Lecture 2

- Topics for today:
    - Some Finance Theory for Pricing Derivatives and Contingent Claims
    - Random number generation for Monte Carlo simulations
    - Parallel processing across multiple CPU processors
    - Parallel processing for Monte Carlo simulation on GPU's

- Reading assignments for this lecture

  Cuda documentation for CURAND

  1st paper by L'Ecuyer and Simard

  Paper on skip ahead techniques by Bradley, du Toit, Giles, Tong, and Woodhams

  See p. 162, Figure 1 for C code to run MRG32k3a of "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators," by Pierre L'Ecuyer, 47 (January-Febvruary 1999).

- Problem Set 1: are all students now set up with ability to run Cuda C code on GPU's

- Problem Set 2 due in 2 weeks

Lecture 2 - Random Number Generators and Parallel Processing for Monte Carlo Simulation

2.1 Applications of Monte Carlo simulation in Finance

- Many of the asset pricing problems in Finance can be solved by simulating the model and numerically computing the expected value of the discounted cash flows.

- For example, one can value a call or put option in the Black-Scholes model by doing the following: (1) simulate the lognormal diffusion process for stock prices, (2) compute the payoff of the option at expiration, (3) discount with the interest rate, and (4) compute the average across a large number of simulations.

- Of course, we would not do this if we have a fast analytic solution for the option price, which is the case for European calls and puts in the Black-Scholes model. But we are likely to be interested in pricing more complex derivatives structures and we may want to use more complex models.

- A brief review of the theory for arbitrage-free risk neutral pricing. See Hull's textbook or Darrell Duffie, *Dynamic Asset Pricing*, any edition. The more general theory was presented in the CIR paper, Cox, Ingersoll, and Ross, "An Intertemporal General Equilibrium Model of Asset Prices," *Econometrica* 53 (1985). Their well known paper on the term structure of interest rates is a separate paper in the same issue of *Econometrica*.

Lecture 2.1 - Applications of Monte Carlo simulation in Finance (cont.)

- CIR results for asset pricing, Theorem 3

THEOREM 3: *The price of any contingent claim satisfies the partial differential equation*

$$(31)\qquad \tfrac{1}{2}(\operatorname{var} W)F_{WW}+\sum_{i=1}^{k}(\operatorname{cov} W, Y_i)F_{WY_i}+\tfrac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}(\operatorname{cov} Y_i, Y_j)F_{Y_iY_j}$$
$$+[r(W, Y, t)W-C^*(W, Y, t)]F_W$$
$$+\sum_{i=1}^{k}F_{Y_i}\left[\mu_i-\left(\frac{-J_{WW}}{J_W}\right)(\operatorname{cov} W, Y_i)-\sum_{j=1}^{k}\left(\frac{-J_{WY_j}}{J_W}\right)(\operatorname{cov} Y_i, Y_j)\right]$$
$$+F_t-r(W, Y, t)F+\delta(W, Y, t)=0,$$

where $r(W, Y, t)$ *is given from equation* (14) *as*

$$r(W, Y, t)=a^{*\prime}\alpha-\left(\frac{-J_{WW}}{J_W}\right)\left(\frac{\operatorname{var} W}{W}\right)-\sum_{i=1}^{k}\left(\frac{-J_{WY_i}}{J_W}\right)\left(\frac{\operatorname{cov} W, Y_i}{W}\right).$$

- We normally drop the wealth state variable, which can be justified by imposing some additional assumptions that CIR cover in the 2md paper on the term structure.

Lecture 2.1 - Applications of Monte Carlo simulation in Finance (cont.)

- CIR results for asset pricing

The existence and uniqueness of a solution to the fundamental valuation equation can be established under some additional regularity conditions.[21] To interpret the solution, consider the following two systems of stochastic differential equations: System I,

$$\begin{aligned} dW(t) &= [a^{*\prime}\alpha W - C^*]\,dt + a^{*\prime}GW\,dw(t), \\ dY(t) &= \mu(Y,t)\,dt + S(Y,t)\,dw(t); \end{aligned} \tag{35a}$$

and System II,

$$\begin{aligned} dW(t) &= [a^{*\prime}\alpha W - \phi_W - C^*]\,dt + a^{*\prime}GW\,dw(t), \\ dY(t) &= [\mu(Y,t) - [\phi_{Y_1} \quad \cdots \quad \phi_{Y_k}]']\,dt + S(Y,t)\,dw(t), \end{aligned} \tag{35b}$$

- Use the risk adjusted, or risk neutral, processes in CIR System II. We normally drop the wealth equation.

Lecture 2.1 - Applications of Monte Carlo simulation in Finance (cont.)

- CIR results for asset pricing

LEMMA 4: *The unique solution to* (31) *with boundary conditions* (34) *is also given by*

$$
\begin{aligned}
(37)\qquad F(W, Y, t, T) = \hat{E}_{W,Y,t}\Bigg[& \Theta(W(T), Y(T)) \\
& \times\left[\exp\left(-\int_t^T r(W(u), Y(u), u)\, du\right)\right] I(\tau \geqslant T) \\
& + \Psi(W(\tau), Y(\tau), \tau) \\
& \times\left[\exp\left(-\int_t^\tau r(W(u), Y(u), u)\, du\right)\right] I(\tau < T) \\
& + \int_t^{\tau \wedge T} \delta(W(s), Y(s), s) \\
& \times\left[\exp\left(-\int_t^s r(W(u), Y(u), u)\, du\right)\right] ds\Bigg],
\end{aligned}
$$

where $\hat{E}$ *denotes expectation with respect to System II, and* $I(\cdot)$ *and* τ *are as defined in* (36).

Lecture 2.1 - Applications of Monte Carlo simulation in Finance (cont.)

- Example for the 3 factor Hull-White model

$$
\begin{aligned}
dr &= [\kappa_0(\theta_0(t) + y_1 + y_2 - r) - \lambda_0\sigma_0^2(t)]\,dt + \sigma_0(t)dz_0 \\
dy_1 &= [\kappa_1(\theta_1 - y_1) - \lambda_1\sigma_1^2(t)]\,dt + \sigma_1(t)dz_1 \\
dy_2 &= (-\kappa_2 y_2 - \lambda_2\sigma_2^2(t))\,dt + \sigma_2(t)dz_2
\end{aligned}
$$

- Example for Local Volatility model

$$dS = (r(t) - \delta(t))S\,dt + \sigma(t,S)\,dz\,.$$

- Example of an FX model, with stochastic volatility

$$dS/S = \left(r_d - r_f\right)dt + \sqrt{v(t)}\,dz_1$$

Stochastic differentials for $v(t)$, $r_d(t)$, $r_d(y)$, $r_f(y)$, and $y(t)$

Lecture 2 - Random Number Generators and Parallel Processing for Monte Carlo Simulation (cont.)

2.2 Random number generators

- Different types of random number generators:

  - true hardware random number generators (HRNG)
  - Pseudo random number generators (PRNG) and cryptographically secure random number generators
  - quasi random number generators (example: Sobol sequences)

- **PRNG Example: Linear congruential generators (LCG)**

  Starting from an initial seed, n_0 calculate $n_i = (13^{13} \times n_{i-1}) \bmod 2^{59}$ and $x_i = \frac{n_i}{2^{59}}$

  x_i is a number between 0 and 1 and will appear to be random with a uniform $U(0,1)$ distribution.

  pseudo random number generators are sequential.

2.2 Pseudo random number generators (cont.)

- For multi-threading and parallel processing on GPU's, we will need to control the seed for the random number generator. This is done by using the step ahead method. Suppose that we want to run 100,000 simulations on each thread. We have a starting seed for the 1st thread. The starting seed for the 2nd thread should be the seed after running the 1st 100,000 simulations.

- Start with $n_0 = 1$, and run the seed calculation 100,000 times to get n_{100000} . Set $A_{100000} = n_{100000}$. This coefficient is an integer that we can apply to a starting seed n_0 to step ahead 100,000 simulations. Then for any initial seed, n_0 (as in $n_0 = 394885$), calculate

$$n_{100000} = (A_{100000} \times n_0) \bmod 2^{59}$$

- Then apply A_{100000} again to skip ahead another 100,000 simulations

$$n_{200000} = (A_{100000} \times n_{100000}) \bmod 2^{59}$$

- Repeat

$$n_{300000} = (A_{100000} \times n_{200000}) \bmod 2^{59}$$

2.2 Pseudo random number generators (cont.)

The LCG random number generators are fast, but there are pseudo random number generators with better statistical properties. Two currently popular pseudo RNG's are MRG32k3a and the Mersenne Twister.

MRG32k3a is a combined multiple recursive number generator

$$\begin{aligned} y_{1,n} &= \left(a_{12}\, y_{1,n-2} + a_{13}\, y_{1,n-3}\right) \mod m_1, \\ y_{2,n} &= \left(a_{21}\, y_{2,n-1} + a_{23}\, y_{2,n-3}\right) \mod m_2, \\ x_n &= \left(y_{1,n} + y_{2,n}\right) \mod m_1, \end{aligned} \tag{1}$$

for all $n \geq 3$ where

$$\begin{aligned} a_{12} &= 1403580 & a_{13} &= -810728 & m_1 &= 2^{32} - 209, \\ a_{21} &= 527612 & a_{23} &= -1370589 & m_2 &= 2^{32} - 22853\,. \end{aligned}$$

Generate $z_n = x_n / m_1$ as a uniform $U(0,1)$ random variable.

2.2 Pseudo random number generators (cont.)

- Double precision code can be copied directly from the paper by L'Ecuyer, p. 162, Figure 1.

```
double MRG32k3a(double *dseed)
{
    //  This code is an exact copy of the C code in L'Ecuyer (Operations Research 1999)
    int k;
    double p1, p2;
    p1 = a12*dseed[1] - a13n*dseed[2];
    k = p1 / m1;
    p1 -= k*m1;
    if (p1 < 0.0) p1 += m1;
    dseed[2] = dseed[1]; dseed[1] = dseed[0]; dseed[0] = p1;
    p2 = a21*dseed[3] - a23n*dseed[5];
    k = p2 / m2;
    p2 -= k*m2;
    if (p2 < 0.0) p2 += m2;
    dseed[5] = dseed[4]; dseed[4] = dseed[3]; dseed[3] = p2;
    if (p1 <= p2) return ((p1 - p2 + m1)*norm);
    else return ((p1 - p2)*norm);
}
```

- The 6 seeds are stored as double precision numbers.

2.2 Pseudo random number generators (cont.)

- There is also a formula for stepping ahead in the seeds for MRG32k3a, covered in the paper by Bradley, du Toit, Giles, Tong, and Woodhams.

$$Y_{i,n} = \begin{pmatrix} y_{i,n} \\ y_{i,n-1} \\ y_{i,n-2} \end{pmatrix}$$

for $i = 1, 2$. It follows that the two recurrences in (1) above can be represented as

$$Y_{i,n+1} = A_i Y_{i,n} \mod m_i$$

for $i = 1, 2$ where each A_i is a 3×3 matrix, and therefore

$$Y_{i,n+p} = A_i^p Y_{i,n} \mod m_i \qquad (2)$$

for any $p \geq 0$.

$$A_1 = \begin{pmatrix} 0 & a_{12} & a_{13} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \text{ and } A_2 = \begin{pmatrix} a_{21} & 0 & a_{23} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

2.2 Pseudo random number generators (cont.)

Mersenne Twister – MT 19937

- Currently the most popular pseudo RNG is the Mersenne Twister. There are several versions and the one identified as MT 19937 is the one typically used in software packages. Mersenne Twister is the default RNG in Python, Matlab, R, and other software applications.

- This RNG has a very long period, pow(2, 19937), and it runs fast relative to other RNG's. The LNG example has one initial integer seed, and MRG32k3a has 6 double precision seeds. MT 19937 has 623 32 bit integers for state vector (seeds). For the actual calculation, see M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp. 3-30, (1998) DOI:10.1145/272991.272995

- Jumping ahead or skipping ahead is complex, and applications vary. For a recent paper on running the Mersenne Twister in parallel by one of the authors who originated the algorithm see the following link: https://arxiv.org/pdf/1005.4973v3.pdf

2.2 Pseudo random number generators (cont.)

- If we want to run simulations with a pseudo RNG in parallel, we need to be able to quickly run a jump ahead, or skip ahead, algorithm to determine initial seeds for each thread. We would like to be able to replicate a Monte Carlo simulation run sequentially with one that we run in parallel across multiple processors, or in parallel on a GPU. The ability to replicate is necessary for testing.

- In most current applications with the Mersenne Twister, an independent vector of the initial state is applied for each thread. The parallel simulation results will be different from the simulation results performed with one long sequential sequence. This may not be a critical issue for many applications.

- Mutli-threading across CPU processors. Generate a sequence of 25,000 uniform random numbers across 10 threads (1 per CPU processor) with MRG32k3a. Need to calculate

  A_i^{25000} for $i = 1, 2$. Let $Y_{i,0}$ be the set of initial seeds used for the 1st thread.

  Use $Y_{i,25000} = A_i^{25000} Y_{i,0} \bmod m_i$ as the starting seed on the second thread

  For the 3rd thread start with $Y_{i,50000} = A_i^{25000} Y_{i,25000} \bmod m_i$

2.2 Pseudo random number generators (cont.)

- Mutli-threading across CPU processors. Generate a sequence of 25,000 uniform random numbers across 10 threads (1 per CPU processor) with MRG32k3a.

  Continue repeating these calculations to get starting seeds for the remaining threads

- The Divide & Conquer method can be used to speed up the calculation of A_i^n for i = 1, 2, for large n. Compute

  $A_i^2 = A_i \cdot A_i \bmod m_i$, then $A_i^4 = A_i^2 \cdot A_i^2 \bmod m_i$, $A_i^8 = A_i^4 \cdot A_i^4 \bmod m_i$, ……,

  Continue with 2, 4, 8, 16, 32, 64, 128, 256, 1024, 2018, …. until you reach the largest exponent less than the desired n. Then multiply by A_i (and take mod m_i at each multiplication) to reach A_i^n for i = 1, 2

- This calculation produces the jump ahead or skip ahead for multi-threading across multiple CPU processors.

2.2 Pseudo random number generators (cont.)

- The calculation on the previous slide produces the jump ahead or skip ahead for multi-threading across multiple CPU processors. Use the same method to produce the starting seed for each simulation path that will be run on a thread on a GPU. Run each simulation path in parallel on a GPU. Suppose that the simulation model uses 500 simulations of the uniform $U(0,1)$ distribution on each path. Then calculate A_i^{500} for $i = 1, 2$, to calculate the starting seed for each path and pass these starting seeds to the GPU.

- Suppose that you are multi-threading across 4 processors, each one working with a separate GPU: each processor will run 25,000 simulation paths with 500 sequential simulations of the uniform distribution on each path. Compute A_i^{500} for $i = 1, 2$, for skipping ahead to get starting seeds for each path to run on GPU. Calculate $A_i^{500\times 25000}$ for $i = 1, 2$, to get the starting seeds for each thread on the CPU processors. 500 times 25000 is a large number so that you do want to use the divide & conquer method.

2.2 Pseudo random number generators (cont.)

- In many of our financial models, we need to simulate normally distributed random variables. There are several fast methods for simulating normal random variables. The Box Muller method simulates 2 normals by simulating 2 $U(0,1)$ random variables. For simulations of U_1 and U_2, calculate

$$R^2 = -2\ln U_1\ ,\ \Theta = 2\pi U_2, \quad \text{then} \quad Z_0 = R\cos\Theta = \sqrt{-2\ln U_1}\cos(2\pi U_2)$$
$$Z_1 = R\sin\Theta = \sqrt{-2\ln U_1}\sin(2\pi U_2)$$

  Z_0, Z_1 are independent standard normal random variables.

- Alternatively, one can calculate a standard normal random variable by simulating a number between 0 and 1 with the $U(0,1)$ and then calculate the inverse normal. Let $N(z)$ be the standard normal distribution function. Calculate z from $N(z) = U(0,1)$, or simply $z = N^{-1}\big(U(0,1)\big)$. There are fast algorithms for calculating the inverse normal function. I recommend this method, particularly for running simulations of normal random variables on the GPU. Use MRG32k3a to simulate $U(0,1)$ random numbers and the inverse normal distribution function to transform these to normal random variables.

Lecture 2 - Random Number Generators and Parallel Processing for Monte Carlo Simulation

2.3 Test Results for pseudo random number generators

- See the test results in the paper by L'Ecuyer and Simard

- Pseudo random number generators are not really random, but the dependency is nonlinear and sequences generated by pseudo random number generators appear to be random.

- One can perform statistical tests on pseudo random number generators to determine if they are sufficiently random. In Finance, we use simulation to compute expected values, and we would like to know that Monte Carlo simulation with a pseudo RNG is converging to the expected value that we are trying to calculate. We want a pseudo RNG with good statistical properties.

- The paper by L'Ecuyer and Simard, published 10 years ago, provides the results from a battery of tests run on a number of pseudo RNG's. Go to Table I of the paper. Currently popular RNG's included are MRG32k3a and Mersenne Twister (MT19937 in Table).

2.3 Test Results for pseudo random number generators (cont.)

Table I. Results of Test Batteries Applied to Well-Known RNGs

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush		Crush		BigCrush	
LCG(2^{24}, 16598013, 12820163)	24	3.9	0.66	14		—		—	
LCG(2^{31}, 65539, 0)	29	3.3	0.65	14		125	(6)	—	
LCG(2^{32}, 69069, 1)	32	3.2	0.67	11	(2)	106	(2)	—	
LCG(2^{32}, 1099087573, 0)	30	3.2	0.66	13		110	(4)	—	
LCG(2^{46}, 5^{13}, 0)	44	4.2	0.75	5		38	(2)	—	
LCG(2^{48}, 25214903917, 11)	48	4.1	0.65	4		21	(1)	—	
Java.util.Random	47	6.3	0.76	1		9	(3)	21	(1)
LCG(2^{48}, 5^{19}, 0)	46	4.1	0.65	4		21	(2)	—	
LCG(2^{48}, 33952834046453, 0)	46	4.1	0.66	5		24	(5)	—	
LCG(2^{48}, 44485709377909, 0)	46	4.1	0.65	5		24	(5)	—	
LCG(2^{59}, 13^{13}, 0)	57	4.2	0.76	1		10	(1)	17	(5)
LCG(2^{63}, 5^{19}, 1)	63	4.2	0.75			5		8	
LCG(2^{63}, 9219741426499971445, 1)	63	4.2	0.75			5	(1)	7	(2)
LCG(2^{31}–1, 16807, 0)	31	3.8	3.6	3		42	(9)	—	
LCG(2^{31}–1, $2^{15} - 2^{10}$, 0)	31	3.8	1.7	8		59	(7)	—	
LCG(2^{31}–1, 397204094, 0)	31	19.0	4.0	2		38	(4)	—	
LCG(2^{31}–1, 742938285, 0)	31	19.0	4.0	2		42	(5)	—	
LCG(2^{31}–1, 950706376, 0)	31	20.0	4.0	2		42	(4)	—	
LCG(10^{12}–11, 427419669081, 0)	39.9	87.0	25.0	1		22	(2)	34	(1)
LCG(2^{61}–1, $2^{30} - 2^{19}$, 0)	61	71.0	4.2			1	(4)	3	(1)

2.3 Test Results for pseudo random number generators (cont.)

Wichmann-Hill	42.7	10.0	11.2	1		12	(3)	22	(8)
CombLec88	61	7.0	1.2			1			
Knuth(38)	56	7.9	7.4			1	(1)	2	
ran2	61	7.5	2.5						
CLCG4	121	12.0	5.0						
Knuth(39)	62	81.0	43.3				(1)	3	(2)
MRGk5-93	155	6.5	2.0						
DengLin (2^{31}–1, 2, 46338)	62	6.7	15.3		(1)	11	(1)	19	(2)
DengLin (2^{31}–1, 4, 22093)	124	6.7	14.6		(1)	2		4	(2)
DX-47-3	1457	—	1.4						
DX-1597-2-7	49507	—	1.4						
Marsa-LFIB4	287	3.4	0.8						
CombMRG96	185	9.4	2.0						
MRG31k3p	185	7.3	2.0				(1)		
MRG32k3a	191	10.0	2.1						
MRG63k3a	377	—	4.3						
LFib(2^{31}, 55, 24, +)	85	3.8	1.1	2		9		14	(5)
LFib(2^{31}, 55, 24, −)	85	3.9	1.5	2		11		19	
ran3		2.2	0.9		(1)	11	(1)	17	(2)
LFib(2^{48}, 607, 273, +)	638	2.4	1.4			2		2	
Unix-random-32	37	4.7	1.6	5	(2)	101	(3)	—	
Unix-random-64	45	4.7	1.5	4	(1)	57	(6)	—	
Unix-random-128	61	4.7	1.5	2		13		19	(3)
Unix-random-256	93	4.7	1.5	1	(1)	8		11	(1)

(*continues*)

2.3 Test Results for pseudo random number generators (cont.)

Table I. Results of Batteries of Tests Applied on Well-Known RNGs (continued)

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush		Crush		BigCrush	
Knuth-ran_array2	129	5.0	2.6			3		4	
Knuth-ranf_array2	129	11.0	4.5						
SWB(2^{24}, 10, 24)	567	9.4	3.4	2		30		46	(2)
SWB(2^{24}, 10, 24)[24, 48]	566	18.0	7.0			6	(1)	16	(1)
SWB(2^{24}, 10, 24)[24, 97]	565	32.0	12.0						
SWB(2^{24}, 10, 24)[24, 389]	567	117.0	43.0						
SWB(2^{32}–5, 22, 43)	1376	3.9	1.5		(1)	8		17	
SWB(2^{31}, 8, 48)	1480	4.4	1.5		(2)	8	(2)	11	
Mathematica-SWB	1479	—	—	1	(2)	15	(3)	—	
SWB(2^{32}, 222, 237)	7578	3.7	0.9			2		5	(2)
GFSR(250, 103)	250	3.6	0.9	1		8		14	(4)
GFSR(521, 32)	521	3.2	0.8			7		8	
GFSR(607, 273)	607	4.0	1.0			8		8	
Ziff98	9689	3.2	0.8			6		6	
T800	800	3.9	1.1	1		25	(4)	—	
TT800	800	4.0	1.1			12	(4)	14	(3)
MT19937	19937	4.3	1.6			2		2	
WELL1024a	1024	4.0	1.1			4		4	
WELL19937a	19937	4.3	1.3			2	(1)	2	
LFSR113	113	4.0	1.0			6		6	
LFSR258	258	6.0	1.2			6		6	
Marsa-xor32 (13, 17, 5)	32	3.2	0.7	5		59	(10)	—	
Marsa-xor64 (13, 7, 17)	64	4.0	0.8	1		8	(1)	7	

2.3 Test Results for pseudo random number generators (cont.)

Summary:

- The statistical properties of the linear congruential generators (LNG) are not so good: these RNG's do not pass some of the tests for randomness. The LNG presented in previous slides is in the 11th row of the 1st table. This LNG was the default in older versions of the NAG library.

- MRG32k3a passes all of the statistical tests performed by L'Ecuyer and Simard.

- Both MRG32k3a and the Mersenne Twister perform well in the tests, and have very long periods. The length of the period indicates how many simulations one can perform before the sequence repeats itself.

- I recommend using MRG32k3a, because one can easily apply the stepping ahead algorithm.

- By applying the step ahead algorithm, one can generate pseudo random numbers in parallel across multiple threads on CPU processors or on GPU's and replicate exactly the results obtained from running the pseudo RNG sequentially.

- By setting the initial seeds, one can replicate the results of a Monte Carlo simulation, and this feature is useful for model testing.

Lecture 2 - Random Number Generators and Parallel Processing for Monte Carlo Simulation

2.4 Multi-threading and code for running pseudo random number generators

- The engineers at Nvidia have included MRG32k3a and the step ahead algorithm in the Cuda library, CURAND. This library also includes a number of other popular random number generators.

- Classroom examples:

    1) Code for multi-threading across CPU processors
    2) Code for stepping ahead with MRG32k3a and running normal simulations in parallel
    3) Code to run MRG32k3a to normal simulations on the GPU with stepping ahead on each thread of the GPU calculation
    4) Example running $U(0,1)$ with normal inverse versus the normal simulation in CURAND. The Curand normal random number generator uses the Box-Mueller method and simulates independent normal in pairs.
    5) CURAND examples for the GPU

- The massive parallel processing makes a big difference when you run Monte Carlo simulations for a pricing problem in which you must simulate many time steps from time zero to expiration. For these simulations, you should program and run each simulation path as a thread on the GPU.

2.4 Multi-threading and code for running pseudo random number generators (cont.)

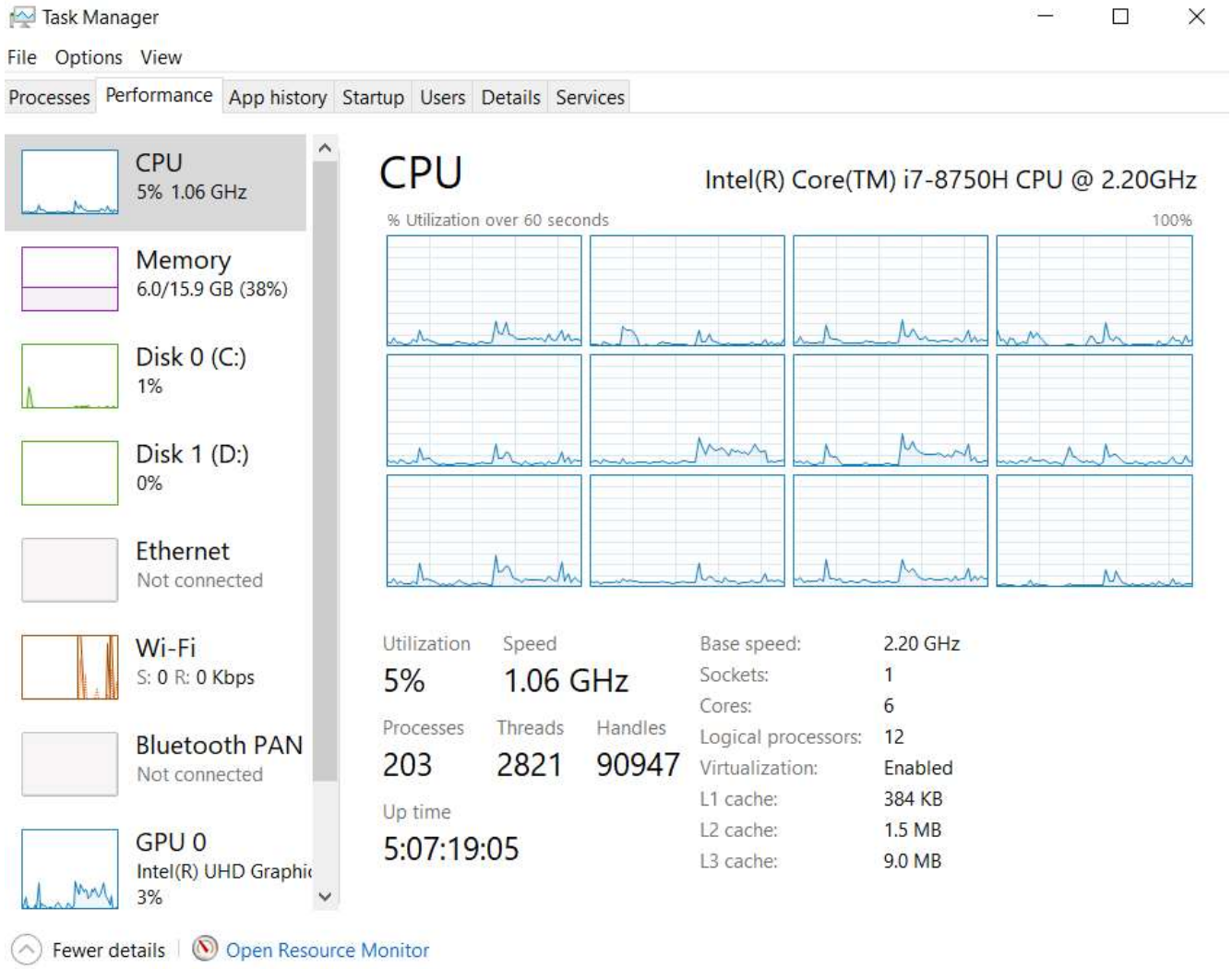
CPU's have multiple cores with multiple threads available

Intel Core i7-8750H CPU @ 2.20GHz, with Max Turbo Frequency 4.10 GHz
6 cores / 12 threads, Max Memory Size 64 Gbytes,
Approximately $400

Intel Core i9-8950K CPU @ 2.90GHz, with Max Turbo Frequency 4.80 GHz
6 cores / 12 threads, Max Memory Size 64 Gbytes,
Approximately $583

Intel's high-end server CPU's have many more cores and threads

Intel Xeon Platinum 8160 Processor (for servers)
Base Frequency 2.10GHz, with Max Turbo Frequency 3.70GHz
24 cores / 48 threads, Max Memory Size 768 Gbytes
Approximately $4,700


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU
5% 1.06 GHz
Memory
6.0/15.9 GB (38%)
Disk 0 (C:)
1%
Disk 1 (D:)
0%
Ethernet
Not connected
Wi-Fi
S: 0 R: 0 Kbps
Bluetooth PAN
Not connected
GPU 0
Intel(R) UHD Graphi
3%
CPU
Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
% Utilization over 60 seconds
100%
Utilization
5%
Speed
1.06 GHz
Processes
203
Threads
2821
Handles
90947
Up time
5:07:19:05
Base speed: 2.20 GHz
Sockets: 1
Cores: 6
Logical processors: 12
Virtualization: Enabled
L1 cache: 384 KB
L2 cache: 1.5 MB
L3 cache: 9.0 MB
Fewer details
Open Resource Monitor

2.4 Multi-threading and code for running pseudo random number generators (cont.)

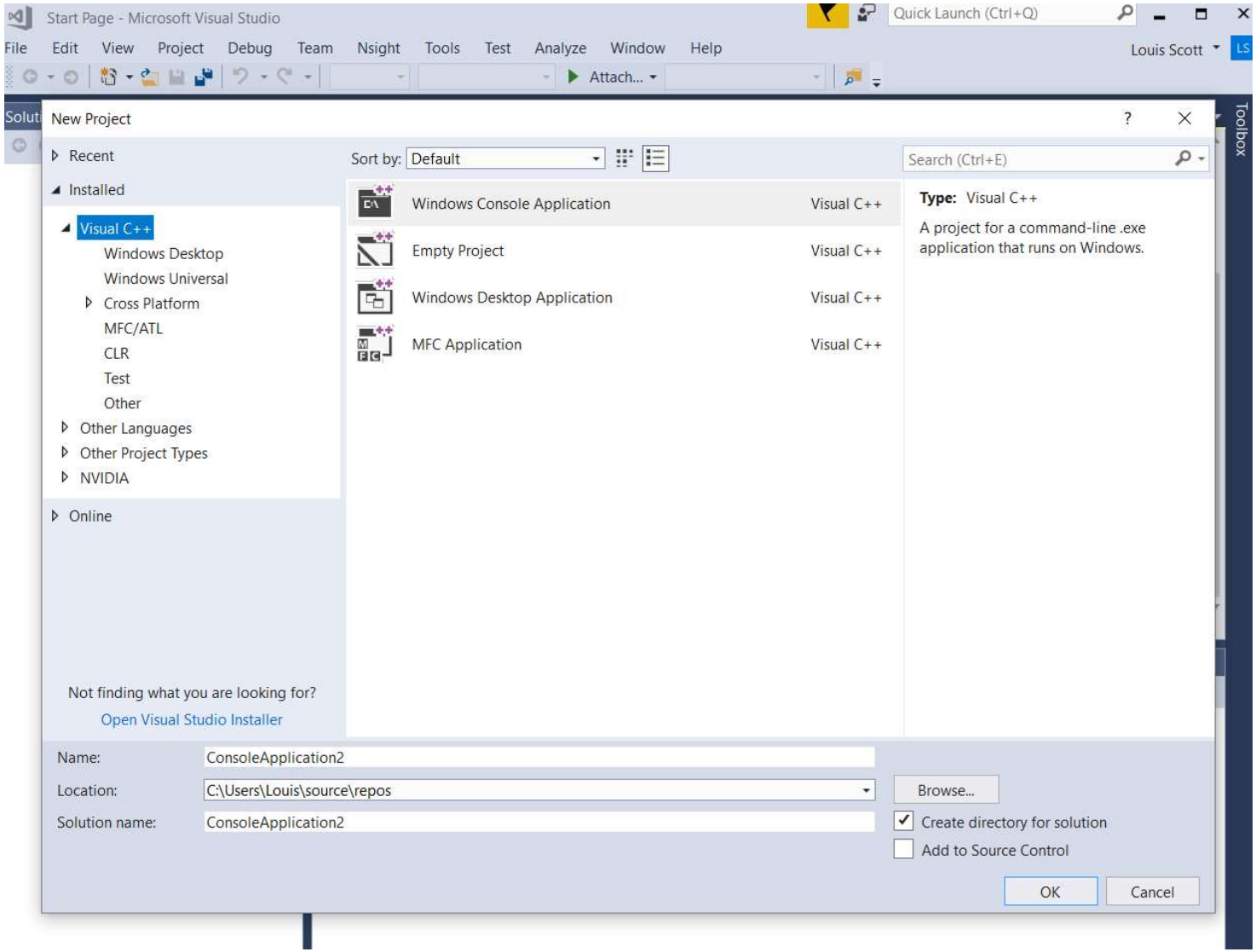
- Multi-threading in C/C++ on Windows operating systems
- Several methods are available
- I recommend using **beginthreadex** and **_thread_data_t** structure
- Organize the programming for your model, and identify the portions that can be calculated simultaneously, in parallel

  Monte Carlo simulation: simulations along a path are normally performed sequentially, but each independent simulation path can be executed in parallel. So run a smaller number of simulations on each thread

  Finite difference algorithms for derivative pricing run sequentially by starting at expiration and working backwards. At each time step, there are calculations that can be run in parallel, or run multiple strikes on different threads.

  The calculations inside **for** loops can be candidates for parallel processing. Recursions must be calculated sequentially. Summations to calculate averages and expected values are not easy to make parallel. One strategy for summation, or reductions, is called **divide and conquer**: break the summation into pieces and calculate the pieces in parallel and then sum the pieces. I do the summations for expected values sequentially on the CPU.

- Do **example** – open a new project in Visual Studio


Start Page - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Debug Team Nsight Tools Test Analyze Window Help
Louis Scott
Attach...
Toolbox
New Project
Recent
Installed
Visual C++
Windows Desktop
Windows Universal
Cross Platform
MFC/ATL
CLR
Test
Other
Other Languages
Other Project Types
NVIDIA
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Windows Console Application Visual C++
Empty Project Visual C++
Windows Desktop Application Visual C++
MFC Application Visual C++
Type: Visual C++
A project for a command-line .exe application that runs on Windows.
Name: ConsoleApplication2
Location: C:\Users\Louis\source\repos
Solution name: ConsoleApplication2
Browse...
Create directory for solution
Add to Source Control
OK
Cancel

2.4 Multi-threading and code for running pseudo random number generators (cont.)

- See Windows_Multi-Threading_Skeleton_CPU , which is a skeleton example for multi-threading in Windows
- Go to Visual Studio example:

```
1   //  Example of Multi-threading in C++ on Windows operating system
2   // include necessary header files
3   #include <malloc.h>
4   #define MAX_NUM_THREADS 8
5   using namespace std;
6
7   /* prototype */
8   unsigned __stdcall RunThread(void *param);
9   int MyParallelFunction(int jThread);
10
11  /* Global Variables */
12  //  Data and parameters required by functions running on multiple thread
13  int nThreads;
14  double Data, *Params;
15  //  ........
16  //  Results from each thread need to be stored in global memory
17  double *Result;
18
19  HANDLE threads[MAX_NUM_THREADS];
20  FILE *fout;
21
22  /* create thread argument struct for Run_Thread() */
23  typedef struct _thread_data_t {
24      int tid;
25      //  double stuff;
26  } thread_data_t;
27
28  int _tmain(int argc, _TCHAR* argv[])
29  {
30      int i, j, nThreads;
31      double *A, *B, x, y;
```

2.4 Multi-threading and code for running pseudo random number generators (cont.)

- See the SVJump_Model_MonteCarlo_MRG32k3a_64 project/folder, which has a program to simulate a complex stock price model across multiple threads on a CPU. Uses the step ahead algorithm to initial set seeds for each thread.

- See the Test_MRG32k3a_GPU project/folder, which has a program to simulate uniform random variables on the GPU. Uses the step ahead algorithm to set seeds, and Curand

- See the Test_MRG32k3a_Normal_GPU project/folder, which has a program to simulate normal random variables for a stock price process on the GPU. Uses the step ahead algorithm to set seeds, and Curand.

- Go to Visual Studio examples →