An Introduction to OpenMP, MPI and CUDA

How to make your Monte Carlo Code 150 times faster

Matthew Willyard

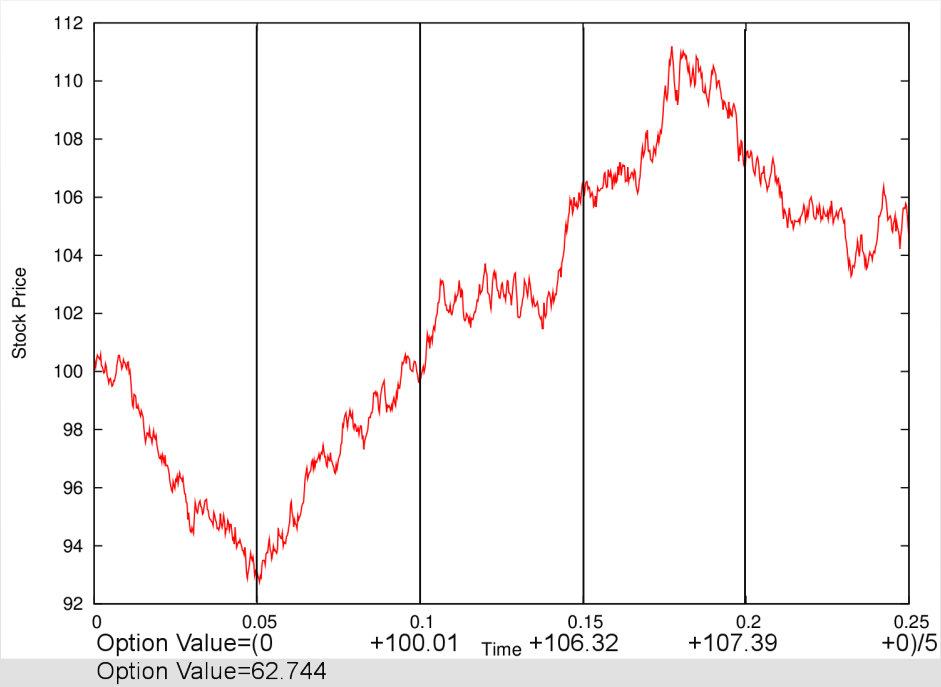
Numerical Methods Seminar
Florida State University

October 18, 2010

Outline

Ratchet Option[Pap02]

Figure: Example Ratchet Option



- Used to hedge equity-linked index annuities

Monte Carlo Simulation of Ratchet Option

- Assume geometric Brownian Motion: $dS_t = \mu S_t dt + \sigma S_t dW_t$

Algorithm for Ratchet Option

```
Value=0
do j=1,Number_of_Simulations
    V=0
    S_old = S_0
    Call Normal_Random_Number_Generator(Z⃗)
    do i=1,Number_of_Time_Steps
        S_new = S_old * e^((r - 1/2 σ^2)Δt + σ√Δt Z(i))
        if (S_new > S_old) then
            V = V + S_new
        endif
        S_old = S_new
    enddo
    Value=Value+V/Number_of_Time_Steps
enddo
Option_Price= Value/Number_of_Simulations
```

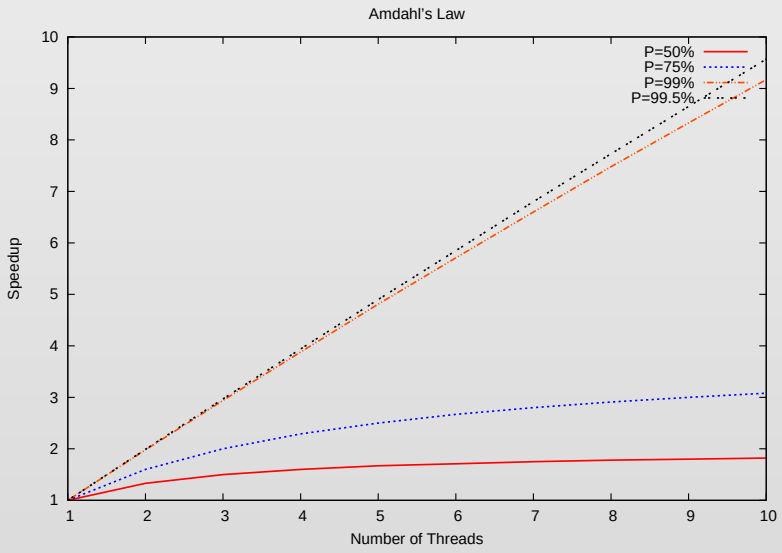
Advantages/Disadvantages of Monte Carlo

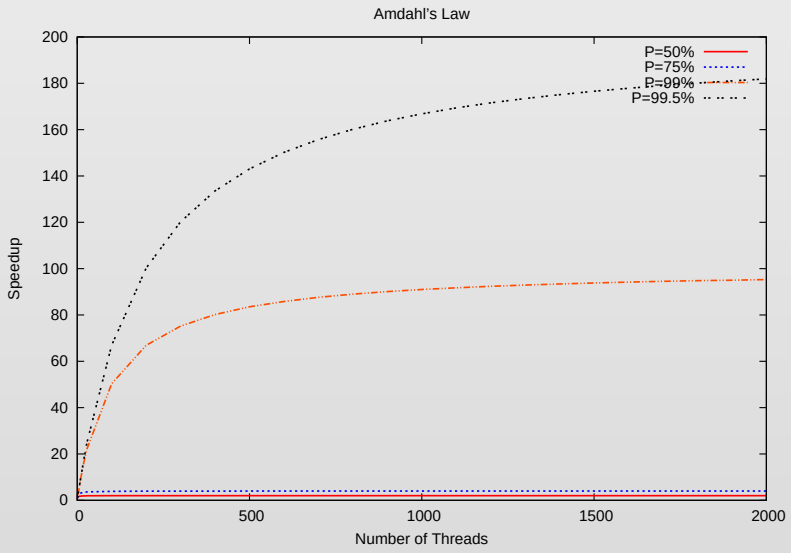
- Advantages
  - Algorithms are simple
  - Versatile
- Disadvantage
  - Slow Convergence
    - $O(N^{-\frac{1}{2}})$ for Monte Carlo
    - $O(N^{-1}(\log N)^d)$ for Quasi-Monte Carlo

Parallel Monte Carlo

- Monte Carlo is "embarrassingly" parallel
- Correlation ruins results
- Amdahl's Law

$$\text{Absolute Speedup} = \frac{\text{Sequential Algorithm Time}}{\text{Parallel Algorithm Time}} = \frac{1}{(1-P) + \frac{P}{N}}$$





Random Number Generators

- Hybrid Pseudorandom Number Generator[HT07]:
  - 3 Tausworth Generators[L'E96] combined with 1 Linear Congruential Generator[PTVF92]
- Quasirandom Number Generator:
  - Random Scrambled[OSG09] Random Start[Ö08] Halton Sequence

Overview

1x

Sequential Code

- Ordinary Code

5x

OpenMP

- Compiler Directives

6x

Open MPI: Open Source High Performance Computing

- Message Passing

150x

NVIDIA CUDA

- Programming for GPU

Open Multi-Processing(OpenMP)

API for explicitly direct multi-threaded, shared memory parallelism

Open Multi-Processing(OpenMP)

API for explicitly direct multi-threaded, shared memory parallelism

- Advantages
  - Portable
    - Works with C/C++ and Fortran
    - Works on Unix/Linux and Windows
  - Easy to implement

Open Multi-Processing(OpenMP)

API for explicitly direct multi-threaded, shared memory parallelism

- Advantages
  - Portable
    - Works with C/C++ and Fortran
    - Works on Unix/Linux and Windows
  - Easy to implement
- Disadvantages
  - Shared memory only
  - Large overhead
  - Threads executed in non-deterministic order

How OpenMP works

Sequential

```
Value=0
Call Init_Random_Number_Generator()
do j=1,Number_of_Simulations
    V=0
    S_old = S_0
    Call Normal_Random_Number_Generator(Z)
    do i=1,Number_of_Time_Steps
        S_new = S_old * e^((r-1/2 σ^2)Δt+σ√Δt Z(i))
        if (S_new > S_old) then
            V = V + S_new
        endif
        S_old = S_new
    enddo
    Value=Value+V/Number_of_Time_Steps
enddo
Option_Price= Value/Number_of_Simulations
```

OpenMP

```
!$omp parallel default(private) shared(Value)
Value=0
Call Init_Random_Number_Generator()
!$omp do schedule(static) REDUCTION(+:Value)
do j=1,Number_of_Simulations
    V=0
    S_old = S_0
    Call Normal_Random_Number_Generator(Z)
    do i=1,Number_of_Time_Steps
        S_new = S_old * e^((r-1/2 σ^2)Δt+σ√Δt Z(i))
        if (S_new > S_old) then
            V = V + S_new
        endif
        S_old = S_new
    enddo
    Value=Value+V/Number_of_Time_Steps
enddo
!$omp end do
!$omp end parallel
Option_Price= Value/Number_of_Simulations
```

Message Passing Interface(MPI)

MPI is a specification for message passing libraries.

Message Passing Interface(MPI)

MPI is a specification for message passing libraries.

- Advantages
  - Portable
    - Works with C/C++ and Fortran
    - Works on Unix/Linux and Windows
  - Works on shared and distributed memory parallel systems

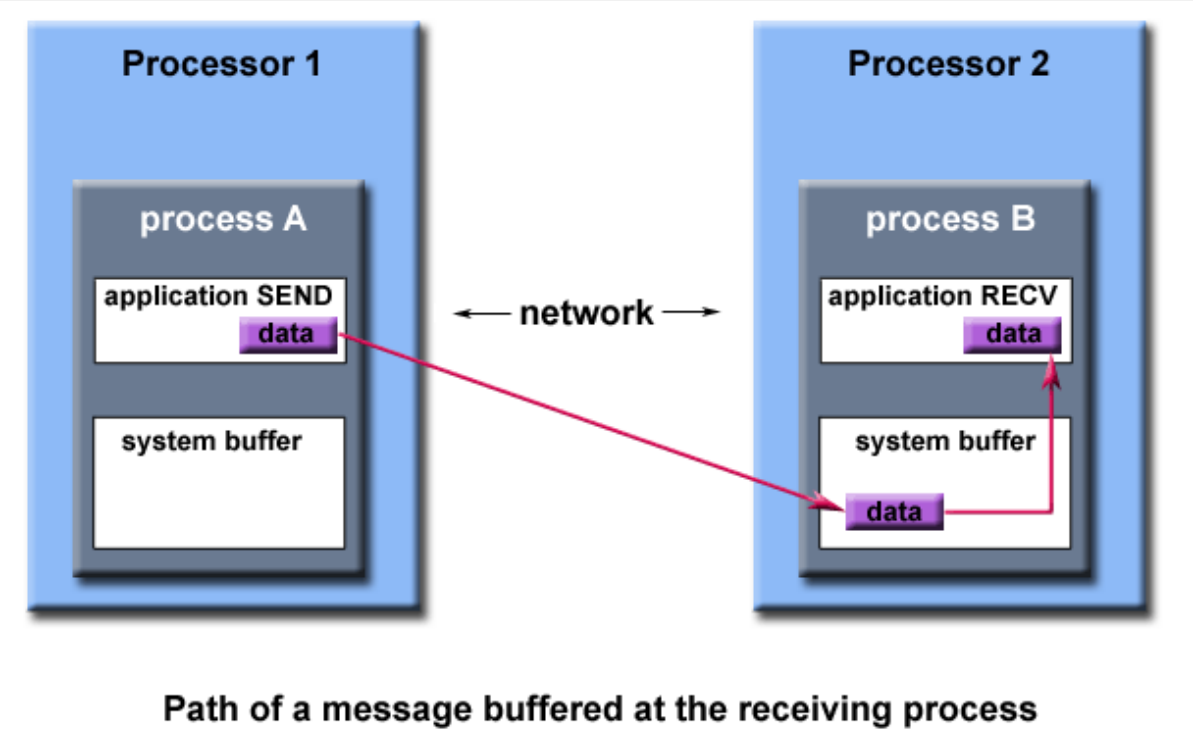
Message Passing Interface(MPI)

MPI is a specification for message passing libraries.

- Advantages
  - Portable
    - Works with C/C++ and Fortran
    - Works on Unix/Linux and Windows
  - Works on shared and distributed memory parallel systems
- Disadvantages
  - Must install MPI
  - Dynamic load balancing is difficult
  - Requires significant rewriting of sequential code
  - Communication must be explicitly coded

How MPI works

Figure: Illustration of MPI Threads[1]



[1] image from https://computing.llnl.gov/tutorials/mpi/

How MPI works

OpenMP

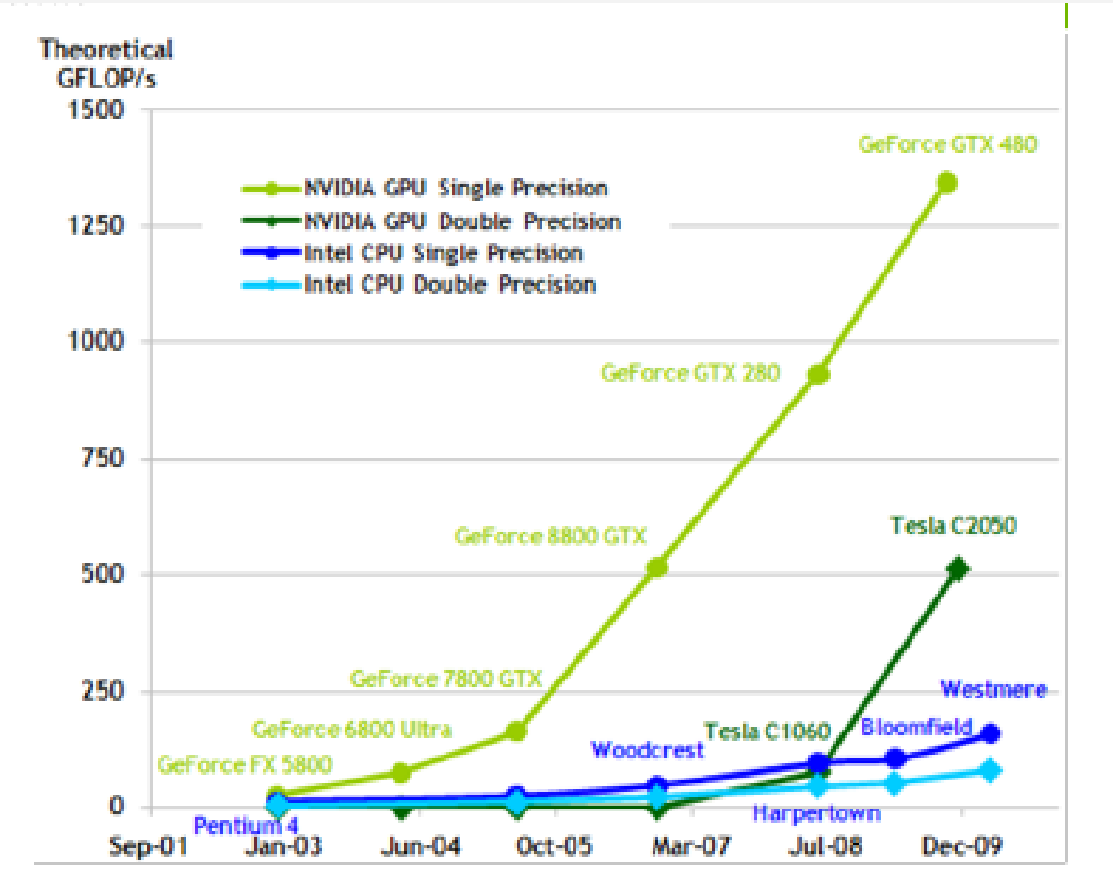
```
!$omp parallel default(private) shared(Value)
Value=0
Call Init_Random_Number_Generator()
!$omp do schedule(static) REDUCTION(+:Value)
do j=1,Number_of_Simulations
    V=0
    S_old = S_0
    Call Normal_Random_Number_Generator(Z)
    do i=1,Number_of_Time_Steps
        S_new = S_old * e^((r-1/2 σ^2)Δt+σ√ΔtZ(i))
        if (S_new > S_old) then
            V = V + S_new
        endif
        S_old = S_new
    enddo
    Value=Value+V/Number_of_Time_Steps
enddo
!$omp end do
!$omp end parallel
Option_Price= Value/Number_of_Simulations
```

MPI

```
use mpi
integer, parameter :: master = 0
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, NThrs, ierr)
Value=0
Call Init_Random_Number_Generator()
do j=1,Number_of_Simulations/NThrs
    V=0
    S_old = S_0
    Call Normal_Random_Number_Generator(Z)
    do i=1,Number_of_Time_Steps
        S_new = S_old * e^((r-1/2 σ^2)Δt+σ√ΔtZ(i))
        if (S_new > S_old) then
            V = V + S_new
        endif
        S_old = S_new
    enddo
    Value=Value+V/Number_of_Time_Steps
enddo
call MPI_REDUCE(Value, V, 1, MPI_DOUBLE_PRECISION,
    MPI_SUM, master, MPI_COMM_WORLD, ierr)
Option_Price= V/Number_of_Simulations
call MPI_FINALIZE(ierr)
```

Why use the graphics card?

Figure: GPU architecture.[2]



[2] image from NVIDIA CUDA Programming Guide 3.2RC

Compute Unified Device Architecture(CUDA)

- NVIDIA CUDA is a general purpose parallel computing architecture that allows programs to run on NVIDIA graphics processing units (GPUs).
- Example NVIDA graphics card: GTX 260
  - 27 multiprocessors at 1.24 GHz
  - each multiprocessor has 8 cores for a total of 216 cores
  - Capable of running 27,648 threads concurrently

CUDA

- Advantages
  - Supports Linux, OS X, and Windows
  - Grants access to computation power of GPUs
- Disadvantages
  - Requires an NVIDA graphics card G8X series and onwards
  - Requires all GPU code to be written in stripped down C.

How CUDA works

Code run on Host

1. Set block_size and the number_of_threads you want
2. Create seeds for the random number generators on the host
3. Create arrays for seeds on device using cudaMalloc
4. Copy seeds from host to the device using cudaMemcpy
5. Create array for Option values on device using cudaMalloc
6. Call the kernel function on the device
   - ex.

```
DigitalOptionPRNG<<<dimGrid,dimBlock>>>(z1_d,z2_d,z3_d,z4_d,Value_d,NT,SimsThread);
```

7. Copy array of option values from device to the host
8. Average the option values to find approximation.
9. Free arrays allocated on device using cudaFree

How CUDA works

MPI

```
use mpi
integer, parameter :: master = 0
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, NThrs, ierr)
Value=0
Call Init_Random_Number_Generator()
do j=1,Number_of_Simulations/NThrs
    V=0
    S_old = S_0
    Call Normal_Random_Number_Generator(Z⃗)
    do i=1,Number_of_Time_Steps
        S_new = S_old * e^((r-1/2 σ^2)Δt+σ√ΔtZ(i))
        if(S_new > S_old)then
            V = V + S_new
        endif
        S_old = S_new
    enddo
    Value=Value+V/Number_of_Time_Steps
enddo
call MPI_REDUCE(Value, V, 1, MPI_DOUBLE_PRECISION,
    MPI_SUM, master, MPI_COMM_WORLD, ierr)
Option_Price= V/Number_of_Simulations
call MPI_FINALIZE(ierr)
```

Code run on Device

```
__global__ void DigitalOptionPRNG(
unsigned *seed1,unsigned *seed2,
unsigned *seed3,unsigned *seed4,
float* V_d,unsigned NT,unsigned SimsThread){
  unsigned int index
      =blockIdx.x*blockDim.x+threadIdx.x;
  if( index<NT ){
    unsigned z1=seed1[index];
    unsigned z2=seed2[index];
    unsigned z3=seed3[index];
    unsigned z4=seed4[index];
    for(int j=0;j<SimsThread;j++){
      V=0
      S_old = S_0
      for(int i=0;i<NumTimeSteps;i++){
        Z(i)=DeviceNormalRNG(z1,z2,z3,z4)
        S_new = S_old * e^((r-1/2 σ^2)Δt+σ√ΔtZ(i))
        if(S_new > S_old)
          V = V + S_new
          S_old = S_new
        }
        Value=Value+V/Number_of_Time_Steps
    }
    V_d[index]=Value/SimsThread;
  }
}
```

Absolute Speedup

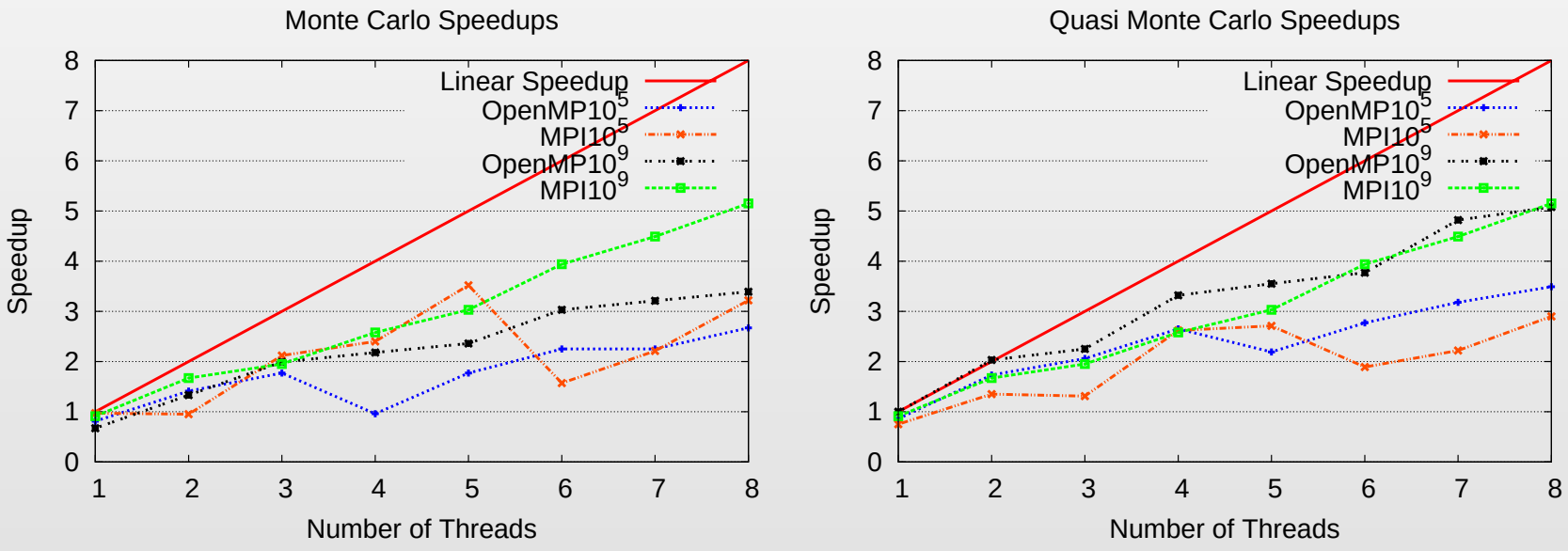
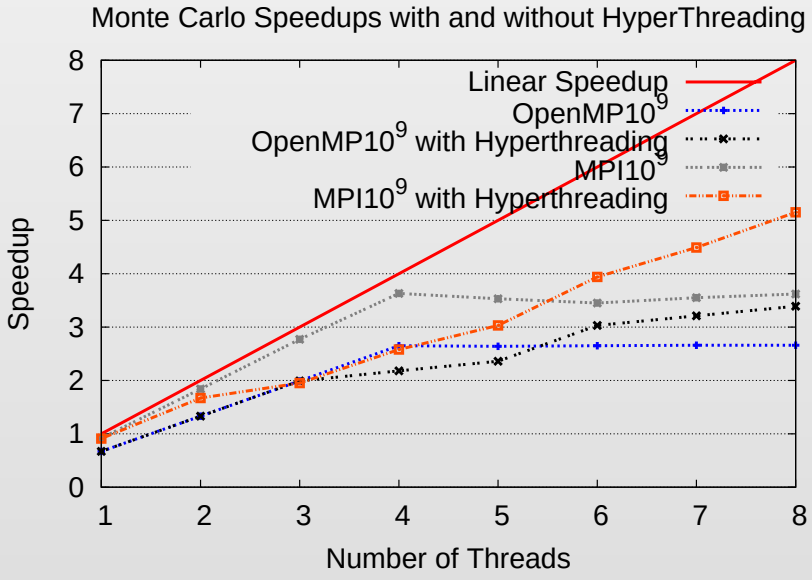
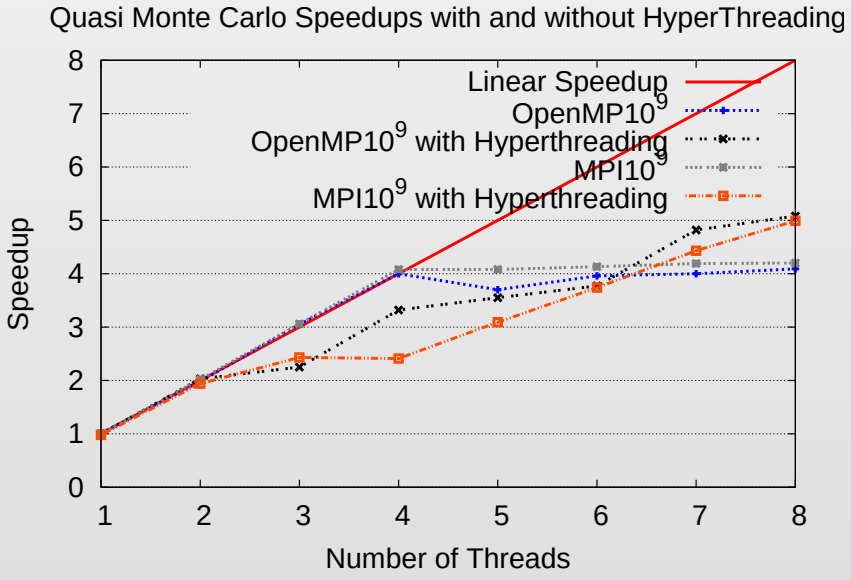


Figure: Speedup for Monte Carlo and Quasi Monte Carlo

Impact of Hyperthreading

Figure: Speedup with and without Hyperthreading





Time Comparison for Monte Carlo

Monte Carlo

Method	Number of Threads	Error	Time(s)	Speedup
Sequential	1	4.39E-4	514.59	-
OpenMP	8	8.19E-4	151.78	3.39
MPI	8	8.74E-4	99.84	5.15
MPI(two computers)	10	5.98E-4	83.59	6.16
MPI(two computers)	18	1.38E-4	89.26	5.77
MPI(two computers)	28	7.36E-4	84.10	6.12
CUDA	27,648	3.51E-4	3.36	152.96

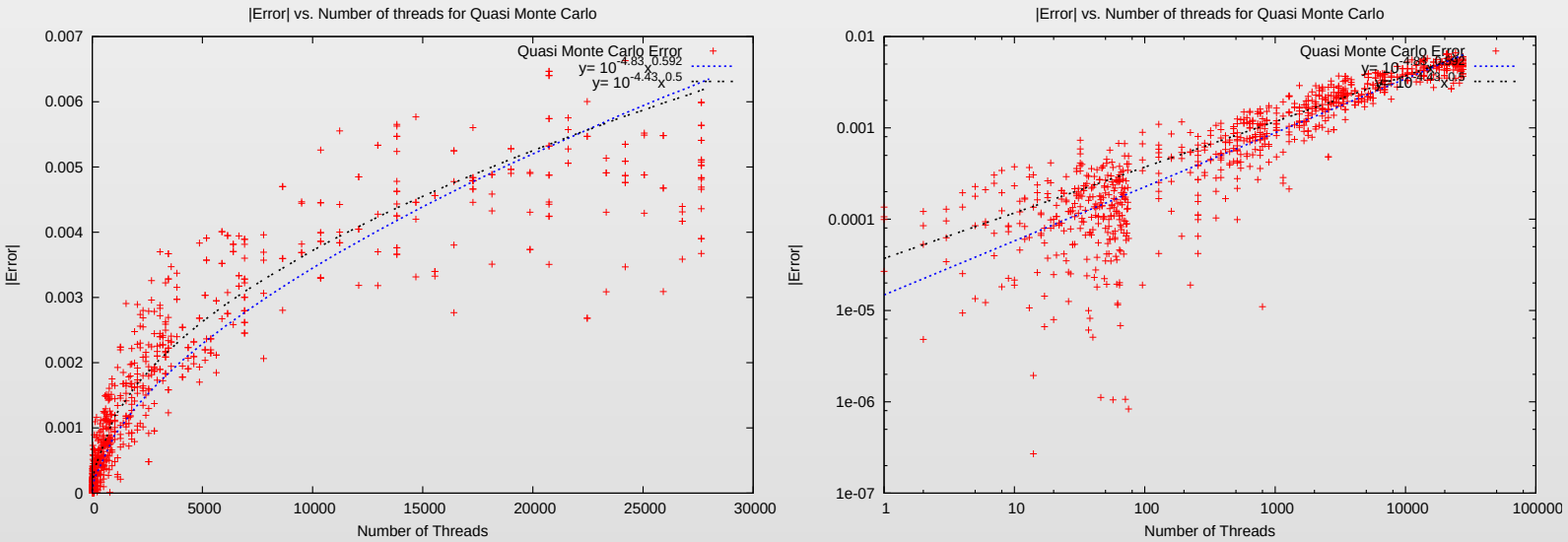
Time Comparison for Quasi Monte Carlo

Quasi Monte Carlo

Method	Number of Threads	Error	Time(s)	Speedup
Sequential	1	5.68E-6	3675.27	-
OpenMP	8	5.99E-6	724.04	5.08
MPI	8	7.33E-7	737.15	4.99
MPI(two computers)	10	3.60E-6	894.97	4.11
MPI(two computers)	18	3.29E-7	637.01	5.77
MPI(two computers)	28	9.78E-7	628.75	5.85
CUDA	27,648	3.12E-3	36.88	99.65
CUDA	1728	4.00E-4	117.1	31.39

Error of Quasi Monte Carlo Method vs. Number of Threads

Figure: |Error| vs. Number of Threads for Quasi Monte Carlo



Conclusions

- OpenMP and MPI both provide similar significant speedups for Monte Carlo programs.
- On a single computer OpenMP is simple to implement.
- For multiple computers MPI must be used and requires a little more effort from the programmer.
- Using the CUDA with a GPU offers the potential for amazing speedups but even greater care must be taken with the random number generator.

Further Reading

- OpenMP tutorial at Lawrence Livermore by Blaise Barney
  - http://www.llnl.gov/computing/tutorials/openMP/
- MPI tutorial at Lawrence Livermore by Blaise Barney
  - http://www.llnl.gov/computing/tutorials/mpi/
- Open MPI: Open source MPI-2 implementation
  - http://www.open-mpi.org/
- CUDA homepage
  - http://www.nvidia.com/cuda
- Dr. Dobb's guide to CUDA
  - http://www.ddj.com/architect/207200659

- Lee W. Howes and David Thomas, *Efficient random number generation and application using cuda*, GPU Gems 3 (2007).
- P. L'Ecuyer, *Maximally equidistributed combined tausworthe generators*, Mathematics of Computation (1996).
- Giray Ökten, *Generalized von neumann-kakutani transformation and random-start scrambled halton sequences*, December 2008.
- Giray Ökten, Manan Shah, and Yevgeny Goncharov, *Random and deterministic digit permutations of the halton sequence*, January 2009.
- A. Papgeorgiou, *The brownian bridge does not offer a consisten advantage in quasi-monte carlo integration*, Journal of Complexity (2002), 171–186.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in c*, Cambridge University Press, 1992.