Computational finance on GPUs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford-Man Institute for Quantitative Finance

Oxford University Mathematical Institute

SIAM Conference on Parallel Processing for Scientific Computing 2010

Acknowledgements: Tom Bradley (NVIDIA), Robert Tong, Jacques du Toit (NAG)

and financial support from CRL, EPSRC, NVIDIA

Outline

- computational finance
- scientific computing on GPUs
- random number generation
- finite difference applications
- current developments

Computational Finance

Might seem a bad time to be in this business, but not for an academic:

- clear need for better models
- regulators (and internal risk management) are demanding more simulation
- computational finance accounts for 10% of Top500 supercomputers
  - primarily throughout computing, huge number of small tasks
- still plenty of students as the job situation recovers
- my own research is on faster simulation
  - better mathematics (improved Monte Carlo method)
  - better computing (use of GPUs)

Computational Finance

What does “computational finance” cover?

- options pricing – investment banks
  - Monte Carlo methods (60%)
  - PDEs / finite difference methods (30%)
  - other semi-analytic methods (10%)
- high-frequency algorithmic trading – hedge funds
- actuarial science – pension funds, insurance industry

Options pricing is the big one in terms of number / size of computers used.

Computational Finance

What is an “option”?

Suppose you run an airline, and you’re concerned the price of aviation fuel may rise in a year’s time – what do you do?

- might try to fix the price now (futures contract)
  but that might lock in a very high price
- alternatively, can hedge the risk by buying an option
  (effectively as a form of insurance)

A call option gives the airline the right (but not an obligation) to buy the fuel at a fixed price K.

Option value

Suppose the fuel price is S in a year's time when the option matures:

- if $S > K$, it is worthwhile to exercise the option, and the value is $S - K$ per unit quantity
- if $S < K$, the option is worthless

Hence, the payoff value is $P = \max(0, S - K)$

To assess the current value of this option we need to model the behavior of the fuel price, but key is we cannot predict it with precision – the model is stochastic to reflect the uncertainty / unpredictability of future events.

SDEs in Finance

In computational finance, stochastic differential equations are used to model the behaviour of

- stocks
- interest rates
- exchange rates
- electricity/gas demand
- crude oil prices
- …

The stochastic term accounts for the uncertainty of unpredictable day-to-day events.

SDEs in Finance

Examples:

- Geometric Brownian motion (Black-Scholes model for stock prices)

$$\mathrm{d}S = r\,S\,\mathrm{d}t + \sigma\,S\,\mathrm{d}W$$

- Cox-Ingersoll-Ross model (interest rates)

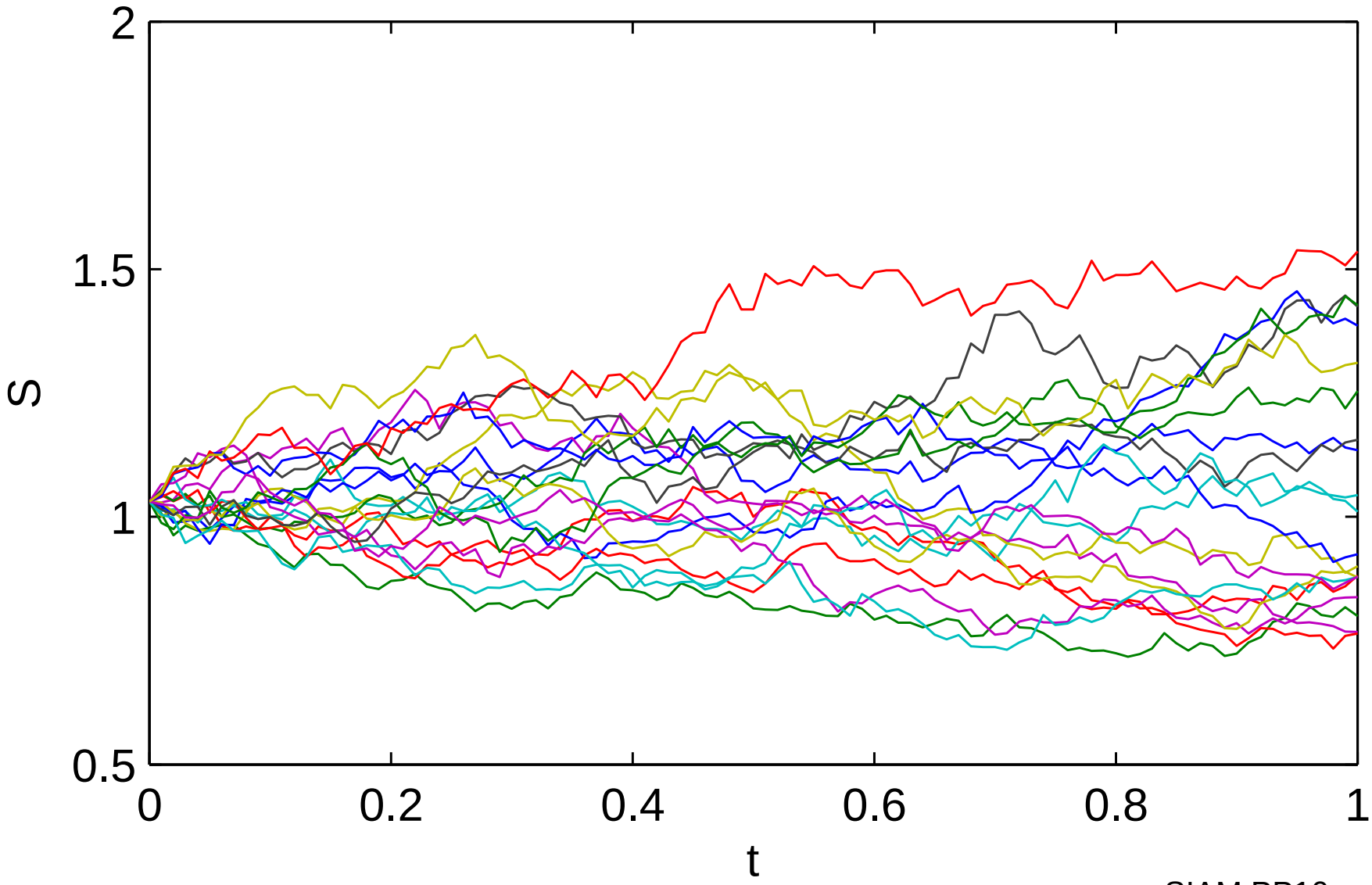
$$\mathrm{d}r = \alpha(b - r)\,\mathrm{d}t + \sigma\,\sqrt{r}\,\mathrm{d}W$$

$W(t)$ is a Wiener variable: for any $q<r<s<t$, $W(t)-W(s)$ is Normally distributed with mean 0 and variance $t-s$, independent of $W(r)-W(q)$.

(More complex models have multiple SDEs with different correlated Wiener inputs)

SDEs in Finance

Geometric Brownian motion: $r = 0.05,\ \sigma = 0.2$



Monte Carlo simulation

For a general scalar SDE

$$\mathrm{d}S(t) = a(S,t)\,\mathrm{d}t + b(S,t)\,\mathrm{d}W(t)$$

an Euler-Maruyama discretisation with timestep h gives

$$\widehat{S}_{n+1} = \widehat{S}_n + a(\widehat{S}_n, t_n)\,h + b(\widehat{S}_n, t_n)\,\Delta W_n$$

where ΔW_n is a Normal random variable with mean 0 and variance h.

Simplest estimator for expected payoff $\mathbb{E}[P]$ is an average from N independent path simulations:

$$\widehat{Y} = N^{-1} \sum_{i=1}^{N} \widehat{P}^{(i)}$$

Monte Carlo simulation

It's trivially parallel – only issue is that each path needs an independent set of random numbers

Why is it so costly?

- the banks don't just need the price at the time of sale, they have to re-value all of their contracts every night to see if they are making or losing money
- they also need to assess the sensitivity of the value to changes in various parameters
- aim of the banks is to hold offsetting risks, so if one goes down, another goes up – the banks really don't want to gamble

PDE formulation

The Monte Carlo approach simulates multiple possible futures and averages over them

An alternative point of view derives a PDE for the option value $V(S,t)$ which depends on time t and the value of the underlying asset (e.g. fuel price) at that time

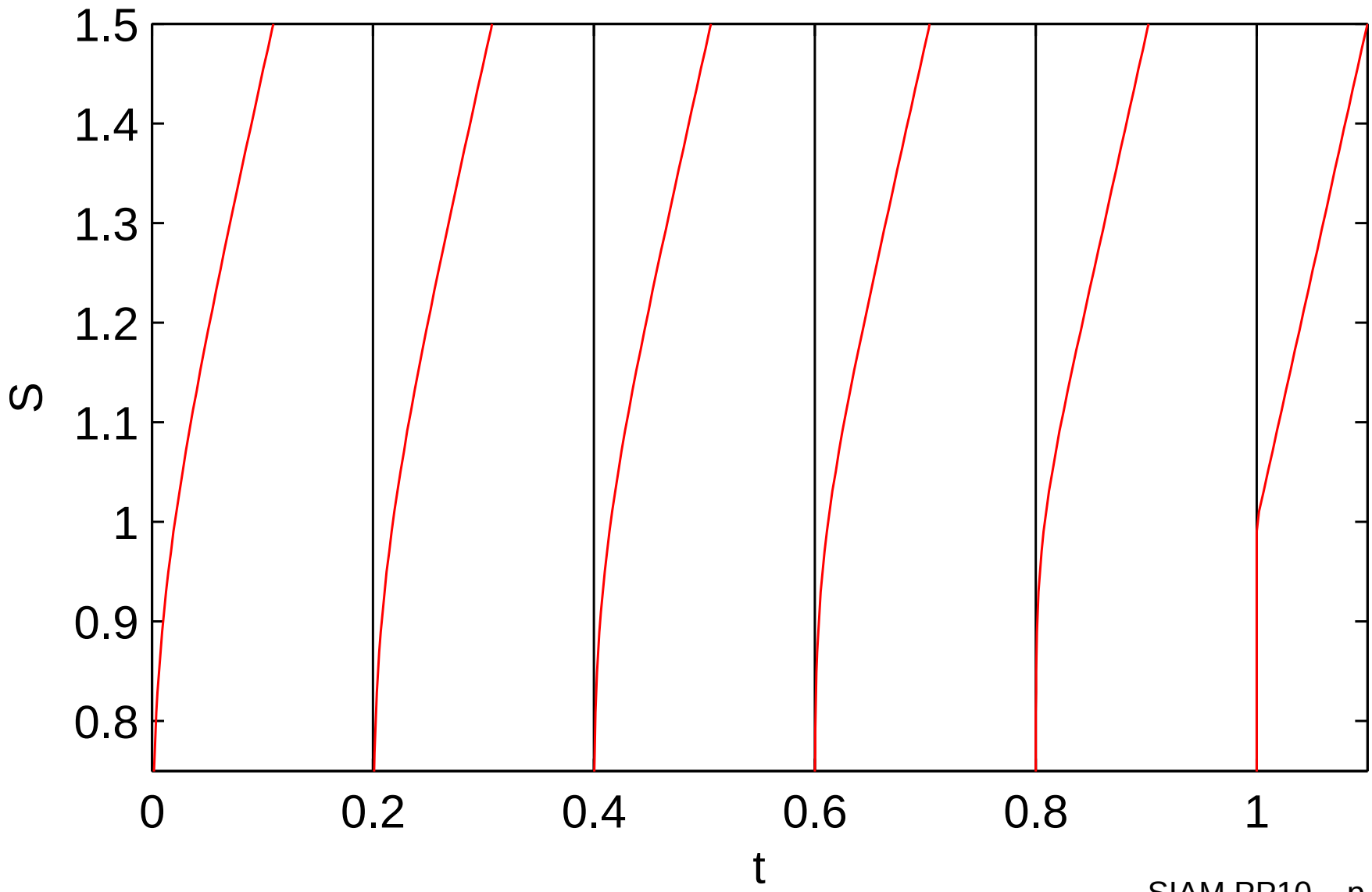
Black-Scholes PDE (Geometric Brownian Motion)

$$V_t + r\,S\,V_S + \tfrac{1}{2}\sigma^2 S^2 V_{SS} = r\,V$$

solved backwards in time from known payoff value at time T

PDE formulation

Black-Scholes call option value: $r = 0.05,\ \sigma = 0.2$



PDE formulation

"Curse of dimensionality"

- with more complex models with D SDEs, each one leads to an extra dimension in the PDE
- cost of Monte Carlo is linear in D;
  cost of PDE approach is exponential
- typically, PDE approach is used up to dimension $D = 3$
- example: exchange rate product with SDEs for local interest rate, foreign interest rate, and exchange rate

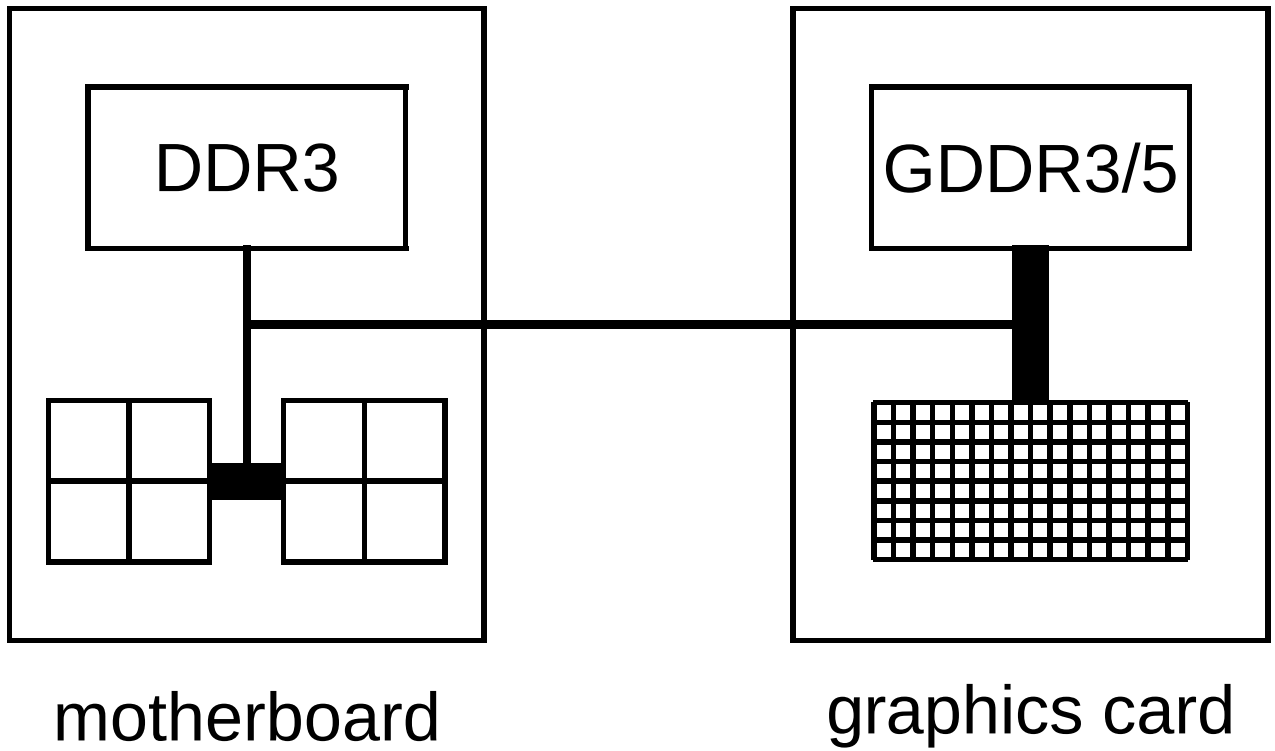
PDE formulation

Numerical methods:

- simple structured grids (up to 100 points in each dimension)
- simple central difference spatial approximations
- usually explicit or Crank-Nicolson time-marching (often ADI in higher dimensions)
- fairly straightforward OpenMP parallelisation on shared-memory systems

GPU hardware

Typically, a PCIe graphics card with a many-core GPU sits inside a PC/server with one or two multicore CPUs:



GPU hardware

- CPUs have up to 6 cores (each with a SSE vector unit) and 10-30 GB/s bandwidth to main system memory
- NVIDIA GPUs have up to 30×8 cores on a single chip and 100+ GB/s bandwidth to graphics memory
- offer 50–100$\times$ speedup relative to a single CPU core
- roughly 10$\times$ speedup relative to two quad-core Xeons
- also 10$\times$ improvement in price/performance and energy efficiency

How is this possible? Much simpler cores (SIMD units, no out-of-order execution or branch prediction) designed for vector computing, not general purpose

Emergence of GPUs

- AMD, IBM (Cell) and Intel (Larrabee) all producing or developing GPUs too
- NVIDIA has a good headstart on software side with CUDA environment
- new OpenCL software standard (based on CUDA and pushed by Apple) will probably run on all platforms
- driving applications are:
  - computer games “physics”
  - video (e.g. HD video decoding)
  - computational science
  - computational finance
  - oil and gas

Why GPUs will stay ahead?

Technical reasons:

- SIMD units means larger proportion of chip devoted to floating point computation (but CPUs will respond with longer vector units – AVX)
- tightly-coupled fast graphics memory means much higher bandwidth

Commercial reasons:

- CPUs driven by price-sensitive office/home computing; not clear these need vastly more speed
- CPU direction may be towards low cost, low power chips for mobile and embedded applications
- GPUs driven by high-end applications – prepared to pay a premium for high performance

Use in computational finance

- Bloomberg has a large cluster:
  - 48 NVIDIA Tesla units, each with 4 GPUs
  - alternative to buying 2000 CPUs
- BNP Paribas has a small cluster:
  - 2 NVIDIA Tesla units
  - replacing 250 dual-core CPUs
  - factor 10x savings in power (2kW vs. 25kW)
- lots of other banks doing proof-of-concept studies
  - my impression is that IT groups are keen, but quants are concerned about effort involved
- Several ISV's now offer software based on CUDA

Programming

Big breakthrough in GPU computing has been NVIDIA's development of CUDA programming environment

- C plus some extensions and some C++ features
- host code runs on CPU, CUDA code runs on GPU
- explicit movement of data across the PCIe connection
- very straightforward for Monte Carlo applications, once you have a random number generator
- significantly harder for finite difference applications (but will be much easier with next-generation GPU)

My experience

- Random number generation (mrg32k3a/Normal):
  - 2500M values/sec on GTX 280
  - 70M values/sec/core on Xeon using Intel's VSL
- LIBOR Monte Carlo testcase:
  - 100x speedup on GTX 280 compared to single thread on Xeon
- 3D PDE application:
  - factor 50x speedup on GTX 280 compared to single thread on Xeon
  - factor 10x speedup compared to two quad-core Xeons

GPU results are all single precision – double precision is currently 2-4 times slower, no more than factor 2 in future

Random number generation

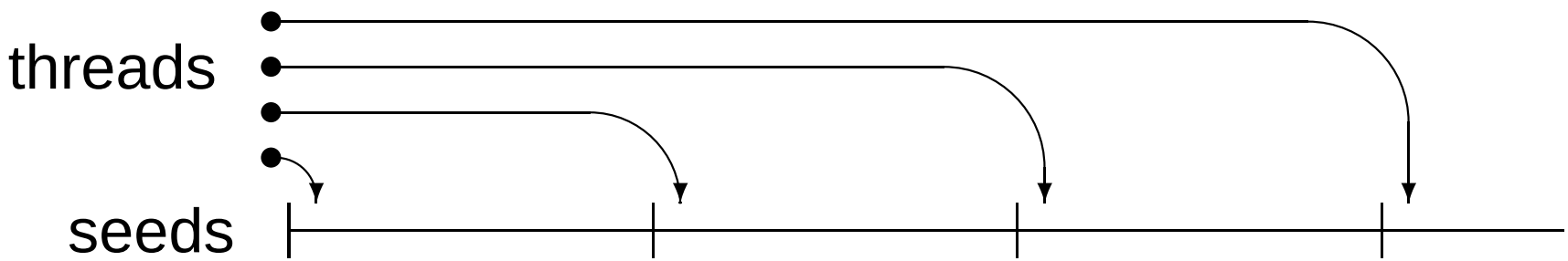
Main challenge for Monte Carlo simulation is parallel random number generation

- want to generate same random numbers as in sequential single-thread implementation
- two key steps:
  - generation of $[0, 1]$ uniform random number
  - conversion to other output distributions (e.g. unit Normal)
- many of these problems are already faced with multi-core CPUs and cluster computing
- NVIDIA does not provide a RNG library, so I have developed one with NAG

Random number generation

Key issue in uniform random number generation:

- when generating 10M random numbers, might have 5000 threads and want each one to compute 2000 random numbers
- need a “skip-ahead” capability so that thread n can jump to the start of its “block” efficiently (usually $\log N$ cost to jump N elements)



Random number generation

mrg32k3a (Pierre L'Ecuyer, '99, '02)

- popular generator in Intel MKL and ACML libraries
- pseudo-uniform $(0, 1)$ output is

$$(x_{n,1} - x_{n,2} \mod m_1) / m_1$$

where integers $x_{n,1}$, $x_{n,2}$ are defined by recurrences

$$\begin{aligned} x_{n,1} &= a_1\, x_{n-2,1} - b_1\, x_{n-3,1} \mod m_1 \\ x_{n,2} &= a_2\, x_{n-1,2} - b_2\, x_{n-3,2} \mod m_2 \end{aligned}$$

$a_1 = 1403580,\ b_1 = 810728,\ m_1 = 2^{32} - 209,$
$a_2 = 527612,\ b_2 = 1370589,\ m_2 = 2^{32} - 22853.$

Random number generation

- Both recurrences are of the form

$$y_n = A\, y_{n-1} \mod m$$

where y_n is a vector $y_n = (x_n,\ x_{n-1},\ x_{n-2})^T$ and A is a 3×3 matrix. Hence

$$y_{n+2^k} = A^{2^k} y_n \mod m = A_k\, y_n \mod m$$

where A_k is defined by repeated squaring as

$$A_{k+1} = A_k\, A_k \mod m, \qquad A_0 \equiv A.$$

Can generalise this to jump N places in $O(\log N)$ operations.

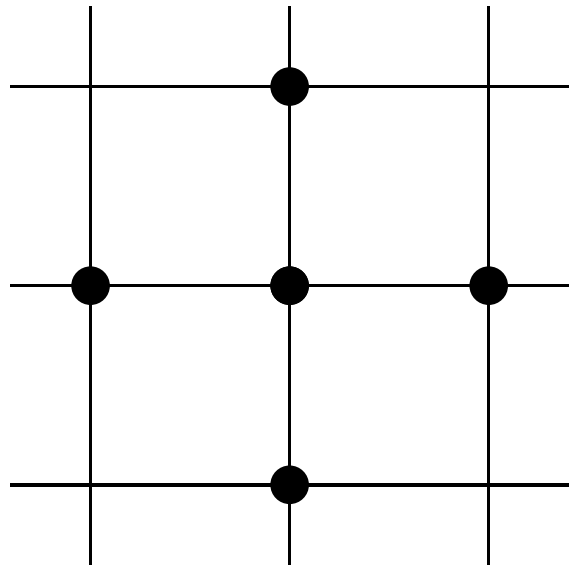
Random number generation

- output distributions:
  - uniform
  - exponential: trivial
  - Normal: Box-Muller or inverse CDF
  - Gamma: using “rejection” methods which require a varying number of uniforms and Normals to generate 1 Gamma variable
- producing Normals with **mrg32k3a**:
  - 2400M values/sec on a 216-core GTX260
  - 70M values/sec on a Xeon using Intel’s VSL
- have also implemented a **Sobol** generator to produce quasi-random numbers
  - 6500M Normals/sec on a 216-core GTX260 using an inverse CDF implementation

Finite Difference Model Problem

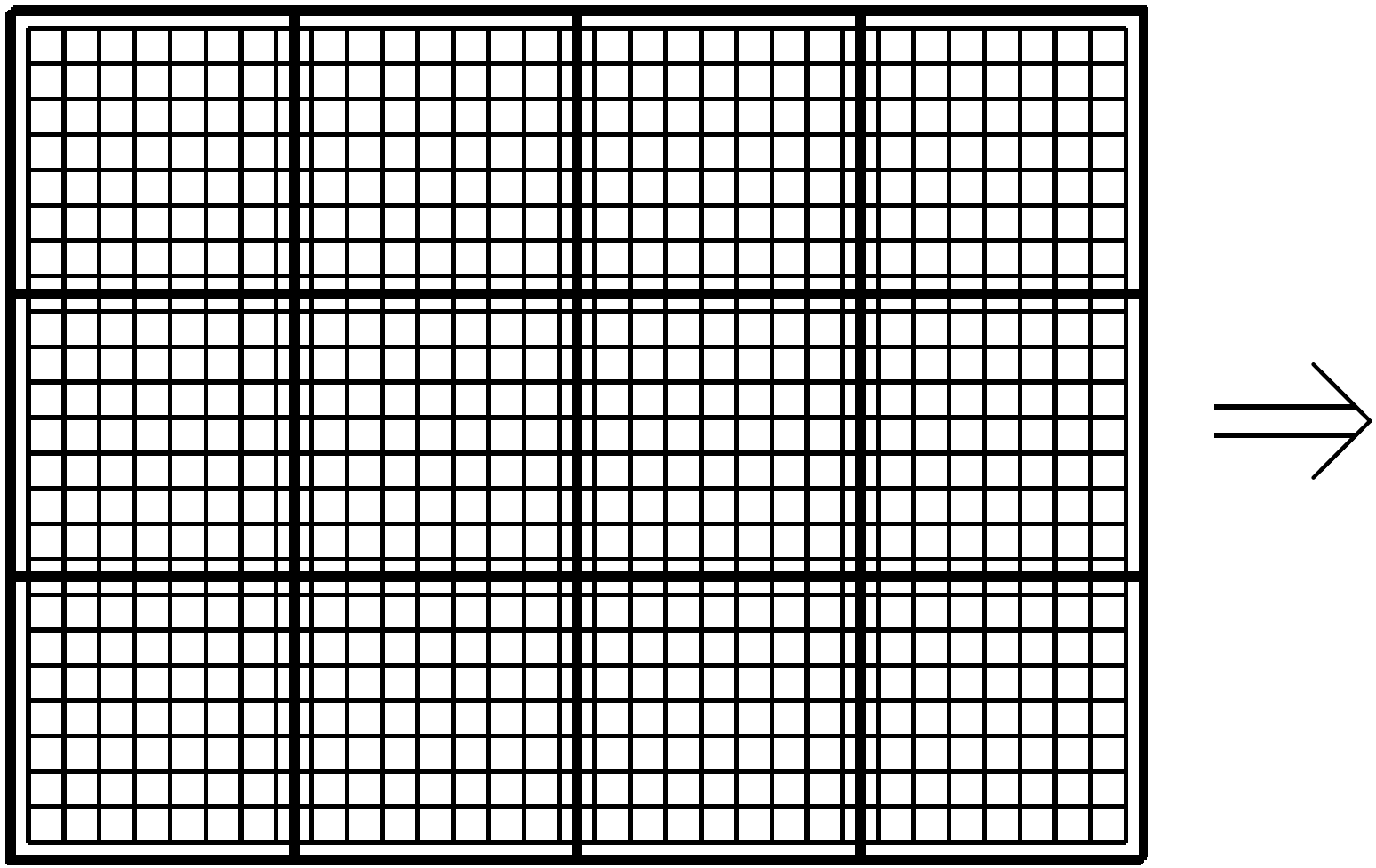
Jacobi iteration to solve discretisation of Laplace equation

$$V_{i,j}^{n+1} = \frac{1}{4}\left(V_{i+1,j}^{n} + V_{i-1,j}^{n} + V_{i,j+1}^{n} + V_{i,j-1}^{n}\right)$$



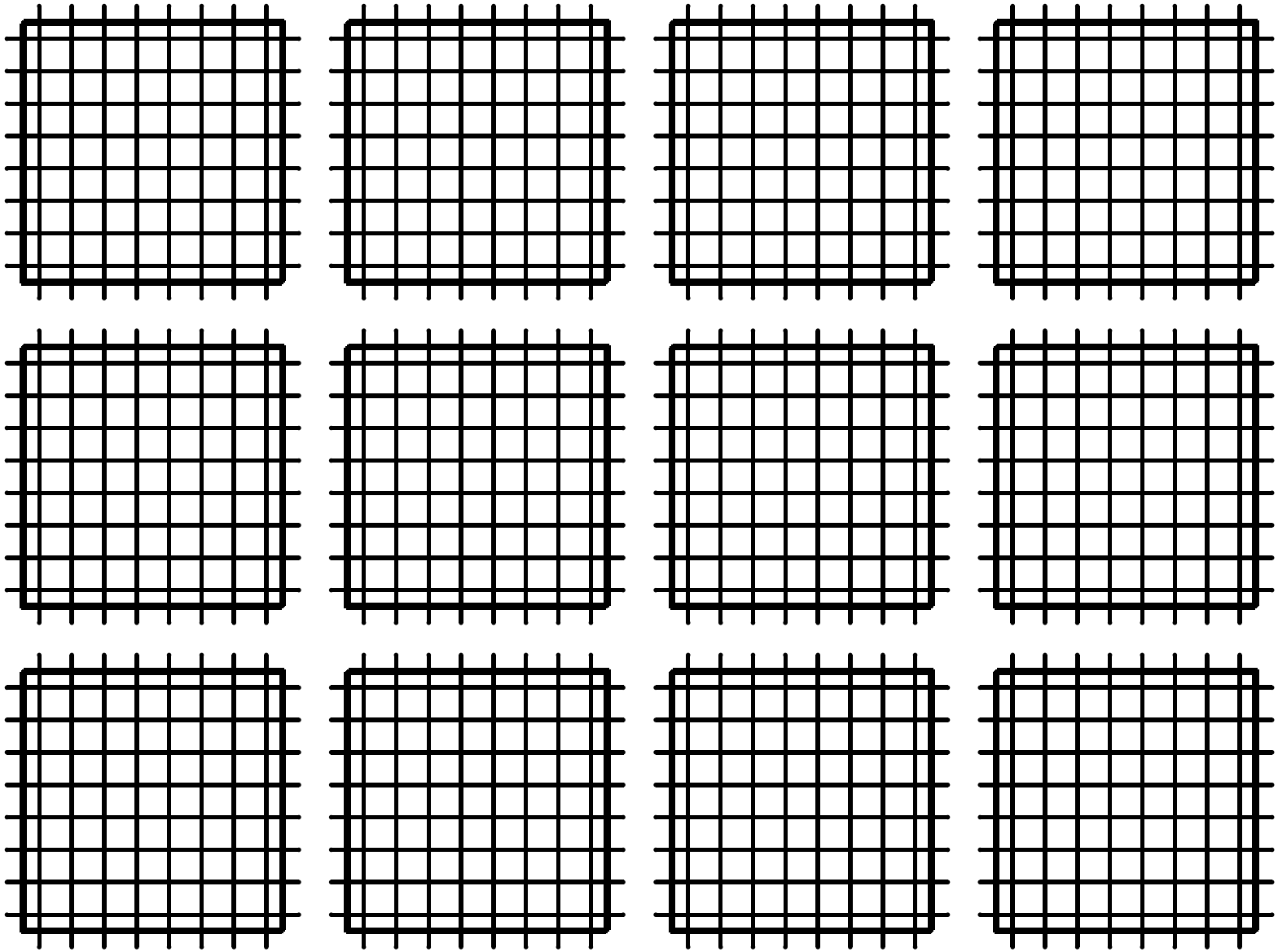
How should this be programmed?

Finite Difference Model Problem

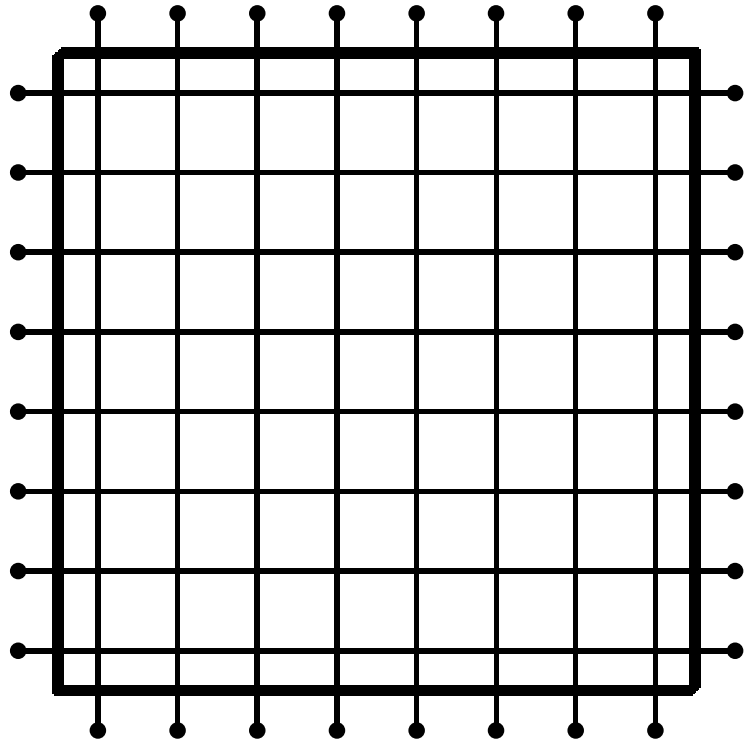


Key idea: take ideas from distributed-memory parallel computing and partition grid into pieces

Finite Difference Model Problem



Finite Difference Model Problem



Each block of threads will work with one of these grid blocks, reading in old values (including the “halo nodes” from adjacent partitions) then computing and writing out new values

Finite Difference Model Problem

Old data is loaded into shared memory:

- each thread loads in the data for its grid point and maybe one halo point
- data is then available for neighbouring threads when they need it
- each thread computed its new value and writes it to graphics memory
- this is slightly tedious, manually programming to duplicate what is done in a cache in a CPU; will be much simpler on new GPUs which have a cache

Finite Difference Model Problem

3D finite difference implementation:

- insufficient shared memory for whole 3D block, so hold 3 working planes at a time
- key steps in kernel code:
  - load in $k=0$ z-plane (inc x and y-halos)
  - loop over all z-planes
    - load $k+1$ z-plane
    - process k z-plane
    - store new k z-plane
- $50\times$ speedup relative to Xeon single core, compared to $5\times$ speedup using OpenMP with 8 cores.

More on Finite Differences

ADI implicit time-marching:

- in MPI parallelisation, a fixed data partitioning means tridiagonal solution spread across several processors – makes it hard to get good performance
- in CUDA parallelisation, all data is held in graphics memory, so can use a different partitioning for each phase of computation
- when solving tri-diagonal equations in a particular direction, each thread handles a separate line in that direction – simple and efficient
- again roughly 10$\times$ speedup compared to two quad-core Xeons

Current developments

NVIDIA: new “Fermi” GPUs shipping soon

- 512 SP cores (1.5 TFlops), 256 DP cores (750 GFlops)
- L1 / L2 cache – will simplify programming

AMD: new GPUs out now with OpenCL support

IBM: Cell hard to program – terminating further development for scientific computing

Intel: Larrabee GPU badly delayed (2011 or 2012?) but watch for AVX vectors in mainstream CPUs

Cray and other systems vendors are building GPU supercomputers

Further information

Monte Carlo and finite difference test codes
`www.maths.ox.ac.uk/~gilesm/hpc/`

CUDA course with practicals
`www.maths.ox.ac.uk/~gilesm/cuda/`

NAG numerical routines for GPUs
`www.nag.co.uk/numeric/GPUs/`

NVIDIA's CUDA homepage
`www.nvidia.com/object/cuda_home.html`

NVIDIA's computational finance page
`www.nvidia.com/object/computational_finance.html`