Asian Option Pricing on Intel® MIC Architecture

Shuo Li
Developer Relations Division
Software and Services Group, Intel Corporation
2111 NE 25th Ave, Portland, OR 97124, USA
shuo.li@intel.com

James Lin
Center for High Performance Computing
Shanghai Jiaotong University
800 Dongchuan Rd, Rm 201 Shanghai, 200240 China
james@sjtu.edu

Abstract
In this paper, we discuss the problem of pricing one exotic option, the strong path dependent Asian option using the Black–Scholes model and compare how the pricing algorithm can be implemented on Intel® Many Integrated Core or MIC Architecture and achieve impressive performance gains. We can demonstrate that a 2-year contract with 252 times steps and 1,000,000 samples can be priced in approximately one fifth of a second on Intel® Xeon Phi™ 7120p.

Keywords: Parallel Processors, Statistical computing, Algorithm, Finance.

1 Introduction
In this paper, we look at exotic options and Asian options specifically. We start with the payoff function and the algorithm of pricing Asian options and give a Monte Carlo solution for the arithmetic average Asian option.

The focus of this paper would be on how to parallelize the Monte Carlo method on the Intel® Many Integrated Core architecture using a combination of vectorization and parallelization technique. We also include a brief introduction to the MIC architecture and programming model to facilitate the development of a high performance many-core algorithm.

We conclude the paper with a performance measurement of our algorithm on the first generation MIC product Intel® Xeon Phi™ Coprocessor 7210p.

European and American call and put options are what is termed plain vanilla options. They have standard well-defined properties and trade actively. Their prices or implied volatilities are quoted by exchanges or by interdealer brokers on a regular basis.

Nonstandard contracts on over-the-counter derivatives market are called exotic options, or simply exotics. These exotics are important to a derivatives dealer because they are generally much more profitable than plain vanilla options.

Exotic products are developed for a number of reasons. Sometimes they meet a genuine hedging need in the market; sometimes there are tax, accounting, legal, or regulatory reasons why corporate treasurers, fund managers, and financial institutions find exotic contracts attractive; Sometimes the products can also be designed to reflect a view on potential future movements in particular market variables.

2 Asian Options
Asian option is a typical exotic options, where the payoff depends on the average of the price of the underlying asset during the life of the option. Its name come from the first price averaging crude oil option contract the Banker’s Trust Tokyo office offered in 1987.

Since the Asian option’s underlying variable is averaged over a period of time, it has a lower volatility and hence renders much cheaper relative to their European vanilla counterpart. The pricing of Asian options becomes one of the most interesting area in Quantitative Finance because there is no closed-form analytical solution. Any numerical methods could applied, mostly Monte Carlo method and binomial.

In this paper, we demonstrate that for geometric averaging Asian options, we can leverage the Black-Scholes-Merton formula and derive a closed-form analytical solution. Second, we use a simple a raw Monte Carlo simulation method to price the arithmetic averaging Asian options.

In general, the payoff function an arithmetic averaging Asian call can be expressed as:

$$\Phi(S) = \mathbb{E} \left[ \frac{1}{T} \int_0^T S(t) dt - K \right]$$

While a geometric averaging Asian call can be express as:

$$\Phi(S) = \mathbb{E} \left[ \exp \left( \frac{1}{T} \int_0^T \log(S(t)) dt \right) - K \right]$$

In practice, we usually assume that the underlying price averaging takes place in discrete time. We further make the assumption that it happens m times a year. Then the payoff function for call and put in arithmetical and geometric have become

$$\Phi(S) = \mathbb{E} \left[ \frac{1}{m+1} \sum_{i=0}^{m} \left( \frac{S_i}{m} \right) - K \right]$$

Where m is the number of times a year the price averaging takes place; $S_0$ is the stock price at time 0, and $K$ is the option strike price and T is the time to expiry. We use $C^{\text{avg}}(S_0, K, T, \sigma, r)$ to represent asina call option for stock price $S_0$, strike price $K$ expire in T time with constant volatility of $\sigma$, and risk free rate of $r$. 

Copyright © 2015, Australian Computer Society, Inc. This paper appeared at 13th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2015), Sydney, Australia, January 2015. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 163. B. Javadi and S. K. Garg, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.
2.1 Geometric Averaging Asian Option

Since the Black-Scholes model assumes that the underlying asset follows a log-normal distribution in continuous time, the products of log-normal distributed random variables are also log-normal distributed. However, the same cannot be said about the sum. This leads us to an analytical solution for the geometric averaging Asian options.

Let’s $C^g$ be a vanilla European call option. We can prove that

$$C^g(S_0, K, T, \sigma_r, \rho) = e^{-\rho T} C_{0g}^g(S_0, K, T, \sigma_\sigma, \rho)$$

with $C^0(S_0, K, T, \sigma_\sigma, \rho)$ is the European options with risk free rate $\rho$ and the volatility $\sigma_\sigma$, where

$$d_2 = \frac{\ln \left( \frac{S_0}{K} \right) + \left( \rho - \frac{\sigma_\sigma^2}{2} \right) T}{\sigma_\sigma \sqrt{T}}$$

From Black-Scholes-Merton Formula, we have

$$C^g(S_0, K, T, \sigma_\sigma, \rho) = S_0 N(d_1) - K e^{-\rho T} N(d_2)$$

where $N(x)$ is the cumulative normal distribution function and $d_1$ and $d_2$ are defined in Black-Scholes-Merton Formula as:

$$d_1 = \frac{\ln \left( \frac{S_0}{K} \right) + \left( \rho + \frac{\sigma_\sigma^2}{2} \right) T}{\sigma_\sigma \sqrt{T}}$$

Now we have demonstrated that geometric averaging Asian options can be simply priced analytically using Black-Scholes-Merton Formula. We can now focus on making the arithmetic averaging Asian options tractable by using another numerical methods, such as Monte Carlo.

2.2 Arithmetic Averaging Asian Option

Monte Carlo uses statistical sampling method to arrive the numerical result. You can directly simulate the arithmetic average Asian options by using risk neutral valuation can be written as

$$C = \text{AsianOption} = \frac{\sum_{i=1}^{n} S(t_i)}{n}$$

Now we have demonstrated that geometric averaging Asian options can be simply priced analytically using Black-Scholes-Merton Formula. We can now focus on making the arithmetic averaging Asian options tractable by using another numerical methods, such as Monte Carlo.

3. Sequential Implementation

Our scalar implementation used a nested loops structure. The inner loop traverses through the asset time steps paths from beginning to expiry, while the outer loop creates many samples of such paths so that we can calculate an average value, based on the central limit thereon.

```c
sum = 0;
for i = 1 to n
    Generate S(t_i), S(t_i), S(t_i), ..., S(T);
    aveprice = S(t_i) + S(t_i) + S(t_i) + ... + S(T);
    presentValue = e^{-\rho T} \max(aveprice - K, 0)
    sum = sum + presentValue
end
```

3.1 Architecture Overview

Like the Intel® Xeon® Processor [3], the Intel® Xeon Phi™ Coprocessor is a shared memory multiprocessor environment. In the first product generation, each processor core supports 4 hyper threads and each thread has its own set of 32 512-bit registers. At the centre of each processor core is a 512-bit fully pipelined vector execution engine, which executes most arithmetic instructions with 4-cycle latency and up to 1-cycle throughput-put.

Intel MIC’s memory model is also consistent with that of the host processors. Each thread in a processor core has its own 8K L1 data and 8K L1 Instruction caches, but shares a 512KB L2 cache. MIC caches are coherent at all time. In the first product generation, up to 64 processor cores are connected in a ring architecture through their L2 cache and provides an abstract SMP model to application programmer. [6]

3.2 Programming Model

The Intel® Xeon Phi™ Coprocessor support initial Many-Core Instruction set or IMCI [7]. Intel® Parallel Studio XE 2013 provides C/C++, Fortran Compilers,
MKL Library and VTune Amplifier Performance Analyser.

Intel® Xeon Phi™ allows the application programmers to express two kinds of parallelism: SIMD parallelism offered via the 512-bit vector processing unit and thread parallelism via its 61-core/244 threads many-core architecture. Applications that run best on Intel MIC are those that can extract each of these two types of parallelism from its algorithm. Expressing thread level parallelism can be done by low level programming to native threads, or by using higher level abstractions such as OpenMP, TBB or Cilk™ Plus. SIMD level parallelism can be expressed either by low level intrinsics or by the new explicit vector programming capability which is part of the OpenMP 4.0 standard.

In the next section, we show that we can extract both SIMD parallelism and thread parallelism from pricing the arithmetic average Asian option.

4 Asian Options on Intel MIC Architecture

In arithmetic average Asian option pricing algorithm, it’s obvious that thread parallelism should be applied at the outer loop. SIMD parallelism can be applied at the inner loop. In general, if the inner loop is too small, it makes sense to apply it to the outer loop. In this case, since there is enough work in inner loop, and it can be vectorized, SIMD parallelism can be applied to the inner loop.

In our Asian option example, we apply SIMD parallelism to the inner loop, while thread parallelism to the outer loop.

4.1 Vectorize the Inner Loop

We can further optimize our inner loop so that no repeated computation exists inside the loop. Multiplication reduction can be converted to linear accumulation of the drift component and path calculation. Our inner loop became just 4 statements, 3 of which have a += operator. We can try to see if compiler can vectorize this loop.

```c
double AsianOption(double S, double r, double d, double v, double x, double T, double dT) {
    int N = T/dT;
    vsRngGaussian(VSL_RNG_METHOD_GAUSSIAN_ICDF, Randomstream, N, randn, 0.0, 1.0);
    double dr = (r - d - v*v/2.0)*dT;
    double accum_dr = 0.0;
    double vsqrtdT = v*sqrt(dT);
    double path = 0.0;
    for (int i = 0; i < N; i++){
        path += randn[i];
        accum_dr += dr;
        double ST = S * exp(accum_dr + vsqrtdT*path);
        cumulativePrice += ST;
    }
    double meanPrice = cumulativePrice / N;
    return max(0, meanPrice - x);
}
```

Intel Compiler could not auto vectorize this loop because the first statement has a loop dependency. Each iteration depends on the result of the previous iteration and varies by a different amount and is later being used within the loop. This by itself is not a complicated operation, but compiler does not understand this idiom and recognize the pattern. Moreover, the syntax in OpenMP 4.0 [16] for explicit vector programming does not provide syntax that would support this pattern. We attempt to solve this problem in the following subsection.

4.1.1 Prefix Sum

Prefix sum aka scan add, is a well-known problem in computer science. It takes one input sequence of numbers x0, x1, x2, ... and returns another sequence of numbers which are the sums of prefixes of the input sequence y0, y1, y2, ..., such that

y0 = x0
y1 = x0 + x1
y2 = x0 + x1 + x2

We can take advantage of IMCI or the ISA used by the first generation of MIC product. Intel C/C++ compiler makes it easier for the programmer to manipulate native vector data type by providing vector class libraries (3) which is a thin C++ wrapper for them. For example, F64vec8 is a thin wrapper for __m512d. Our first objective is to create a vector function that returns a vector of prefix_sum for its input sequence.

F64vec8 prefix_sum(F64vec8 v1);

To implement prefix_sum, we have to take advantage of Intel MIC’s ISA. The strategy is to use binary tree add so that only log2N add is need to add N numbers. We can use IMCI’s masks, swizzles and permute instructions;

In the first add, lane 2N is added to 2N+1 the result goes to 2N+1, where N= 0, 1, 2, 3. Use one swizzle pattern CDAB.

In the second add, lanes 4N+1 is added to lanes 4N+2 4N+3, where N= 0, 1. Use one swizzle pattern BBBB.

In the third add, lane 8N+3 is added to lanes 8N+4 8N+5 8N+6 and 8N+7 where N= 0. Get the value in lane 3 to lower 256 lanes many core/44 threads, then permute to the higher 256 lanes as if they were 32 bit integers.

In the vector function, we have to take advantage of Intel MIC’s ISA. The strategy is to use binary tree add so that only log2N add is need to add N numbers. We can use IMCI’s masks, swizzles and permute instructions;

In the first add, lane 2N is added to 2N+1 the result goes to 2N+1, where N= 0, 1, 2, 3. Use one swizzle pattern CDAB.

In the second add, lanes 4N+1 is added to lanes 4N+2 4N+3, where N= 0, 1. Use one swizzle pattern BBBB.

In the third add, lane 8N+3 is added to lanes 8N+4 8N+5 8N+6 and 8N+7 where N= 0. Get the value in lane 3 to lower 256 lanes, then permute to the higher 256 lanes as if they were 32-bit integers then packed double add with the right mask.

F64vec8 prefix_sum(F64vec8 v1) {
    F64vec8 v2 = v1;
    __m512d k1 = 170; //0 1 0 1 0 1 0 1
    v2 = _mm512_mask_add_pd(v1, k1, v1, _MM_SWIZ_REG_CDAB);
    __m512d k2 = 240; //0 0 1 0 0 1 1 1
    v2 = _mm512_mask_add_pd(v2, k2, v2, _MM_SWIZ_REG_BBBB);
    __m512d k3 = 240; //0 0 0 0 1 1 1 1
    v2 = _mm512_mask_add_pd(v2, k3, v2, _MM_SWIZ_REG_BBBB);
    return v2;
}
```

As we discussed earlier in the programming model section, we would have preferred to use explicit vector programming and code at the C/C++ level. However, that would require syntax support for partial sums, which we hope would be added into
4.1.2 Inner Loop
Having implemented \( \text{prefix\_sum}(F64vec8 \ v1) \), we can easily vectorize the rest of the inner loop
\begin{verbatim}
double AsianOptionSIMD(double S, double r, double d, double v,
                     double x, double T, double dT)
{
    _declspec (align(128))
    double ConstVec = *(F64vec8 *)&ConstArray[0];
    double dr = -r - d - \( \sqrt{v/2.0}\) *dT;
    F64vec8 Drvec = Drvec * ConstVec;
    double cumulativePrice = 0.0;
    for (int i = 0; i < N; i+=8)
    {
        F64vec8 RandVec = *(F64vec8 *)&randn[1];
        F64vec8 PrefixSumVec = PrefixSumVec + \( \text{prefix\_sum}(\text{RandVec}) \);
        F64vec8 StVec = StVec + \( \text{exp}(\text{Drvec} \times \text{PrefixSumVec}) \);
        Accumulatevec = Accumulatevec + StVec;
        PrefixSumVec = F64vec8(\text{prefix\_sum}(?)
        Drvec = Drvec + Drvec;
    }
    cumulativePrice = \text{add\_horizontal}(\text{Accumulatevec});
    double meanPrice = cumulativePrice / N;
    return exp(-r*T) * max(0, meanPrice - x);
}
\end{verbatim}
Vectorized inner loop improved the performance by a factor of 3.5. It’s not a factor of 8 because the 128-bit lane crossing operations are very expensive.

4.2 Parallelize Monte Carlo Method
The outer loop has no data dependency. The OpenMP directives work just fine with one caveat. The random number generator that we used in the baseline version needs to be revisited. Passing different seeds to the same generator to create multiple pseudorandom number generators designed especially for large scale Monte Carlo applications running in a distributed computing environment. Matsumoto [4] specified set of maximum of 6024 MT2203 pseudorandom number generators as an addition to the MT19937 generator. Every MT2203 generator in the set is mutually independent of each other and each of them has a repeating cycle equal to \( 2^{2203} \).

To define, create and initialize a random stream for each worker thread, you can use the following code sequence.

In OpenMP, \textit{parallel}-clause and \textit{for}-clause are different. OpenMP runtime creates the worker threads when it first sees the \textit{parallel}-clause. Code in the parallel section is executed on each thread and variables defined inside parallel section are private by default. It’s an ideal place for the code that setups the parallel execution environment, identifies the worker thread, earmark per thread input data, memory buffer allocation, etc. In our example, each thread needs to know its ID, based on that ID a clustered MT2203 random number stream is created.