Get more power for your compute-heavy workloads

At Principled Technologies, we tested a Dell EMC™ PowerEdge™ R840 server, powered by top-of-the-line Intel® Xeon® Platinum 8180M processors and Intel SSD DC P4500 NVMe drives. We compared this 14th generation offering to a 12th generation Dell EMC PowerEdge R820 server with Intel Xeon E5-4650 processors to assess the computational strength of the newer system.

The new PowerEdge R840 was able to run 4.5 times as many Monte Carlo simulations per minute as the 12th generation server—meaning its calculations were 3.5 times faster than the older system.

Statistical error in Monte Carlo simulations decreases as the number of simulations increases. To that end, we tested each solution to see how many simulations each could complete in a fixed amount of time. We found that statistical error on the PowerEdge R840 simulations was 54 percent lower than that of simulations on the 12th generation server.

The Monte Carlo workload we tested predicts a portfolio value at some point in the future, and it takes more CPU work to carry a simulation further into the future. The newer solution's speed enabled it to simulate 1.2 times further into the future than the Dell PowerEdge R820 solution, for comparable run times.

APRIL 2018
A PRINCIPLED TECHNOLOGIES REPORT
Commissioned by Dell Technologies

Section A: System configuration information

On 04/11/2018, we finalized the hardware and software configurations we tested. Updates for current and recently released hardware and software appear often, so unavoidably these configurations may not represent the latest versions available when this report appears. For older systems, we chose configurations representative of typical purchases of those systems. We concluded hands-on testing on 04/16/2018.

	Dell EMC PowerEdge R840	Dell EMC PowerEdge R820
BIOS name and version	Dell BIOS 0.2.5	Dell BIOS 2.4.1
Non-default BIOS settings	N/A	N/A
Operating system name and version/build number	CentOS 7.4 kernel 3.10.0-693.21.1.el7.x86_64	CentOS 7.4 kernel 3.10.0-693.21.1.el7.x86_64
Date of last OS updates/patches applied	4/10/2018	4/11/2018
Power management policy	Performance Per Watt (DAPC)	Performance Per Watt (DAPC)
Processor		
Number of processors	4	4
Vendor and model	Intel Xeon Platinum 8180M	Intel Xeon E5-4650
Core count (per processor)	28	8
Core frequency (GHz)	2.50	2.70
Stepping	4	7

	Dell EMC PowerEdge R840	Dell EMC PowerEdge R820
Memory module(s)		
Total memory in system (GB)	3,072	384
Number of memory modules	48	48
Vendor and model	Samsung M386A8K40BM2-CTD	Samsung M393B1G70BH0-YK0
Size (GB)	64	8
Type	PC4-21300	PC3L-12800R
Speed (MHz)	2,666	1,600
Speed running in the server (MHz)	2,666	1,600
Storage controller		
Vendor and model	PERC H740P	PERC H710
Cache size	4096 MB	512 MB
Firmware version	50.0.1-0639	21.3.4-0001
Driver version	07.701.17.00-rh1	07.701.17.00-rh1
Local storage		
Number of drives	6	6
Drive vendor and model	Samsung PM863a	SanDisk LT0200MO
Drive size	1.92 TB	200 GB

	Dell EMC PowerEdge R840	Dell EMC PowerEdge R820
Drive information (speed, interface, type)	SATA SSD	SAS SSD
NVMe storage		
Number of drives	6	N/A
Drive vendor and model	Intel DC P4500	N/A
Drive size	1.0 TB	N/A
Drive information (speed, interface, type)	PCIe NVMe SSD	N/A
Network adapter		
Vendor and model	Broadcom BCM5720	Intel I350-t
Number and type of ports	4x Gigabit Ethernet	4x Gigabit Ethernet
Driver version	tg3 3.137	igb 5.4.0-k
Cooling fans		
Vendor and model	Delta Electronics PFM0612XHE-SM02	SanAce60 9GA0612P1J611
Number of cooling fans	6	6
Power supplies		
Vendor and model	Dell 095HR5	Dell 0CC6WF
Number of power supplies	2	2
Wattage of each (W)	1600	1100

Section B: Detailed testing methodology

This section shows our methods for OS installation and server configuration, as well as how we installed and ran the Financial Monte Carlo application.

Installing and configuring the operating system

1. Boot the server from the CentOS 7.4 (1708) minimal-installation DVD.
2. From the options menu, select Install CentOS.
3. Select the language and keyboard type you wish to use for this installation.
4. Choose default disk partitioning on one volume. Do not partition any remaining volumes.
5. Choose Minimal Install.
6. Disable kdump.
7. Apply Network configuration.
    a. Enable the first active network port.
    b. Assign the static IP address and hostname from the following table:

System	hostname	IP Address	Routing Prefix
Dell EMC PowerEdge R840	test-r840	10.215.1.151	/16
Dell EMC PowerEdge R820	test-r820	10.215.1.150	/16

    c. Set the IP addresses of the gateway and DNS server to 10.220.0.1 and 10.41.0.10, respectively.

8. Set the time zone to Eastern/US, and enable NTP to sync only from 10.40.0.1.
9. Click Install.
10. Set the root password.
11. When the install completes, reboot the system.
12. Log into the system as root.
13. Update the OS software, and install additional packages.

```
yum -y update
yum -y install gcc gcc-c++ make autoconf automake flex libtool numactl\
  sysstat bash-completion wget lsof sysstat
```

14. Stop and disable the following unnecessary services:

```
for i in postfix.service firewalld.service ; do
  systemctl disable $i
  systemctl stop $i
done
```

15. Disable SELinux with the following command:

```
sed -i 's/^SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config
```

16. Set the tuning profile with the following command:

```
tuned-adm profile balanced
```

17. Reboot the system.

Preparing the Financial Monte Carlo application

We obtained a financial Monte Carlo program from Intel's Developer Zone at the following URL (download will begin when you click): https://software.intel.com/sites/default/files/managed/0e/30/MonteCarloSample_12_31_14.tar.gz. Intel modified the original code to illustrate performance with Intel processors and compilers. See the archive's documentation for details on the code's history. We placed the source code under content management via git. In the following section, we present our source-code modification. They fall under four key areas:

1. We implemented random number generation for our environment, which did not use Intel's MKL math libraries.
2. We modified the complied options for GCC to reflect the current architectures and compiler capabilities.
3. We added code to compute an unbiased estimator of the statistical uncertainty in the valuation of the portfolio.
4. We added the option to parallelize the code via OpenMP. We choose two "hot spots" to parallelize, and we implemented a simple parallelization scheme, distinct from that chosen by Intel.

In addition, we created run scripts to control which CPU cores (hyperthreads) ran each instance of the program. For each system, we used the 4.8.5 version of the GCC compiler from the CentOS distribution.

```
$ rpm -qa | grep gcc
gcc-4.8.5-16.el7_4.2.x86_64
gcc-c++-4.8.5-16.el7_4.2.x86_64
libgcc-4.8.5-16.el7_4.2.x86_64

$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/4.8.5/lto-wrapper
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
```

```
--infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla
--enable-bootstrap --enable-shared --enable-threads=posix --enable-checking=release
--with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions
--enable-gnu-unique-object --enable-linker-build-id --with-linker-hash-style=gnu
--enable-languages=c,c++,objc,obj-c++,java,fortran,ada,go,lto --enable-plugin
--enable-initfini-array --disable-libgcj --with-isl=/builddir/build/BUILD/gcc-4.8.5-
20150702/obj-x86_64-redhat-linux/isl-install --with-cloog=/builddir/build/BUILD/gcc-
4.8.5-20150702/obj-x86_64-redhat-linux/cloog-install --enable-gnu-indirect-function
--with-tune=generic --with-arch_32=x86-64 --build=x86_64-redhat-linux
Thread model: posix
gcc version 4.8.5 20150623 (Red Hat 4.8.5-16) (GCC)
```

The table below shows simulation data and settings for each of our tests.

	Dell EMC PowerEdge R840	Dell EMC PowerEdge R820
Application scale-out test		
Simulations	1,960,000	1,960,000
Simulation length	40	40
Instances	224	64
Compiler options	default	default
Monte Carlo statistical uncertainty test		
Simulations	1,960,000	1,450,000
Simulation length	40	40
Instances	224	64
Compiler options	default	default

Simulation length test		
Simulations	1,960,000	1,960,000
Simulation length	44	20
Instances	1	1
Compiler options	default + -fopenmp	default + -fopenmp

Performance testing of the financial Monte Carlo application

Because several parameters controlling either the portfolio or the simulation were set in the source code, we re-compiled the application for each of the three tests. We give the values of the parameters and any GCC compiler options in the following table. Note that our default compiler options were as follows: `-std=c++11 -Ofast -flto -march=native`

Testing application scale out

We ran increasing numbers of independent instances of the Monte Carlo program with the following scripts, one for the R820, and the other for the R840.

```
# R820
for i in 0 1 3 7 15 31 47 63 ; do
  echo test $i >> test-$i
  for j in $(seq 0 $i) ; do
    numactl -C $j ./release/MonteCarlo &
  done | tee -a test-r820-$i
  wait
done
# nos. of instances that are multiples of 64
for i in {1..2} ;  do for j in $(seq 0 63) ; do
```

```
  numactl -C $j ./release/MonteCarlo &
  done
done | tee test-r820-b

# R840
for i in 0 1 3 7 15 31 63 95 111 127 143 159 175 191 223 ; do
  echo test $i >> test-$i
  for j in $(seq 0 $i) ; do
    numactl -C $j ./release/MonteCarlo &
  done | tee -a test-r840-$i
  wait
done
# nos. of instances that are multiples of 224
for i in {1..2} ;  do for j in $(seq 0 223) ; do
  numactl -C $j ./release/MonteCarlo &
  done
done | tee test-r840b
```

Testing Monte Carlo statistical uncertainty

We ran independent instances of the Monte Carlo program up to the maximum number of hyperthreads on the server with the following scripts, one for the R820, and the other for the R840.

```
# R820
for i in {0..63} ; do
  numactl -C $i ./release/MonteCarlo &
done

# R840
for i in {0..223} ; do
  numactl -C $i ./release/MonteCarlo &
done
```

Testing simulation length

We ran one instance of the application, parallelized over the maximum of physical cores on the system: namely, 112 for the Dell EMC PowerEdge R840, and 32 for the Dell EMC PowerEdge R820.

```
# R820
OMP_NUM_THREADS=32 ./release/MonteCarlo
# R840
OMP_NUM_THREADS=112 ./release/MonteCarlo
```

Section C: Our modifications to the financial Monte Carlo source-code distribution

In this section, we show how we modified the original Monte Carlo source code for our testing purposes.

```
# git diff HEAD

diff --git a/.gitignore b/.gitignore
index 151080d..8cfb621 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1 +1,5 @@
 release/
+Build.bat
+MonteCarlo.sln
+MonteCarlo.vcxproj
+MonteCarlo.vcxproj.filters
diff --git a/Makefile b/Makefile
index 7a606e3..ea29b26 100644
@@ -19,6 +19,8 @@
 SOURCES := $(wildcard $(SRCDIR)/*.cpp)
 OBJECTS := $(patsubst $(SRCDIR)/%,$(BUILDDIR)/%,$(SOURCES:.cpp=.o))

+$(OBJECTS): src/monte_carlo.h
+
 $(TARGET): $(OBJECTS)
        @echo " Linking..."
        $(CXX) $^ $(LIBFLAGS) -o $(TARGET)
@@ -30,7 +32,9 @@
 icpc: $(TARGET)

 gcc: CXX := g++
-gcc: CFLAGS := -O2 -mfpmath=sse -flto
```

```
+#gcc: CFLAGS := -O2 -mfpmath=sse -flto
gcc: CFLAGS := -std=c++11 -Ofast -flto -march=native #-fopenmp
+gcc: LIBFLAGS := $(CFLAGS) -lm
 gcc: $(TARGET)
diff --git a/src/main.cpp b/src/main.cpp
index c387e20..469183b 100644
@@ -41,12 +41,8 @@
                volatility[i] = c_volatility_val;
        }

-       // create normal distribution either using MKL, or setting all values to 0.3
-#ifdef IS_USING_MKL
+       // create normal distribution
        float_point_precision
*normal_distribution_rand=initialize_normal_dist(c_normal_dist_mean,
c_normal_dist_std_dev);
-#else
-       float_point_precision *normal_distribution_rand=initialize_normal_dist(0,0);
-#endif

 #ifndef __INTEL_COMPILER
 #ifdef PERF_NUM
@@ -56,9 +52,10 @@
        CUtilTimer timer;
        printf("Starting serial, scalar Monte Carlo...\n");
        timer.start();
-       float_point_precision payoff =
calculate_monte_carlo_paths_scalar(initial_LIBOR_rate, volatility,
normal_distribution_rand, discounted_swaption_payoffs);
+       std::tuple<float_point_precision, float_point_precision> payoff;
+        payoff = calculate_monte_carlo_paths_scalar(initial_LIBOR_rate, volatility,
normal_distribution_rand, discounted_swaption_payoffs);
        timer.stop();
-       printf("Calculation finished. Average discounted payoff is %.6f. Time taken is
%.0fms\n", payoff, timer.get_time()*1000.0);
```

```
+       printf("Calculation finished. Average discounted payoff is %.6f (%.6f) for %d
runs. Time taken is %.0fms\n", std::get<0>(payoff),
sqrt(std::get<1>(payoff)/c_num_simulations), c_num_simulations, timer.get_time()*1000.0);
 #ifdef PERF_NUM
        avg_time += time;
        }
@@ -188,7 +185,14 @@

 // Returns an array of random numbers pulled from a normal distribution
 // If MKL is enabled, a Gaussian distribution is used
-// if MKL is not enabled, 0.3 is used for all "random" values (pass 0 for mean and
std_dev)
+// if MKL is not enabled, a Gaussian distribution from std library is used
+
+#ifndef IS_USING_MKL
+#include <random>
+#endif
+
+#include <omp.h>
+
 float_point_precision *initialize_normal_dist(float_point_precision mean,
float_point_precision std_dev)
 {
 #if _MSC_VER && !__INTEL_COMPILER
@@ -208,8 +212,14 @@
 #endif
     vslDeleteStream( &vslstream);
 #else
+
+        std::random_device rd;
+        std::mt19937 gen{rd()};
+        std::normal_distribution<float_point_precision> d{mean, std_dev};
+
+#pragma omp parallel for
        for(int i=0; i<c_num_simulations*c_time_steps; ++i) {
```

```
-                zloc[i] = 0.3;
+                zloc[i] = d(gen);
         }
 #endif // IS_USING_MKL
diff --git a/src/monte_carlo.h b/src/monte_carlo.h
index 067f763..8704c4b 100644
@@ -22,6 +22,7 @@
 #include <cstdio>
 #include <cmath>
 #include <cassert>
+#include <tuple>

 #ifdef __INTEL_COMPILER
 #include <cilk/cilk.h>
@@ -36,9 +37,9 @@

 //GCC uses __attribute__(noinline) at end of function declaration
 //This #defines Microsoft's/Intel's __declspec(noinline) to nothing
-#ifdef __GNUC__
-#define __declspec(noinline)
-#endif
+//#ifdef __GNUC__
+//#define __declspec(noinline)
+//#endif

 //Determines whether float is single precision (float) or double precision (double)
 //This should be defined in the preprocessor as either DOUBLE or SINGLE
@@ -63,7 +64,9 @@
 const float c_zero = 0.0f;
 const float c_one_half = 0.5f;
 const float c_hundred = 100.0f;
+#ifdef IS_USING_MKL
 #define exp expf
+#endif
```

```
 #endif // SINGLE or DOUBLE floating point precision

@@ -71,7 +74,7 @@

 //Number of simulations that Monte Carlo runs
 //Must be a multiple of c_simd_vector_length
-const int c_num_simulations = 96000;
+const int c_num_simulations = 196000;
 //The interval at which the future LIBOR rate is recalculated
 //Otherwise known as the LIBOR interval
 const float_point_precision c_reset_interval = 0.25;
@@ -85,12 +88,15 @@
 const float_point_precision c_strike_prices[]  = {.045,.05,.055,.045,.05,.055,.045,.05,
                                                   .055,.045,.05,.055,.045,.05,.055 };

-//number of different possible forward rates calculated for the portfolio
-const int c_num_forward_rates=80;
+
 //Number of time steps each possible forward rate goes through
-//Each step uses the seed of a random number from a normal distrubition
+//Each step uses the seed of a random number from a normal distribution
 const int c_time_steps = 40;

+//number of different possible forward rates calculated for the portfolio
+//assumes the list of maturities is in ascending order
+const int c_num_forward_rates= c_time_steps + c_lengths[c_num_options-1];
+
 //Amount price varies over time
 //Typically determined as function of time to maturity
 //In this example, however, it remains constant
@@ -142,7 +148,7 @@
 //Calls calculate_path_for_swaption_kernel_array using a for loop
 //returns the average of all simulations
 __declspec(noinline)
-float_point_precision calculate_monte_carlo_paths_scalar(
```

```
+std::tuple<float_point_precision,float_point_precision>
calculate_monte_carlo_paths_scalar(
     float_point_precision *__restrict initial_LIBOR_rate,
     float_point_precision *__restrict volatility,
        float_point_precision *__restrict normal_distribution_rand,
diff --git a/src/simulations.cpp b/src/simulations.cpp
index 79b7fd8..da20fb4 100644
--- a/src/simulations.cpp
+++ b/src/simulations.cpp
@@ -20,28 +20,41 @@
 // one calls scalar code using cilk_for, and one calls array notation using cilk_for.

 #include "monte_carlo.h"
+#include <omp.h>

 // Description:
 // Calls scalar kernel using linear for loop
 // [in]: initial_LIBOR_rate, volatility, normal_distribution_rand,
discounted_swaption_payoffs
 // [out]: average discounted payoff
 __declspec(noinline)
-float_point_precision calculate_monte_carlo_paths_scalar(
+std::tuple<float_point_precision, float_point_precision>
calculate_monte_carlo_paths_scalar(
     float_point_precision *__restrict initial_LIBOR_rate,
     float_point_precision *__restrict volatility,
-       float_point_precision *__restrict normal_distribution_rand,
+    float_point_precision *__restrict normal_distribution_rand,
     float_point_precision *__restrict discounted_swaption_payoffs
 )
 {
+
+#pragma omp parallel for
     for (int path=0; path<c_num_simulations; ++path) {
         calculate_path_for_swaption_kernel_scalar(initial_LIBOR_rate, volatility,
```

```
                        normal_distribution_rand+(path*c_time_steps),
discounted_swaption_payoffs+path);
     }
-       float_point_precision total_payoff = c_zero;
-       for(int i=0; i<c_num_simulations; ++i) {
-               total_payoff += discounted_swaption_payoffs[i];
-       }
-    return total_payoff/c_num_simulations;
+
+    float_point_precision total_payoff = c_zero;
+    float_point_precision total_payoff2 = c_zero;
+
+    for(int i=0; i<c_num_simulations; ++i) {
+        total_payoff += discounted_swaption_payoffs[i];
+    }
+    total_payoff /= c_num_simulations;
+
+    for(int i=0; i<c_num_simulations; ++i) {
+        total_payoff2 += (discounted_swaption_payoffs[i] -
total_payoff)*(discounted_swaption_payoffs[i] - total_payoff);
+    }
+    total_payoff2 /= c_num_simulations;
+
+    return std::make_tuple(total_payoff, total_payoff2);
 }

 #ifdef __INTEL_COMPILER
@@ -62,9 +75,9 @@
                calculate_path_for_swaption_kernel_array(initial_LIBOR_rate, volatility,
                        normal_distribution_rand+(path*c_time_steps),
discounted_swaption_payoffs+path);
     }
-       for(int i=0; i<c_num_simulations; ++i) {
-               total_payoff += discounted_swaption_payoffs[i];
-       }
```

```
+    for(int i=0; i<c_num_simulations; ++i) {
+        total_payoff += discounted_swaption_payoffs[i];
+    }
     return total_payoff/c_num_simulations;
 }

diff --git a/src/kernel.cpp b/src/kernel.cpp
index 79b7fd8..da20fb4 100644
--- a/src/kernel.cpp
+++ b/src/kernel.cpp
@@ -39,7 +39,11 @@
     float_point_precision *discounted_swaption_payoffs
 )
 {
+#if __GNUC__
+    float_point_precision forward_LIBOR_rates[c_num_forward_rates] __attribute__
((aligned (64))) ;
+#else
     __declspec(align(64)) float_point_precision
forward_LIBOR_rates[c_num_forward_rates];
+#endif
     // initial_LIBOR_rate holds constant values, but could be filled with real, varied
values
        for(int i=0;i<c_num_forward_rates;++i) {
         forward_LIBOR_rates[i] = initial_LIBOR_rate[i];
```

ABOUT PRINCIPLED TECHNOLOGIES

Principled Technologies, Inc.
1007 Slater Road, Suite 300
Durham, NC, 27703
www.principledtechnologies.com

We provide industry-leading technology assessment and fact-based marketing services. We bring to every assignment extensive experience with and expertise in all aspects of technology testing and analysis, from researching new technologies, to developing new methodologies, to testing with existing and new tools.

When the assessment is complete, we know how to present the results to a broad range of target audiences. We provide our clients with the materials they need, from market-focused data to use in their own collateral to custom sales aids, such as test reports, performance assessments, and white papers. Every document reflects the results of our trusted independent analysis.

We provide customized services that focus on our clients' individual requirements. Whether the technology involves hardware, software, Web sites, or services, we offer the experience, expertise, and tools to help our clients assess how it will fare against its competition, its performance, its market readiness, and its quality and reliability.

Our founders, Mark L. Van Name and Bill Catchings, have worked together in technology assessment for over 20 years. As journalists, they published over a thousand articles on a wide array of technology subjects. They created and led the Ziff-Davis Benchmark Operation, which developed such industry-standard benchmarks as Ziff Davis Media's Winstone and WebBench. They founded and led eTesting Labs, and after the acquisition of that company by Lionbridge Technologies were the head and CTO of VeriTest.