Slides from FYS4411 Lectures

Morten Hjorth-Jensen & Gustav R. Jansen

[1]Department of Physics and Center of Mathematics for Applications
University of Oslo, N-0316 Oslo, Norway

Spring 2012

Topics for Week 8, February 20. - 24.

Parallelization, onebody densities and blocking

- Repetition from last week
- MPI programming and access to titan.uio.no
- Onebody densities
- Statistical analysis and blocking

Project work this week: finalize 1a and 1b. Start implementing importance sampling and exercise 1c.

Importance sampling, what we want to do

We need to replace the brute force Metropolis algorithm with a walk in coordinate space biased by the trial wave function. This approach is based on the Fokker-Planck equation and the Langevin equation for generating a trajectory in coordinate space. This is explained later.
For a diffusion process characterized by a time-dependent probability density $P(x,t)$ in one dimension the Fokker-Planck equation reads (for one particle/walker)

$$\frac{\partial P}{\partial t} = D\frac{\partial}{\partial x}\left(\frac{\partial}{\partial x} - F\right)P(x,t),$$

where F is a drift term and D is the diffusion coefficient.
The new positions in coordinate space are given as the solutions of the Langevin equation using Euler's method, namely, we go from the Langevin equation

$$\frac{\partial x(t)}{\partial t} = DF(x(t)) + \eta,$$

with η a random variable, yielding a new position

$$y = x + DF(x)\Delta t + \xi,$$

where ξ is gaussian random variable and Δt is a chosen time step.

Importance sampling, what we want to do

The process of isotropic diffusion characterized by a time-dependent probability density $P(\boldsymbol{x}, t)$ obeys (as an approximation) the so-called Fokker-Planck equation

$$\frac{\partial P}{\partial t} = \sum_i D \frac{\partial}{\partial \boldsymbol{x_i}} \left(\frac{\partial}{\partial \boldsymbol{x_i}} - \boldsymbol{F_i} \right) P(\boldsymbol{x}, t),$$

where $\boldsymbol{F_i}$ is the i^{th} component of the drift term (drift velocity) caused by an external potential, and D is the diffusion coefficient. The convergence to a stationary probability density can be obtained by setting the left hand side to zero. The resulting equation will be satisfied if and only if all the terms of the sum are equal zero,

$$\frac{\partial^2 P}{\partial \boldsymbol{x_i}^2} = P \frac{\partial}{\partial \boldsymbol{x_i}} \boldsymbol{F_i} + \boldsymbol{F_i} \frac{\partial}{\partial \boldsymbol{x_i}} P.$$

Importance sampling, what we want to do

The drift vector should be of the form $\boldsymbol{F} = g(\boldsymbol{x})\frac{\partial P}{\partial \boldsymbol{x}}$. Then,

$$\frac{\partial^2 P}{\partial \boldsymbol{x_i}^2} = P\frac{\partial g}{\partial P}\left(\frac{\partial P}{\partial \boldsymbol{x_i}}\right)^2 + Pg\frac{\partial^2 P}{\partial \boldsymbol{x_i}^2} + g\left(\frac{\partial P}{\partial \boldsymbol{x_i}}\right)^2.$$

The condition of stationary density means that the left hand side equals zero. In other words, the terms containing first and second derivatives have to cancel each other. It is possible only if $g = \frac{1}{P}$, which yields

$$\boxed{\boldsymbol{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T,} \tag{1}$$

which is known as the so-called *quantum force*. This term is responsible for pushing the walker towards regions of configuration space where the trial wave function is large, increasing the efficiency of the simulation in contrast to the Metropolis algorithm where the walker has the same probability of moving in every direction.

Importance Sampling

The Fokker-Planck equation yields a (the solution to the equation) transition probability given by the Green's function

$$G(y,x,\Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}} \exp\left(-(y-x-D\Delta t F(x))^2/4D\Delta t\right)$$

which in turn means that our brute force Metropolis algorithm

$$A(y,x) = \min(1, q(y,x))),$$

with $q(y,x) = |\Psi_T(y)|^2/|\Psi_T(x)|^2$ is now replaced by

$$q(y,x) = \frac{G(x,y,\Delta t)|\Psi_T(y)|^2}{G(y,x,\Delta t)|\Psi_T(x)|^2}$$

See program vmc_importance.cpp for example.

Importance sampling, new positions, see code vmc_importance.cpp under the programs link

```
for (variate=1; variate <= max_variations; variate
    ++){
    // initialisations of variational parameters
       and energies
    beta += 0.1;
    energy = energy2 = delta_e = 0.0;
    //  initial trial position, note calling with
       beta
    for (i = 0; i < number_particles; i++) {
      for ( j=0; j < dimension; j++) {
        r_old[i][j] = gaussian_deviate(&idum)*sqrt(
           timestep);
      }
    }
    wfold = wave_function(r_old, beta);
    quantum_force(r_old, qforce_old, beta, wfold);
```

Importance sampling, new positions in function vmc_importance.cpp

```
// loop over monte carlo cycles
for (cycles = 1; cycles <= number_cycles;
    cycles++){
  // new position
  for (i = 0; i < number_particles; i++) {
    for ( j=0; j < dimension; j++) {
      // gaussian deviate to compute new
          positions using a given timestep
      r_new[i][j] = r_old[i][j] +
          gaussian_deviate(&idum)*sqrt(timestep
          )+qforce_old[i][j]*timestep*D;
```

Importance sampling, new positions in function vmc_importance.cpp

```
//  we move only one particle at the time
        for (k = 0; k < number_particles; k++) {
          if ( k != i) {
            for ( j=0; j < dimension; j++) {
              r_new[k][j] = r_old[k][j];
            }
          }
        }
        //          wave_function_onemove(r_new,
            qforce_new, &wfnew, beta);
        wfnew = wave_function(r_new, beta);
        quantum_force(r_new, qforce_new, beta,
            wfnew);
```

Importance sampling, new positions in function vmc_importance.cpp

```
//  we compute the log of the ratio of the
    greens functions to be used in the
//  Metropolis-Hastings algorithm
greensfunction = 0.0;
for ( j=0; j < dimension; j++) {
  greensfunction += 0.5*(qforce_old[i][j]+
      qforce_new[i][j])*
    (D*timestep*0.5*(qforce_old[i][j]-
       qforce_new[i][j])-r_new[i][j]+r_old
       [i][j]);
}
greensfunction = exp(greensfunction);
```

Importance sampling, new positions in function vmc_importance.cpp

```
// The Metropolis test is performed by
    moving one particle at the time
if(ran2(&idum) <= greensfunction*wfnew*
   wfnew/wfold/wfold ) {
  for ( j=0; j < dimension; j++) {
    r_old[i][j] = r_new[i][j];
    qforce_old[i][j] = qforce_new[i][j];
  }
  wfold = wfnew;
  .....
```

Importance sampling, Quantum force in function vmc_importance.cpp

```
void  quantum_force(double **r, double **qforce,
    double beta, double wf)
{
  int i, j;
  double wfminus, wfplus;
  double **r_plus, **r_minus;

  r_plus = (double **) matrix( number_particles,
      dimension, sizeof(double));
  r_minus = (double **) matrix( number_particles,
      dimension, sizeof(double));
  for (i = 0; i < number_particles; i++) {
    for ( j=0; j < dimension; j++) {
      r_plus[i][j] = r_minus[i][j] = r[i][j];
    }
  }
...
```

Importance sampling, Quantum force in function vmc_importance.cpp, brute force derivative

```
    // compute the first derivative
    for (i = 0; i < number_particles; i++) {
      for (j = 0; j < dimension; j++) {
        r_plus[i][j] = r[i][j]+h;
        r_minus[i][j] = r[i][j]-h;
        wfminus = wave_function(r_minus, beta);
        wfplus  = wave_function(r_plus, beta);
        qforce[i][j] = (wfplus-wfminus)*2.0/wf/(2*h);
        r_plus[i][j] = r[i][j];
        r_minus[i][j] = r[i][j];
      }
    }

} // end of quantum_force function
```

Going Parallel with MPI

You will need to parallelize the codes you develop.
Task parallelism: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulation or integrations are examples of this. It is almost embarrassingly trivial to parallelize Monte Carlo codes.
MPI is a message-passing library where all the routines have corresponding C/C++-binding

```
MPI_Command_name
```

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

```
MPI_COMMAND_NAME
```

What is Message Passing Interface (MPI)? Yet another library!

MPI is a library, not a language. It specifies the names, calling sequences and results of functions or subroutines to be called from C or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C or C++ are compiled with ordinary compilers and linked with the MPI library.
MPI is a specification, not a particular implementation. MPI programs should be able to run on all possible machines and run all MPI implementetations without change.
An MPI computation is a collection of processes communicating with messages.
See chapter 4.7 of lecture notes for more details.

MPI

MPI is a library specification for the message passing interface, proposed as a standard.

- independent of hardware;
- not a language or compiler specification;
- not a specific implementation or product.

A message passing standard for portability and ease-of-use.
Designed for high performance.
Insert communication and synchronization functions where necessary.

Demands from the HPC community

In the field of scientific computing, there is an ever-lasting wish to do larger simulations using shorter computer time. Development of the capacity for single-processor computers can hardly keep up with the pace of scientific computing:

- processor speed
- memory size/speed

Solution: parallel computing!

The basic ideas of parallel computing

- Pursuit of shorter computation time and larger simulation size gives rise to parallel computing.
- Multiple processors are involved to solve a global problem.
- The essence is to divide the entire computation evenly among collaborative processors. Divide and conquer.

A rough classification of hardware models

- Conventional single-processor computers can be called SISD (single-instruction-single-data) machines.
- SIMD (single-instruction-multiple-data) machines incorporate the idea of parallel processing, which use a large number of process- ing units to execute the same instruction on different data.
- Modern parallel computers are so-called MIMD (multiple-instruction- multiple-data) machines and can execute different instruction streams in parallel on different data.

Shared memory and distributed memory

- One way of categorizing modern parallel computers is to look at the memory configuration.
- In shared memory systems the CPUs share the same address space. Any CPU can access any data in the global memory.
- In distributed memory systems each CPU has its own memory. The CPUs are connected by some network and may exchange messages.

Different parallel programming paradigms

- **Task parallelism** Â the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulation is one example. Integration is another. However this paradigm is of limited use.
- **Data parallelism** Â use of multiple threads (e.g. one thread per processor) to dissect loops over arrays etc. This paradigm requires a single memory address space. Communication and synchronization between processors are often hidden, thus easy to program. However, the user surrenders much control to a specialized compiler. Examples of data parallelism are compiler-based parallelization and OpenMP directives.

Today's situation of parallel computing

- Distributed memory is the dominant hardware configuration. There is a large diversity in these machines, from MPP (massively parallel processing) systems to clusters of off-the-shelf PCs, which are very cost-effective.
- Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware. It is primarily used for the distributed memory systems, but can also be used on shared memory systems.

In these lectures we consider only message-passing for writing parallel programs.

Overhead present in parallel computing

- **Uneven load balance**: not all the processors can perform useful work at all time.
- **Overhead of synchronization.**
- **Overhead of communication**.
- Extra computation due to parallelization.

Due to the above overhead and that certain part of a sequential algorithm cannot be parallelized we may not achieve an optimal parallelization.

Parallelizing a sequential algorithm

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This is the difficult part,
- Distribute the global work and data among P processors.

Process and processor

- We refer to process as a logical unit which executes its own code, in an MIMD style.
- The processor is a physical device on which one or several processes are executed.
- The MPI standard uses the concept process consistently throughout its documentation.

Bindings to MPI routines

MPI is a message-passing library where all the routines have corresponding C/C++-binding

MPI_Command_name

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

MPI_COMMAND_NAME

The discussion in these slides focuses on the C++ binding.

Communicator

- A group of MPI processes with a name (context).
- Any process is identified by its rank. The rank is only meaningful within a particular communicator.
- By default communicator MPI_COMM_WORLD contains all the MPI processes.
- Mechanism to identify subset of processes.
- Promotes modular design of parallel libraries.

Some of the most important MPI routines

- MPI_ Init - initiate an MPI computation
- MPI_Finalize - terminate the MPI computation and clean up
- MPI_Comm_size - how many processes participate in a given MPI communicator?
- MPI_Comm_rank - which one am I? (A number between 0 and size-1.)
- MPI_Send - send a message to a particular process within an MPI communicator
- MPI_Recv - receive a message from a particular process within an MPI communicator

The first MPI C/C++ program

Let every process write ”Hello world” on the standard output. This is program2.cpp under MPI/chapter07.

```
using namespace std;
#include <mpi.h>
#include <iostream>
int main (int nargs, char* args[])
{
int numprocs, my_rank;
//   MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
cout << "Hello world, I have  rank " << my_rank <<
   " out of "
     << numprocs << endl;
//  End MPI
MPI_Finalize ();
```

The Fortran program

```
PROGRAM hello
INCLUDE "mpif.h"
INTEGER:: size, my_rank, ierr

CALL  MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
WRITE(*,*)"Hello world, I've rank ",my_rank," out
    of ",size
CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

Note 1

The output to screen is not ordered since all processes are trying to write to screen simultaneously. It is then the operating system which opts for an ordering. If we wish to have an organized output, starting from the first process, we may rewrite our program as in the next example (program3.cpp), see again chapter 4.7 of lecture notes.

Ordered output with MPI_Barrier

```
int main (int nargs, char* args[])
{
 int numprocs, my_rank, i;
 MPI_Init (&nargs, &args);
 MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
 for (i = 0; i < numprocs; i++) {
 MPI_Barrier (MPI_COMM_WORLD);
 if (i == my_rank) {
 cout << "Hello world, I have  rank " << my_rank <<
         " out of " << numprocs << endl;}
      MPI_Finalize ();
}
```

Note 2

Here we have used the *MPI_Barrier* function to ensure that that every process has completed its set of instructions in a particular order. A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here *MPI_COMM_WORLD* have called *MPI_Barrier*. The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like *MPI_ALLREDUCE* to be discussed later, have the same property; viz. no process can exit the operation until all processes have started. However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to a have a synchronized action.

Ordered output with MPI_Recv and MPI_Send

```
.....
int numprocs, my_rank, flag;
MPI_Status status;
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
if (my_rank > 0)
MPI_Recv (&flag, 1, MPI_INT, my_rank-1, 100,
            MPI_COMM_WORLD, &status);
cout << "Hello world, I have  rank " << my_rank <<
    " out of "
<< numprocs << endl;
if (my_rank < numprocs-1)
MPI_Send (&my_rank, 1, MPI_INT, my_rank+1,
           100, MPI_COMM_WORLD);
MPI_Finalize ();
```

Note 3

The basic sending of messages is given by the function *MPI_SEND*, which in C/C++ is defined as

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)}
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable **buf** is the variable we wish to send while **count** is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

Note 4

Once you have sent a message, you must receive it on another task. The function **MPI_RECV** is similar to the send call.

```
int MPI_Recv( void *buf, int count, MPI_Datatype
    datatype,
              int source,
              int tag, MPI_Comm comm, MPI_Status *
                  status )
```

The arguments that are different from those in *MPI_SEND* are **buf** which is the name of the variable where you will be storing the received data, **source** which replaces the destination in the send command. This is the return ID of the sender.
Finally, we have used **MPI_Status status;** where one can check if the receive was completed.
The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.
Armed with this wisdom, performed all hello world greetings, we are now ready for serious work.

Strategies

- Develop codes locally, run with some few processes and test your codes. Do benchmarking, timing and so forth on local nodes, for example your laptop. You can install MPICH2 on your laptop (most new laptos come with dual cores). You can test with one node at the lab.
- When you are convinced that your codes run correctly, you start your production runs on available supercomputers, in our case titan.uio.no.

How do I run MPI on the machines at the lab (MPICH2)

The machines at the lab are all quad-cores

- Compile with mpicxx or mpic++
- Set up collaboration between processes and run

```
mpd --ncpus=4 &
#  run code with
mpiexec -n 4 ./nameofprog
```

Here we declare that we will use 4 processes via the $-ncpus$ option and via $-n4$ when running.

- End with

```
mpdallexit
```

Can I do it on my own PC/laptop?

Of course:

- go to `http://www.mcs.anl.gov/research/projects/mpich2/`
- or go to `http://www.open-mpi.org/`
- follow the instructions and install it on your own PC/laptop

I don't have windows as operating system and need dearly your feedback.

Integration algos

The trapezoidal rule (example6.cpp)

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right).$$

Another very simple approach is the so-called midpoint or rectangle method. In this case the integration area is split in a given number of rectangles with length h and heigth given by the mid-point value of the function. This gives the following simple rule for approximating an integral

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given rectangle. This is used in program5.cpp.

Dissection of example program5.cpp

```
1   //     Reactangle rule and numerical integration
2   using namespace std;
3   #include <mpi.h>
4   #include <iostream>

5   int main (int nargs, char* args[])
6   {
7       int numprocs, my_rank, i, n = 1000;
8       double local_sum, rectangle_sum, x, h;
9       //   MPI initializations
10      MPI_Init (&nargs, &args);
11      MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
12      MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
```

Dissection of example program5.cpp

```
13      //   Read from screen a possible new vaue of
    n
14      if (my_rank == 0 && nargs > 1) {
15         n = atoi(args[1]);
16      }
17      h = 1.0/n;
18      //  Broadcast n and h to all processes
19      MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD
    );
20      MPI_Bcast (&h, 1, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
21      //  Every process sets up its contribution
    to the integral
22      local_sum = 0.;
```

Dissection of example program5.cpp

After the standard initializations with MPI such as MPI_Init, MPI_Comm_size and MPI_Comm_rank, MPI_COMM_WORLD contains now the number of processes defined by using for example

```
mpiexec -np 10 ./prog.x
```

In line 4 we check if we have read in from screen the number of mesh points n. Note that in line 7 we fix $n = 1000$, however we have the possibility to run the code with a different number of mesh points as well. If my_rank equals zero, which correponds to the master node, then we read a new value of n if the number of arguments is larger than two. This can be done as follows when we run the code

```
mpiexec -np 10 ./prog.x 10000
```

Dissection of example program5.cpp

The MPI routine MPI_Bcast transfers data from one task to a group of others. The format for the call is in C++ given by the parameters of

```
MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);.
MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

in a case of a double. The general structure of this function is

```
MPI_Bcast( void *buf, int count, MPI_Datatype
    datatype, int root, MPI_Comm comm).
```

All processes call this function, both the process sending the data (with rank zero) and all the other processes in MPI_COMM_WORLD. Every process has now copies of n and h, the number of mesh points and the step length, respectively.

We transfer the addresses of n and h. The second argument represents the number of data sent. In case of a one-dimensional array, one needs to transfer the number of array elements. If you have an $n \times m$ matrix, you must transfer $n \times m$. We need also to specify whether the variable type we transfer is a non-numerical such as a logical or character variable or numerical of the integer, real or complex type.

Dissection of example program5.cpp

```
23      for (i = my_rank; i < n; i += numprocs) {
24        x = (i+0.5)*h;
25        local_sum += 4.0/(1.0+x*x);
26      }
27      local_sum *= h;
```

In line 17 we define also the step length *h*. In lines 19 and 20 we use the broadcast function MPI_Bcast. We use this particular function because we want data on one processor (our master node) to be shared with all other processors. The broadcast function sends data to a group of processes.

Dissection of example program5.cpp

```
28      if (my_rank == 0) {
29        MPI_Status status;
30        rectangle_sum = local_sum;
31        for (i=1; i < numprocs; i++) {
32          MPI_Recv(&local_sum,1,MPI_DOUBLE,
    MPI_ANY_SOURCE,500,
                          MPI_COMM_WORLD,&status);
33          rectangle_sum += local_sum;
34        }
35        cout << "Result: " << rectangle_sum  <<
    endl;
36      }  else
37        MPI_Send(&local_sum,1,MPI_DOUBLE,0,500,
    MPI_COMM_WORLD);
38      // End MPI
39      MPI_Finalize ();
40      return 0;
41    }
```

Dissection of example program5.cpp

In lines 23-27, every process sums its own part of the final sum used by the rectangle rule. The receive statement collects the sums from all other processes in case my_rank == 0, else an MPI send is performed. If we are not the master node, we send the results, else they are received and the local results are added to final sum. The above can be rewritten using the MPI_allreduce, as discussed in the next example.

The above function is not very elegant. Furthermore, the MPI instructions can be simplified by using the functions MPI_Reduce or MPI_Allreduce. The first function takes information from all processes and sends the result of the MPI operation to one process only, typically the master node. If we use MPI_Allreduce, the result is sent back to all processes, a feature which is useful when all nodes need the value of a joint operation. We limit ourselves to MPI_Reduce since it is only one process which will print out the final number of our calculation, The arguments to MPI_Allreduce are the same.

MPI_reduce

Call as

```
MPI_reduce( void *senddata, void* resultdata, int
    count,
      MPI_Datatype datatype, MPI_Op, int root,
          MPI_Comm comm)
```

The two variables *senddata* and *resultdata* are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable *count* represents the total dimensionality, 1 in case of just one variable, while MPI_Datatype defines the type of variable which is sent and received.
The new feature is MPI_Op. It defines the type of operation we want to do. In our case, since we are summing the rectangle contributions from every process we define MPI_Op = MPI_SUM. If we have an array or matrix we can search for the largest og smallest element by sending either MPI_MAX or MPI_MIN. If we want the location as well (which array element) we simply transfer MPI_MAXLOC or MPI_MINOC. If we want the product we write MPI_PROD.
MPI_Allreduce is defined as

```
MPI_Alreduce( void *senddata, void* resultdata, int
     count,
          MPI_Datatype datatype, MPI_Op, MPI_Comm
              comm) }.
```

Dissection of example program6.cpp

```
//     Trapezoidal rule and numerical integration
    usign MPI, example program6.cpp
using namespace std;
#include <mpi.h>
#include <iostream>

//      Here we define various functions called by
    the main program

double int_function(double );
double trapezoidal_rule(double , double , int ,
    double (*)(double));

//    Main function begins here
int main (int nargs, char* args[])
{
  int n, local_n, numprocs, my_rank;
  double a, b, h, local_a, local_b, total_sum,
      local_sum;
  double  time_start, time_end, total_time;
```

Dissection of example program6.cpp

```
//  MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
time_start = MPI_Wtime();
//  Fixed values for a, b and n
a = 0.0 ; b = 1.0;  n = 1000;
h = (b-a)/n;     // h is the same for all
    processes
local_n = n/numprocs;
// make sure n > numprocs, else integer division
    gives zero
// Length of each process' interval of
// integration = local_n*h.
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
```

Dissection of example program6.cpp

```
  total_sum = 0.0;
  local_sum = trapezoidal_rule(local_a, local_b,
      local_n,
                                  &int_function);
  MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE,
                MPI_SUM, 0, MPI_COMM_WORLD);
  time_end = MPI_Wtime();
  total_time = time_end-time_start;
  if ( my_rank == 0) {
    cout << "Trapezoidal rule = " <<  total_sum <<
       endl;
    cout << "Time = " <<  total_time
         << " on number of processors: "  <<
            numprocs  << endl;
  }
  // End MPI
  MPI_Finalize ();
  return 0;
}  // end of main program
```

Dissection of example program6.cpp

We use MPI_reduce to collect data from each process. Note also the use of the function MPI_Wtime. The final functions are

```
//  this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate
```

Dissection of example program6.cpp

Implementation of the trapezoidal rule.

```
//  this function defines the trapezoidal rule
double trapezoidal_rule(double a, double b, int n,
                        double (*func)(double))
{
  double trapez_sum;
  double fa, fb, x, step;
  int    j;
  step=(b-a)/((double) n);
  fa=(*func)(a)/2. ;
  fb=(*func)(b)/2. ;
  trapez_sum=0.;
  for (j=1; j <= n-1; j++){
    x=j*step+a;
    trapez_sum+=(*func)(x);
  }
  trapez_sum=(trapez_sum+fb+fa)*step;
  return trapez_sum;
}  // end trapezoidal_rule
```

How do I use the titan.uio.no cluster?

hpc@usit.uio.no

- Computational Physics requires High Performance Computing (HPC) resources
- USIT and the Research Computing Services (RCS) provides HPC resources and HPC support
- Resources: `titan.uio.no`
- Support: 14 people
- Contact: `hpc@usit.uio.no`

Titan

https://wiki.uio.no/usit/suf/vd/hpc/index.php/TITAN

Getting started

Batch systems

- A batch system controls the use of the cluster resources
- Submits the job to the right resource
- Monitors the job while executing
- Restarts the job in case of failure
- Takes care of priorities and queues to control execution order of unrelated jobs

Sun Grid Engine

- SGE is the batch system used on Titan
- Jobs are executed either interactively or through job scripts
- Useful commands: `showq`, `qlogin`, `sbatch`
- `http://hpc.uio.no/index.php/Titan_User_Guide`

Getting started

Modules

- Different compilers, MPI-versions and applications need different sets of user environment variables
- The `modules` package lets you load and remove the different variable sets
- Useful commands:
  - List available modules: `module avail`
  - Load module: `module load <environment>`
  - Unload module: `module unload <environment>`
  - Currently loaded: `module list`
- `http://hpc.uio.no/index.php/Titan_User_Guide`

Example

Interactively

```
# login to titan
$ ssh titan.uio.no
# ask for 4 cpus
$ qlogin --account=fys3150 --ntasks=4
# start a job setup, note the punktum!
$ source  /site/bin/jobsetup
# we want to use the intel module
$ module load intel
$ module load openmpi/1.2.8.intel
$ mkdir -p fys3150/mpiexample/
$ cd fys3150/mpiexample/
# Use program6.cpp from the course pages, see chapter 4
# compile the program
$ mpic++ -O3 -o program6.x program6.cpp
# and execute it
$ mpirun ./program6.x
$ Trapezoidal rule = 3.14159
$ Time = 0.000378132 on number of processors: 4
```

The job script

job.sge

```
#!/bin/sh
# Call this file job.slurm
# 4 cpus with mpi (or other communication)
#SBATCH --ntasks=4
# 10 mins of walltime
#SBATCH --time=0:10:00
# project fys3150
#SBATCH --account=fys3150
# we need 2000 MB of memory per process
#SBATCH --mem-per-cpu=2000M
# name of job
#SBATCH --job-name=program5

source /site/bin/jobsetup

# load the module used when we compiled the program
module load scampi

# start program
mpirun ./program5.x

#END OF SCRIPT
```

Example

Submitting

```
# login to titan
$ ssh titan.uio.no
# we want to use the module scampi
$ module load scampi
$ cd fys3150/mpiexample/
# compile the program
$ mpic++ -O3 -o program5.x program5.cpp
# and submit it
$ sbatch job.slurm
$ exit
```

Example

Checking execution

```
# check if job is running:
$ showq –u mhjensen
ACTIVE JOBS--------------------
JOBNAME            USERNAME      STATE   PROC    REMAINING            STARTTIME

883129             mhjensen    Running      4    10:31:17   Fri Oct  2 13:59:25

     1 Active Job      2692 of 4252 Processors Active (63.31%)
                        482 of  602 Nodes Active      (80.07%)

IDLE JOBS----------------------
JOBNAME            USERNAME      STATE   PROC      WCLIMIT            QUEUETIME


0 Idle Jobs

BLOCKED JOBS----------------
JOBNAME            USERNAME      STATE   PROC      WCLIMIT            QUEUETIME


Total Jobs: 1   Active Jobs: 1   Idle Jobs: 0   Blocked Jobs: 0
```

Tips and admonitions

Tips

- Titan FAQ: `http://www.hpc.uio.no/index.php/FAQ`
- man-pages, e.g. `man sbatch`
- Ask us

Admonitions

- Remember to exit from qlogin-sessions; the resource is reserved for you untill you exit
- Don’t run jobs on login-nodes; these are only for compiling and editing files

Definition of onebody density, needed in 1d

The harmonic oscillator-like functions for so-called $n_x = n_y = 0$ waves are rather simple.
This means that if we use just the harmonic oscillator-like wave functions, our ground state for the two electron dot is

$$\Phi(\mathbf{r_1}, \mathbf{r_2}) = C \exp\left(-\omega(r_1^2 + r_2^2)/2\right).$$

and the onebody density is defined as

$$\rho(\mathbf{r}_1) = \int d\mathbf{r}_2 \left| C \exp\left(-\omega(r_1^2 + r_2^2)/2\right)\right|^2,$$

if we use just the Harmonic oscillator wave functions. Remember that these are eigenfunctions of the unperturbed problem.

Definition of onebody density, needed in 1d

With the onebody density defined as

$$\rho(\mathbf{r}_1) = \int d\mathbf{r}_2 \left| C \exp\left(-\omega(r_1^2 + r_2^2)/2\right)\right|^2,$$

your tasks are to find the constant C and then calculate the density for only a harmonic oscillator state. Plot it as a function of x and y for the ground state.

Definition of onebody density, needed in 1d

In the next step the pure harmonic oscillator wave function is replaced by the optimal trial wave function from our Monte Carlo calculations, namely Ψ_T. This gives a new density given by

$$\rho(\mathbf{r}_1) = \int d\mathbf{r}_2 \left|\Psi_T\left(\mathbf{r}_1, \mathbf{r}_2\right)\right|^2 .$$

Your task then is to compute the density for the ground state with the correlations baked in and compare the result with the one obtained with the pure harmonic oscillator. You have to compare this for different values of ω in order to study the role of correlations.

Why blocking?

Statistical analysis, see chapter 11.2 of lecture notes

- Monte Carlo simulations have to be treated as *computer experiments*
- The results can be analysed with the same statistical tools as we would use when analyzing experimental data.
- As in all experiments, we are looking for expectation values and an estimate of how accurate they are, i.e., possible sources for errors.

Why blocking?

Statistical analysis

- As in other experiments, Monte Carlo experiments have two classes of errors:
  - Statistical errors
  - Systematical errors
- Statistical errors can be estimated using standard tools from statistics
- Systematical errors are method specific and must be treated differently from case to case. (In VMC a common source is the step length or time step in importance sampling)

What is blocking?

Blocking

- Blocking is a cheap (in terms of CPU expenditure) way of estimating statistical errors
- Say that we have a set of samples from a Monte Carlo experiment
- Assuming (wrongly) that our samples are uncorrelated our best estimate of the standard deviation of the mean $\langle \mathcal{M} \rangle$ is given by

$$\sigma = \sqrt{\frac{1}{n} \left(\langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 \right)}$$

- If the samples are correlated we can rewrite our results to show that

$$\sigma = \sqrt{\frac{1 + 2\tau/\Delta t}{n} \left(\langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 \right)}$$

where τ is the correlation time (the time between a sample and the next uncorrelated sample) and Δt is time between each sample

What is blocking?

Blocking

- If $\Delta t \gg \tau$ our first estimate of σ still holds
- Much more common that $\Delta t < \tau$
- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathcal{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking

- If $\Delta t \gg \tau$ our first estimate of σ still holds
- Much more common that $\Delta t < \tau$
- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathcal{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking

- If $\Delta t \gg \tau$ our first estimate of σ still holds
- Much more common that $\Delta t < \tau$
- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathcal{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking

- If $\Delta t \gg \tau$ our first estimate of σ still holds
- Much more common that $\Delta t < \tau$
- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathcal{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking

- If $\Delta t \gg \tau$ our first estimate of σ still holds
- Much more common that $\Delta t < \tau$
- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathcal{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking

- If $\Delta t \gg \tau$ our first estimate of σ still holds
- Much more common that $\Delta t < \tau$
- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathcal{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking

- Problem: We don't know τ or it is too expensive to compute
- Solution: Make a plot of std. dev. as a function of block size
- The estimate of std. dev. of correlated data is too low $\rightarrow$ the error will increase with increasing block size until the blocks are uncorrelated, where we reach a plateau
- When the std. dev. stops increasing the blocks are uncorrelated

What is blocking?

Blocking

- Problem: We don't know τ or it is too expensive to compute
- Solution: Make a plot of std. dev. as a function of block size
- The estimate of std. dev. of correlated data is too low $\rightarrow$ the error will increase with increasing block size until the blocks are uncorrelated, where we reach a plateau
- When the std. dev. stops increasing the blocks are uncorrelated

What is blocking?

Blocking

- Problem: We don't know τ or it is too expensive to compute
- Solution: Make a plot of std. dev. as a function of block size
- The estimate of std. dev. of correlated data is too low $\rightarrow$ the error will increase with increasing block size until the blocks are uncorrelated, where we reach a plateau
- When the std. dev. stops increasing the blocks are uncorrelated

What is blocking?

Blocking

- Problem: We don't know τ or it is too expensive to compute
- Solution: Make a plot of std. dev. as a function of block size
- The estimate of std. dev. of correlated data is too low $\rightarrow$ the error will increase with increasing block size until the blocks are uncorrelated, where we reach a plateau
- When the std. dev. stops increasing the blocks are uncorrelated

Implementation

Main ideas

- Do a parallel Monte Carlo simulation, storing all samples to files (one per process)
- Do the statistical analysis on these files, independently of your Monte Carlo program
- Read the files into an array
- Loop over various block sizes
- For each block size n_b, loop over the array in steps of n_b taking the mean of elements $in_b, \ldots, (i+1)n_b$
- Take the mean and variance of the resulting array
- Write the results for each block size to file for later analysis

Implementation

Main ideas

- Do a parallel Monte Carlo simulation, storing all samples to files (one per process)
- Do the statistical analysis on these files, independently of your Monte Carlo program
- Read the files into an array
- Loop over various block sizes
- For each block size n_b, loop over the array in steps of n_b taking the mean of elements $in_b, \ldots, (i+1)n_b$
- Take the mean and variance of the resulting array
- Write the results for each block size to file for later analysis

Implementation

Main ideas

- Do a parallel Monte Carlo simulation, storing all samples to files (one per process)
- Do the statistical analysis on these files, independently of your Monte Carlo program
- Read the files into an array
- Loop over various block sizes
- For each block size n_b, loop over the array in steps of n_b taking the mean of elements $in_b, \ldots, (i+1)n_b$
- Take the mean and variance of the resulting array
- Write the results for each block size to file for later analysis

Implementation

Main ideas

- Do a parallel Monte Carlo simulation, storing all samples to files (one per process)
- Do the statistical analysis on these files, independently of your Monte Carlo program
- Read the files into an array
- Loop over various block sizes
- For each block size n_b, loop over the array in steps of n_b taking the mean of elements $in_b, \ldots, (i+1)n_b$
- Take the mean and variance of the resulting array
- Write the results for each block size to file for later analysis

Implementation

Parallel file output

- The total number of samples from all processes may get very large
- Hence, storing all samples on the master node is not a scalable solution
- Instead we store the samples from each process in separate files
- Must make sure these files have different names

String handling

```
ostringstream ost;
ost << "blocks_rank" << my_rank << ".dat";
blockofile.open(ost.str().c_str(), ios::out | ios::
    binary);
```

Implementation

Parallel file output

- Having separated the filenames it's just a matter of taking the samples and store them to file
- Note that there is no need for communication between the processes in this procedure

File dumping

```
all_energies = new double[number_cycles+1];
mc_sampling(max_variations, number_cycles,
    cumulative_e, cumulative_e2,
            all_energies);

blockofile.write((char*)(all_energies+1),
                 number_cycles*sizeof(double));
blockofile.close();
```

Implementation

Reading the files

- Reading the files is only about mirroring the output
- To make life easier for ourselves we find the filesize, and hence the number of samples by using the C function `stat`

File loading

```
struct stat result;
if(stat("blocks_rank0.dat", &result) == 0){
  local_n = result.st_size/sizeof(double);
  n = local_n*n_procs;
}

double* mc_results = new double[n];
for(int i=0; i<n_procs; i++){
  ostringstream ost;
  ost << "blocks_rank" << i << ".dat";
  ifstream infile;
  infile.open(ost.str().c_str(), ios::in | ios::binary);
  infile.read((char*)&(mc_results[i*local_n]),result.st_size);
  infile.close();
}
```

Implementation

Blocking

- Loop over block sizes $in_b, \ldots, (i+1)n_b$

Loop over block sizes

```
for(int i=0; i<n_block_samples; i++){
   block_size = min_block_size+i*block_step_length;
   blocking(mc_results, n, block_size, res);
   mean  = res[0];
   sigma = res[1];
   outfile << block_size << "\t" << mean << "\t"
           << sqrt(sigma/((n/block_size)-1.0))
           << endl;
}
```

Implementation

Blocking

- The blocking itself is now just a matter of finding the number of blocks (note the integer division) and taking the mean of each block
- Note the pointer aritmetic: Adding a number *i* to an array pointer moves the pointer to element *i* in the array

Blocking function

```
void blocking(double *vals, int n_vals, int
    block_size, double *res){
  int n_blocks = n_vals/block_size;
  double* block_vals = new double[n_blocks];
  for(int i=0; i<n_blocks; i++)
    block_vals[i] = mean(vals+i*block_size,
        block_size);
  meanvar(block_vals, n_blocks, res);
}
```