymbol{B} \sum\nolimits_{i=1}^{m\times n} \boldsymbol{s_i}, \tag{4}$$

where *J* is again the spin-spin interaction strength and ***B*** is the external magnetic field strength. Some normalization constants may be used to adjust dimensionality of the equation to the common energy units. If *J*>0 the system is ferromagnetic, otherwise, if *J*<0, it is paramagnetic. Indices i and j are sampling all available pairs of neighboring spins on the grid, excluding the double counts of *ij* and *ji* pairs. The upper bound for the first sum is determined by the range of interaction, which is a simple pairwise interaction, and by the number of spins within this range on the simulation grid.

To study these systems further we need to consider the thermodynamic partition function and density of states. For canonical ensemble the partition function Z is given by the following expression [12]

$$Z = \sum_{\substack{config\,. \\ space}} e^{-E_r/k_B T}. \qquad (5)$$

Summation is run over all configuration space, k_b is the Boltzmann constant, E_r and T are the enumerated energy values and the temperature for this particular configuration.

High degree of the system's degeneracy, that is different E_r may have the same values, allows us to describe all possible configurations only in terms of energy, that is

$$Z = \sum_{E} g(E) e^{-E/k_B T}, \qquad (6)$$

where *g(E)* is the density of states.

For our analysis we need to keep track of the two values, that is $H(E)$ – the number of times this particular configuration is observed and $g(E)$ – calculated density of states. Probability of the system transition to the next configuration is given by an expression

$$p(E_1 \to E_2) = min\left[\frac{g(E_1)}{g(E_2)}, 1\right]. \qquad (7)$$

At the very beginning $H(E)$ is taken to be zero everywhere. The unknown density of states *g(E)* is taken to be one for any value of *E*.

Both values are modified every time any configuration with this particular value of E is visited. For $g(E)$ modification factor f is chosen to be bigger than 1 and equal $e \approx 2.71828$ for this particular implementation. It is better to keep track of $g(E)$ changes through the following expression $\ln[g(E)] \rightarrow \ln[g(E)] + \ln[f]$. As soon as the condition of flatness for $H(E)$ is met we change modification factor to a new value $f_{i+1} = \sqrt{f_i}$ and repeat the procedure starting with $H(E)$ equals zero everywhere again. Measure of flatness is chosen to be $mean(H(E))/max(H(E))*100\%$. $H(E)$ is incremented by 1 every time the state with particular E is visited or the system stays with the old value of E. After multiple iterations, g(E) should converge to the real density of states for the simulated system.

Thus, to find this density of states flat histogram random walk method of Wang and Landau goes through following steps:

1. Choose at random any spin $\boldsymbol{s_i}$ and flip its sign so $\boldsymbol{s_i}`=-\boldsymbol{s_i}$.
2. Calculate the total energy according to Eq. (4).
3. Calculate the transition probability according to Eq. (7) and keep the updated configuration if the criterion is met. If not, reverse the spin flip and keep the current configuration.

Repeat the previous steps to achieve the proper flatness of the histogram.

As were told, every instance of visiting the different configuration with certain value of energy E or keeping the old one, when the transition rules in Eq. (7) are not met, is followed by an increase $\ln[g(E)] \rightarrow \ln[g(E)] + \ln[f]$ and $H(E) \rightarrow H(E)+1$.

These steps, repeated multiple times on the order of the number of sites in the simulation grid represent a single Monte Carlo step. To calculate any observable value for a given temperature or energy range, Monte Carlo steps have to be repeated numerous times. Different physical properties, including magnetization, specific heat, density of states etc, could be calculated via similar algorithms [9, 10].

Splitting the energy range between the processes require some additional thoughts and design. The different energy range pieces should overlap. Firstly, because it is required to cover the whole energy range under investigation without the gaps. Secondly, this overlap in the further development of the Wang-Landau sampling algorithm allows us to propagate the certain spin configuration between the processes. Special attention was paid to the areas with the highly improbably configurations to avoid a processes queue.

Each configuration (with particular *E* value) is enumerated and has its own slot to accumulate the number of visits *H*(*E*). It is easy to show that the number of these slots, for the m by n 2*D* grid, is 2*nm*+1.

Results and Discussions

Configuration parameters for our eight-core desktop PC are read as follows: Intel Core i7 4790K, 4.0GHz/LGA-1150/22nm/ Haswell/8Mb L3 Cache, DDR-3 16Gb/1866MHz PC14900. *Debian* Linux OS with *gccC*++ compiler and *Octave* packages were deployed.

On Figure 1 we plotted the data from our algorithm implementation for the three separate spin models. These are the Ising, Pott and Domb models. In all cases the grid size is 16^2 and the flatness criterion is 80%. For the Ising model the number of spin's orientations is 2 (spin is up or down), the clock model has *q*=7 and the standard Potts model has *q*=13. For simplicity, magnetic field ***B*** is taken to be zero everywhere.

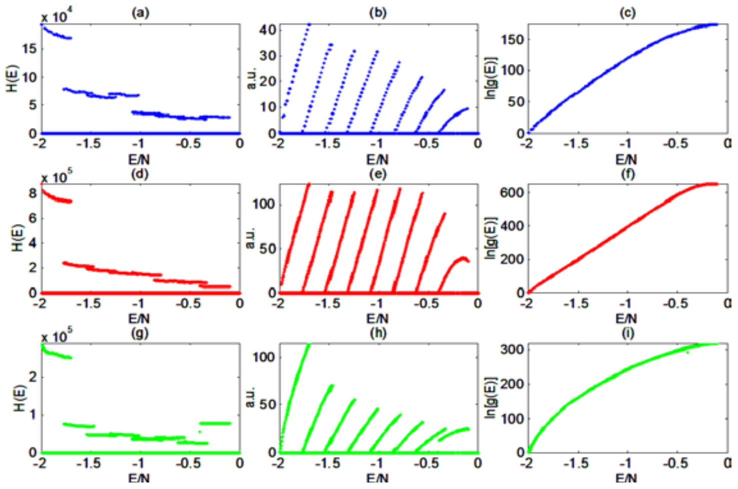


Figure 1. Three columns, from left to the right: Accumulated histogram of the *H*(*E*) visits, intermediate density function data and final reconstructed density of state *g*(*E*) function. Three rows, from top to the bottom: The first row is the Ising model, the second is clock model and the third is the standard Potts model

First left column of the plots on Figure 1 represents *H*(*E*) – the number of visits per energy value accumulated for each separate energy range for all three models right after the last modification of *g*(*E*), before flashing the data to hard drive. The separate energy ranges and their overlap extent are clearly visible on all plots. The second column is the density of states *g*(*E*) before stitching them to a single function. The thick line of zeros at the bottom represents the states included into 2*mn*+1 range but not accessible due to the system configuration and its interaction Hamiltonian. The models were implemented in *C*++ language. The output data were later displayed and further processed with *Octave* script. The third column is the final reconstructed density function.

For the sake of simplicity, we studied only the region from -2 to 0 [E/N]. In case of the Ising model the positive part is easily reconstructed by the mirror reflection. In all other cases the energy range could be expanded as far as it necessary.

One can see that the difference between the number of visits may be significant for the different pieces of the same model, see subplots (a),(d) and (g), nonetheless the first derivative (the factor determining the quality of stitching) for the two *g*(*E*) pieces at the connection points, as could be understood from the process, is mainly dependent on the smoothness of *H*(*E*).

Every data piece head is already normalized to zero in *C*++ script. The final data postprocessing is done in *Octave* package. At this stage, depending on the number of artefacts in the data, each head of the consecutive data piece is aligned with the tail of the previous. The head of the first data piece is normalized to 2. The latter is done because there are only two states with all spins aligned in one direction. If head-to-tail stitching is incorrect we are automatically switching to the different but overlapping in the energy space points. Another observation worth mentioning is that systems with multiple degrees of freedom have more states available (compare subplots (b) and (h)). Thus it takes more iterations to visit all of them (compare subplots (a) and (g)).

Conclusions

We have implemented Wang-Landau algorithm to calculate density of states based on the Monte Carlo method with importance sampling in 2D spin glasses for the Ising, Domb and Potts models as an example of multithreading and performance optimization in scientific computing.

We reported a generalized set of rules represented in terms of algorithms and programs to construct a density of stated produced by a multithread algorithm.

The algorithm works fine for all type of models and only requires correct definition of the interaction Hamiltonian in *C*++ script.

Significant speedup compare to the linear model on the multicore computer has been observed.

The model and its development represent the computational basis for the whole generation of the quantum algorithms for approximating partition.

Acknowledgments

This research was supported by grant №3824/ГФ4 provided by the Science Committee at the Ministry of Science and Education of Republic of Kazakhstan to the principal investigator at the National Nanotechnology Laboratory of Open Type, Physics and Technology Department, al-Farabi Kazakh National University.

References

1. Wang F., Landau D.P. Efficient, multiple-range random walk algorithm to calculate the density of states // Phys. Rev. Lett. – 2011. – 86. – Pp. 2050–2053.
2. Metropolis N., Rosenbluth A.W. Equation of State Calculations by Fast Computing Machines // Journal of Chemical Physics. – 1953. – 21. – Pp. 1087-1092.
3. Wang J.-S., Swendsen R.H. Cluster Monte Carlo algorithms // Physica. – A: Statistical Mechanics and its Applications. – 1990. – 167(3). – Pp. 565-579.
4. Wolff U. Collective Monte Carlo Updating for Spin Systems // Physical Review Letters. – 1989. – 62(4). – Pp. 361-364.
5. Altevogt P., Linke A. Parallelization of the two-dimensional Ising model on a cluster of IBM RISC system / 6000 workstations // Parallel Computing. – 1993. – 19(9). – Pp. 1041-1052.
6. Heermann D.W., Burkitt A.N. Parallel Algorithms in Computational Science. Springer-Verlag New York, Inc. – New York, NY, USA, 1991.

7. Open MPI project [web data base].-Information and resources. – http://www.open-mpi.org/
8. Chapman B. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). – The MIT Press: Scientific and Engineering Computation edition, 2007. – 384 p.
9. Domb C., Green M.S. Ising Model. Phase Transitions and Critical Phenomena. – Academic Press, 1974. – V.3. – Pp. 357-484.
10. Potts R.B. Some Generalized Order-Disorder Transformations // Proceedings of the Cambridge Philosophy Society. – 1952. – 48(1). – Pp. 106-109.
11. Ising E. Beitrag zur Theorie des Ferromagnetismus // Zeitschrift für Physik. – 1925. – 31. – Pp. 253-258.
12. Zhang C., Ma J. Simulation via direct computation of partition functions // Phys. Rev. E. – 2007. – 76. – Pp. 036708-15.

Questions:

1. What is the density of states?
2. Describe the vector and standard Pott's model.
3. What are the differences between the basic spin grid models?
4. What is importance sampling?
5. How is Wang-Landau algorithm is incorporated in the biased sampling scheme?
6. Describe the convergence process for the Wang-Landau sampling algorithm.
7. What is the spin cluster?
8. How we could multithread the Wang-Landau algorithm?
9. Determine the partition function for canonical ensemble.
10. What is the system degeneracy and how it affects the partition function?
11. What is the transition probability for the Wang-Landau algorithm?
12. How we measure the flatness of the histogram?
13. How the flatness of the histogram could affect the stitching procedure?

Program code

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <mpi.h>
#include <iomanip>
#include <sys/time.h>
using namespace std;
const double pi = 3.14159;
const int qq=10;
const int m=16, n=16;
int nn;

// timing functions
```

```
double get_wall_time(){
struct timeval time;
if (gettimeofday(&time,NULL)){ return 0;}
return (double)time.tv_sec + (double)time.tv_usec*.000001;}

double get_cpu_time(){return (double)clock() / CLOCKS_PER_SEC;}
double E_nergy(int *s[], double* cos_theta[], int m, int n, double J){
double s_um=0;
int up,down,left,right;

for(int i=0; i<m; i++){
for(int j=0; j<n; j++){
i==0 ? up=m-1 : up=i-1;
i==m-1 ? down=0 : down=i+1;
j==0 ? left=n-1: left=j-1;
j==n-1 ? right=0 : right=j+1;

s_um=s_um+(cos_theta[s[i][j]][s[down][j]]+
cos_theta[s[i][j]][s[up][j]]+cos_theta[s[i][j]][s[i][right]]+
cos_theta[s[i][j]][s[i][left]]);}}

return s_um*J/2;}

double dE(int *s[], double* cos_theta[], int m, int n, int i, int j, int nn ,double J){
double d_e=0;
int up,down,left,right;

i==0 ? up=m-1 : up=i-1;
i==m-1 ? down=0 : down=i+1;
j==0 ? left=n-1: left=j-1;
j==n-1 ? right=0 : right=j+1;

d_e=(
cos_theta[nn][s[down][j]]+
cos_theta[nn][s[up][j]]+
cos_theta[nn][s[i][right]]+
cos_theta[nn][s[i][left]])-

(cos_theta[s[i][j]][s[down][j]]+
cos_theta[s[i][j]][s[up][j]]+
cos_theta[s[i][j]][s[i][right]]+
cos_theta[s[i][j]][s[i][left]]);
return d_e*J;}

int main(int argc, char *argv[]){
```

```
// Start Timers
double wall0 = get_wall_time();
double cpu0 = get_cpu_time();

int id, ntasks;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
MPI_Comm_rank(MPI_COMM_WORLD, &id);

const clock_t begin_time = clock();

cout<<setprecision(3);

double r_atio=0.1;
double J=-1;
double E1t=-3*(m*n),E2t=-3*(m*n); // Just the arbitrary numbers to start

double l_ow=-2, u_pper=-0.1, l_ength=1.1*((u_pper-l_ow)/ntasks);
double small_piece=(u_pper-l_ow-l_ength)/(ntasks-1);

double E_min=l_ow+small_piece*double(id), E_max=E_min+l_ength;
long int total_steps=90000000;

double f=2.71828;
// Introduce arrays to keep log_gE and H_E data. it covers the whole energy range
int n_odes=m*n*4+1;
double* log_gE = new double[n_odes];
double* log_gE_total = new double[n_odes*ntasks];
double* H_E = new double[n_odes];
double* H_E_copy = new double[n_odes];
double* H_E_total = new double[n_odes*ntasks];

double* EpN = new double[n_odes];
for(int i=0;i<n_odes;i++){
log_gE[i]=0;H_E[i]=0;EpN[i]=double(i-2*m*n);}
double* cost_heta = new double[qq*qq];
double** cos_theta = new double*[qq];
// initialize it
for(int i = 0; i < qq; ++i){
cos_theta[i] = cost_heta + qq*i;}
// precomputing the array
for(int t = 0; t < qq; t++){
for (int q = 0; q < qq; q++){
// clock model
// cos_theta[q][t] = cos(2*pi*double(q-t)/(qq-1));
// standard Pott with delta function
// does not work properly...in the begining and for qq=2
// need to check how it's interact with ising...
(q==t)?(cos_theta[q][t]=1):(cos_theta[q][t]=0);

// Ising
// (q==t)?(cos_theta[q][t]=1):(cos_theta[q][t]=-1);
}}

// create the main, basic 2D array of spins
int* s_data = new int[m*n];
int** s = new int*[m];
// initialize it
for(int i = 0; i < m; i++){
s[i] = s_data + n*i;}

for(int j=0;j<n;j++){
for(int i=0;i<m;i++){
s[i][j] = qq-1;}} // filling array with MAX value

E1t=E_nergy(s, cos_theta, m, n, J)/(m*n);

while((E1t/(m*n)<E_min)||(E1t/(m*n)>E_max)){
int i = rand()%m; int j = rand()%n; // the indices should fall within the boundaries

//int nn=s[i][j]; // save the old spin value of the site
// need for sure another value... yes..otherwise no flip is observed
//while(nn==s[i][j]){
//          nn =rand()%qq;} // nn takes values from 0 to qq-1, because array indexed from 0 ==> cos(2pi*nn/qq)....
// nn takes values from 0 to qq-1, because array indexed from 0 ==> cos(2pi*nn/qq)....
int nn =rand()%qq;

s[i][j] = nn;
E1t=E_nergy(s, cos_theta, m, n, J);}

cout<<id<<"\t"<<E_min<<"\t"<<E_max<<"\t"<<E1t/(m*n)<<"\n";
log_gE[int(E1t)+2*m*n]+=log(f);
H_E[int(E1t)+2*m*n]+=1;

int c_ount=0;
srand (time(NULL)+id);
while(c_ount<=total_steps){c_ount++;
```

```
for (int ii=0;ii<m*n;ii++){

// the indices should fall within the boundaries
int i = rand()%m; int j = rand()%n;
// nn takes values from 0 to qq-1, because array indexed from 0 ==> cos(2pi*nn/qq)....
int nn =rand()%qq;

// should stay here. before I've change the matrix s
float d_e=dE(s, cos_theta, m, n, i,j,nn,J);

int nn_old=s[i][j];s[i][j]=nn;

E2t=E1t+d_e;
if((E_min<=E2t/(m*n)) && (E2t/(m*n)<E_max)){

if(exp(log_gE[int(E1t)+2*m*n]-log_gE[int(E2t)+2*m*n])>=
rand()/(RAND_MAX + 1.0)){
E1t=E2t;
log_gE[int(E2t)+2*m*n]+=log(f);
H_E[int(E2t)+2*m*n]+=1;

}else{// walker stays.
log_gE[int(E1t)+2*m*n]+=log(f);
H_E[int(E1t)+2*m*n]+=1;
// no spin configuration update here. Reverse s[][]
s[i][j]=nn_old;}

}else{
log_gE[int(E1t)+2*m*n]+=log(f);
H_E[int(E1t)+2*m*n]+=1;//}
// no spin configuration update here. Reverse s[][]
s[i][j]=nn_old;}
}

int max_H=1;
for(int i=0;i<n_odes;i++){
if(H_E[i]>=max_H){max_H=H_E[i];}}

int min_H=max_H;
for(int i=0;i<n_odes;i++){
if(H_E[i]>1 && H_E[i]<min_H){min_H=H_E[i];}}

(min_H!=max_H)?(r_atio=double(min_H)/double(max_H)):(r_atio=0);

if(r_atio>=0.85){
```

```
// Stop timers
double wall1 = get_wall_time();
double cpu1 = get_cpu_time();

if(wall1-wall0>=60){

cout<<setprecision(3)<<id<<"\t["<<min_H<<":"<<max_H<<"]\t["<<E_min<<":"
<<E_max<<"]\t"<<c_ount<<"\t"<<setprecision(5)<<f<<setprecision(3)<<"\t"<<
(wall1-wall0)/60<<"m\t"<<(cpu1-cpu0)/60<<"m\n";}
else{
cout<<setprecision(3)<<id<<"\t["<<min_H<<":"<<max_H<<"]\t["<<E_min<<":"
<<E_max<<"]\t"<<c_ount<<"\t"<<setprecision(5)<<f<<setprecision(3)<<"\t"<<
(wall1-wall0)<<"s\t"<<(cpu1-cpu0)<<"s\n";}

// Start Timers
double wall0 = get_wall_time();
double cpu0 = get_cpu_time();

f=sqrt(f);
for(int i=0;i<n_odes;i++){H_E_copy[i]=H_E[i];H_E[i]=0;}

c_ount=0;}

if (f<=1.0001){break;}}

MPI_Gather(H_E_copy,n_odes,MPI_DOUBLE,H_E_total,n_odes,MPI_DOUBLE,0,MPI_COMM_WORLD);

MPI_Gather(log_gE,n_odes,MPI_DOUBLE,log_gE_total,n_odes,MPI_DOUBLE,0,MPI_COMM_WORLD);

// stitching pieces of different energy range together
if(id==0){
ofstream gfile("log_gE_HE.txt");
H_E_total[0]=ntasks; // need to save number of tasks for Matlab reader
for(int j=0;j<ntasks;j++){
for(int i=0;i<n_odes;i++){
gfile<<EpN[i]/(m*n)<<"\t"<<log_gE_total[i+j*n_odes]<<"\t"
<<H_E_total[i+j*n_odes]<<"\n";}}
gfile.close();}

// dynamic arrays clean up
```

```
//************************
delete[] EpN;

delete[] s_data;
delete[] s;

delete[] cost_heta;
delete[] cos_theta;

delete[] log_gE;
delete[] H_E_copy;
delete[] H_E;
delete[] H_E_total;
delete[] log_gE_total;

MPI_Finalize();

// Stop timers
// double wall1 = get_wall_time();
//double cpu1 = get_cpu_time();
//
//cout<<id<<"\t"<<(wall1-wall0)/60<<"\t"<<(cpu1-cpu0)/60<<"\n";

return 0;}
```

Matlab Script for Data Reconstruction

```
%close all;
clear all; clc;
load('log_gE_HE.txt')

c_olor=['k*' 'rp' 'go' 'bv' 'm^' 'c*'];
co_lor=c_olor(1+round(rand*5));

%subplot(2,1,1)
figure(1)
d_ata=log_gE_HE(:,2);
for i=1:length(d_ata)
if d_ata(i)~=0
% plot(log_gE_HE(i,1),d_ata(i),co_lor);hold on;drawnow;
% axis square;axis tight;
end
end
title('g(E)')
%subplot(2,1,2)
figure(2)
d_ata=log_gE_HE(:,3);
```

```
for i=1:length(d_ata)
if d_ata(i)~=0
%plot(log_gE_HE(i,1),d_ata(i),co_lor);hold on;drawnow;
%axis square;axis tight;
end
end
title('H(E)');
ylabel('ln[g(E)]');
xlabel('E/N');

% stitching procedure
ntasks=log_gE_HE(1,3);
%ntasks=8;
n_odes=floor(length(log_gE_HE(:,1))/ntasks)

for i=2:ntasks
% find the index of the first nonzero_right element in the data subset...
non_zero_right=i*n_odes;
for j=1:n_odes-1 % moving backward through subset
if log_gE_HE(i*n_odes-j,2)~=0
non_zero_right=i*n_odes-j;
end
end

s_hift=log_gE_HE(non_zero_right-n_odes,2)-log_gE_HE(non_zero_right,2)

% rise/lower the whole RIGHT subset of data according to the findings of the previous loop
for j=1:n_odes
if log_gE_HE((i-1)*n_odes+j,2)~=0
log_gE_HE((i-1)*n_odes+j,2)=log_gE_HE((i-1)*n_odes+j,2)+s_hift;
end
end
end

figure(3)
sub_truct=0;
for i=1:length(d_ata)
if d_ata(i)~=0
sub_truct=d_ata(i);break;
end
end
d_ata=log_gE_HE(:,2);
points_counter=0;
for i=1:length(d_ata)
```

```
points_counter=points_counter+1;
if mod(points_counter,n_odes)==0
co_lor=c_olor(1+round(rand*5))
end
if d_ata(i)~=0
%           plot(log_gE_HE(i,1),d_ata(i)-199993,co_lor);hold on;drawnow;
plot(log_gE_HE(i,1),d_ata(i)-d_ata(1),co_lor);hold on;drawnow;
%           plot(log_gE_HE(i,1),d_ata(i),co_lor);hold on;drawnow;
%           plot(log_gE_HE(i,1),d_ata(i)/d_ata(1),co_lor);hold on;drawnow;
axis square;axis tight;
end
end
title('g(E)')
ylabel('ln[g(E)]');
xlabel('E/N');
%print -deps -color single_cluster.eps
%print -dpng -color single_cluster.png
```

5

MICROSOFT VISUAL STUDIO IDE WITH OPENMP ADD-ON USED TO BUILD A RADIOLOGICAL PHANTOM, ITS PROJECTIONS AND RECONSTRUCTED CROSS-SECTIONS

Introduction

OpenMP API has been developed as an open standard for parallel programing in *C*, *C++* and *Fortran* languages. It has a full a set of compiler directives, library routines and environment variables that are dedicated for programming and running the multi-threaded applications on multiprocessor systems with the shared memory.

The most exemplary usage has been done and widely available on the Microsoft Visual Studio integrated development environment.

Several reasons are mentioned to motivate one to use OpenMP: First, OpenMP is the most widely standard for SMP (symmetric multiprocessing) systems; Second, it supports three differentmodern languages (Fortran, C, C++), and it has been implemented by many vendors, and the last but not the least is that OpenMP is arelatively small and simple specification, and it supports incremental parallelism whereincremental parallelism is a technique for parallelizing an existing serial program, in which the parallelization is introduced as a sequence of incremental changes, parallelizing one loop at a time. After each transformation the program is tested to ensure that its behavior is the same as the original program, making it much easier to insure that no undetected bugs have been introduced.

Nowadays, a lot of research is done on OpenMP including the major commercial software products.

When one want to compare OpenMPand OpenMPI, he should know that MPI which stands for Message Passing Interface has become accepted as a portable, cross-platform, style of parallel programming. MPI in general is considered as difficult to program and doesn't support incrementalparallelization of an existing sequential program, though one could really see it as a matter of preferences. MPI was initiallydefined for client/server typeof programming to run across a network, and so includes time and resources costlysemantics for message queuing and selection, and requires the existences of wholly separatememories.

A significant part of OpenMP functionality is implemented using compiler directives. They must be explicitly placed by the user at the proper execution level, which will allow the program to run in parallel mode. In C/C++, the OpenMP directives are defined by the #pragma wordexisting both in C and C ++ standards, and used to specify additional instructions to the compiler. The use of the special key "omp" directive indicates that the commands are related to OpenMP and in order to avoid accidental coincidence of the names of OpenMP directives with other names.

Methods

Our task is to create a numerical phantom sized up to 1024 by 1024 by 1024 voxels and provide for each voxel an information for the type of the material and attenuation coefficents μ, so later on this information could be used to create a projection in the illuminatinggamma ray field.

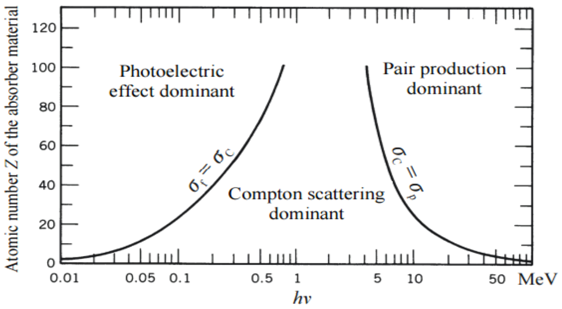


Figure 1. Gamma rays attenuation processes as a function of energy, see [1]

When we are talking about the gamma rays attenuation by matter we need to consider three basic mecanisms of the gamma ray scattering, which is the photoelectric effect, Compton scattering, and pair production. Depending on the situation one or two of these meachnism may become the more prominent than the rest of them, see Figure 1.

For the sake of numerical simulations, all three of them a usually combined into the single μ value, depending on the energy of gamma ray quanta, see figure 2.

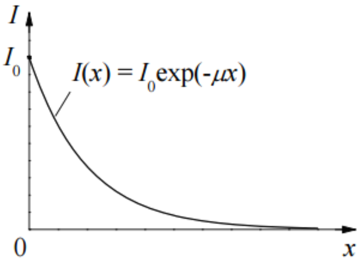


Figure 2. Gamma rays attenuation processes as a function of distance, see [1]

In this case, the gamma rays attenuation process is reduced to a simple exponential form

$$I=I_0*exp(-\mu x), \qquad (1)$$

where I and I0 are the intensities after and before the gamma ray traverses the sample, μ is the linear attenuation coefficient measured in inverse cantimeters. There are variations of this formula where the density of the matter is involved. In this case, the units of mu should ne given in inverse cantimeters squared.

This research is using the tables of X-Ray Mass Attenuation Coefficientsand Mass Energy-Absorption Coefficientsfrom 1 keV to 20 MeV for Elements Z = 1 to 92 and 48 Additional Substances of Dosimetric Interest provided by the Physical Measurement Laboratory at National Institute of Standards and Technology, Gaithersburg, MD, USA [2].

Sample illumination setup is given on the next figure 3. The major parameters introduced onm this figure are the source elevation above the central line, distances from the source to the center of the sample and from the center of the sample to the detector, bothe measured along the central ray. As we can see the phantom is placed in the center of coordinate system and could rotate around all three axises. Geometrical dimensions of the detector plate are given as well. Detector could be moved in the *ZY* plane as well, in case it is needed by the shifted image of the phantom's projection.

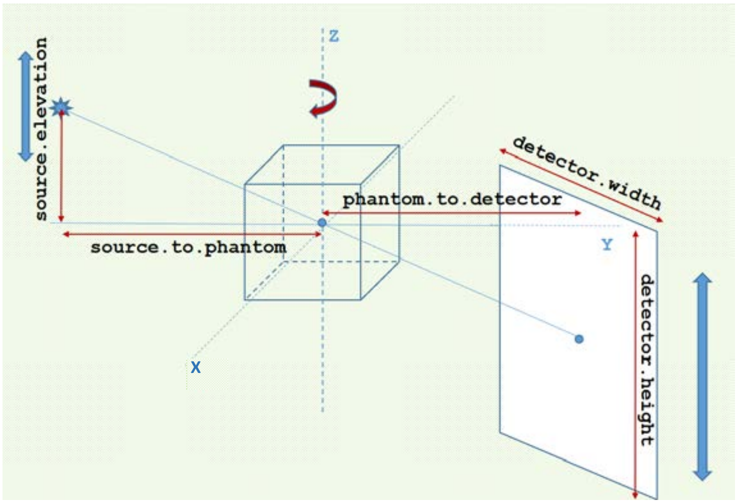


Figure 3. Phantom illumination setup

Results and Discussions

The separate Phantom class has been written to trace a projection according to the task, see the appendices to this chapter with the program code.

Three projections, with nonzero elevation of the x-ray source, and the additional modulation coefficient proportional to the inverse square of the distance from the source to the detector cell are given, see figure 4. First projection (a) is the cube filled with inclussions of different nature, such whater, blood etc. The second projection (b) is the original projection with Gaussian noise added. The third one is different from the original by addition of the white noise.

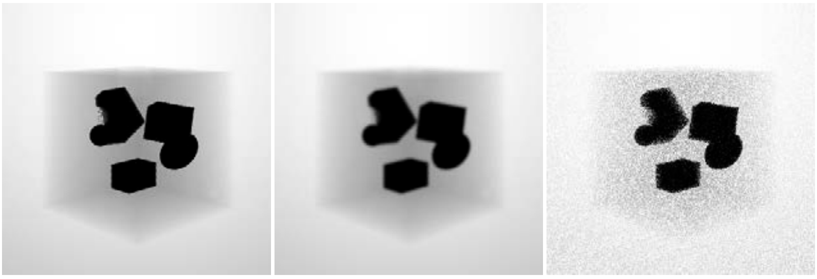
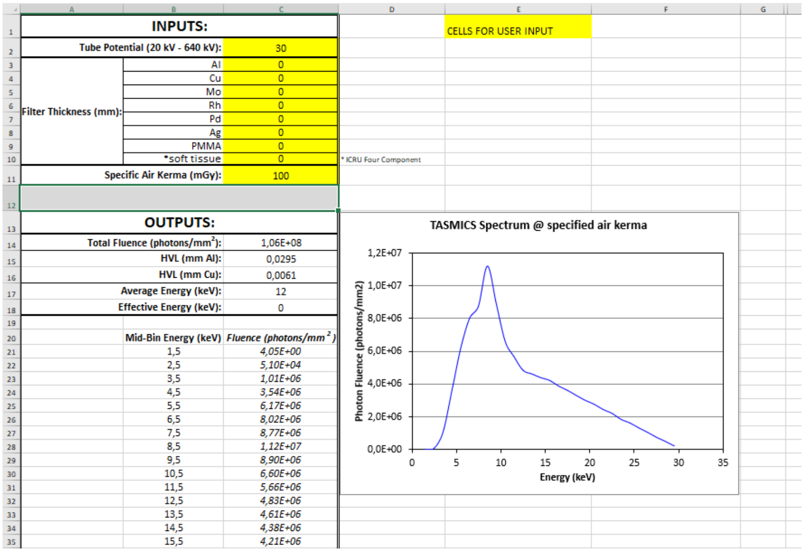

(a) original projection (b) Gaussian noise added (c) white noise added

Figure 4. Multiple projections generated under different registration conditions

The sample energy spectrum data are given by the following picture provided by the x-ray source producing company.



INPUTS:		
Tube Potential (20 kV - 640 kV):		30
Filter Thickness (mm):	Al	0
	Cu	0
	Mo	0
	Rh	0
	Pd	0
	Ag	0
	PMMA	0
	*soft tissue	0
Specific Air Kerma (mGy):		100
OUTPUTS:		
Total Fluence (photons/mm^2):		1,06E+08
HVL (mm Al):		0,0295
HVL (mm Cu):		0,0061
Average Energy (keV):		12
Effective Energy (keV):		0
	Mid-Bin Energy (keV)	***Fluence (photons/mm^2)***
	1,5	*4,05E+00*
	2,5	*5,10E+04*
	3,5	*1,01E+06*
	4,5	*3,54E+06*
	5,5	*6,17E+06*
	6,5	*8,02E+06*
	7,5	*8,77E+06*
	8,5	*1,12E+07*
	9,5	*8,90E+06*
	10,5	*6,60E+06*
	11,5	*5,66E+06*
	12,5	*4,83E+06*
	13,5	*4,61E+06*
	14,5	*4,38E+06*
	15,5	*4,21E+06*

Figure 5. Sample data sheet for the number of photons per energy bin as provided by the manufacturer

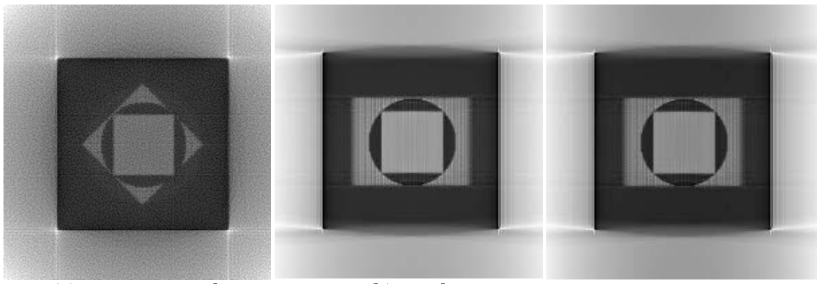
Names of the elements used for the phantom generation is taken from the input_file.txt text file, see the end of the chapter and mu data is extracted according to the energy bins stepping used in the x-ray source spectrum description.

The backprojection according to the Radon transform is done in a seoarate class *Loadprojection.cpp*, see figure 6.

In both cases, see the program code below, we have parallelized the loop scaning the voxels of the detectors from top to the buttom. This approach was good enough to produce a reasonable performance.

Phantom which has been used to reconstruct these slices is shown below on figure 6. As one can see, we have pretty good resolution and feature's visibility.

Artifacts, visible on the pictures as the light rays streaking from the sharp angles of the phantom originates from the beam hardening.



(a) coronal (b) axial (c) sagittal

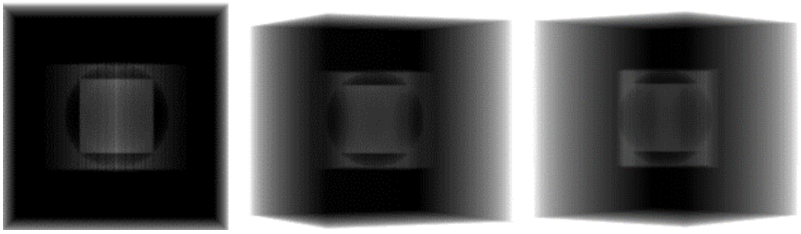
Figure 6. Reconstructed sagittal, axial and coronal slices of the arbitrary phantom

Beam hardening is seen with polychromatic gamma ray spectra. As the ray passes through the body, low energy gamma quanta are attenuated, absorbedand scattered more easily, while the remaining high energy quanta stay on course and do not diminish in number drastically. Thus, beam transmission does not follow the simple exponential decay seen with a monochromatic ray, see figure 2. This is observed to be a particular problem with high atomic number materials such as bone, iodine, or metal. Compared to low atomic number materials such as water, or for the low energy X-ray tubes, these high atomic number materials have dramatically increased attenuation at lower energies thus making the passing beam «harder».

This particular crossections have been reconstructed from the 361 projections taken from the phantomwith one degree step, see figure 7.

Acknowledgments

This research was commissioned and fundedby the ADANI Company, producing and manufacturing Medical, Security, Spectroscopy and Inspection X-ray Systems, Minsk, Belarus.



(a) α=0^0 (b) α=26^0 (c) α=52^0

Figure 7. Sample projections from the reconstructed phantom taken ad different angles

References

1. http://www.nuclear-power.net/nuclear-power/reactor-physics/interaction-radiation-matter/interaction-gamma-radiation-matter/gamma-ray-attenuation/
2. https://www.nist.gov/pml/x-ray-mass-attenuation-coefficients
3. Kak A.C. and Slaney M. Principles of Computerized Tomographic Imaging. IEEE Press. – New York, NY, USA, 1988.

Questions:

1. What is the motivation behind the building the radiological numerical phantom.
2. What is main differences between *OpenMP* and *OpenMPI*? What are their?
3. How we could introduce dynamically allocated arrays into a parallel region?
4. What are the most obvious sources of the memory leaks in OpenMP parallel programming?
5. What is the difference between projections altered with Gaussian and with white noise?
6. What are the main featrures of OpenCV that have been used in this program?
7. How we address the phantom rotations in our simulations?

8. What are the main mechanisms of gamma ray attenuation in matter?
9. What is registered by the detector arrays? If it is an intensity or the number of photons striking the detector, then how we can get the attenuation coeeficient distribution?
10. What should be closely monitored to maintain a proper performance of the multiple nested loops construction?

Program code

Phantom.h

```
#ifndef PHANTOM_H
#define PHANTOM_H
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <ctime>
#include <cstring>
#include <sstream>
#include <omp.h>

#include<algorithm>

#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#define SSTR( x ) static_cast< std::ostringstream &>( \
( std::ostringstream() << std::dec << x ) ).str()

class Phantom {
int w_idth, d_epth, h_eight, interpolation_points, elements_count;
// Blurred and noise added projection paramenters
int gaussblur_kernel, gaussnoise_sigma, gaussnoise_mean;
float a_multiplier;
public:
// Phantom variables
uchar *ph_antom;

Phantom(int, int, int);
~Phantom();

void create_phantom(uchar);
void grow_random_defects(int, int, int, int);
void save_phantom();
void o_bject(int,int,int,int,int,int,int,float, float, float,int);
void set_blur_and_noise(int,float,int,int);
// Projections variables
int x_source, y_source, z_source;
int x_detector, y_detector, z_detector ;
void create_projection(int, int, int, float, int, int, float*[], float*[], int,int, float, int, int, float, float, int);};
#endif
```

Phantom.cpp

```
#include "Phantom.h"
using namespace cv;
using namespace std;

const float pi = 3.14159f;
const float rad_grad = 0.01745f;

Phantom::Phantom(int a, int b, int c) {
w_idth = a; d_epth = b; h_eight = c;
ph_antom = new (std::nothrow) uchar[w_idth*d_epth*h_eight];
if (ph_antom == 0) { std::cout << "\n Error assigning memory \n"; }
}

Phantom::~Phantom(){
// This destructor function for class Phantom deallocates the directory's
// list of Entry objects.
delete [] ph_antom;}

// main bulk phantom structure
void Phantom::create_phantom(uchar phantom_material){
std::fill(ph_antom, ph_antom + w_idth*d_epth*h_eight, phantom_material);}

void Phantom::grow_random_defects(
int material_1,int quantity_1, int material_2,int quantity_2){
int defect_material[2], defect_quantity[2];
defect_material[0]=material_1; defect_material[1]=material_2;
defect_quantity[0]=quantity_1; defect_quantity[1]=quantity_2;

srand(static_cast<unsigned char>(time(NULL)));
for (int d_efects=0;d_efects<=1;++d_efects){
for(int ie=1;ie<=defect_quantity[d_efects];ie++){
// randomly choose defects shape. Two shapes are available: ellipsoids and cuboids
```

```
int defect_type=rand()%2;
// defect's random dimensions
int max_size=w_idth/4;
int min_size=w_idth/8;

int widt_h= min_size + rand()%max_size;
int dept_h= min_size + rand()%max_size;
int heigh_t=min_size + rand()%max_size;
// location of the defect's principal corner
int x=widt_h+ rand()%(w_idth-widt_h);
int y=dept_h+ rand()%(d_epth-dept_h);
int z=heigh_t+ rand()%(h_eight-heigh_t);
// defect's random orientations
float alpha= float(rand())/RAND_MAX*2*pi;
float beta = float(rand())/RAND_MAX*2*pi;
float gamma= float(rand())/RAND_MAX*2*pi;
// 3D rotation matrices
float Rx[3][3]={1,0,0,
0,cos(alpha),-sin(alpha),
0,sin(alpha),cos(alpha)};
float Ry[3][3]={cos(beta),0,sin(beta),
0,1,0,
-sin(beta),0,cos(beta)};
float Rz[3][3]={cos(gamma),-sin(gamma),0,
sin(gamma),cos(gamma),0,
0,0,1};
for(int k=0;k<widt_h;++k){
for(int l=0;l<dept_h;++l){
for(int m=0;m<heigh_t;++m){
// consecutive rotations about three major axis as defined by the input_file.txt
// rotation around X axis
float k1=k*Rx[0][0]+l*Rx[0][1]+m*Rx[0][2];
float l1=k*Rx[1][0]+l*Rx[1][1]+m*Rx[1][2];
float m1=k*Rx[2][0]+l*Rx[2][1]+m*Rx[2][2];
// rotation around Y axis
float k2=k1*Ry[0][0]+l1*Ry[0][1]+m1*Ry[0][2];
float l2=k1*Ry[1][0]+l1*Ry[1][1]+m1*Ry[1][2];
float m2=k1*Ry[2][0]+l1*Ry[2][1]+m1*Ry[2][2];
// rotation around z axis + translation + transform to index notation
int X= static_cast<int>(k2*Rz[0][0]+l2*Rz[0][1]+m2*Rz[0][2]+x);
int Y= static_cast<int>(k2*Rz[1][0]+l2*Rz[1][1]+m2*Rz[1][2]+y);
int Z= static_cast<int>(k2*Rz[2][0]+l2*Rz[2][1]+m2*Rz[2][2]+z);
// growing the defect inside the phantom's matrix
if((X>=0)&&(Y>= 0)&&(Z>=0)&&(X<=w_idth-1)
&&(Y<=d_epth-1)&&(Z<=h_eight-1)&&
// next line makes sure that voids destroys everything
(ph_antom[X + Y*w_idth + Z*w_idth*d_epth]!=0)){
if(defect_type==0){// ellipsoids
if( pow((k-widt_h/2.0),2)/(widt_h*widt_h/4.0)+
pow((l-dept_h/2.0),2)/(dept_h*dept_h/4.0)+
pow((m-heigh_t/2.0),2)/(heigh_t*heigh_t/4.0)<=1)
{ph_antom[X + Y*w_idth + Z*w_idth*d_epth]=defect_material[d_efects];}}
else{
//cuboids
ph_antom[X + Y*w_idth + Z*w_idth*d_epth]=defect_material[d_efects];}}
}}}}}}

void Phantom::o_bject(int object_shape,
int object_x,int object_y,int object_z,
int object_width,int object_depth,int object_height,
float object_alpha, float object_beta, float object_gamma,
int object_id){

float Rx[3][3]={1,0,0,
0,cos(object_alpha),-sin(object_alpha),
0,sin(object_alpha),cos(object_alpha)};
float Ry[3][3]={cos(object_beta),0,sin(object_beta),
0,1,0,
-sin(object_beta),0,cos(object_beta)};
float Rz[3][3]={cos(object_gamma),-sin(object_gamma),0,
sin(object_gamma),cos(object_gamma),0,
0,0,1};
for(int k=0;k<object_width;++k){
for(int l=0;l<object_depth;++l){
for(int m=0;m<object_height;++m){
// rotation around X axis
float k1=k*Rx[0][0]+l*Rx[0][1]+m*Rx[0][2];
float l1=k*Rx[1][0]+l*Rx[1][1]+m*Rx[1][2];
float m1=k*Rx[2][0]+l*Rx[2][1]+m*Rx[2][2];
// rotation around Y axis
float k2=k1*Ry[0][0]+l1*Ry[0][1]+m1*Ry[0][2];
float l2=k1*Ry[1][0]+l1*Ry[1][1]+m1*Ry[1][2];
float m2=k1*Ry[2][0]+l1*Ry[2][1]+m1*Ry[2][2];
// rotation around z axis + translation + transform to index notation
int X=
static_cast<int>(k2*Rz[0][0]+l2*Rz[0][1]+m2*Rz[0][2]+float(object_x));
int Y=
static_cast<int>(k2*Rz[1][0]+l2*Rz[1][1]+m2*Rz[1][2]+float(object_y));
int Z = static_cast<int>(k2*Rz[2][0] + l2*Rz[2][1] + m2*Rz[2][2] +
float(object_z));
// growing the defect inside the phantom's matrix
if((X>=0)&&(Y>=0)&&(Z>=0)&&(X<=w_idth-1)&&
(Y<=d_epth-1)&&(Z<=h_eight-1)&&
// next line makes sure that voids destroys everything
(ph_antom[X + Y*w_idth + Z*w_idth*d_epth])!=0){
if(object_shape==0){// ellipsoids
if( pow((k-object_width/2.0),2)/(object_width*object_width/4.0)+
pow((l-object_depth/2.0),2)/(object_depth*object_depth/4.0)+
pow((m-object_height/2.0),2)/(object_height*object_height/4.0)<=1)
{ph_antom[X + Y*w_idth + Z*w_idth*d_epth]=object_id;}}
else{ //cuboids
ph_antom[X + Y*w_idth + Z*w_idth*d_epth]=object_id;}}
}}}}

void Phantom::save_phantom(){
// saving generated data
// I'm using three elements of the phantom data to save info about size of the matrix
double dtime1 = omp_get_wtime();
FILE* fout;
fopen_s(&fout, "phantom.txt", "wb");
for (int k = 0; k < h_eight; ++k) {
for (int j = 0; j < d_epth; ++j) {
for (int i = 0; i < w_idth; ++i) {
int c_ounter = i + j*w_idth + k*w_idth*d_epth;
if (c_ounter == 0) { fprintf(fout, "%u\t", w_idth); }
else if (c_ounter == 1) { fprintf(fout, "%u\t", d_epth); }
else if (c_ounter == 2) { fprintf(fout, "%u\t", h_eight); }
else { fprintf(fout, "%u\t", ph_antom[c_ounter]); }
}fprintf(fout, "\n");}}
fclose(fout);
dtime1 = omp_get_wtime() - dtime1;
std::printf("time to save phantom to text file in seconds = %f\n\n", dtime1);
}

void Phantom::set_blur_and_noise(int gaussblurkernel, float a_value,
int noisemean, int gaussnoisesigma) {
gaussblur_kernel = gaussblurkernel;
gaussnoise_sigma = gaussnoisesigma;
gaussnoise_mean=noisemean;
a_multiplier =a_value;}

void Phantom::create_projection(int y_source, int z_source, int y_detector,
float theta_phantom,
int w_detector, int h_detector,
float** interpolated_mu, float** s_pectrum , int interpolation_points,
int elements_count,float total_photons, int image_compression, int voxels_per_cm,
float t_before, float t_after, int mamo){
float a_ngle = theta_phantom;
if (mamo == 1) { theta_phantom = 0; };

// t_before and t_after are used to cut the time of calculations over the empty space
cout << y_source << "\t" << z_source << "\t" << y_detector << "\t"
<< t_before << "\t" << t_after << "\n";

double dtime = omp_get_wtime();

float c_os=cos(theta_phantom*rad_grad);
float s_in=sin(theta_phantom*rad_grad);

float y_dist = float(y_detector - y_source);
float y_dist_cos = y_dist*c_os;
float y_dist_sin = y_dist*s_in;

float y_source_cos = static_cast<int>(round(float(y_source)*c_os + w_idth / 2.0f));
float y_source_sin = static_cast<int>(round(float(y_source)*s_in - d_epth / 2.0f));
float z_source_add = static_cast<int>(round(float(z_source) + h_eight / 2.0f));

float voxel_size = 1.0f / float(voxels_per_cm); // cm

// need to move the center of detector back and fourth to keep projection in the center
// depending on the source's elevation
float z_detector = float(z_source*y_detector) / float(abs(y_source));
// in case of mammography the detector is not moving
if (mamo == 1) { z_detector = 0; }

// original projection
Mat img_float = cv::Mat(h_detector, w_detector, CV_32FC1, Scalar::all(0));
Mat img;// = cv::Mat(h_detector, w_detector, CV_16UC1); // original projection

// gauss blur added projection
Mat g_auss;
// random white gaussian noise
Mat n_oise = cv::Mat(h_detector, w_detector, CV_16UC1);

// to keep sqrt(1+a*img_float) data
Mat s_um;
Mat noise_float; // random white gaussian noise type float
```

```
// some auxilary noise conversion data, see algorithm below
Mat add_noise = cv::Mat(h_detector, w_detector, CV_32FC1);
Mat img_noise = cv::Mat(h_detector, w_detector, CV_32FC1);
Mat img_noise_uchar = cv::Mat(h_detector, w_detector, CV_16UC1);

omp_set_dynamic(0); // Explicitly disable dynamic teams
// Get the number of processors in this system
int c_ores = omp_get_num_procs();
// Now set the number of threads
omp_set_num_threads(c_ores);

#pragma omp parallel
{
float * modified_photons;
modified_photons = new (std::nothrow) float[interpolation_points];
if (modified_photons == 0) { std::cout << "\n Error assigning memory \n"; }

float *e_xp;
e_xp = new (std::nothrow) float[interpolation_points*(elements_count + 1)];
if (e_xp == 0) { std::cout << "\n Error assigning memory \n"; }

// precompute exponent' argument for given elements and spectrum
// for one voxel step!!!!
for (int i = 0; i < interpolation_points; i++) {
for (int j = 1; j <= elements_count; j++) {
// interpolated_mu[j][interpolation_points] is density rho
// overall expression is attenuation with one voxel step
// our dt is about one voxel in all directions
e_xp[j + i*elements_count] =
float(exp(-interpolated_mu[j][i] * interpolated_mu[j][interpolation_points]
* voxel_size));
}}

// we multithread over the rows of the detector matrix
#pragma omp for
for (int j = 0; j<img_float.rows; j++) {

float z_dist = float(j) - float(h_detector) / 2.0f - z_detector - float(z_source);

for (int i = 0; i<img_float.cols; i++) {
// we have number of photons... should we square it?
float i_ntensity = 0;
// copying the source's spectrum at the origin
for (int k = 0; k < interpolation_points; k++) {
modified_photons[k] = s_pectrum[k][1];}
```

```
float x_dist = float(i) - float(w_detector) / 2.0f;
float x_dist_sin = x_dist*s_in;
float x_dist_cos = x_dist*c_os;
float xcos_ysin = x_dist_cos - y_dist_sin;
float xsin_ycos = x_dist_sin + y_dist_cos;

// let us assume that the sample is int he air chamber...
int inde_x = 1; // the air!!!

// distance in voxels
float dist_to_voxel = pow(x_dist*x_dist+y_dist*y_dist +z_dist*z_dist, 0.5);
for (int energy_bins = 0; energy_bins < interpolation_points; energy_bins++) {
modified_photons[energy_bins] *=
float(exp(-interpolated_mu[inde_x][energy_bins]
* interpolated_mu[inde_x][interpolation_points]* voxel_size*dist_to_voxel));}

float dt = pow((y_dist*y_dist + x_dist*x_dist + z_dist*z_dist), -0.5f);
float r2 = (y_dist*y_dist + x_dist*x_dist + z_dist*z_dist)*(voxel_size*voxel_size);

int Xt, Yt, Zt;
// tracing only through the region occupied by a phantom
for(float t= t_before;t<=t_after;t+=dt){
if (mamo == 0) {
// we have to move the origin to the center of the phantom by adding w_idth/2 etc.
// need two lines below for right solution
// fancy way to speed up calculations by static_cast<int>
Xt = static_cast<int>(xcos_ysin*t - y_source_sin); //
Yt = static_cast<int>(xsin_ycos*t + y_source_cos);
Zt = static_cast<int>(z_dist*t + z_source_add);
}
else {
// this part has not been modified for long time.
// phantom is immediately beside the detector MAMO case
Xt = static_cast<int>((                    (x_dist*t + floor(w_idth / 2.0f))));
Yt = static_cast<int>((float(y_source) + (y_dist*t + floor(d_epth / 2.0f))));
Zt = static_cast<int>((float(z_source) + (z_dist*t + floor(h_eight / 2.0f))));
}

// if the ray hits the phantom we do calculations
if( (Xt>=0) && (Yt >= 0) && (Zt >= 0) && (Xt<w_idth) && (Yt<d_epth)
&& (Zt<h_eight) ){
int inde_x = ph_antom[Xt + Yt*w_idth + Zt*w_idth*d_epth];
if (inde_x != 0) {// we did not strike the void
for (int energy_bins = 0; energy_bins < interpolation_points; energy_bins++){
modified_photons[energy_bins]*=e_xp[inde_x + energy_bins*elements_count]
/= e_xp[1 + energy_bins*elements_count];}} // reversing the effect of air voxel
}}

// accumulate illumination from the whole spectrum
for (int energy_bins = 0; energy_bins < interpolation_points; energy_bins++) {
i_ntensity += modified_photons[energy_bins];}
//i_ntensity +=modified_photons[9];}

// in case we need to attenuate the intensity proportionally to 1/r^2
//i_ntensity /= r2;
img_float.at<float>(h_detector - 1 - j, w_detector - 1 - i) = i_ntensity;
}}}

//recreate/overwrite/redifine the dynamic variable in order to delete them
// bc does not work another way
#pragma omp parallel
{
float * modified_photons;
modified_photons = new (std::nothrow) float[interpolation_points];
if (modified_photons == 0) { std::cout << "\n Error assigning memory \n"; }
delete[]modified_photons;

float *e_xp;
e_xp = new (std::nothrow) float[interpolation_points*(elements_count + 1)];
if (e_xp == 0) { std::cout << "\n Error assigning memory \n"; }
delete[]e_xp;

}

normalize(img_float, img_float, 0, 65535, CV_MINMAX);
img_float.convertTo(img, CV_16UC1);

vector<int> compression_params;
compression_params.push_back(CV_IMWRITE_PNG_COMPRESSION);
compression_params.push_back(image_compression);
string pa_th =
"C:\\Users\\Martha\\Google Drive\\reconstruction_64_bit\\projections\\
old_projections\\air_chamber_900_200_512\\";
//string pa_th =
"C:\\Users\\Martha\\Google Drive\\reconstruction_64_bit\\projections\\
old_projections\\no_r2_900_200_mono_1024\\";
```

```
//string pa_th = "C:\\Users\\Martha\\Google Drive\\
reconstruction_64_bit\\projections\\old_projections\\no_r2_200_200_1024\\";
//string pa_th = "C:\\Users\\Martha\\Google Drive\\
reconstruction_64_bit\\projections\\old_projections\\no_r2_200_200\\";
//string pa_th = "C:\\Users\\Martha\\Google Drive\\
reconstruction_64_bit\\projections\\old_projections\\without_r2\\";
string projectio_n="projection_";
string extensio_n=".png";
string extension_gauss= "_gauss.png";
string extension_rand = "_rand.png";
string extension_noise = "_noise.png";

string counte_r = SSTR(a_ngle);

string filenam_e = pa_th+projectio_n + counte_r + extensio_n;
string filename_gauss = projectio_n + counte_r + extension_gauss;
string filename_noise = projectio_n + counte_r + extension_noise;

std::cout << filenam_e<<"\n";
std::cout << filename_gauss << "\n";
std::cout << filename_noise << "\n";

// 1. Gaussian blur
GaussianBlur(img, g_auss, Size(gaussblur_kernel, gaussblur_kernel), 0, 0);
// 2. Adding white noise
// another uchar to float conversion, i.e. working with smaller 0-255 range of values now
img.convertTo(img_float, CV_32FC1);
float a = a_multiplier;
sqrt(1+a*img_float, s_um); // s_um is float

// unsigned char like in original projection
randn(n_oise, gaussnoise_mean, gaussnoise_sigma);
n_oise.convertTo(noise_float, CV_32FC1); // conversion without scaling
multiply(noise_float,s_um, add_noise); // multiplication

// addition of two floats
img_noise = img_float + add_noise;
img_noise.convertTo(img_noise_uchar, CV_16UC1);
// clipping in the line above to 65535 value is done automaticaly

cv::imwrite(filenam_e.c_str(), img, compression_params);
//cv::imwrite(filename_gauss.c_str(), g_auss, compression_params);
//cv::imwrite(filename_noise.c_str(), img_noise_uchar, compression_params);

dtime = omp_get_wtime() - dtime;
```

```
std::printf("elapsed build time in seconds = %f\n\n", dtime);
}
```

Loadprojection.h

```
#ifndef LOADPROJECTION_H
#define LOADPROJECTION_H

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <ctime>
#include <cstring>
#include <sstream>
#include <omp.h>

#include<algorithm>

#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#define SSTR( x ) static_cast< std::ostringstream &>( \
( std::ostringstream() << std::dec << x ) ).str()

class Loadprojection {
int y_source, z_source, y_detector;
int w_detector, h_detector, projections_number;
public:
int slice_size;
float *o_bject;
Loadprojection(int, int, int,int,int,int);
~Loadprojection();
void load_images();
};
#endif
```

Loadprojection.cpp

```
#include "Loadprojection.h"

using namespace cv;
using namespace std;

const float pi = 3.14159f;
const float rad_grad = 0.01745f;

Loadprojection::Loadprojection(int ysource, int zsource, int ydetector, int wdetector,

int hdetector, int projectionsnumber) {
y_source = ysource;  z_source = zsource;
y_detector = ydetector;
w_detector = wdetector; h_detector = hdetector;
projections_number = projectionsnumber;
slice_size = 130;
o_bject = new (std::nothrow) float[slice_size * slice_size * slice_size];
if (o_bject == 0) { std::cout << "\n Error assigning memory \n"; }
for (int k = 0; k < slice_size; ++k) {
for (int j = 0; j < slice_size; ++j) {
for (int i = 0; i < slice_size; ++i) {
int c_ounter = i + j * slice_size + k * slice_size * slice_size;
o_bject[c_ounter] = 0;
}}}}

Loadprojection::~Loadprojection() {
delete[] o_bject;}

// main bulk phantom structure
void Loadprojection::load_images() {

double dtime = omp_get_wtime();

vector<int> compression_params;
compression_params.push_back(CV_IMWRITE_PNG_COMPRESSION);
//compression_params.push_back(image_compression);
compression_params.push_back(0);

int t_arget = 65;
int d_elta = 5;

// number of projections should be = number of rays...
projections_number = 360;
float r_ange = 2.0f*pi;
float db = r_ange / float(projections_number);

float start_view_angle = 0;
float view_angle;
```

```
Mat c_onvolved = cv::Mat(h_detector, w_detector, CV_32FC1, Scalar::all(0));
Mat reconstructed_image = cv::Mat(slice_size, slice_size, CV_32FC1,
Scalar::all(0));
Mat reconstructed_scaled;

Mat IMG_float = cv::Mat(h_detector, w_detector, CV_32FC1, Scalar::all(0));
Mat img_float = cv::Mat(h_detector, w_detector, CV_32FC1, Scalar::all(0));
Mat IMG = cv::Mat(h_detector, w_detector, CV_16UC1, Scalar::all(0));
Mat img_normalized = cv::Mat(h_detector, w_detector, CV_16UC1, Scalar::all(0));

Mat slice_normalized=cv::Mat(slice_size, slice_size, CV_16UC1, Scalar::all(0));
Mat slice_all = cv::Mat(slice_size, slice_size, CV_32FC1, Scalar::all(0));

float *g_na;
//filter_width should be at least >=2 * w_detector;
int filter_width = 2 * w_detector;
g_na = new (std::nothrow) float[filter_width];
if (g_na == 0) { std::cout << "\n Error assigning memory \n"; }
for (int ig = 0; ig < filter_width; ig++) { g_na[ig] = 0; }
// precompute g_na filter values
int center_point = static_cast<int>(floor(filter_width / 2.0f));
int A = 1;
g_na[center_point] = 1 / float(8 * A*A);
for (int ni = 1; ni <= int(floor(filter_width / 2.0f)) + 1; ni++) {
int index_left = center_point - ni;
if ((index_left >= 0) && (ni % 2 != 0)) {
g_na[index_left] = -1 / (2 * ni*ni*pi*pi*A*A);}

int index_right = center_point + ni;
if ((index_right <= filter_width) && (ni % 2 != 0)) {
g_na[index_right] = -1 / (2 * ni*ni*pi*pi*A*A); }
}

view_angle = start_view_angle;
string extensio_n = ".png";
string a_ffix;
for (int r_otate = 0; r_otate < projections_number; r_otate++) {
Mat slice = cv::Mat(slice_size, slice_size, CV_32FC1, Scalar::all(0));
Mat v_isits = cv::Mat(slice_size, slice_size, CV_32FC1, Scalar::all(0));
float R2 = float(y_source*y_source);
float dist_to_screen = float(-y_source + y_detector);
float y_dist = float(y_detector - y_source);
float c_os = cos(view_angle);
float s_in = sin(view_angle);
float y_dist_cos = y_dist*c_os;
float y_dist_sin = y_dist*s_in;

float y_source_cos = float(y_source)*c_os + slice_size / 2.0f;
float y_source_sin = float(y_source)*s_in - slice_size / 2.0f;
float z_source_add = float(z_source)+ slice_size / 2.0f;

int rotation_angle = r_otate*int(180/pi*r_ange/projections_number);
string counte_r = SSTR(rotation_angle);
string projectio_n = "projection_";
string p_ath = "C:\\Users\\Martha\\Google Drive\\reconstruction_64_bit\\
projections\\old_projections\\air_chamber_900_200_512\\";
//string p_ath = "C:\\Users\\Martha\\Google Drive\\reconstruction_64_bit\\
projections\\old_projections\\no_r2_200_200_1024\\";
//string p_ath = "C:\\Users\\Martha\\Google Drive\\reconstruction_64_bit\\
projections\\old_projections\\no_r2_200_200_mono\\";
//string p_ath = "C:\\Users\\Martha\\Google Drive\\reconstruction_64_bit\\
projections\\old_projections\\without_r2\\";
//string p_ath = "C:\\Users\\Martha\\Google Drive\\
reconstruction_64_bit\\projections\\old_projections\\no_r2_200_200_1024\\";

string filenam_e = p_ath + projectio_n + counte_r + extensio_n;
cout << filenam_e << "\n";
// LOAD image

IMG = imread(filenam_e, CV_LOAD_IMAGE_UNCHANGED);
//IMG = abs(IMG - 65535);
//Mat IMG = cv::Mat(h_detector, w_detector, CV_16UC1, Scalar::all(65535));

if (!IMG.data) // Check for invalid input
{
cout << "Could not open or find the image" << std::endl;
}

IMG.convertTo(IMG_float, CV_32FC1);

// This scaling is good for taking logarithm of data
IMG_float = IMG_float * 35.0f;
//IMG_float = 0;
IMG_float = IMG_float + 2e6f;


// convolve a single line of projection data with g_na
// Q(n*a)=R(n*a)*gna(n*a)
// f*g[n]=sum_m { f[m].g[n-m] }
// g[n-m]=gna(n*a) is symmetric. which makes it easier
```

```
float Rna_prime, Dna, g_na_data;

for (int ck = 0; ck < h_detector; ck++) {
float dist_to_screen = float(-y_source + y_detector);
float z_dist = float(ck) - float(h_detector) / 2.0f;
for (int ci = 0; ci < w_detector; ci++) {
float x_dist = float(ci) - float(w_detector) / 2.0f;
// !!! NEED TO CHECK !!!
Dna = dist_to_screen / pow(dist_to_screen*dist_to_screen + x_dist*x_dist +
z_dist*z_dist, 0.5f);
//incorrect Dna = -y_source / pow(y_source*y_source + x_dist*x_dist +
z_dist*z_dist, 0.5f);
//Dna = 1.0f;
img_float.at<float>(h_detector - 1 - ck, w_detector - 1 - ci) =
abs(log(IMG_float.at<float>(h_detector - 1 - ck, w_detector - 1 - ci)))*Dna;}


for (int ci = 0; ci < w_detector; ci++) {
float Qna = 0;

for (int cj = 0; cj < w_detector; cj++) {
Rna_prime = 0;
if (cj < w_detector) {
Rna_prime = img_float.at<float>(h_detector - 1 - ck, w_detector - 1 - cj);}

g_na_data = g_na[center_point + ci - cj];
Qna = Qna + Rna_prime*g_na_data;}

c_onvolved.at<float>(h_detector - 1 - ck, w_detector - 1 - ci) = Qna;
}}

float t_before = (float(-y_source) - float(slice_size) / pow(2.0f,0.5)) / (float(-
y_source) + float(y_detector));
float t_after = (float(-y_source) + float(slice_size) / pow(2.0f, 0.5)) / (float(-
y_source) + float(y_detector));

// Explicitly disable dynamic teams
omp_set_dynamic(0);
// Get the number of processors in this system
int c_ores = omp_get_num_procs();
// Now set the number of threads
omp_set_num_threads(c_ores);

#pragma omp parallel for
for (int ck = 0; ck < h_detector; ck++) {
float z_dist = float(ck) - float(h_detector) / 2.0f;
```

```
for (int ci = 0; ci < w_detector; ci++) {

float x_dist = float(ci) - float(w_detector) / 2.0f;

float x_dist_sin = x_dist*s_in;
float x_dist_cos = x_dist*c_os;
float xcos_ysin = x_dist_cos - y_dist_sin;
float xsin_ycos = x_dist_sin + y_dist_cos;

float dt = pow((y_dist*y_dist + x_dist*x_dist + z_dist*z_dist), -0.5f);
int Xt, Yt, Zt;

for (float t = t_before; t <= t_after; t += dt) {
// we have to move the origin to the center of the phantom by adding w_idth/2 etc.
// need two lines below for right solution
// fancy way to speed up calculations by static_cast<int>
Xt = static_cast<int>(xcos_ysin*t - y_source_sin);

Yt = static_cast<int>(xsin_ycos*t + y_source_cos);
Zt = static_cast<int>(z_dist*t + z_source_add);

if ((Zt < slice_size-1) && (Xt < slice_size-1) && (Yt < slice_size-1) &&
(Zt > 0) && (Yt >0) && (Xt >0) ) {

//if ((Yt >= t_arget - d_elta) && (Yt <= t_arget + d_elta))
//if ((Zt>= t_arget - d_elta) && (Zt <= t_arget + d_elta))
if ((Xt >= t_arget - d_elta) && (Xt <= t_arget + d_elta))
{
// wrong...non uniform along the Z axis...
float half_slice = float(slice_size) / 2.0f;
//float s = pow(float((half_slice -Zt)*(half_slice - Zt)+((half_slice - Yt)*(half_slice –
Yt))),0.5);
// xt*sin or xt*cos..
float U2 = pow(float(y_source*y_source)/
(abs(float(y_source))+
(float(Xt) - half_slice)*sin(view_angle) + (float(Yt)-
half_slice)*cos(view_angle)),2.0);
U2 = 1.0f;

//a_ffix = "XZ"; slice.at<float>(Zt, Xt)
//a_ffix = "XY"; slice.at<float>(Xt, Yt)
a_ffix = "ZY"; slice.at<float>(Zt, Yt)
+= c_onvolved.at<float>(h_detector - 1 - ck, w_detector - 1 - ci)*U2;
v_isits.at<float>(Zt, Yt) += 1;
}}}}}

for (int i0 = 0; i0 < slice_size; i0++) {
for (int j0 = 0; j0 < slice_size; j0++) {
if (v_isits.at<float>(i0, j0)!=0) {
slice.at<float>(i0, j0)/= (v_isits.at<float>(i0, j0));}}}

slice_all = slice_all + slice;

for (int i0 = 0; i0 < slice_size; i0++) {
for (int j0 = 0; j0 < slice_size; j0++) {
if (i0==0){slice_all.at<float>(i0, j0) = slice_all.at<float>(1, j0);}
if (j0 ==0) { slice_all.at<float>(i0, j0) = slice_all.at<float>(i0,1);}
if (j0 == slice_size-1) { slice_all.at<float>(i0, j0) = slice_all.at<float>(i0, slice_size - 2);}
if (i0 == slice_size-1) { slice_all.at<float>(i0, j0) = slice_all.at<float>(slice_size - 2,j0); }}}
slice_all.at<float>(0, 0) = slice_all.at<float>(1, 1);

normalize(slice_all, slice_normalized, 0, 65535, CV_MINMAX);
slice_normalized.convertTo(slice_normalized, CV_16UC1);
imshow("reconstructed_image", slice_normalized);
waitKey(1);

// rotate the object to build another projection at the different view angle
view_angle += db;
}

string n_ame = "_slice_";
string pat_h = "C:\\Users\\Martha\\Desktop\\slices\\";
string c_ounter = SSTR(t_arget);
string time_stamp = SSTR(round(omp_get_wtime()));
string rotation_s = SSTR(projections_number);
string slic_e = SSTR(slice_size);
string filenam_e = pat_h + rotation_s + "_" + a_ffix +n_ame + c_ounter +"_"+slic_e + "_"+time_stamp+ extensio_n;
cv::imwrite(filenam_e, slice_normalized, compression_params);

dtime = omp_get_wtime() - dtime;
std::printf("elapsed build time in seconds = %f\n\n", dtime);

waitKey(0);
delete[]g_na;
}
```

input_file.txt to define the phantom geometry

Selecting path to the attenuation coefficients database and spectrum description text file

```
1    path.mu.data:Attenuation coefficients\\NIST compounds\\
2    path.spectrum.data:100kV.txt
```

choosing the type of computer tovography, which is the regular CT or mamography

```
3    mammography.no

4    element.id:Vacuum.0
5    element.id:Air, Dry.1
6    element.id:Polymethyl Methacrylate.2
7    element.id:Water, Liquid.3
8    element.id:Blood, Whole.4
```

CT scan geometry

```
9    source.to.phantom:200
10   source.elevation:0
11   phantom.to.detector:200
```

Detector’s geometry

```
12   detector.width:1024
13   detector.height:1024
```

Number and angular spacing of the projections

```
14   start.angle:0 // from lesser number to bigger one
15   end.angle:360
16   projections.number:361
17   image.compression:0

18   voxels.per.cm:128
```

Noise and blurring parameters

```
19   gauss.blur.kernel:11 // should be ODD!!

20   a.multiplier:0.10
21   gauss.noise.mean:0
22   gauss.noise.sigma:5
```

Phantom geometry

```
23   phantom.width:128
24   phantom.depth:128
25   phantom.height:128
26   phantom.id:3
```

```
27   save.phantom.no

28   defects.quantity:0
29   defects.id:0

30   voids.quantity:0
31   voids.id:0

Voids and defect descriptions
32   object.shape:1
33   object.x:64
34   object.y:19
35   object.z:32
36   object.width:64
37   object.depth:64
38   object.height:64
39   object.alpha:0
40   object.beta:0
41   object.gamma:45
42   object.id:1

43   object.shape:0
44   object.x:32
45   object.y:32
46   object.z:32
47   object.width:64
48   object.depth:64
49   object.height:64
50   object.alpha:0
51   object.beta:0
52   object.gamma:0
53   object.id:3

54   object.shape:1
55   object.x:42
56   object.y:42
57   object.z:42
58   object.width:45
59   object.depth:45
60   object.height:45
61   object.alpha:0
62   object.beta:0
63   object.gamma:0
64   object.id:0
```

TABLE OF CONTENT

Educational issue

Kussainov Arman Sainovitch

**SELECTED PROBLEMS
IN PARALLEL COMPUTING
AND MULTITHREADING
IN NUCLEAR PHYSICS**

Educational-methodical manual

Typesetting and
cover design *G. Kaliyeva*

Cover design used photos from sites
www.goodwp.com

IB №11576
Signed for publishing 22.12.2017. Format 60x84 $^{1}/_{16}$. Offset paper.
Digital printing. Volume 6,12 printer’s sheet. 60 copies. Order №6864.
Publishing house «Qazaq university»
Al-Farabi Kazakh National University
KazNU, 71 Al-Farabi, 050040, Almaty

Printed in the printing office of the «Kazakh University» publishing house.