PHYS-E0412, Computational Physics, Lecture 6, 12 February 2019

Ilja Makkonen

Aalto University
School of Science

Learning objectives for week 6

- Case study of many-body quantum simulation: Interacting particles in a harmonic oscillator studied using Variational (Quantum) Monte Carlo
- VMC wave function ansatz and parameter optimisation
- Basic principles of parallel programming and architectures used in high-performance computing (hands-on stuff possibly at later homeworks?)

Homework 6:

Finding the variational ground state wave function and energy of a Helium atom (S=0) using Variational Monte Carlo.

The stationary many-body Schrödinger equation

$$\hat{H}\Psi_n = E_n\Psi_n, \text{ where}$$

$$\hat{H} = -\frac{1}{2}\sum_i \nabla_i^2 + \frac{1}{2}\sum_i\sum_{j\neq i}\frac{1}{|\mathbf{r}_i - \mathbf{r}_j|} + \sum_i V_{\text{ext}}(\mathbf{r}_i)$$

for like particles in external potential (unit charge, like for electrons), and

$$\Psi = \Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N),$$

is a high-dimensional (3^N) object.

Wave function symmetry/ antisymmetry (”statistics”)

The probability density of a two-particle wave function of indistinguishable particles should not change when the particles are interchanged,

$$|\Psi(\mathbf{r}_1, \mathbf{r}_2)|^2 = |\Psi(\mathbf{r}_2, \mathbf{r}_1)|^2$$

We can achieve this in two ways:

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = \pm\Psi(\mathbf{r}_2, \mathbf{r}_1)$$

The particles whose wave functions are symmetric (”+” sign) under particle interchange have integral or zero intrinsic spin, and are termed bosons. Particles whose wave functions which are anti-symmetric (”-” sign) under particle interchange have half-integral intrinsic spin, and are termed fermions. Consider, what happens when the particles approach one another?

Wave function symmetry/ antisymmetry (”statistics”)

Consider a two-body noninteracting system. Let’s try to construct the system as a linear combination of products of two single particle wave functions (since particles are indistinguishable, we cannot know which term would describe the system)

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = A\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) + B\psi_a(\mathbf{r}_2)\psi_b(\mathbf{r}_1)$$

There are only two correctly normalised combinations possible

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = \frac{1}{\sqrt{2}}[\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) \pm \psi_a(\mathbf{r}_2)\psi_b(\mathbf{r}_1)]$$

In the case of Fermions (”-”), if *a=b*, then, $\Psi = 0$ implying that no two fermions can occupy the same state. Bosons do not obey this principle and many of them can be found in the lowest (ground) state. These properties result in Fermi-Dirac and Bose-Einstein statistics for non-interacting fermions and bosons, respectively.

Variational quantum Monte Carlo

Any wave function Ψ satisfies:

$$E_0 \leq E = \frac{\int \Psi^* H \Psi \; d\mathbf{R}}{\int \Psi^* \Psi \; d\mathbf{R}} \; ,$$

if it has correct particle statistics.

E_0 is the ground state energy.

The integrals are high-dimensional: $d \times N$, where

d is the dimension of the space and N number of particles

$\mathbf{R}$ contains all these coordinates.

Invent a wave function Ψ and calculate the energy as above.

Very simple!

Variational quantum Monte Carlo

We can use Monte Carlo integration with Metropolis sampling:

$$E = \frac{\int \Psi^* H \Psi \, d\mathbf{R}}{\int \Psi^* \Psi \, d\mathbf{R}} = \frac{\int |\Psi|^2 \frac{H\Psi}{\Psi} \, d\mathbf{R}}{\int |\Psi|^2 \, d\mathbf{R}} = \lim_{M\to\infty} \frac{1}{M} \sum_{i=1}^{M} E_L(\mathbf{R}_i) = \langle E_L \rangle_{|\Psi|^2} \, ,$$

where the local energy is

$$E_L = \frac{H\Psi}{\Psi}$$

Constant for exact eigenstates

and $\mathbf{R}_i$ is sampled from $|\Psi(\mathbf{R})|^2$.

If you have an analytic many-body wave function,
Monte Carlo is very good in extracting information out of it!

Warming up with harmonic oscillator

Hamiltonian:

$$H = -\frac{1}{2}\frac{d^2}{dx^2} + \frac{1}{2}x^2 \ ,$$

and trial wave function

$$\psi(x) = e^{-\alpha x^2} \ .$$

Local energy:

$$E_L = \frac{H\psi}{\psi} = -\frac{1}{2}\frac{d^2\psi}{dx^2}\frac{1}{\psi} + \frac{1}{2}x^2 = \alpha - 2\alpha^2 x^2 + \frac{1}{2}x^2 \ .$$

Notice that potential part in E_L is always simple: $\frac{V\psi}{\psi} = V$.

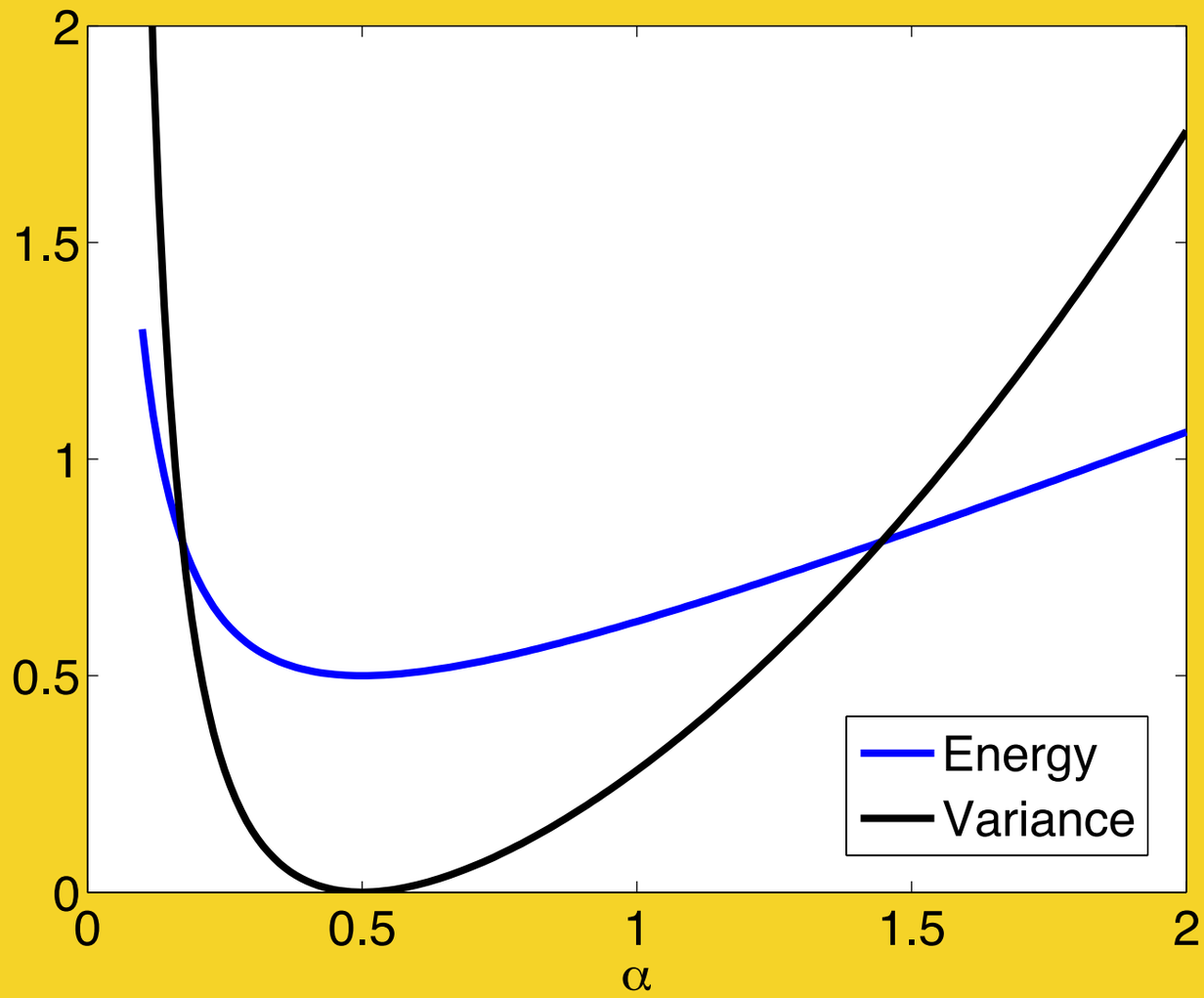
Gaussian integrals are doable: $\langle x^2 \rangle = \frac{\int x^2\psi^2}{\int \psi^2} = \frac{1}{4\alpha}$. So energy is $E(\alpha) = \frac{1}{8\alpha} + \frac{\alpha}{2}$.

Variance of local energy: $\sigma^2_{E_L} = \frac{(1-4\alpha^2)^2}{32\alpha^2}$

We use Monte Carlo for integrals later, as no analytic results for real problems!

Warming up with harmonic oscillator

Energy and variance as a function of the parameter in the wave function:



Both curves show a minimum at the same parameter value.

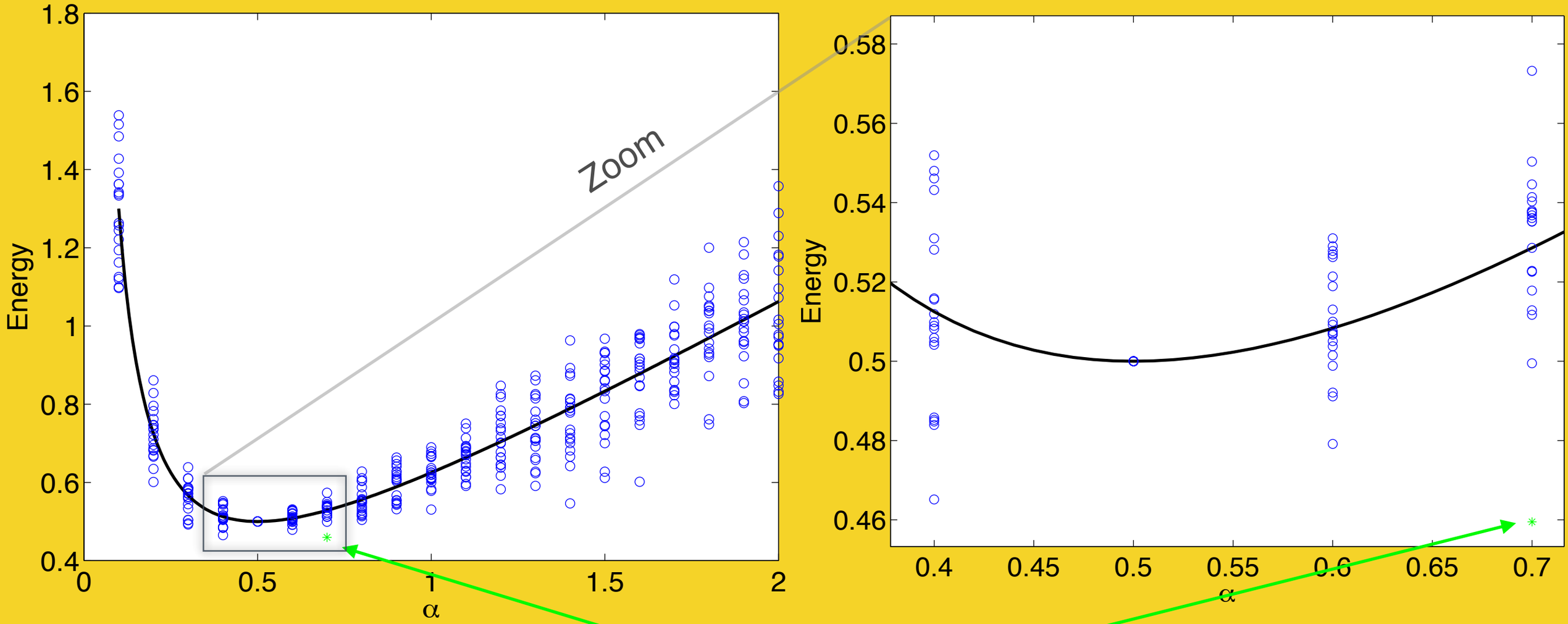
The variance is zero at that point.

These are the analytic results.

Warming up with harmonic oscillator

Calculating the energy stochastically, many simulations.

Sample coordinate from the density, and evaluate local energy, gives circles below.
Lines are the analytic results.



Stochastic noise make it somewhat hard to locate the minimum of energy.

You might think that the minimum is at 0.4 if you have only a few simulations.

Simplest VMC

Usually wave function contains tunable parameters α.

- Invent a many-body wave function Ψ with a few parameters.
- Calculate E.
- Vary components of α and locate minimum of energy.
- Calculate other observables with optimal parameters.

Very limited! Optimization is very slow.

We actually try to optimize a function $\lim_{M\to\infty} \frac{1}{M} \sum_{i=1}^{M} E_L(\mathbf{R}_i)$

For a finite M, cost function has noise:

Mathematically this is optimization of a noisy function.

Traditional optimizations methods typically fail.

Metropolis

The high-dimensional integrals again calculated using Monte Carlo.

N_W walkers perform N_S metropolis steps.

For each walker and step:

- Move the walker $\mathbf{R} \to \mathbf{R}'$ by e.g. moving one of particles.
- Calculate the ratio $p = |\Psi(\mathbf{R}')/\Psi(\mathbf{R})|^2$.
- Accept move if $p > r$ (r random number $[0, 1]$)
- Make measurements (like local energy).

Finish when observables are converged.

Walkers are uncorrelated, can be directly used for error estimate.

Energy derivative

Derivative of the energy with respect to one of the parameters.

Assume real Ψ, and denoting $\Psi' = \frac{\partial \Psi}{\partial \alpha}$ below.

$$
\begin{aligned}
\frac{\partial E}{\partial \alpha} &= \frac{\left(\int \Psi' H \Psi + \int \Psi H \Psi'\right) \int \Psi^2 - 2 \int \Psi' \Psi \int \Psi H \Psi}{\left(\int \Psi^2\right)^2} \\
&= \frac{2 \int \Psi^2 \frac{\Psi'}{\Psi} \frac{H\Psi}{\Psi}}{\int \Psi^2} - \frac{2 \int \Psi^2 \frac{\Psi'}{\Psi} \int \Psi^2 \frac{H\Psi}{\Psi}}{\int \Psi^2 \int \Psi^2} \\
&= 2 \left\langle \frac{\Psi'}{\Psi} \frac{H\Psi}{\Psi} \right\rangle - 2 \left\langle \frac{\Psi'}{\Psi} \right\rangle \left\langle \frac{H\Psi}{\Psi} \right\rangle \\
&= 2 \left\langle \frac{\Psi'}{\Psi} E_L \right\rangle - 2 \left\langle \frac{\Psi'}{\Psi} \right\rangle \langle E_L \rangle
\end{aligned}
$$

In $\langle \ldots \rangle$, $\mathbf{R}$ is sampled from Ψ^2.

If E_L is constant, derivative is zero.

Optimization in VMC

Sample $|\Psi(\mathbf{R}, \alpha_i)|^2$ with m walkers $\{\mathbf{R}_j\}_{j=1}^m$ using e.g. Metropolis algorithm.

Change parameters according to rule:

$$\alpha_{i+1} = \alpha_i - \gamma_i \nabla_\alpha E$$

where ∇-term is done as in previous page.

Gradient of energy with respect to parameters, now $\langle \ldots \rangle$ is an average over walkers.

γ_i is a damping factor, ensuring convergence.

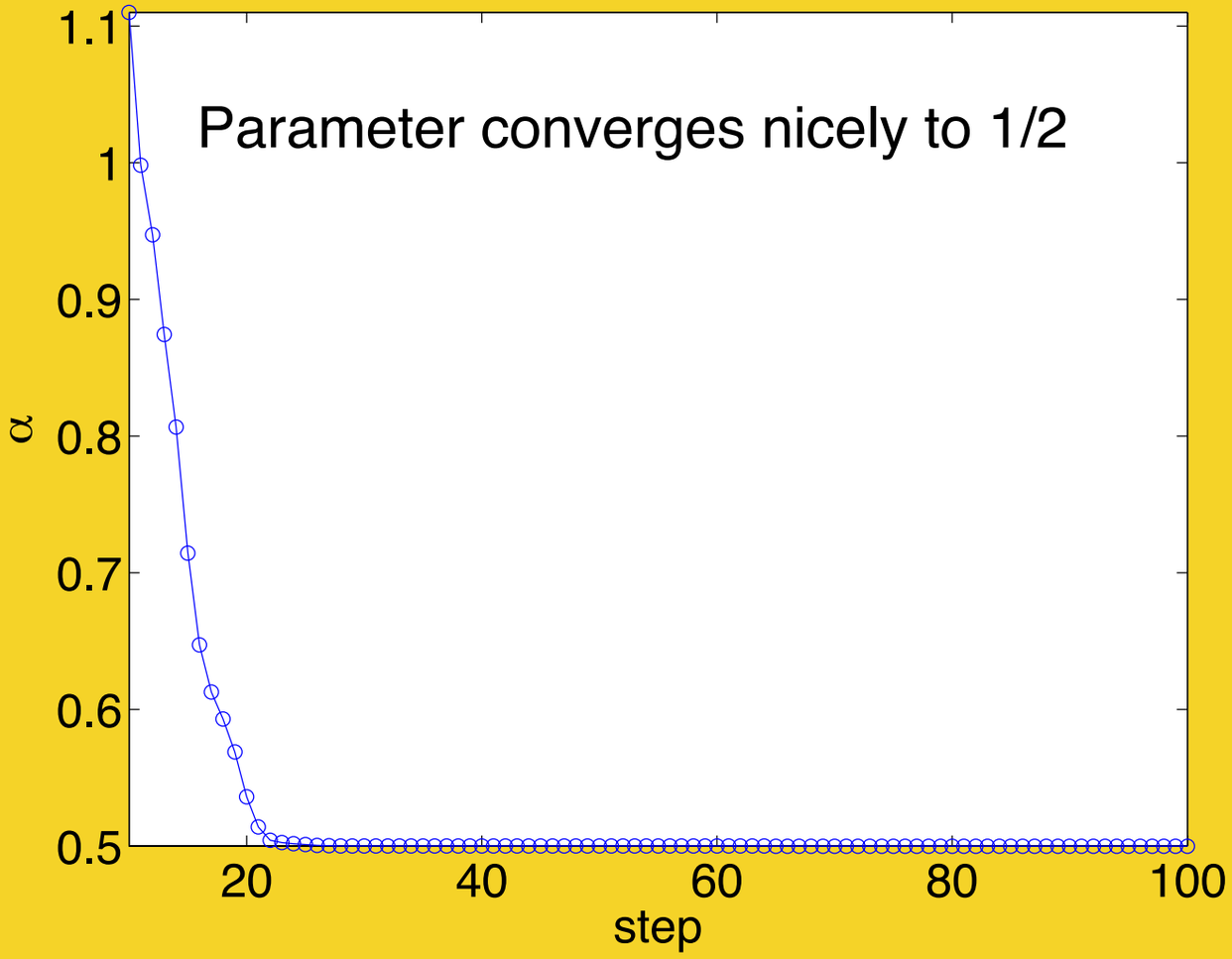
Parameters move to direction that lowers energy.

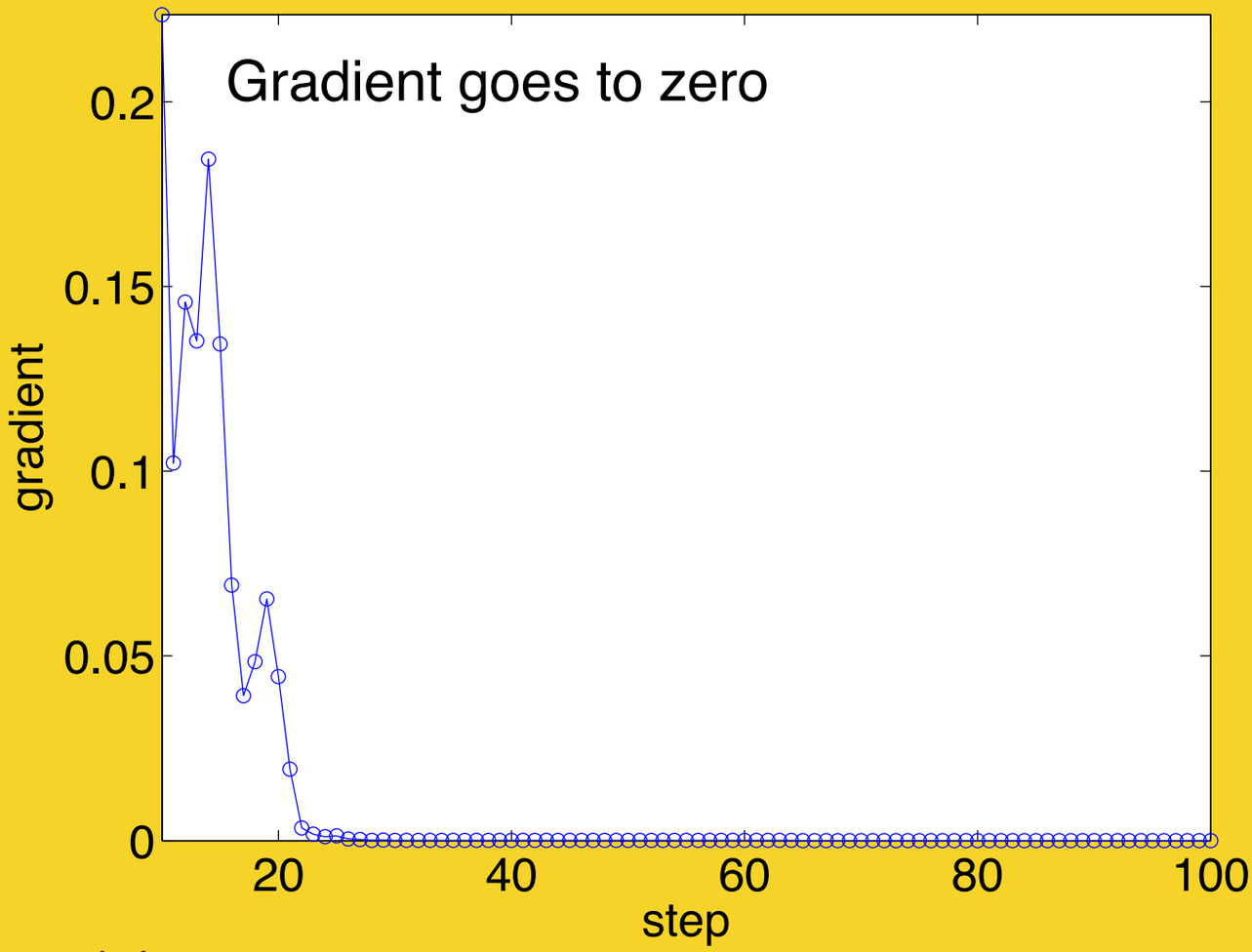
Walkers sample density (that changes).

When converged, gradient is zero on average.

$\nabla_\alpha E$ still fluctuates (in typical cases).

VMC, pure harmonic

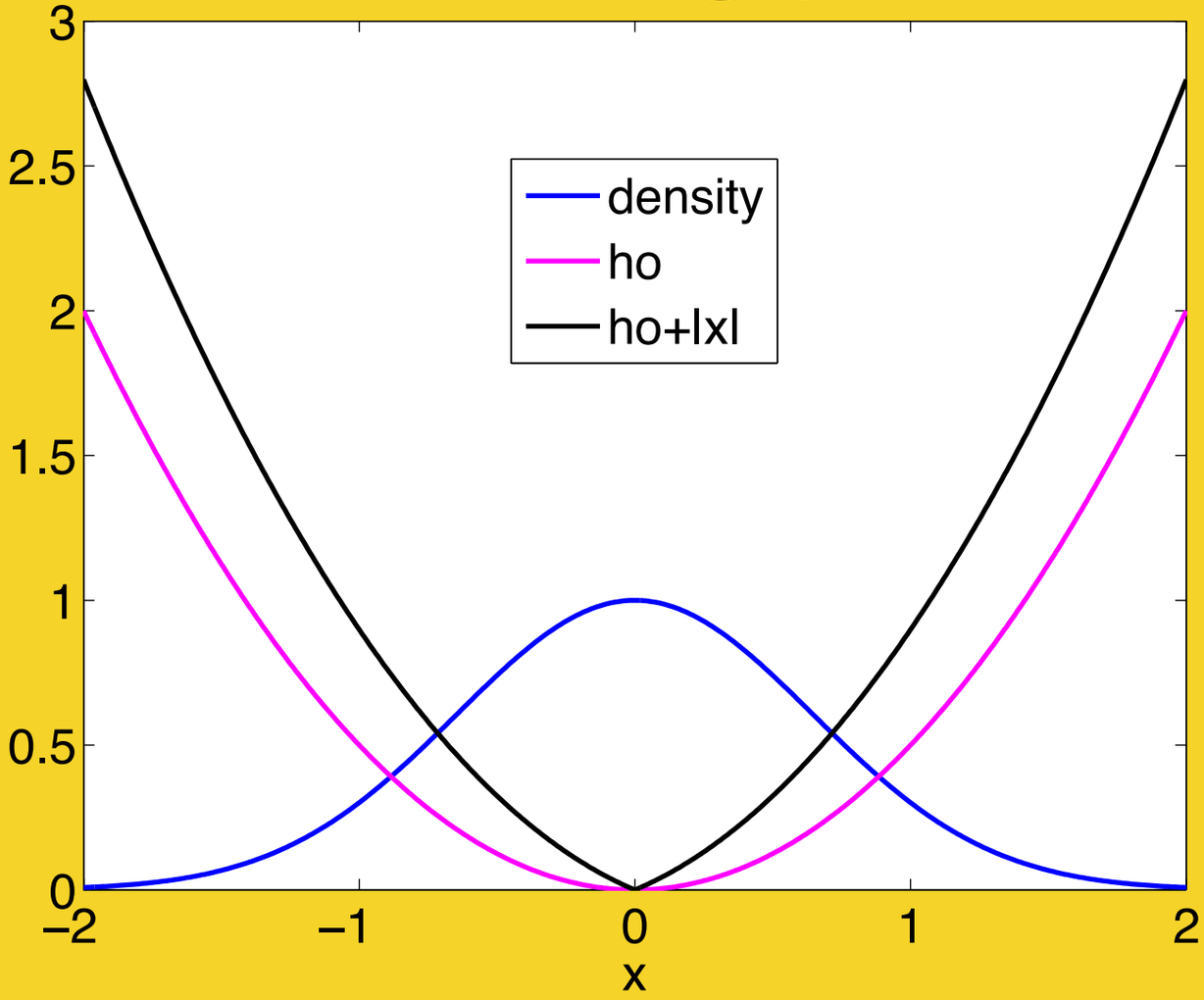


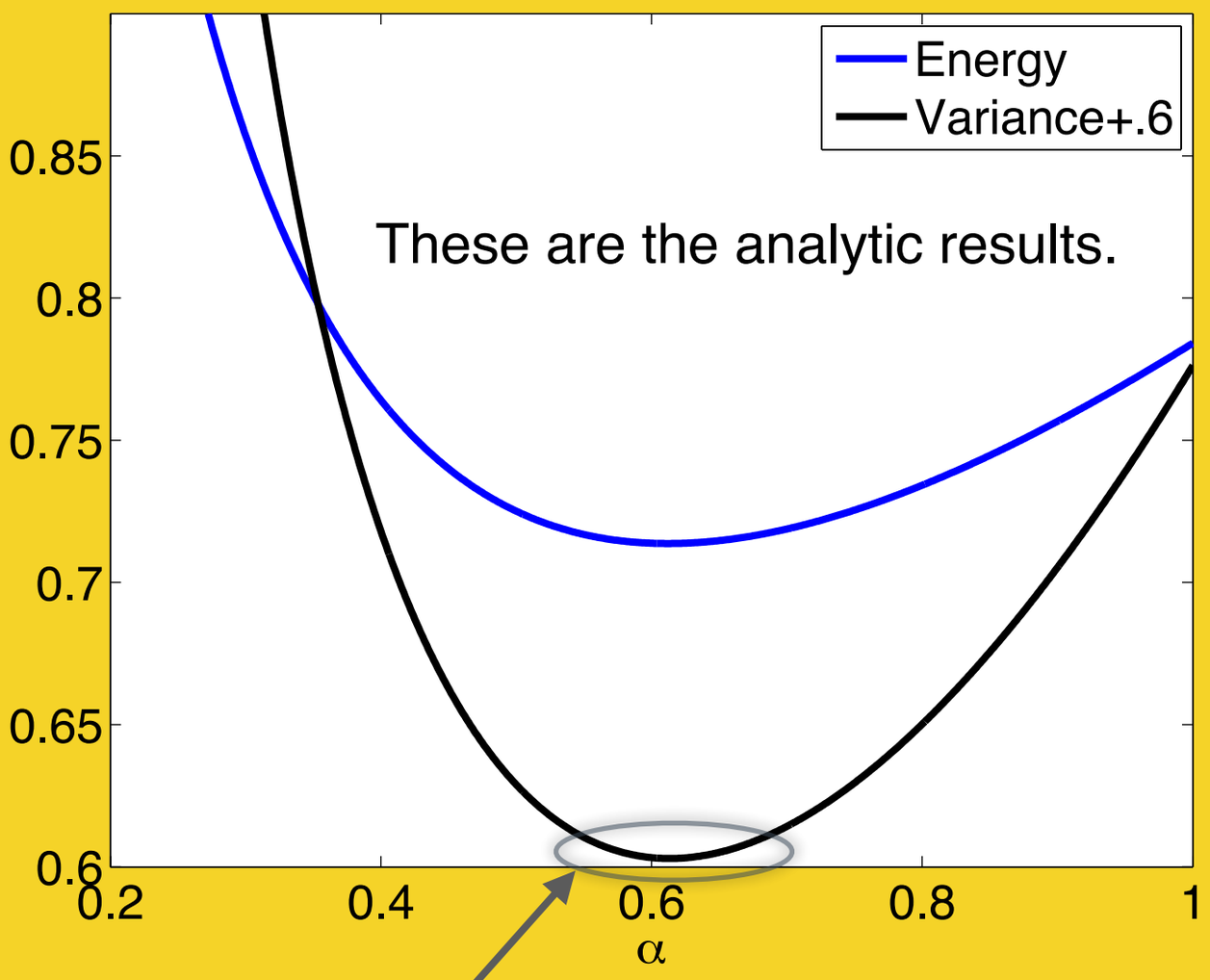


This is not a typical case, as the optimal wave function is the exact ground state to the problem.

Next, a more “typical” case…

Perturbing potential with |x|





Same trial wave function, gaussian.

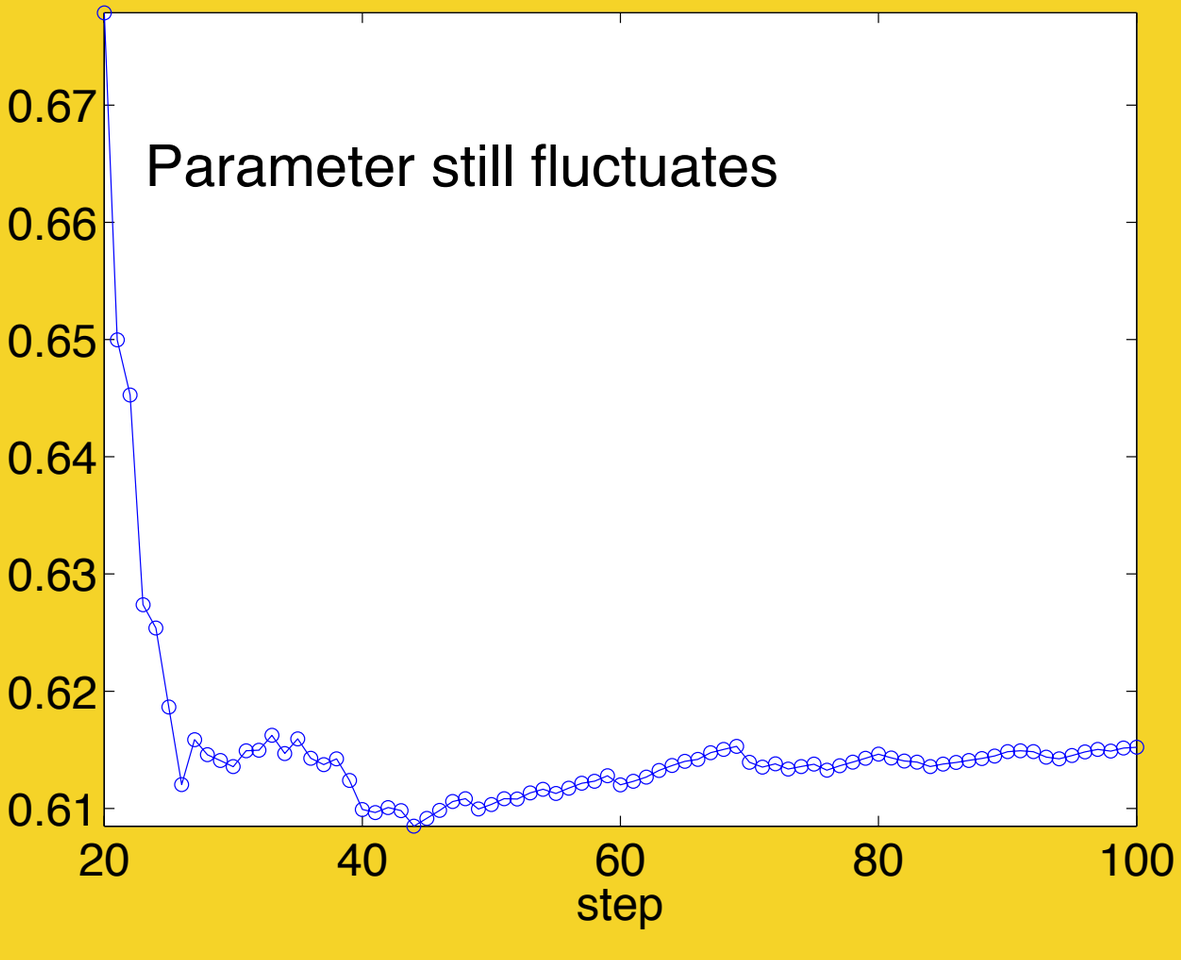
This does not solve the problem exactly anymore.

The variance is now finite even for the optimal parameter.

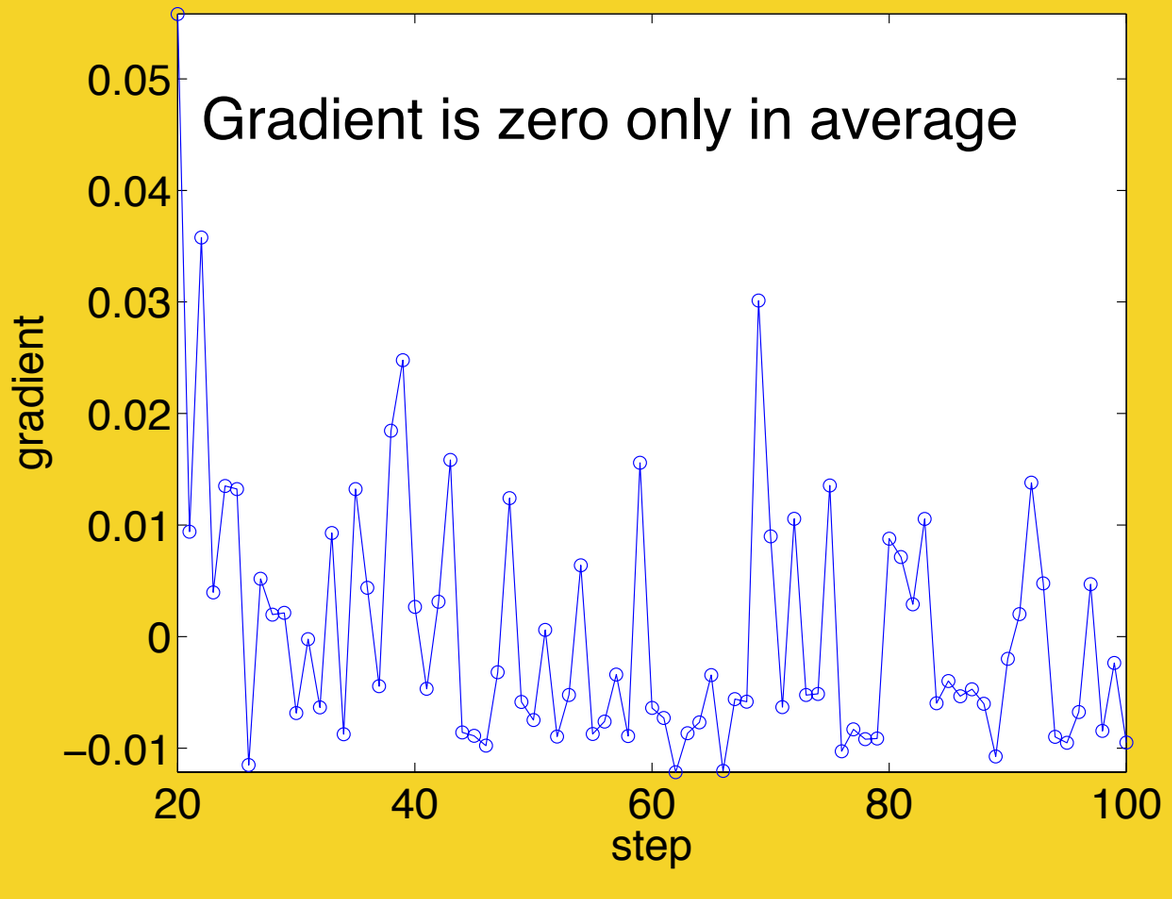
Strength of the linear term around 0.4 [actual value is 1/sqrt(2*pi) for simpler plotting].

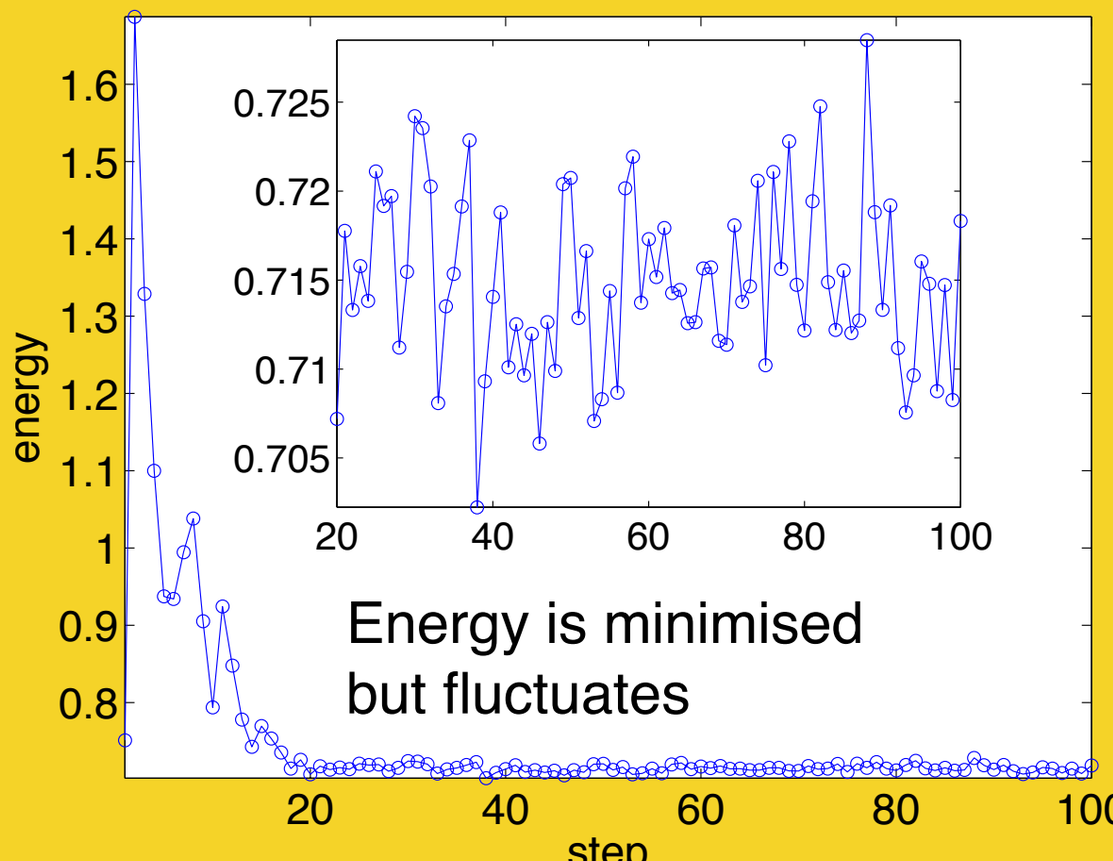
These are again analytic results, but we try to do the wave function optimisation now with a Monte Carlo technique.

VMC, |x|+harmonic



We see that the simulation reaches the correct parameter value range, and the gradient fluctuates around zero there.





Correlated fermion wave function

Very often used, simple generalization of Slater determinant:

$$\Psi(\mathbf{R}) = \det{}_{\uparrow}[\psi_i(\mathbf{r}_j)]\det{}_{\downarrow}[\psi_k(\mathbf{r}_l)] \prod_{i<j}^{N} J(r_{ij}) \ ,$$

where J is the Jastrow pair-correlation factor.

Has explicit dependence on the inter-electron distances, captures correlation.

Usually, both ψ and J have variational parameters.

Example: Two electrons in 2D ho

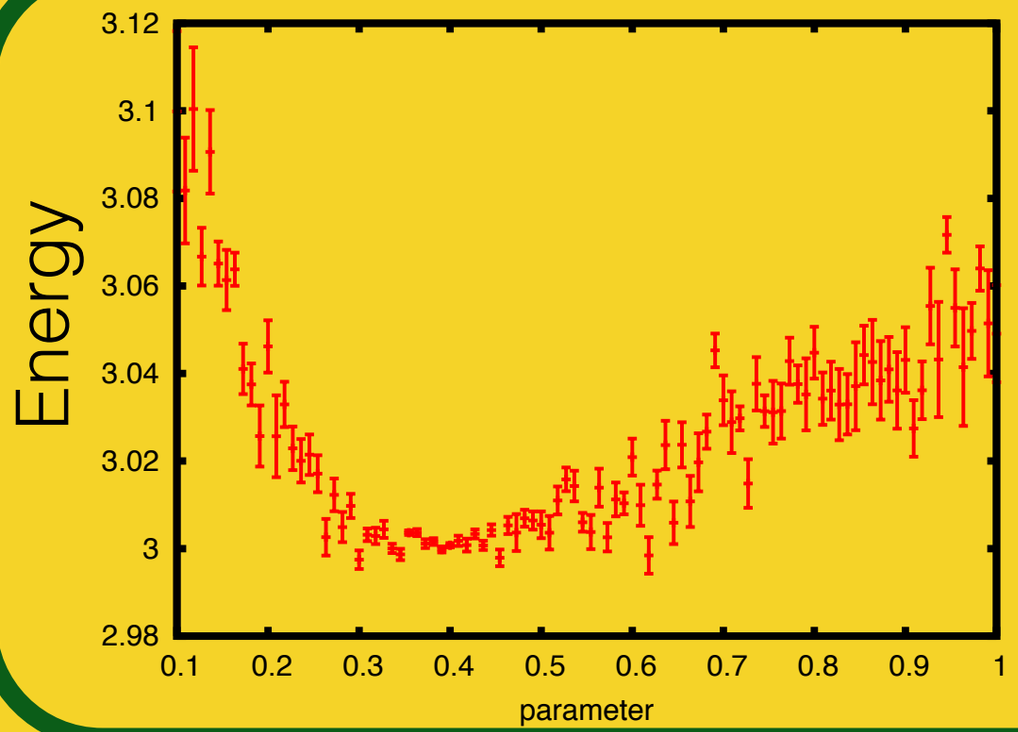
$$H = -\frac{1}{2}\sum_{i=1}^{2}(\nabla_i^2 - r_i^2) + \frac{1}{r_{12}} \ , \quad \psi = e^{-r_1^2/2}e^{-r_2^2/2}(1 + r_{12})$$

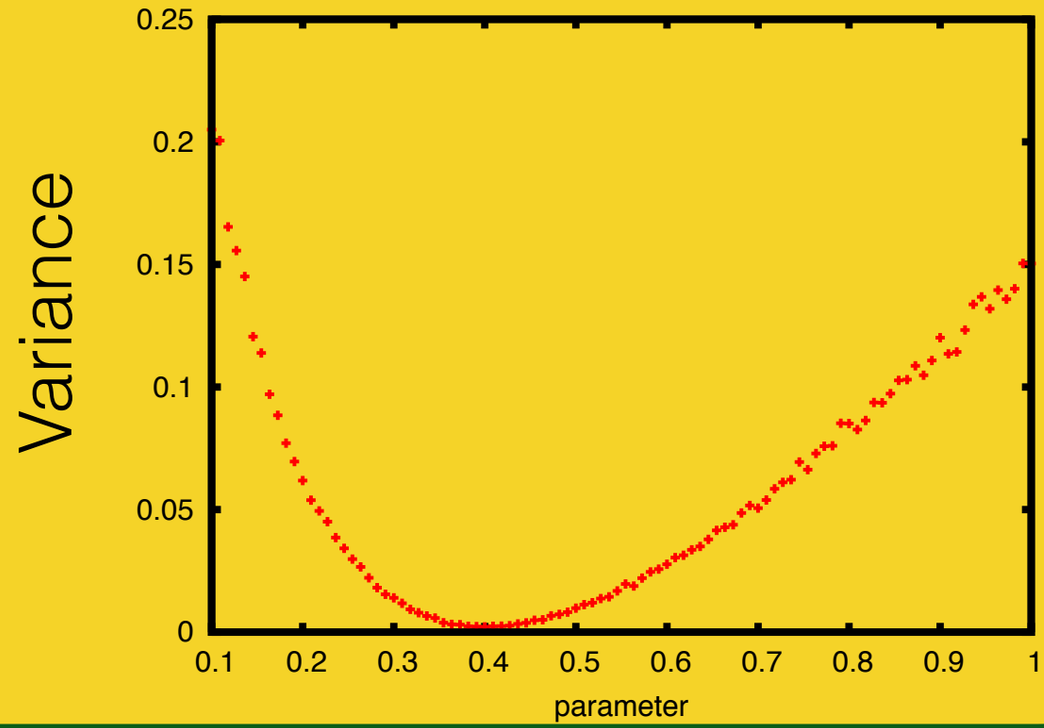
exact ground state with $E = 3$.

Generalization to any interaction strength: C/r_{12}, approximation $J(r_{12}) = \exp(Cr_{12}/(1 + \alpha r_{12}))$.

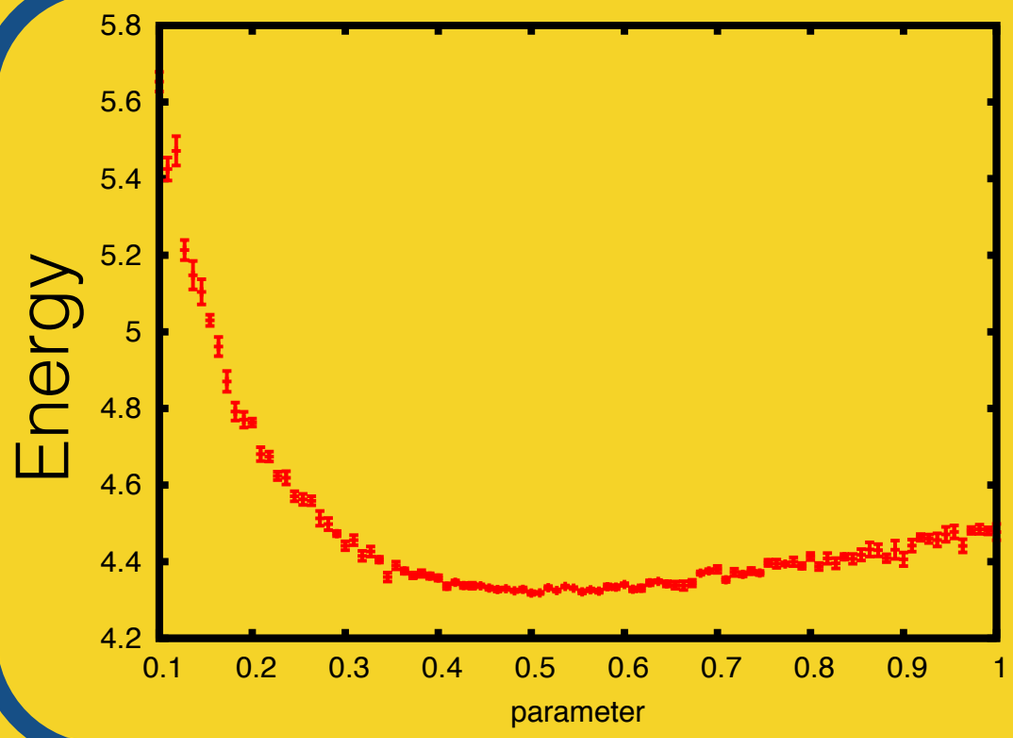
Results for two particles in 2D ho

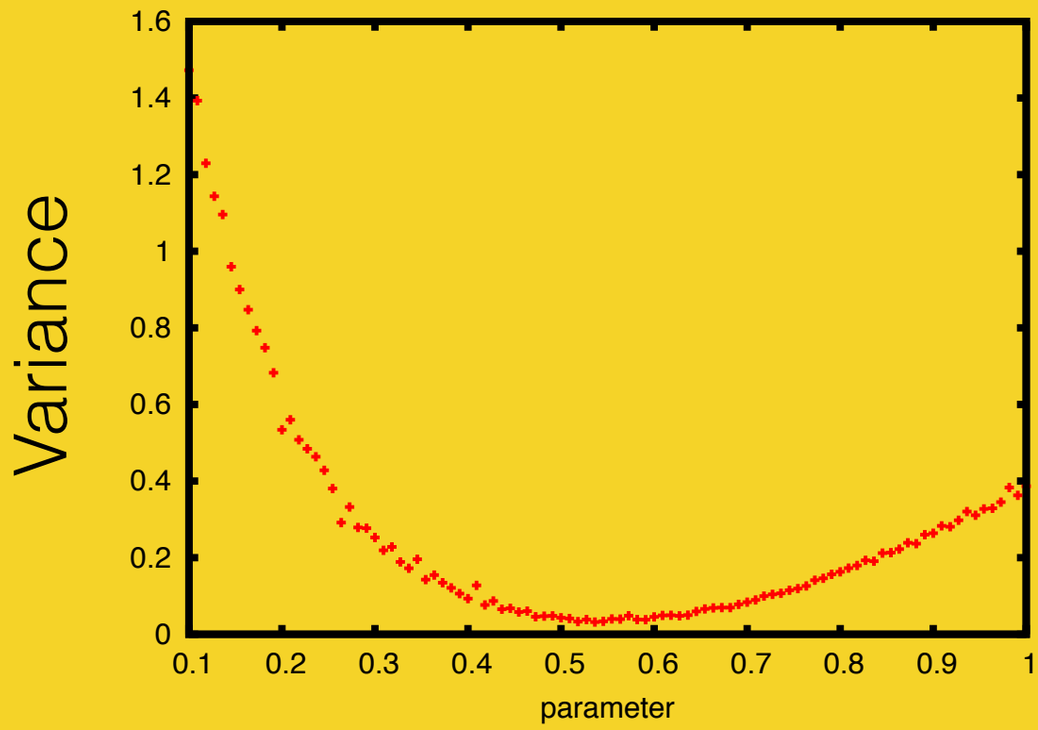
C=1





C=3





Correlated wave function Helium (Homework 6)

Very often used, simple generalization of Slater determinant:

$$\Psi(\mathbf{R}) = \det[\psi_i(\mathbf{r}_j)] \det[\psi_k(\mathbf{r}_l)] \prod J(r_{ij}) \ ,$$

where J is the Jastrow pair-correlation factor.

Has explicit dependence on the inter-electron distances, captures correlation.

Usually, both ψ and J have variational parameters.

Example: **Helium atom**

$$H = -\frac{1}{2}\sum_{i=1}^{2}(\nabla_i^2 + \frac{4}{r_i}) + \frac{1}{r_{12}} \ , \quad \psi = \exp(-\alpha r_1)\exp(-\alpha r_2)(1+\beta r_{12})$$

Trial wave function

Homework 6

Variational (Quantum) Monte Carlo

This week's problem is to calculate variationally the interacting total energy of a He atom (spin singlet, $S = 0$), using Variational Monte Carlo (VMC).
The related Fortran program for interacting bosons in harmonic confinement in 2D (`HO_2D.f90`), that was discussed during the lecture on Tuesday, should be useful for solving this homework.

1. Modify the example program so that it works in 3D (this is trivial), uses the correct nuclear potential instead of the harmonic one, and has a more appropriate trial wave function of the form

$$\Psi(r_1, r_2) = \exp(-\alpha r_1) \exp(-\alpha r_2) \exp(r_{12}/(1 + \beta r_{12})) \tag{1}$$

   Hint: Now you have 2 variational parameters instead of 1. It might be easiest from the point of view of Problem 2 to put both of these in a single array.

   Calculate the variational energy and variance of the local energy using parameter values $\alpha = 2$ and $\beta = 0.5$. (3 p)

2. Modify the VMC parameter optimization routine so that it can handle more parameters than just 1. Please also try switching the Jastrow pair-correlation factor to $(1 + \beta r_{12})$. How low can you go in energy and variance, what is the Jastrow and its parameter values then? What does the variance tell you about the trial wave function? (2 p)

   A side note: In principle you can experiment freely with the trial wave function to try to optimize it further as long as you conserve the symmetry/antisymmetry properties relevant for your system. However, in the case of the Coulomb interaction there is the problem that the potential energy diverges at close distances. The wave function (and thereby the kinetic energy) can be corrected to cancel this singularity by taking into account the so-called "Kato's cusp conditions". Both of the above Jastrow forms obey these. If you see instabilities / divergencies in your experiments you are probably violating them.

Helium atom (atomic units used)

$$\exp(-2r_1)\exp(-2r_2)$$

```
Energy is   -2.750165045570168
 +/-   9.800667894136265 8E-003
variance     1.148828847904341 4
```

$$\exp(-\alpha r_1)\exp(-\alpha r_2)$$

```
Energy is   -2.859094712349666
 +/-   5.219723617465824 9E-003
variance    0.921309854117879 47
```

Compare with the best fermion mean-field (Hartree-Fock) energy -2.862 a.u.

Taking the electron-electron coordinate to the wave function,
captures “correlation” of electrons:

$$\exp(-\alpha r_1)\exp(-\alpha r_2)\exp(r_{12}/(1+\beta r_{12}))$$

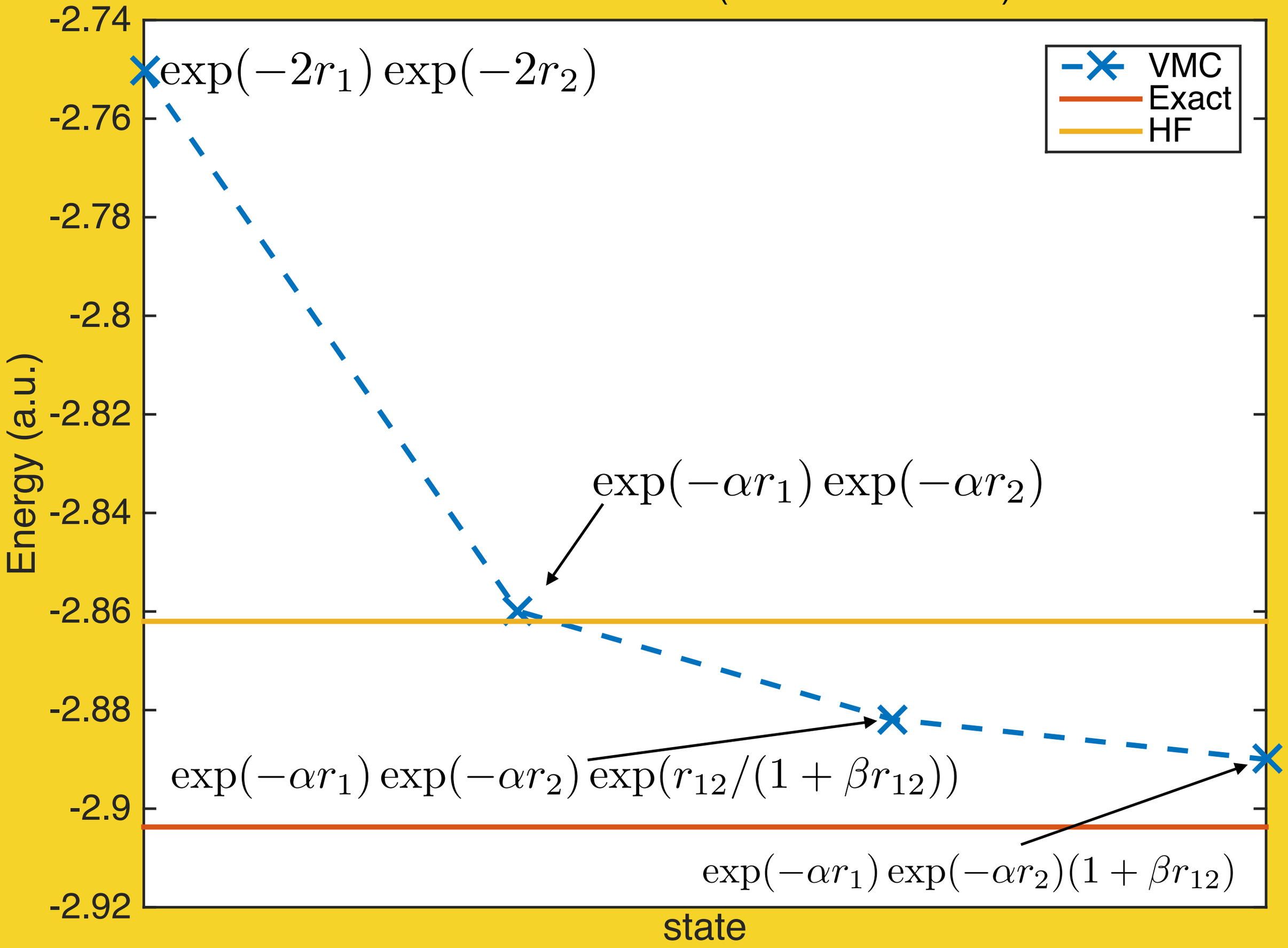
```
Energy is   -2.881872211742003 9
 +/-   4.758238774659062 3E-003
variance    0.313146948839314 38
```

$$\exp(-\alpha r_1)\exp(-\alpha r_2)(1+\beta r_{12})$$

```
Energy is   -2.890006006984346 1
 +/-   2.890313721384115 5E-003
variance    0.189102691907800 4
```

“Exact” energy -2.90372 a.u.

Helium atom (atomic units used)



Project on this direction?

This could be continued on an Project, for example on

1) Mean-field calculation for N bosons in a trap

2) Quantum Monte Carlo for some many-body problem

Reminder: The Project

Small individual project work, ***~40 hours of work***

Computational physics problem coded and analysed.
Small report returned to lecturers
Mandatory part of the course, 30% weight in assessment

Around 6 pages pdf, structured like

- Background
- Theory and method
- Results from simulations
- Conclusions and further studies

Deadline in May, but a title and a brief introduction of the project have to be submitted and approved by around late March!
If you want to start earlier, just contact first one of us lecturers!

Physical limitations on single processor speed forces to seek other options for faster calculations. We need:

Parallel computing

Using more than one processing unit (CPU) to solve a computational problem.

Not all problems can be solved efficiently in this way.

But for problems that this works, we get:

A. Reduction of the run time (at best, time gets divided by the number of processors)

B. We can solve bigger problems

Even all modern processors are “multicore”

Top supercomputers at top500.org

Summit - IBM
DOE/SC/Oak Ridge National Laboratory
United States
No. 1 system since June 2018

Finland (CSC) currently and in the near future

- **Sisu (Cray):** 1688 TFlop/s, 37th on Top500 in Nov. 2014
- **New environment under construction in 2019-20**:

  2.0 + 6.4 Petaflops

Summit (ORNL): Specifications and Features

- **Processor:** IBM POWER9™ (2/node)
- **GPUs:** 27,648 NVIDIA Volta V100s (6/node)
- **Nodes:** 4,608
- **Measured cores (source: Top500):** 2,397,824
- **Node Performance:** 42T
- **Theoretical Peak (source: Top500):** 200,795 TFlop/s
- **Memory/node:** 512GB DDR4 + 96GB HBM2
- **NV Memory/node:** 1600GB
- **Total System Memory:** >10PB DDR4 + HBM + Non-volatile
- **Interconnect Topology:** Mellanox EDR 100G InfiniBand, Non-blocking Fat Tree
- **Peak Power Consumption:** 13MW

(Calculate how much would the electricity cost for you)

CSC'S NEW SYSTEM (BY 2020)

Phase 1 (summer 2019)

- Supercomputer partition
- Theoretical peak performance of 2.0 Petaflops
- Latest generation Intel Xeon processors, code name Cascade Lake, 30480 cores
- Compute nodes have a mix of memory sizes, ranging from 96 GB up to 1.5 TB
- 4 PB Lustre parallel storage system by DDN
- Interconnect network HDR InfiniBand 200 Gbps by Mellanox, nodes connected with 100Gbps HDR100 links

Artificial intelligence partition

- The total peak performance of the AI system is 2.5 Petaflops from the GPUs
- Comprises 80 nodes with 4 Nvidia V100 GPUs + 2 CPUs each (320 GPUs)
- Each node has 3.2 TB of fast local storage
- Dual rail HDR100 interconnect network connectivity providing 200Gbps aggregate bandwidth
- This partition is engineered to allow GPU intensive workloads to scale well to multiple nodes

Data management solution

- In the first phase CSC will also offer a new common data management solution for both new and old infrastructure. This system is based on CEPH object storage technology and it provides 12 PB of storage capacity and will be the backbone for providing a rich environment for storing, sharing and analyzing data across the CSC compute infrastructure.

Phase 2 (early 2020)

- Atos BullSequana XH2000 supercomputer with 6.4 Petaflops of theoretical peak performance
- AMD EPYC processors, code name Rome, 200 000 cores
- Each node will be equipped with 256 GB of memory
- 8 PB Lustre parallel storage system
- Interconnect network HDR InfiniBand by Mellanox

€33 million investment altogether

Flynn's taxonomy

Classify the possible cases we can have

S I S D **Single Instruction Stream** **Single Data Stream**	**S I M D** **Single Instruction Stream** **Multiple Data Stream**
M I S D **Multiple Instruction Stream** **Single Data Stream**	**M I M D** **Multiple Instruction Stream** **Multiple Data Stream**

"One CPU simulation"

"GPU"

"Not relevant here"

"Supercomputer"

One CPU is serial and both on the right are parallel

Speedup=

Wall-clock time of serial program execution

divided by

Wall-clock time of parallel program execution

This indicates how much you benefit from parallel computing in terms of real time.

It depends in a non-trivial way on the architecture, number of parallel processors, method used etc.

Example: N independent tasks, N processors, ideal world, speedup is N. (Normally called “embarrassingly parallel”)

Amdahl's law

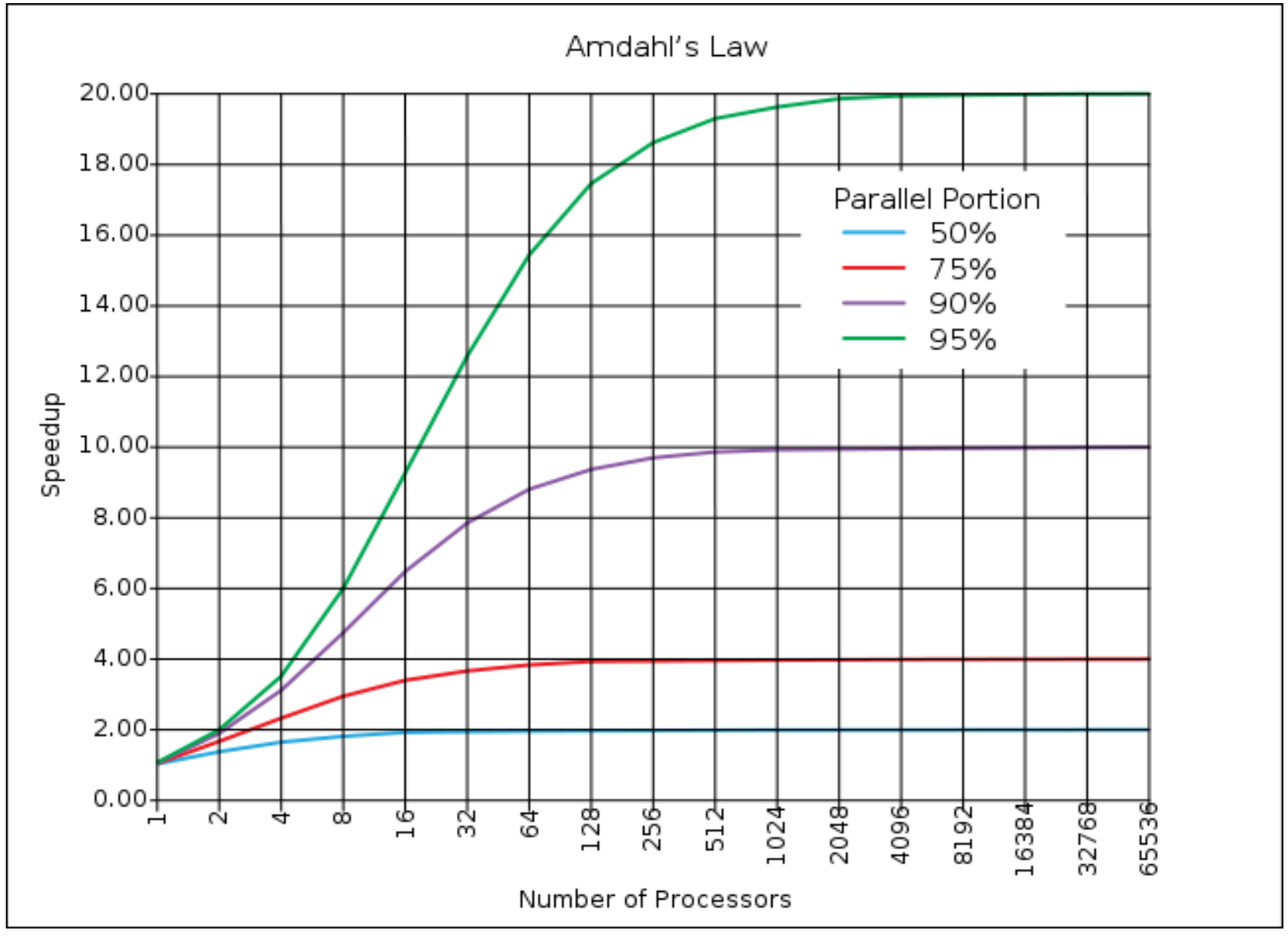
Scalability limits

$$speedup = \frac{1}{\frac{P}{N} + S}$$

P: parallel fraction

S: serial fraction

N: number of CPU cores

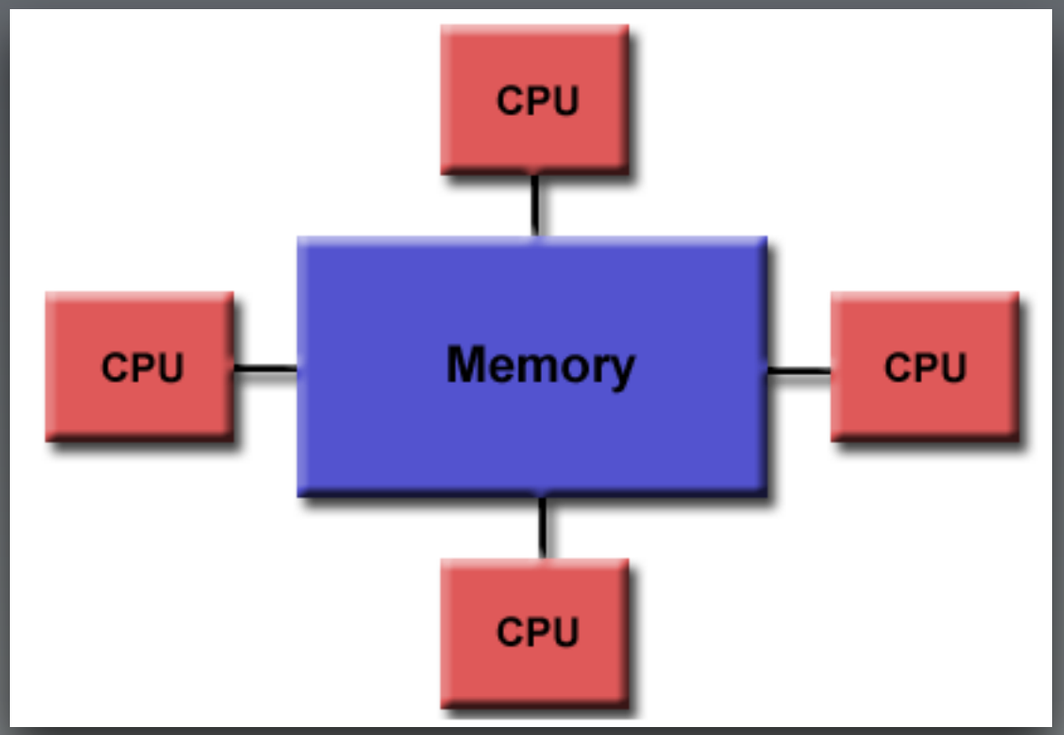


If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

If 90% of the code can be parallelized, maximum speedup = 10, but needs around 1000 processors

It might not make sense to waste resources if scalability is not good.

Shared memory architecture



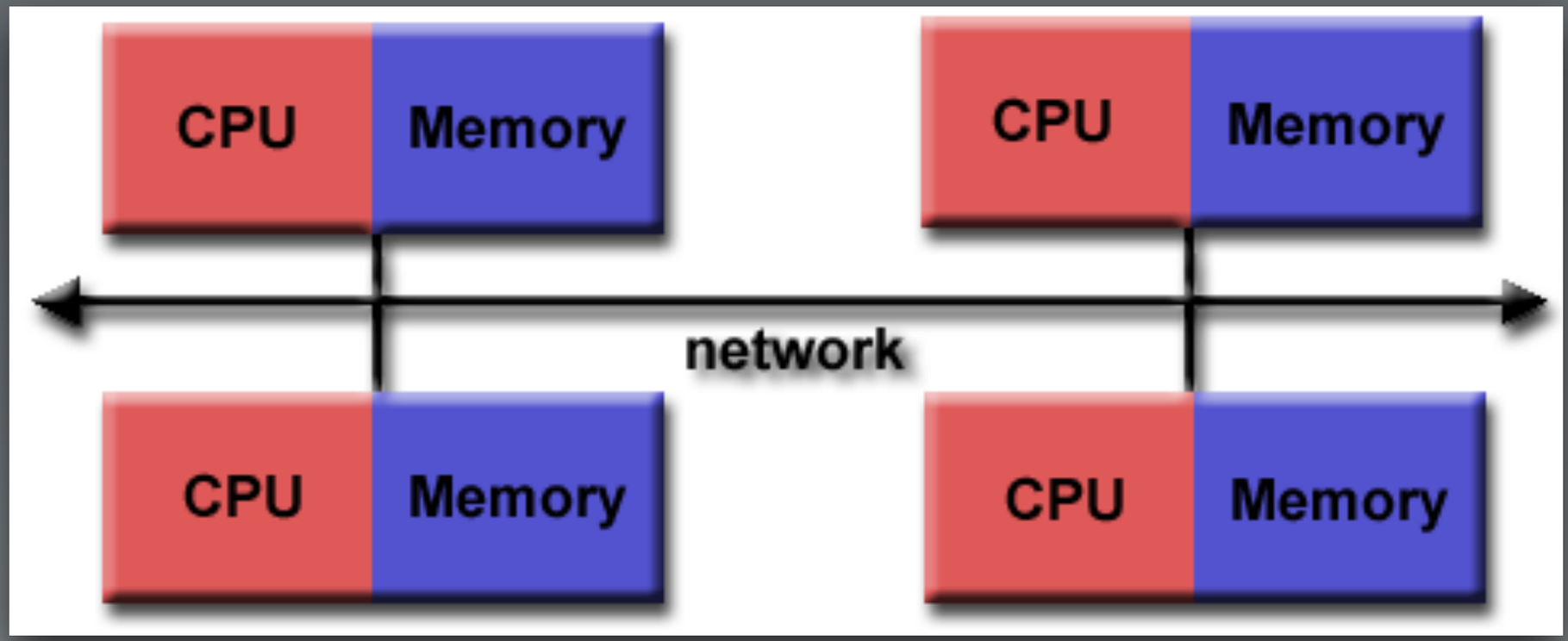
CPUs share same memory.

Easy for programmer as all memory easily available (e.g. OpenMP)

Not practical for huge numbers of CPUs.

Not fully relevant for modern supercomputers.

Distributed memory architecture



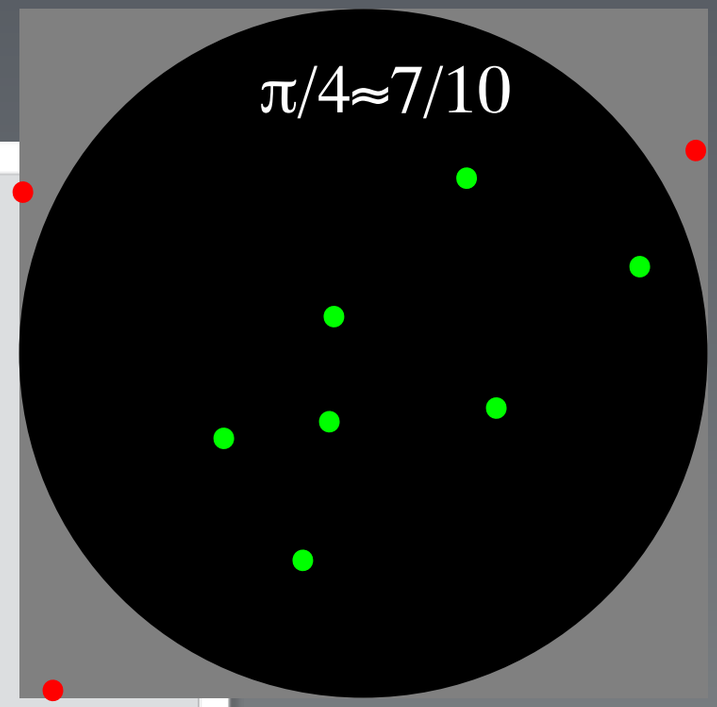
CPUs have their own memory.

Hard for programmer as some data not easily available (e.g. MPI).

One needs to communicate between CPUs to share data.

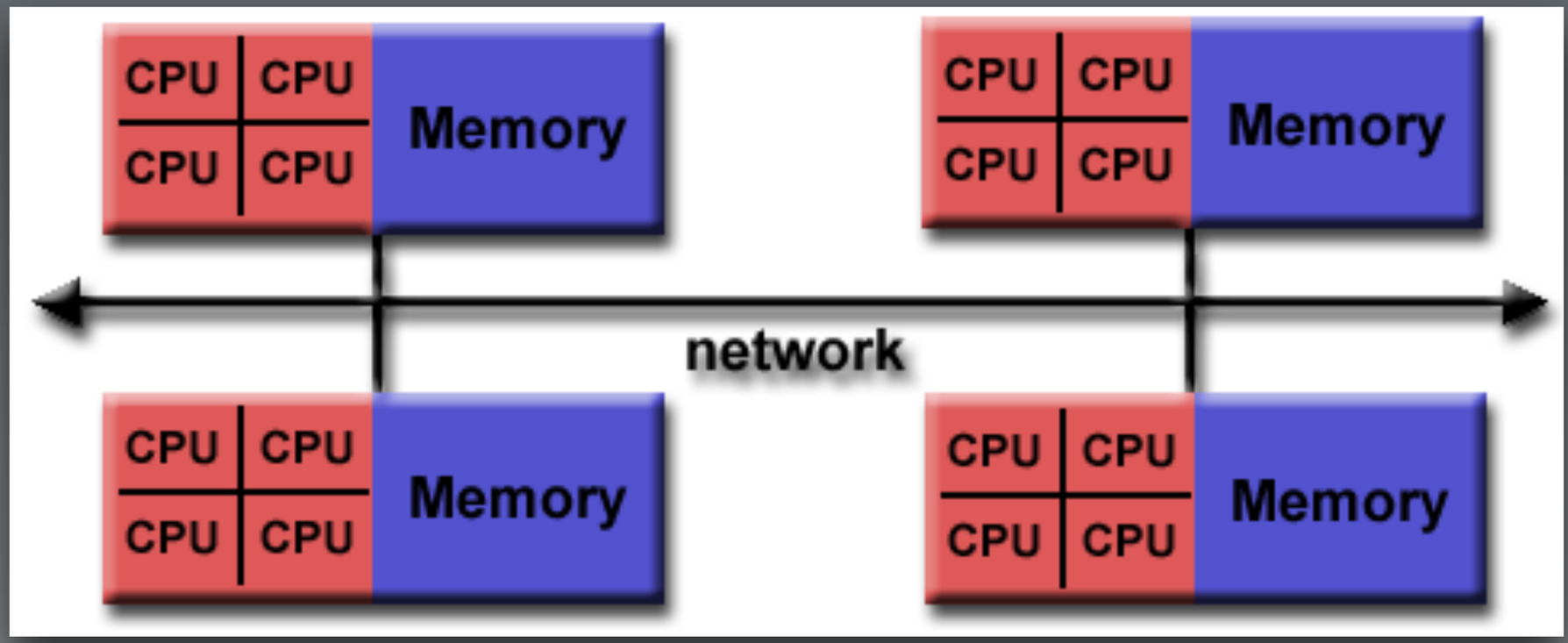
Very easily scalable for huge supercomputers.

“Pseudo” Example




```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER

  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)

else if I am WORKER

  send to MASTER circle_count

endif
```

Hybrid memory architecture



Shared memory for groups of CPUs.

Distributed memory between CPU groups.

Very easily scalable, and this is what supercomputers have.

Hard for the programmer, as some data fast and rest slow.

Matlab

Parallel Computing Toolbox

Perform parallel computations on multicore computers, GPUs, and computer clusters

Parallel Computing Toolbox™ lets you solve computationally and data-intensive problems using multicore processors, GPUs, and computer clusters. High-level constructs—parallel for-loops, special array types, and parallelized numerical algorithms—let you parallelize MATLAB® applications without CUDA or MPI programming. You can use the toolbox with Simulink® to run multiple simulations of a model in parallel.

The toolbox lets you use the full processing power of multicore desktops by executing applications on workers (MATLAB computational engines) that run locally. Without changing the code, you can run the same applications on a computer cluster or a grid computing service (using MATLAB Distributed Computing Server™). You can run parallel applications interactively or in batch.

Parallel for-Loops (parfor)

Run loop iterations in parallel on a parallel pool using the `parfor` language construct

Batch Processing

Offload execution of a function or script to run in a cluster or desktop background

GPU Computing

Transfer data between MATLAB and a graphics processing unit (GPU); run code on a GPU

Distributed Arrays and SPMD

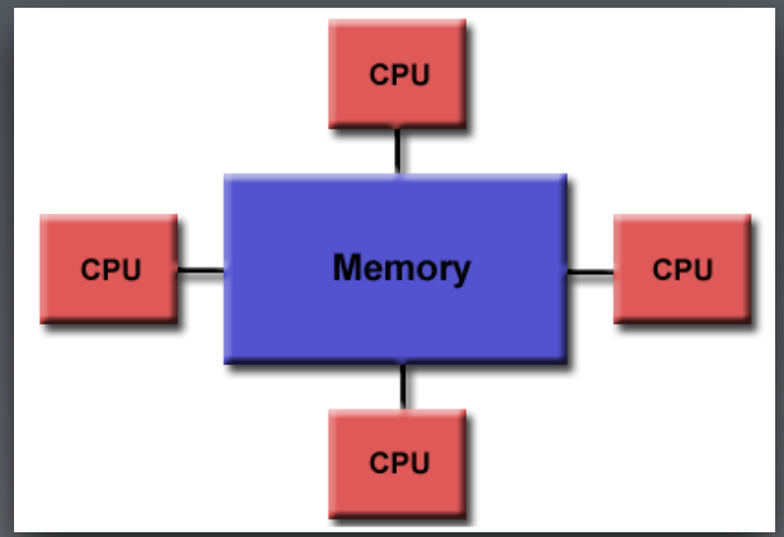
Partition arrays across multiple MATLAB workers for data-parallel computing and simultaneous execution

Big Data

Accelerate mapreduce and datastore programs by running on a parallel pool or Hadoop® cluster

This you can study from the documentation.

OpenMP



For shared memory case, easy parallelisation.

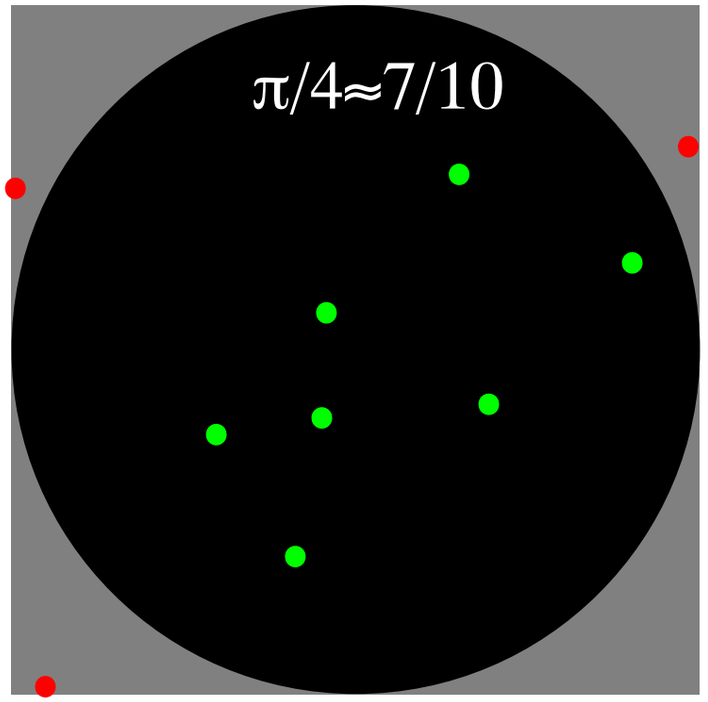
Example: use all cores of your modern desktop/laptop machine.

A Fortran example:

```
program hello90
use omp_lib
integer:: id, nthreads
  !$omp parallel private(id)
  id = omp_get_thread_num()
  write (*,*) 'Hello World from thread', id
  !$omp barrier
  if ( id == 0 ) then
    nthreads = omp_get_num_threads()
    write (*,*) 'There are', nthreads, 'threads'
  end if
  !$omp end parallel
end program
```

Second example:

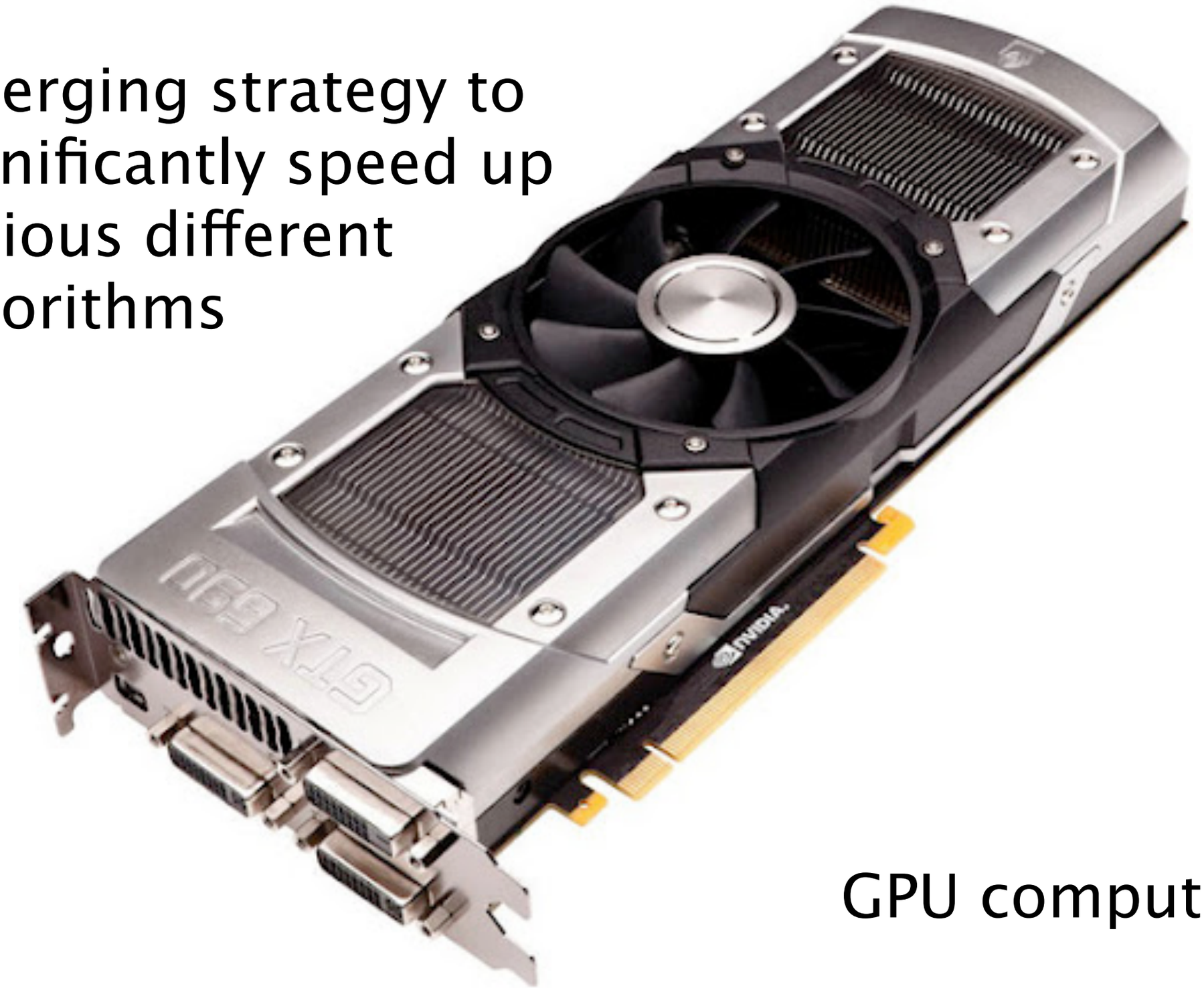
```
program pie
use omp_lib
integer:: id, nthreads
integer*8 :: count, N=10**3
real :: x, y
call random_seed()
!$omp parallel private(id,x,y)
nthreads = omp_get_num_threads()
id = omp_get_thread_num()
call random_number(x)
write (*,*) x, ' is random at thread', id
count=0
!$omp parallel do reduction(+:count)
do i = 1, N*nthreads
   call random_number(x)
   call random_number(y)
   if (x**2+y**2<1.0) then
      count=count+1
   end if
enddo
write (*,*) 'Hello World from thread', id
if ( id == 0 ) then
   write (*,*) 'There are', count, 'inside', real(count)/real(nthreads*N)
   write(*,*) ' vs ', atan(1.d0)
end if
!$omp end parallel
end program
```



You should not expect perfect speedups from omp!

Computing on graphics processor

Emerging strategy to significantly speed up various different algorithms



GPU computing

Computing on graphics processor

Why faster?

CPU

GPU

Computing on graphics processor

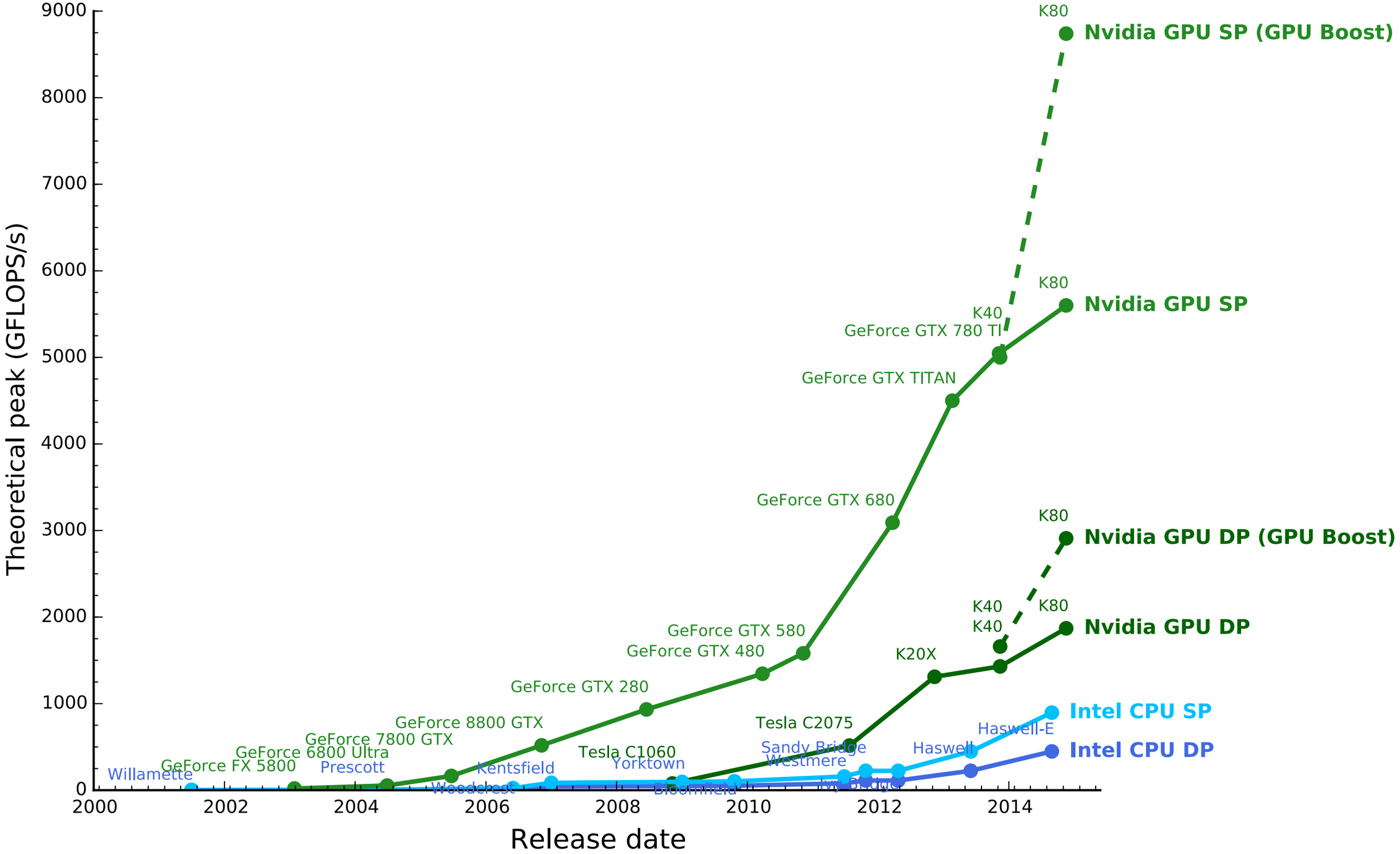
Why faster?

CPU

GPU

Task: transport boxes

Theoretical performance of CPUs and GPUs



CPU VS GPU

- In a nutshell, the CPU typically runs one or a few very fast computational threads, while the GPU runs on the order of ten thousand simultaneous (not so fast) threads

- The hardware in the GPU emphasizes fast data processing over cache and flow control

- To benefit from the GPU, the computation has to be **data-parallel** and **arithmetically intensive**
  - data-parallel: the same instructions are executed in parallel to a large number of independent data elements
  - arithmetic intensity: the ratio of arithmetic operations to memory operations

PROGRAMMING FOR GPUS

Option 1. Writing GPU kernels by hand (typically using C/C++):

- CUDA by Nvidia: Used a lot in academia but restricted to Nvidia hardware.
- OpenCL: Not restricted to Nvidia, code is more portable
- PyCUDA/PyOpenCL: enable you access CUDA/OpenCL APIs from Python.

Option 2. Simplified parallel programming of heterogenous CPU/GPU systems:

- OpenACC. The programmer can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using compiler directives and additional functions (cf. OpenMP). Compiler decides what to do on the GPU and when and how to transfer data. Currently, only a few compilers support OpenACC.

CUDA

- Nvidia's programming model for their GPUs
- Ordinary C/C++ code with special functions that run on the GPU
- **Pros:**
  - Best performance (if you know what to do)
  - Good documentation, a lot of resources online etc.
- **Cons:**
  - Bigger coding effort
  - To get the best performance, some low level hardware knowledge required

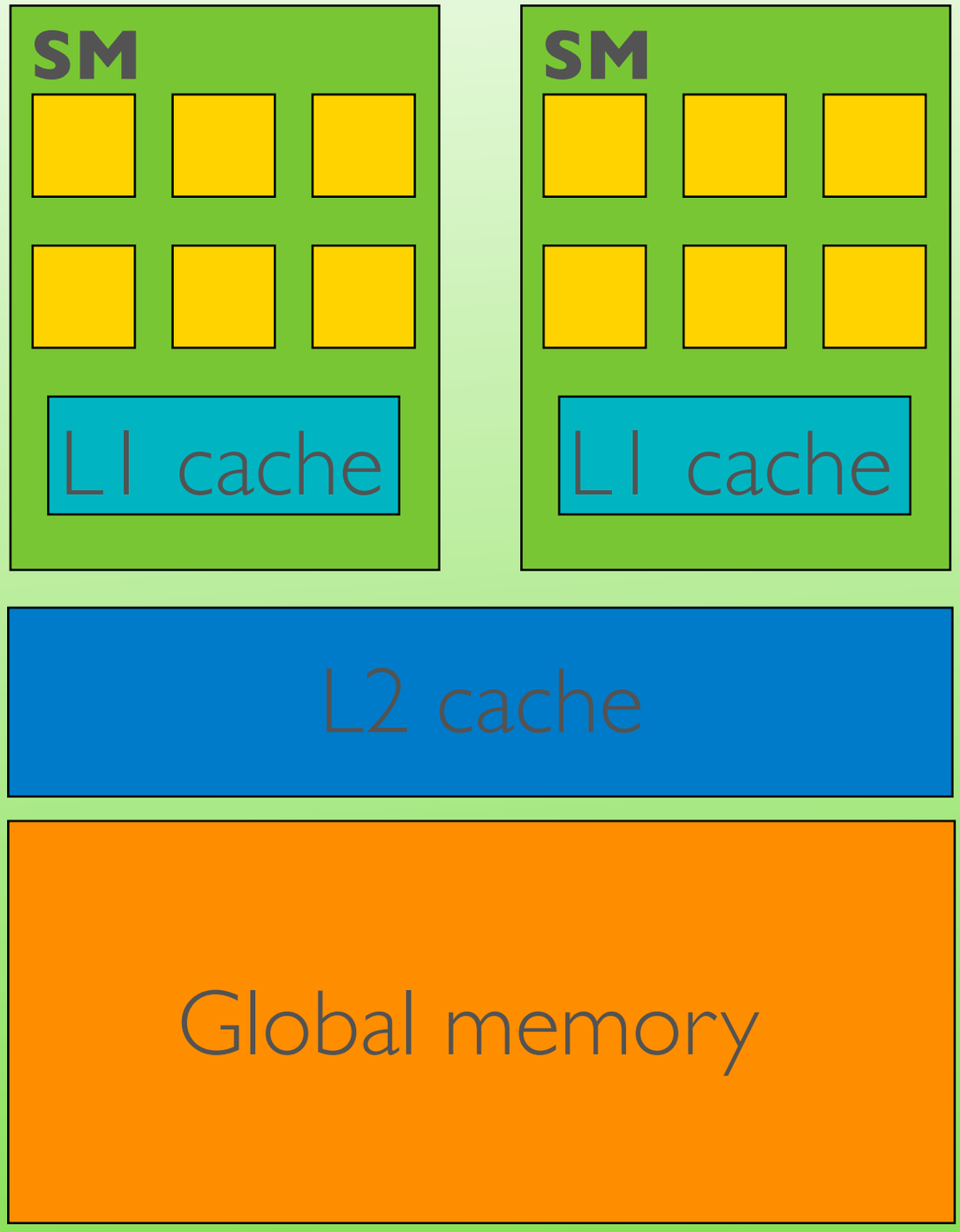
CUDA

- The major difference compared to ordinary C are special functions, called **kernels**, that are executed on the GPU by many threads in parallel

- Each thread is assigned an ID number that allows different threads to operate on different data

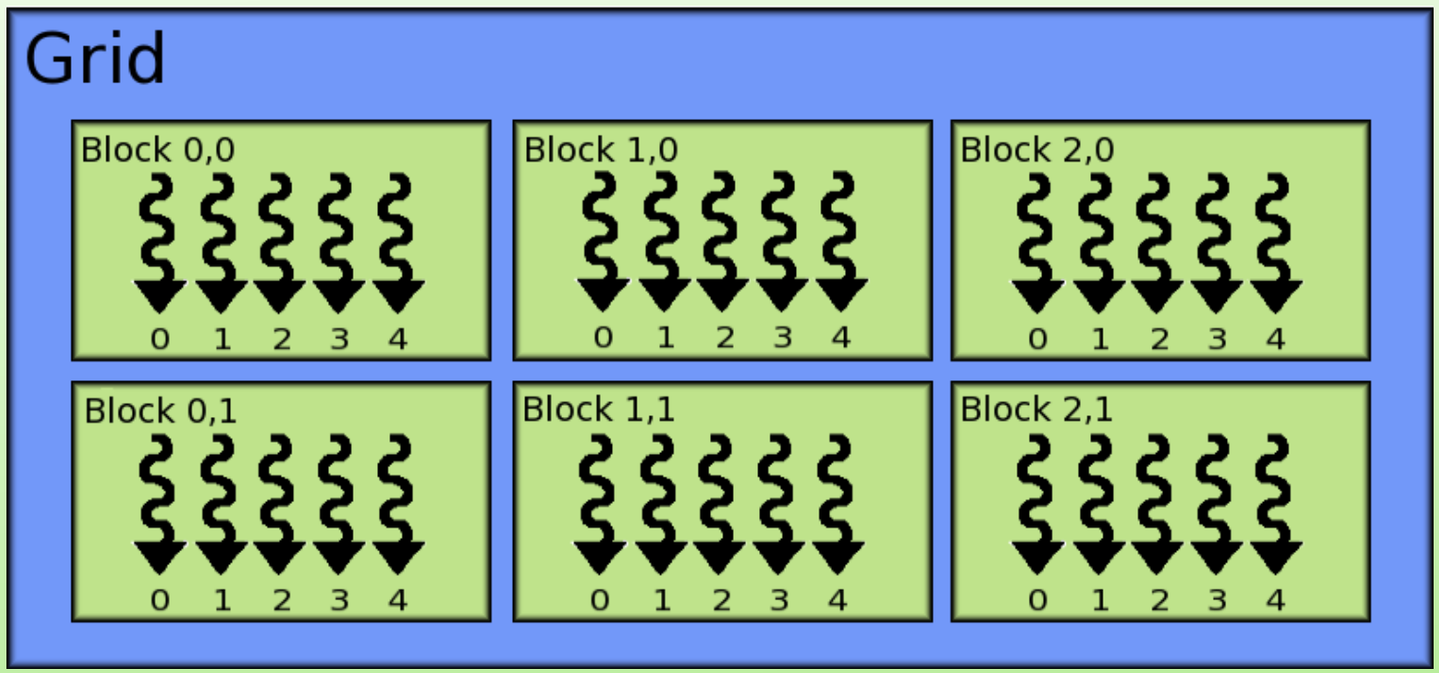
- The typical structure of a CUDA program:
  - 1. Prepare data
  - 2. Allocate memory on the GPU
  - 3. Transfer data to GPU
  - 4. Perform the computation on the GPU
  - 5. Transfer results back to the host side

GPU HARDWARE



- The GPU consists of multiple streaming multiprocessors (SM) that each have tens or hundreds of cores
- Each SM has an L1 cache / shared memory
- There is also an L2 cache shared by all SMs

THREAD HIERARCHY



Thread hierarchy in CUDA

- The **threads** are organized into **blocks** that form a **grid**
- Threads within a block are executed on the same multiprocessor and they can be communicate via the shared memory
- Threads in the same block can be synchronized but different blocks are completely independent

CUDA

```
__global__ void Vcopy(int* A, int* B, int N)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if (id < N)
        B[id] = A[id];
}

int main()
{
    //Host code
    ...

    Vcopy<<<numBlocks, numThreadsPerBlock>>>(A, B, N);
}
```

- Vector copy kernel, from A to B.
- Each thread calculates its global ID number and copies its data at the same time.
- All threads do the same task but for different data (SIMD in Flynn's taxonomy).

ALLOCATION AND TRANSFER

- The GPU memory is separate from the host memory that the CPU sees
- Before any kernels can be run, the necessary data has to transferred to the GPU memory via the PCIe bus
- Memory on the GPU can be allocated with **cudaMalloc**
- Memory transfers between GPU and the host with **cudaMemcpy**
- The transfer of the data is rather slow, compared to normal computer memory operations.
- One needs reasonably large tasks to be done in GPU to overcome the data transfer time.
- Best cases are such that very little data transfer are done but a lot of calculations on GPU

LIBRARIES

- Included in CUDA are many useful libraries, for example
  - **cuFFT** - Fast Fourier transforms
  - **CUBLAS** - CUDA-accelerated linear algebra
  - **cuRAND** - Random number generation
  - **Thrust** - High level interface for many parallel algorithms

- With these, you can use the computational power of the GPUs for many basic tasks without having to write any kernels yourself

THRUST

- Library of parallel algorithms and data structures

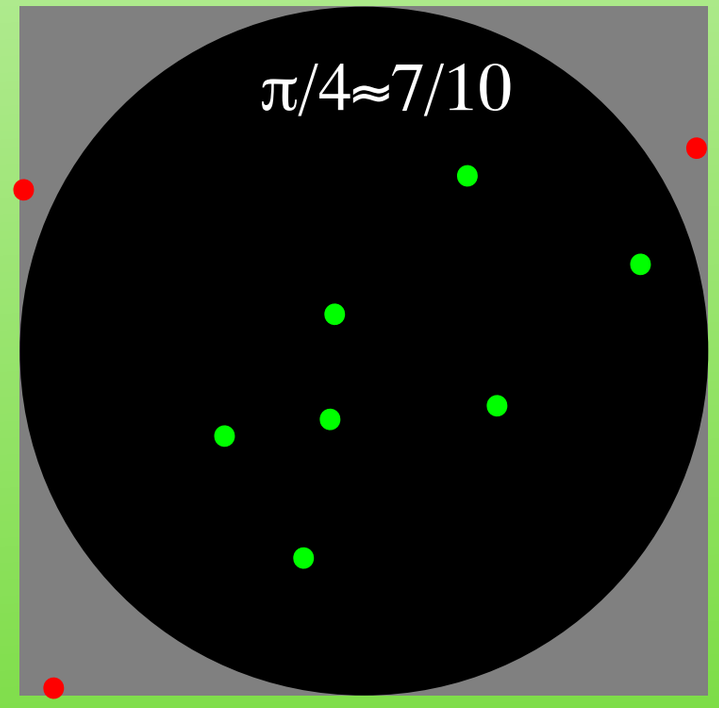
- Much simpler than coding cuda yourself, example:

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per second on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

CURAND

- A library for CUDA-accelerated random number generation
- You can generate random numbers in bulk from the host code or generate individual numbers inside kernels
- Includes many different generation algorithms and distributions

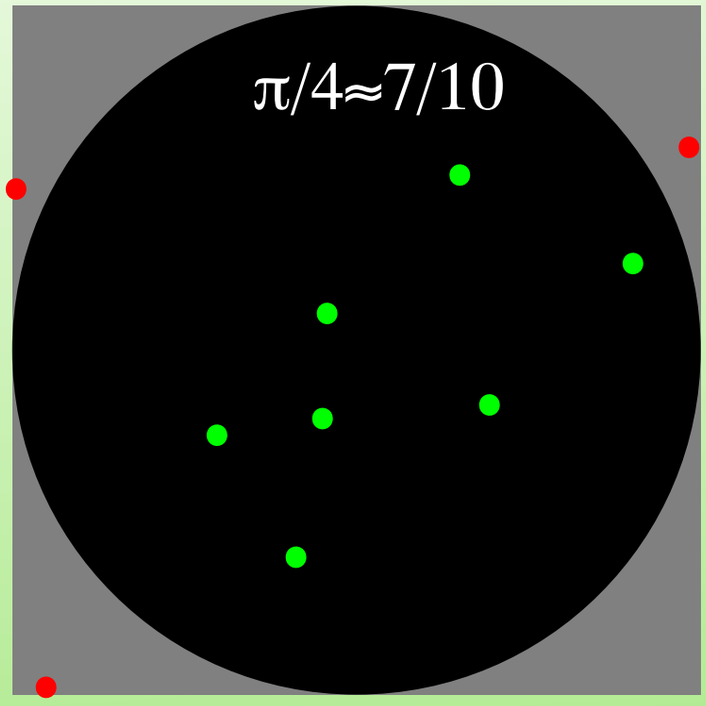
Example of this and thrust and cuda, once more:



CURAND AND THRUST

```
/* GPU kernel */
__global__ void piper4(curandState *state, long long *Res)
{
  long long id = threadIdx.x + blockIdx.x * BLOCKSIZE;
  long long count = 0;
  double x;
  double y;

  /* Copy RNG state to local memory for efficiency */
  curandState localState = state[id];

  /* Np points */
  for(int n = 0; n < Np; n++)
   {
     x = curand_uniform_double(&localState);
     y = curand_uniform_double(&localState);
     if((x*x + y*y) < 1)
      count++;
   }

  /* Copy RNG state back to global memory */
  state[id] = localState;
  /* Store results */
  Res[id] = count;
}
```

Parts of a real code




```
piper4<<<(N/BLOCKSIZE), BLOCKSIZE>>>(devStates, devRes);

devcount = thrust::reduce(dev_ptr, dev_ptr + N);
```

```
GPU approximation for pi with 1048576000 points: 3.14163159179687, error 3.89382070817845e-05
GPU approximation for pi with 1048576000 points: 3.14154328536987, error -4.93682199200762e-05
GPU approximation for pi with 1048576000 points: 3.1415571937561, error -3.54598336897993e-05
Average error is 2.490075481969e-05,
 dividing with the expected error gives 0.491664238439872
GPU took 0.0405897349119186 s each
CPU took 35.0880470275879 s
CPU approximation for pi with 1048576000 points: 3.14171020126343, error 0.00011754767363481
Speedup factor: 864.456176757812
```