1. PERFORMANCE MEASURES

1.1 Floating Point Operations [Ex1]

Peak floating point performance (PP):

$$PP\,[\text{FLOP/s}] = f\,[\text{Hz} = \text{cycle/s}] \cdot c\,[\text{FLOP/cycle}] \cdot v\,[\text{-}] \cdot n\,[\text{-}]$$

f	core frequency in CPU cycles per second (given in Hz)
c	number of FLOP executed in each cycle
v	SIMD width (in number of floats)
n	number of cores (#cores/die * #dies/node)

1.2 System Memory Bandwidth [Ex1]

Theoretical memory bandwidth (PB) for DRAM:

$$PB\,[\text{B/s}] = f\,[\text{Hz} = \text{cycle/s}] \cdot c\,[\text{channel}] \cdot l\,[\text{lane/channel}] \cdot b\,[\text{bit/lane/cycle}] \cdot 0.125\,[\text{B/bit}]$$

f	DRAM frequency
c	number of memory channels
l	width of the memory channel (typically 64 lanes per channel)
b	number of bits moved through a lane in each cycle

If there is more than one die in the node, *PB* has to be multiplied with the number of dies in the node.

2. BRUTUS

Login:

```
> ssh username@brutus.ethz.ch
```

Load/Unload a module:

```
> module load <modulename>
> module unload <modulename>
```

List all currently loaded modules:

```
> module list
```

List all available modules:

```
> module avail
```

Submit a job:

```
> bsub -n <#cores> -W <wall_clock_time> -o <output_file>
       <program_name> <program_args>
```

For example (2 hours wall clock time):

```
> bsub -n 48 -W 02:00 -o output.log ./myprogram 0 1
```

Get status of jobs:

```
> bjobs
```

3. MONTE CARLO ESTIMATOR

3.1 Formula [Ex2]

In Monte Carlo simulations we generate a set of random variables $X_i = f(x_i)$. From this sample we want to extract good estimates of the properties of the function *f*. In the lecture it was shown that the sample mean $\bar{X}$ is a true (i.e. unbiased) estimator for the expectation value E[*X*] and mentioned that the variance can be estimated from the sample mean of squares.

Show that the formula given in the lecture is an unbiased estimator, i.e. prove that

$$\mathrm{E}\left[\frac{N}{N-1}\left(\overline{X^2} - \bar{X}^2\right)\right] = \mathrm{Var}\,X \ .$$

Proof: Let $i = \{1, \ldots, N\}$. Insert the definition of the mean:

$$\mathrm{E}\left[\frac{N}{N-1}\left(\overline{X^2} - \bar{X}^2\right)\right] = \frac{1}{N-1}\mathrm{E}\left[\sum_i X_i^2 - \frac{1}{N}\left(\sum_i X_i\right)\left(\sum_j X_j\right)\right]$$

Split the double sum:

$$= \frac{1}{N-1}\mathrm{E}\left[\sum_i X_i^2 - \frac{1}{N}\sum_i X_i^2 - \frac{1}{N}\sum_i \sum_{j \neq i} X_i X_j\right]$$

X_i and X_j are uncorrelated for i ≠ j. Hence,

$$= \frac{1}{N}\sum_i \mathrm{E}\left[X_i^2\right] - \frac{1}{N(N-1)}\sum_i \sum_{j \neq i} \mathrm{E}\left[X_i\right]\mathrm{E}\left[X_j\right]$$

$$\mathrm{E}\left[X^2\right] - \mathrm{E}\left[X\right]^2 = \mathrm{Var}\,X$$

3.2 Example: Monte Carlo Estimation of π [Ex2]

In this specific case every sample X_i is either 0 or 1. Therefore, $\sum_i X_i^2 = \sum_i X_i$.

The error of the mean is estimated as

$$\Delta = \sqrt{\frac{\mathrm{Var}\,X}{N}} = \sqrt{\frac{1}{N-1}\left(\overline{X^2} - \bar{X}^2\right)} = \sqrt{\frac{1}{N-1}\left(\bar{X} - \bar{X}^2\right)} \ .$$

Any stochastic error decays with $\frac{1}{\sqrt{N}}$.

Siehe auch ML Ex2, Q3, Seite 5 ff.

Dort wird auch das Verhalten bei niedrigen Temperaturen beschrieben.

Example Code

```
// Exercise codes for HPC course
// (c) 2012 Jan Gukelberger, ETH Zurich
#include <random>
#include <iostream>
#include <iomanip>
#include <stdexcept>

typedef double value_type;
typedef std::size_t size_type;
typedef std::mt19937 rng_type;


value_type calcpi4(rng_type& rng, size_type nsamples)
{
   std::uniform_real_distribution<value_type> dist(0,1);
   size_type hits = 0;
   for( size_type i = 0; i < nsamples; ++i )
   {
      value_type x = dist(rng);
      value_type y = dist(rng);
      hits += ( (x*x + y*y) < 1. );
   }

   return hits / value_type(nsamples);
}

int main(int argc, const char** argv)
{
   if( argc != 3 )
      throw std::runtime_error(std::string("usage: ")
                               +argv[0]+" NUM_SAMPLES SEED");
   const size_type nsamples = std::stoul(argv[1]);
   const size_type seed     = std::stoul(argv[2]);
   std::cout << "#threads=1, #samples=" << nsamples
             << ", seed=" << seed << ": " << std::flush;

   rng_type rng(seed);
   value_type mean = calcpi4(rng, nsamples);

   // evaluate results:
   // As X_i \elem {0,1}, we don't need to accumulate the sum
   // of squares explic. but have \overline{X^2}=\overline{X}
   value_type error = std::sqrt(1./(nsamples-1.) *
                                (mean - mean*mean));
   std::cout << std::setprecision(10) << "pi = " << 4*mean
             << " +/- " << 4*error << std::endl;
}
```

4. Error Estimation Code

```
#include <cmath>

class accumulator {
public:
    typedef double value_type;
    typedef std::size_t size_type;

    accumulator() : count_(0), sum_(0.), sum2_(0.) {}

    void operator<< (value_type x)
    {
        sum_  += x;
        sum2_ += x*x;
        ++count_;
    }

    void merge(accumulator const& acc)
    {
        sum_   += acc.sum_;
        sum2_  += acc.sum2_;
        count_ += acc.count_;
    }

    value_type mean() const
    {
        return sum_ / count_;
    }

    value_type error() const
    {
        return std::sqrt( (sum2_ - sum_*sum_/count_)
                          / (count_*(count_-1)) );
    }
private:
    size_type count_;
    value_type sum_, sum2_;
};
```

5. TIME AND MISCELLANEOUS STUFF

Method using C++11 chrono:

```
#include <chrono>
std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
start = std::chrono::high_resolution_clock::now();
end = std::chrono::high_resolution_clock::now();
int elapsed_time =
std::chrono::duration_cast<std::chrono::microseconds>(end-
                                                      start).count();
```

Method using lrt:

```
#include <time.h>
timespec timeStart, timeFinish;
clock_gettime(CLOCK_MONOTONIC, &timeStart);
clock_gettime(CLOCK_MONOTONIC, &timeFinish);
double duration = timeFinish.tv_sec +
                  (double)timeFinish.tv_nsec/1e9 -
                  timeStart.tv_sec -
                  (double)timeStart.tv_nsec/1e9;
```

Get UNIX time:

```
#include <time.h>
time_t seconds;
seconds = time(NULL);
```

Linux: Dateien durchsuchen

```
find . -type f -name *.html -exec grep -l -i "suchstr" '{}' \;
```

6. ATOMIC BARRIER

```
#ifndef HPC12_ATOMICBARRIER_HPP
#define HPC12_ATOMICBARRIER_HPP

#include <atomic>
#include <cassert>
#include <limits>

class atomicbarrier
{
public:
  atomicbarrier(unsigned int count)
  : m_total(count)
  , m_count(count)
  , m_generation(0)
  {
    assert(count != 0);
  }

  void wait()
  {
    // needs to be sequentially consistent
    unsigned int gen = m_generation.load();

    // decrease the count
    // fetch_sub returns the value *before* the operation
    if (m_count.fetch_sub(1)==1) { // needs to be sequentially
                                   // consistent
      // if done reset to new generation of wait
      m_count.store(m_total,std::memory_order_release);
      m_generation.fetch_add(1,std::memory_order_release);
    }
    else
      while (gen ==
             m_generation.load(std::memory_order_relaxed))
            /* run in cricles, scream and shout */ ;
  }

  unsigned int num_waiting() const
  {
    return m_count;
  }

private:
  unsigned int const m_total;
  std::atomic<unsigned int> m_count;
  std::atomic<unsigned int> m_generation;
};

#endif //HPC12_ATOMICBARRIER_HPP
```

7. MY ATOMIC BARRIER

```
#ifndef HPC12_ATOMICBARRIER_HPP
#define HPC12_ATOMICBARRIER_HPP

#include <atomic>
#include <cassert>

class atomicbarrier
{
public:
    atomicbarrier(unsigned int count)
    : a_total(count)
    {
      assert(count != 0);
      a_count = ATOMIC_VAR_INIT(a_total);
      a_generation = ATOMIC_VAR_INIT(0);
    }

    void wait()
    {
      unsigned int gen = a_generation;

      // decrease the count
      if (--a_count==0) {
        // if done reset to new generation of wait and wake up
        // all threads
        a_count = ATOMIC_VAR_INIT(a_total);
        a_generation++;
      }
      else
        while (gen == a_generation) {}
    }

private:
    std::atomic<unsigned int> a_count;
    unsigned int const a_total;
    std::atomic<unsigned int> a_generation;
};

#endif //HPC12_ATOMICBARRIER_HPP
```

Barrier timing

```
#include "atomicbarrier.hpp"
#include <iostream>
#include <thread>
#include <cstdlib>
#include <vector>

int main(int argc, char** argv)
{
  // decide how many threads to use
    std::size_t nthreads =
         std::max(1u, std::thread::hardware_concurrency());
    if (argc==2)
      nthreads = std::atoi(argv[1]);
  std::cout << nthreads << " threads in total.\n";
  int repetitions=100000;
  atomicbarrier b(nthreads);

  std::vector<std::thread> threads(nthreads);

  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] = std::thread( [&b,repetitions] () {
      for (int i=0;i<repetitions;++i)
        b.wait();
    });

  for (std::thread& t : threads)
    t.join();

  std::cout << "All done\n";

  return 0;
}
```

8. METROPOLIS PROPOSAL PROBABILITIES

Metropolis algorithm, box $V_{box} = [0, 1[^2$:

1. Randomly pick a particle, and propose an update $\vec{x}=(x, y) \rightarrow (x+\delta x, y+\delta y)$, where δx, δy are random (uniform) displacements in $[-\Delta, \Delta[$.
2. Compute the energy of the new (proposed) configuration, which is needed for the calculation of the acceptance rate $P_{a,b}=\min\left[1, \frac{A_{b,a} w\left(\{\vec{x}_i\}_b\right)}{A_{a,b} w\left(\{\vec{x}_i\}_a\right)}\right]$, where $w\left(\{\vec{x}_i\}\right)$ is the Boltzmann weight.
3. Accept the new configuration with probability $P_{a,b}$.
4. Measure the quantities you want to estimate after a fixed number of updates.

Boltzmann weight

$$\exp\left[-\frac{\Delta E}{kT}\right]$$

A-priori proposal probabilities

We have to pick 1 random particle over N, and we displace it in two dimensions picking one infinitesimally small interval of size δ over the full range 2Δ. To restore configuration a from configuration b, we need exactly the same steps:

$$A_{a,b}=A_{b,a}=\frac{1}{N}\left(\frac{\delta}{2\Delta}\right)^2$$

Acceptance probability and detailed balance condition

The acceptance probability can be simplified to:

$$P_{a,b}=\min\left[1, \frac{w\left(\{\vec{x}_i\}_b\right)}{w\left(\{\vec{x}_i\}_a\right)}\right]=\min\left[1, \exp\left(-\beta \Delta E\right)\right] \text{ , where } \Delta E=E_b-E_a$$

The full transition rate is:

$$W_{a,b}=A_{a,b} P_{a,b}$$

We have to show that the detailed balance is satisfied, i.e.:

$$\frac{W_{a,b}}{W_{b,a}}=\frac{P_{a,b}}{P_{b,a}}=\exp\left(-\beta \Delta E\right)$$

Proof:
Suppose $\Delta E < 0$, then the left side of the detailed balance becomes

$$\frac{1}{\exp\left[-\beta\left(-\Delta E\right)\right]}$$ which is equal to the right part.

In the opposite case the transition rate $W_{b,a}$ is 1, and the left part trivially becomes the right part.

9. BINNING ANALYSIS

→ ML Ex3 S.3 ff. Und Ex3 accumulator.hpp

10. MUTEX TASK QUEUE EXAMPLE

```
#include <queue>
#include <array>
#include <cmath>
#include <string>
#include <iostream>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>

typedef std::size_t size_type;
typedef double value_type;
typedef std::array<value_type,2> coordinate_type;

#ifndef M_PI
#define M_PI (3.14159265358979)
#endif

/// 2-dimensional trapezoidal rule for rectangular integration
/// region [a,b]
template<class F>
value_type integrate(F f, const coordinate_type& a,
                     const coordinate_type& b, size_type n)
{
  ...
}

/// Oscillating integrand with line singularity x^2+y^3 = 0.1
value_type integrand(const coordinate_type& x)
{
  return sin(M_PI/(x[0]*x[0]+x[1]*x[1]*x[1]-0.1));
}


struct task
{
  coordinate_type a;
  coordinate_type b;
};

struct task_queue
{
  std::mutex mutex;
  std::condition_variable available;
  std::queue<task> queue;
  size_type active = 0;

  size_type regions = 0;
  value_type result = 0.;
};
```

```
void work(task_queue& tasks, size_type n, value_type maxerror)
{
  // when there are no tasks and no active workers we are
  // finished
  std::unique_lock<std::mutex> lk(tasks.mutex);
  while( !tasks.queue.empty() || tasks.active > 0 )
  {
    // wait for available tasks
    while( tasks.queue.empty() && tasks.active > 0 )
      tasks.available.wait(lk);

    // work
    ++tasks.active;
    while( !tasks.queue.empty() )
    {
      // get task
      coordinate_type a = tasks.queue.front().a;
      coordinate_type b = tasks.queue.front().b;
      tasks.queue.pop();
      lk.unlock();

      // integrate
      value_type value = integrate(integrand,a,b,n);
      value_type error = std::abs(value -
                                  integrate(integrand,a,b,n/2));

      // subdivide into quarters by halving along each
      // dimension
      lk.lock();
      if( error > maxerror )
      {
        coordinate_type center = {{0.5*(a[0]+b[0]),
                                   0.5*(a[1]+b[1])}};
        tasks.queue.push({a,center});
        tasks.queue.push({center,b});
        tasks.queue.push({{{a[0],center[1]}},
                          {{center[0],b[1]}}});
        tasks.queue.push({{{center[0],a[1]}},
                          {{b[0],center[1]}}});
        tasks.available.notify_all();
      }
      else
      {
        tasks.result += value;
        ++tasks.regions;
      }
    }
    --tasks.active;
    tasks.available.notify_all();
  }
}
```

```
int main(int argc, const char** argv)
{
  // read integration parameters from command line
  if( argc != 4 )
    throw std::runtime_error(std::string("usage: ")+argv[0]+
              " SEGMENT_SAMPLES MAX_ERROR NUM_THREADS");
  size_type segment_samples = std::stoul(argv[1]);
  value_type max_error      = std::stod (argv[2]);
  size_type nthreads        = std::stoul(argv[3]);
  std::cout.precision(10);

  // full integration volume: [-1,1]^2
  task_queue tasks;
  tasks.queue.push(task({{ {-1,-1}}, {{1,1}} }));

  // run n worker threads
  std::vector<std::thread> workers;
  for( unsigned i = 0; i < nthreads; ++i )
    workers.push_back(std::thread(std::bind(work,
            std::ref(tasks), segment_samples, max_error)));
  for( std::thread& t : workers )
    t.join();

  std::cout << "SEGMENT_SAMPLES = " << segment_samples
            << ", MAX_ERROR = " << max_error
            << ", NUM_THREADS = " << nthreads
            << ", # Regions = " << tasks.regions
            << ", Result = " << tasks.result << std::endl;
}
```

10. OPENMP MONTE CARLO

```
value_type mean = 0;
#pragma omp parallel num_threads(nthreads) reduction(+:mean)
{
  assert( nthreads ==
          static_cast<size_type>(omp_get_num_threads()) );
  // seed RNG with unique ID
  rng_type rng(nthreads*seed+omp_get_thread_num());
  mean = calcpi4(rng, nsamples/double(nthreads)+0.5);
}
```

11. OPENMP N-BODY METROPOLIS

```
double calculate_energy() const
{
  double energy = 0.0;
#pragma omp parallel for schedule(guided) reduction(+:energy)
  for(size_type i=0; i < n_; ++i) {
    double energy_part = 0.0;
    for(size_type j=0; j < i; ++j)
      energy_part += potential_(config_.first[i]-
                     config_.first[j], config_.second[i]-
                     config_.second[j]);
      energy += energy_part;
  }
  return energy;
}

double calculate_energy_diff(size_type p,
       std::pair<double,double> const& new_pos) const
{
  double dE =0.;
#pragma omp parallel for reduction(+:dE)
  for(size_type i=0; i < n_; ++i) {
    if (i != p)
    dE +=   potential_(new_pos.first-config_.first[i],
                       new_pos.second-config_.second[i])
                       - potential_(config_.first[p]
                       -config_.first[i], config_.second[p]
                       -config_.second[i]);
  }
  return dE;
}


#pragma omp parallel
#pragma omp single nowait
std::cout << std::endl << "Using " << omp_get_num_threads()
          << " threads." << std::endl;
```

12. OPENMP TASK QUEUE

```
value_type work(coordinate_type a, coordinate_type b,
                size_type n, value_type maxerror)
{
  value_type value = integrate(integrand,a,b,n);
  value_type error = std::abs(value -
                              integrate(integrand,a,b,n/2));
  // subdivide into quarters by halving along each dimension
  if( error > maxerror )
  {
    double r1, r2, r3, r4;
    coordinate_type center = {{0.5*(a[0]+b[0]),
                               0.5*(a[1]+b[1])}};

#pragma omp task shared(r1, a, b, center, n, maxerror) untied
    r1 = work(a, center, n, maxerror);

#pragma omp task shared(r2, a, b, center, n, maxerror) untied
    r2 = work(center, b, n, maxerror);

#pragma omp task shared(r3, a, b, center, n, maxerror) untied
    r3 = work({{a[0],center[1]}}, {{center[0],b[1]}}, n,
              maxerror);

    r4 = work({{center[0],a[1]}}, {{b[0],center[1]}}, n,
              maxerror);

#pragma omp taskwait
    value = r1+r2+r3+r4;
  }

  return value;
}

int main(int argc, const char** argv)
{
  // read integration parameters from command line
  if( argc != 4 )
    throw std::runtime_error(std::string("usage: ")+argv[0]+
                " SEGMENT_SAMPLES MAX_ERROR NUM_THREADS");
  size_type segment_samples = std::stoul(argv[1]);
  value_type max_error      = std::stod (argv[2]);
  size_type nthreads        = std::stoul(argv[3]);
  std::cout.precision(10);

  value_type result;

  // full integration volume: [-1,1]^2
#pragma omp parallel num_threads(nthreads)
#pragma omp single nowait
  result = work({{-1,-1}}, {{1,1}}, segment_samples,
                max_error);

  std::cout << "SEGMENT_SAMPLES = " << segment_samples
            << ", MAX_ERROR = " << max_error
            << ", NUM_THREADS = " << nthreads
            << ", Result = " << result << std::endl;
}
```

13. N-BODY LENNARD JONES

→ ML Ex4, 2

```
const unsigned DIMENSIONS = 2;

typedef std::size_t size_type;
typedef double scalar_type;
typedef std::array<scalar_type,DIMENSIONS> position;

struct potential
{
  potential(scalar_type rm, scalar_type epsilon,
            scalar_type rc=0)
  :   rm2_(rm*rm)
  ,   eps_(epsilon)
  ,   rc2_(1e10*rm)
  ,   shift_(0)
  {
    // default cut-off radius
    if(rc <= 0 )    rc = 2.5*rm/std::pow(2,1/6.);

    position x = {{}};
    position y = {{rc}};
    assert( x[0] == 0  && x[1] == 0 );
    assert( y[1] == 0 );
    shift_ = -(*this)(x,y,position());
    rc2_ = rc*rc;
    std::cout << "# Potential shift -V(rc=" << rc << ")="
              << shift_ << std::endl;
  }

  /// potential V(r^2)
  scalar_type operator()(scalar_type r2) const
  {
    if( r2 >= rc2_ )    return 0;

    scalar_type s2 = rm2_ / r2;
    scalar_type s6 = s2*s2*s2;
    return eps_*(s6*s6 - 2*s6) + shift_;
  }

  /// potential V(x,y) considering periodic boundaries
  scalar_type operator()(const position& x, const position& y,
                         const position& extent) const
  {
    scalar_type r2 = 0.;
    for( size_type d = 0; d < DIMENSIONS; ++d )
    {
      scalar_type r = dist(x[d],y[d],extent[d]);
      r2 += r*r;
    }

    return (*this)(r2);
  }

  /// compute the Lennard-Jones force particle y exerts on x
  /// and add it to f
  void add_force(position& f, const position& x,
                 const position& y, const position& extent) const
  {
    // distance vector r=x-y, considering periodic boundaries
    position r;
    scalar_type r2 = 0.;
    for( size_type d = 0; d < DIMENSIONS; ++d )
    {
      r[d] = dist(x[d],y[d],extent[d]);
      r2 += r[d]*r[d];
    }

    // potential cut-off
    if( r2 >= rc2_ )    return;

    // s = r_m/r
    // V(s) = eps * (s^12 - s^6)
    r2 = 1/r2;
    scalar_type s2 = rm2_ * r2; // (rm/r)^2
    scalar_type s6 = s2*s2*s2;  // (rm/r)^6
    // common factor
    scalar_type fr = 12*eps_ * (s6*s6 - s6) * r2;
    for( size_type d = 0; d < DIMENSIONS; ++d )
      f[d] += fr * r[d];
  }

  scalar_type cutoff_radius() const {
    return std::sqrt(rc2_); }

private:
  scalar_type dist(scalar_type x, scalar_type y,
                   scalar_type extent) const
  {
    scalar_type r = x-y;
    if     ( r < -extent/2 ) r += extent;
    else if( r >  extent/2 ) r -= extent;
    return r;
  }

  scalar_type rm2_;   // r_m^2
  scalar_type eps_;   // \epsilon
  scalar_type rc2_;   // cut-off radius r_c^2
  scalar_type shift_; // potential shift -V(r_c)
};
```

14. N-BODY: COMPARE MD AND MC

MD: Molecular Dynamics MC: Monte Carlo

Taking long-time averages of the energies in the MD simulation we obtain the system's temperature

$$\frac{d}{2}k_B T = \bar{E}_{\text{kin}}$$

from the mean kinetic energy. With this we can start a MC simulation with the same parameters and compare the MC estimate $\langle E_{\text{pot}} \rangle$ to the MD mean $\bar{E}_{\text{pot}}$.

15. N-BODY VERLET ALGORITHM

→ ML Ex4, 2

```
class simulation
{
public:
  simulation(const position& extent, const potential& pot,
             const std::vector<position>& x,
             const std::vector<position>& v )
  :   extent_(extent)
  ,   potential_(pot)
  ,   x_(x)
  ,   v_(v)
  ,   a_(x.size())
  {
    calculate_forces(a_,x_);
  }

  void evolve(scalar_type dt, size_type steps)
  {
    assert( steps >= 1 );
    configuration xold(x_), aold(a_);
    update_positions(x_,xold,v_,a_,dt);

    for( size_type s = 1; s < steps; ++s )
    {
      std::swap(x_,xold);
      #pragma omp parallel for
      for( size_type i = 0; i < x_.size(); ++i )
      {
        calculate_force(i,a_,xold);
        update_velocity(v_[i],aold[i],a_[i],dt);
        update_position(x_[i],xold[i],v_[i],a_[i],dt);
      }
      std::swap(a_,aold);
    }

    #pragma omp parallel for
    for( size_type i = 0; i < x_.size(); ++i )
    {
      calculate_force(i,a_,x_);
      update_velocity(v_[i],aold[i],a_[i],dt);
    }
  }

  void print_config() const
  {
    for( size_type i = 0; i < x_.size(); ++i )
      std::cout << x_[i] << v_[i] << a_[i] << std::endl;
    std::cout << std::endl;
  }

  std::pair<scalar_type,scalar_type> measure_energies() const
  {
    scalar_type epot = 0, ekin = 0;
    #pragma omp parallel for reduction(+:epot,ekin)
    for( size_type i = 0; i < x_.size(); ++i )
    {
      const position& xx = x_[i];
      const position& vv = v_[i];

      ekin += std::inner_product(vv.begin(),vv.end(),
                                 vv.begin(),scalar_type(0));

      for( size_type j = 0; j < i; ++j )
        epot += potential_(xx,x_[j],extent_);
    }

    return std::make_pair(0.5*ekin,epot);
  }

private:
  typedef std::vector<position> configuration;

  void update_positions(configuration& x,
       const configuration& xold, const configuration& v,
       const configuration& a, scalar_type dt)
  {
    #pragma omp parallel for
    for( size_type i = 0; i < x.size(); ++i )
      update_position(x[i],xold[i],v[i],a[i],dt);
  }

  void update_velocities(configuration& v,
              const configuration& aold,
              const configuration& a, scalar_type dt)
  {
    #pragma omp parallel for
    for( size_type i = 0; i < v.size(); ++i )
      update_velocity(v[i],aold[i],a[i],dt);
  }

  void calculate_forces(configuration& a,
                        const configuration& x)
  {
    #pragma omp parallel for
    for( size_type i = 0; i < x.size(); ++i )
      calculate_force(i,a,x);
  }

  void update_position(position& xx, const position& xxold,
       const position& vv, const position& aa, scalar_type dt)
  {
    for( size_type d = 0; d < DIMENSIONS; ++d )
    {
      // Verlet step
      xx[d] = xxold[d] + vv[d]*dt + 0.5*dt*dt*aa[d];

      // enforce periodic boundaries
      xx[d] = fmod(xx[d],extent_[d]);
      if( xx[d] <  0 )   xx[d] += extent_[d];
      assert( xx[d] >= 0 && xx[d] < extent_[d] );
    }
  }

  void update_velocity(position& vv, const position& aaold,
                       const position& aa, scalar_type dt)
  {
    for( size_type d = 0; d < DIMENSIONS; ++d )
      vv[d] += 0.5*dt*(aaold[d] + aa[d]);
  }

  void calculate_force(size_type i, configuration& a,
                       const configuration& x)
  {
    const position& xx = x[i];
    position& aa = a[i];
    std::fill(aa.begin(),aa.end(),scalar_type(0));

    for( size_type j = 0; j < x.size(); ++j )
    {
      if( j == i ) continue;
      potential_.add_force(aa,xx,x[j],extent_);
    }
  }


  position extent_; /// system extent along each dimension
  potential potential_;

  configuration x_;
  configuration v_;
  configuration a_;
};
```

16. DIFFUSION 2D

Diffusion equation: $\frac{\partial \rho(\boldsymbol{r},t)}{\partial t} = D\nabla^2\rho(\boldsymbol{r},t)$

2D: $\frac{\partial f}{\partial \rho} = D\Delta\rho = D\nabla^2\rho = D\frac{\partial^2\rho}{\partial x^2} + D\frac{\partial^2\rho}{\partial y^2}$

Discrete time: $t = n\cdot\Delta t$

Spatial points: $x_i = i\cdot\Delta x\ ,\quad y_j = j\cdot\Delta y$

Forward-Euler:

$$\frac{\rho_{i,j}^{(n+1)} - \rho_{i,j}^{(n)}}{\Delta t} = D\left(\frac{\rho_{i+1,j}^{(n)} - 2\rho_{i,j}^{(n)} + \rho_{i-1,j}^{(n)}}{\Delta x^2} + \frac{\rho_{i,j+1}^{(n)} - 2\rho_{i,j}^{(n)} + \rho_{i,j-1}^{(n)}}{\Delta y^2}\right)$$

To analyse the stability we perform an von Neumann stability analysis, using a plane wave ansatz:

$$\rho_{i,j}^{n} = \zeta^n e^{i(k_x x_i + k_y y_j)}$$

to solve the differential equation. For stability we require

$$|\zeta| \le 1$$

otherwise the solution will grow and diverge.
From solving the discretised differential equation using our plane wave ansatz we obtain

$$\begin{aligned}\zeta &= 1 - \Delta t\left(\frac{e^{ik_x\Delta x} - 2 + e^{-ik_x\Delta x}}{\Delta x^2} + \frac{e^{ik_y\Delta y} - 2 + e^{-ik_y\Delta y}}{\Delta y^2}\right)\\ &= 1 - \left[\frac{D\Delta t}{\Delta x^2}\right](2\cos(k_x\Delta x) - 2) + \left[\frac{D\Delta t}{\Delta y^2}\right](2\cos(k_y\Delta y) - 2)\\ &= 1 - 4\left[\frac{D\Delta t}{\Delta x^2}\right]\sin^2\left(\frac{k_x\Delta x}{2}\right) - 4\left[\frac{D\Delta t}{\Delta y^2}\right]\sin^2\left(\frac{k_y\Delta y}{2}\right)\end{aligned}$$

which leads to the stability condition

$$4\frac{D\Delta t}{\Delta x^2} + 4\frac{D\Delta t}{\Delta y^2} \le 2$$

The time step should therefore satisfy

$$\Delta t \le \frac{\Delta x^2\Delta y^2}{(\Delta x^2 + \Delta y^2)2D}$$

For equal spacing $\Delta x = \Delta y$ this simplifies to

$$\Delta t \le \frac{\Delta x^2}{4D}\ .$$

17. COMPLETE MPI EXAMPLE

```cpp
// Example solutions for HPC course
// (c) 2012 Jan Gukelberger, ETH Zurich
// (c) 2012 Michele Dolfi, ETH Zurich

#include <random>
#include <iostream>
#include <iomanip>
#include <stdexcept>
#include <cassert>
#include <mpi.h>
#include <chrono>

typedef double value_type;
typedef std::size_t size_type;
typedef std::mt19937 rng_type;


value_type calcpi4(rng_type& rng, size_type nsamples)
{
  std::uniform_real_distribution<value_type> dist(0,1);
  size_type hits = 0;
  for( size_type i = 0; i < nsamples; ++i )
  {
    value_type x = dist(rng);
    value_type y = dist(rng);
    hits += ( (x*x + y*y) < 1. );
  }

  return hits / value_type(nsamples);
}

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  int np, rank;

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&np);


std::chrono::time_point<std::chrono::high_resolution_clock>
start, end;
  if (rank==0)
    start = std::chrono::high_resolution_clock::now();

  if( argc != 3 && rank == 0)
    throw std::runtime_error(std::string("usage: ")+argv[0]+
                             " NUM_SAMPLES SEED");
  const size_type nsamples = std::stoul(argv[1]);
  const size_type seed     = std::stoul(argv[2]);

  if (rank == 0)
    std::cout << "#procs=" << np << ", #samples=" << nsamples
              << ", seed=" << seed << ": " << std::flush;

  rng_type rng(np*seed+rank); // seed RNG with unique ID
  value_type partial = calcpi4(rng,
                               nsamples/double(np)+0.5);
  value_type mean;
  MPI_Reduce(&partial, &mean, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);

  if (rank == 0) {
    // evaluate results:
    // As X_i \elem {0,1}, we don't need to accumulate the sum
    // of squares but have \overline{X^2}=\overline{X}
    mean /= np;
    value_type error = std::sqrt(1./(nsamples-1.) *
                                 (mean - mean*mean));
    std::cout << std::setprecision(10) << "pi = " << 4*mean
              << " +/- " << 4*error << std::endl;

    end = std::chrono::high_resolution_clock::now();
    int elapsed_time =
      std::chrono::duration_cast<std::chrono::microseconds>
                                 (end-start).count();
    std::cout << "elapsed time: " << elapsed_time << "mus\n";
  }

  MPI_Finalize();
}
```

18. SSE

```
#include <x86intrin.h>
#include "aligned_allocator.hpp"
using hpc12::aligned_allocator;
// aligned to the cacheline size
typedef std::vector<scalar_type,
aligned_allocator<scalar_type,64> > positions_type;

/// compute the Lennard-Jones force particle at position x0
scalar_type compute_force(positions_type const& positions,
                          scalar_type x0)
{
  scalar_type rm2   = rm * rm;
  scalar_type eps12 = 12*eps;
  size_type   ndiv4 = N/4;

  __m128 mmx0    = _mm_load1_ps(&x0);
  __m128 mmrm2   = _mm_load1_ps(&rm2);
  __m128 mmeps12 = _mm_load1_ps(&eps12);
  __m128 mmforce = _mm_setzero_ps();

  for (size_type i=0; i<ndiv4; ++i) {
    __m128 mmxi    = _mm_load_ps(&positions[i*4]);
    __m128 mmr     = _mm_sub_ps(mmx0,mmxi);
    __m128 mmrinv  = _mm_rcp_ps(mmr);
    __m128 mmrinv2 = _mm_mul_ps(mmrinv,mmrinv);
    __m128 mms2    = _mm_mul_ps(mmrm2,mmrinv2);
    __m128 mms6    = _mm_mul_ps(mms2,_mm_mul_ps(mms2,mms2));

    __m128 mmpart  = _mm_mul_ps(_mm_mul_ps(mmeps12,
                     _mm_sub_ps(_mm_mul_ps(mms6,mms6),
                     mms6)), mmrinv);

    mmforce = _mm_add_ps(mmforce, mmpart);
  }
  scalar_type forces[4];
  _mm_store_ps(forces,mmforce);

  for (size_type i=ndiv4*4 ; i<N ; ++i) {
    scalar_type r  = x0 - positions[i];
    scalar_type rinv = 1./r;
    scalar_type rinv2 = rinv * rinv;
    scalar_type s2 = rm2 * rinv2;  // (rm/r)^2
    scalar_type s6 = s2*s2*s2;     // (rm/r)^6
    forces[0] += 12*eps * (s6*s6 - s6) * rinv;
  }

  return forces[0]+forces[1]+forces[2]+forces[3];
}
```

19. AVX

```
/// compute the Lennard-Jones force particle at position x0
scalar_type compute_force(positions_type const& positions,
                          scalar_type x0)
{
  scalar_type rm2   = rm * rm;
  scalar_type eps12 = 12*eps;
  size_type   ndiv8 = N/8;

  __m256 mmx0    = _mm256_set1_ps(x0);
  __m256 mmrm2   = _mm256_set1_ps(rm2);
  __m256 mmeps12 = _mm256_set1_ps(eps12);
  __m256 mmforce = _mm256_setzero_ps();

  for (size_type i=0; i<ndiv8; ++i) {
    __m256 mmxi    = _mm256_load_ps(&positions[i*8]);
    __m256 mmr     = _mm256_sub_ps(mmx0,mmxi);
    __m256 mmrinv  = _mm256_rcp_ps(mmr);
    __m256 mmrinv2 = _mm256_mul_ps(mmrinv,mmrinv);
    __m256 mms2    = _mm256_mul_ps(mmrm2,mmrinv2);
    __m256 mms6    = _mm256_mul_ps(mms2,
                           _mm256_mul_ps(mms2,mms2));

    __m256 mmpart  = _mm256_mul_ps(_mm256_mul_ps(mmeps12,
                     _mm256_sub_ps(_mm256_mul_ps(mms6,mms6),
                     mms6)), mmrinv);

    mmforce = _mm256_add_ps(mmforce, mmpart);
  }
  scalar_type forces[8];
  _mm256_store_ps(forces,mmforce);

  for (size_type i=ndiv8*8 ; i<N ; ++i) {
    scalar_type r  = x0 - positions[i];
    scalar_type rinv = 1./r;
    scalar_type rinv2 = rinv * rinv;
    scalar_type s2 = rm2 * rinv2;  // (rm/r)^2
    scalar_type s6 = s2*s2*s2;     // (rm/r)^6
    forces[0] += 12*eps * (s6*s6 - s6) * rinv;
  }

  return std::accumulate(forces, forces+8, 0.);
}
```

20. SSE WITH CUTOFF

```
/// compute the Lennard-Jones force particle at position x0
scalar_type compute_force(positions_type const& positions,
                          scalar_type x0, scalar_type rc)
{
  scalar_type rm2   = rm * rm;
  scalar_type eps12 = 12*eps;
  size_type   ndiv4 = N/4;

  __m128 mmx0    = _mm_load1_ps(&x0);
  __m128 mmrc    = _mm_load1_ps(&rc);
  __m128 mmrm2   = _mm_load1_ps(&rm2);
  __m128 mmeps12 = _mm_load1_ps(&eps12);
  __m128 mmforce = _mm_setzero_ps();
  // bit mask with 1 at the positions of the sign bits within
  // an __m128 vector
  __m128 signmask = _mm_castsi128_ps(
                    _mm_set1_epi32(1u << 31));


  for (size_type i=0; i<ndiv4; ++i) {
    __m128 mmxi    = _mm_load_ps(&positions[i*4]);
    __m128 mmr     = _mm_sub_ps(mmx0,mmxi);
    // compute abs(r) by unsetting the sign bits
    __m128 mmabsr  = _mm_andnot_ps(signmask, mmr);
    // rinv = (abs(r) < rc) ? 1/r : 0
    __m128 mmrinv  = _mm_and_ps( _mm_cmplt_ps(mmabsr,mmrc),
                                 _mm_rcp_ps(mmr) );
    __m128 mmrinv2 = _mm_mul_ps(mmrinv,mmrinv);
    __m128 mms2    = _mm_mul_ps(mmrm2,mmrinv2);
    __m128 mms6    = _mm_mul_ps(mms2,_mm_mul_ps(mms2,mms2));

    __m128 mmpart  = _mm_mul_ps(_mm_mul_ps(mmeps12,
                     _mm_sub_ps(_mm_mul_ps(mms6,mms6),
                     mms6)), mmrinv);

    mmforce = _mm_add_ps(mmforce, mmpart);
  }
  scalar_type forces[4];
  _mm_store_ps(forces,mmforce);

  for (size_type i=ndiv4*4 ; i<N ; ++i) {
    scalar_type r  = x0 - positions[i];
    if( r >= rc ) continue;
    scalar_type rinv = 1./r;
    scalar_type rinv2 = rinv * rinv;
    scalar_type s2 = rm2 * rinv2;  // (rm/r)^2
    scalar_type s6 = s2*s2*s2;     // (rm/r)^6
    forces[0] += 12*eps * (s6*s6 - s6) * rinv;
  }

  return forces[0]+forces[1]+forces[2]+forces[3];
}
```