Best Practices for Developing and Optimizing Threaded Applications

While threading can be a challenge, new software development tools help simplify the process by identifying thread correctness issues and performance opportunities. We present a methodology that has been used to successfully thread many applications and discuss tools that can assist in developing multi-threaded applications.

ABSTRACT
Microprocessor design is experiencing a shift away from a predominant focus on pure performance to a balanced approach that optimizes for power as well as performance. Multi-core processors are capable of greater performance with optimal power consumption by concurrently sharing work and executing tasks on independent execution cores. One technique to fully utilize multi-core processors is to thread the application to enable it to run on multiple processor cores. While threading can be a challenge, new software development tools help simplify the process by identifying thread correctness issues and performance opportunities. We present a methodology that has been used to successfully thread many applications and discuss tools that can assist in developing multi-threaded applications.

1. INTRODUCTION

This paper presents a threading methodology for design, implementation and debugging threads, which has been successfully used and refined on many large applications and libraries by Intel® software engineering teams. The threading methodology consists of four phases. The first phase focuses on performance analysis in order to identify performance optimization opportunities with the goal of optimizing the serial version of the application. This phase also includes discovering which functions in the call tree are potential candidates for introducing threads, so that the maximum benefits from threading will be realized. The second phase involves effectively converting your serial application to a parallel one. The third phase identifies possible correctness issues in your threaded application, such as data races and deadlocks. These threading specific bugs are almost impossible to detect without the aid of effective tools which have recently become available. Finally, the fourth phase looks for performance improvements in your threaded application.

This paper highlights tried and tested techniques to maximize productivity with commercial and open source tools for each phase of the threading methodology. The Black Scholes calculation, a classic application used in the financial services industry, will be used to illustrate the threading methodology. This paper summarizes our work on using the threading methodology to effectively thread a Black Scholes application; future articles will discuss each phase of the threading methodology in greater detail.

2. BLACK SCHOLES APPLICATION

Fischer Black and Myron Scholes seminal paper on the pricing of options had a major impact on the financial services industry, and in part led to the field of financial engineering. This paper uses the Black Scholes algorithm from the "Financial Recipes" website written by Prof. Bernt A Ødegaard, of the Norwegian School of Management and the Central Bank of Norway. The "Financial Recipes" website is located at http://finance.bi.no/~bernt/gcc_prog. In particular, we calculate European call options, a specific class of Black Scholes calculations. The application first generates data to be used in the European call option calculation using standard Monte Carlo simulation techniques (source code also available at the Financial Recipes website), and then performs the Black Scholes calculation on the Monte Carlo data. The application is expected to scale well after threads are introduced, as each calculation is independent, and we should be able to effectively take advantage of additional processors.

2.1 Amdahl's law

Amdahl's law predicts the estimated speedup that can be achieved by converting a serial application to a parallel application. This first step estimates how much benefit you can achieve by threading your application on a particular processor. Also, it gives an estimate of how well the application scales as more processors/cores are added.

```
Tparallel = {(1-P) +P/Number of processors} Tserial + overhead
```

Tserial refers to the time taken to run the application before introducing any threads. Let's assume that we decide to thread a particular region of code, so the P in Amdahl's law refers to the fraction of the calculation that can be parallelized. So, the fraction of the time that is not threaded remains serial and it's given by (1-P). A certain amount of overhead is introduced while parallelizing the application due the synchronization constructs.

```
Scaling = Tserial / Tparallel
```

2.2 System Configuration
Our testing environment consisted of Intel® Xeon® processor 5160 (4GB RAM, two 3.0 GHz dual core processors). The 64-bit operating system was Red Hat Fedora Core release 5 (Bordeaux) with 2.6.15 kernel. The 32 bit application was compiled with Intel® C++ Compiler for Linux* 9.1.045, and g++ 4.1.0 20060304 supplied with Fedora Core 5.

3. THREADING METHODOLOGY
The process of converting a serial to threaded application can be daunting and time consuming without following the proper methodology and without the aid of the right software development tools. The threading methodology discussed in this paper has been effectively used for many years to thread high performance applications. Also, this is an iterative process which you can continue until you have achieved the desired performance.

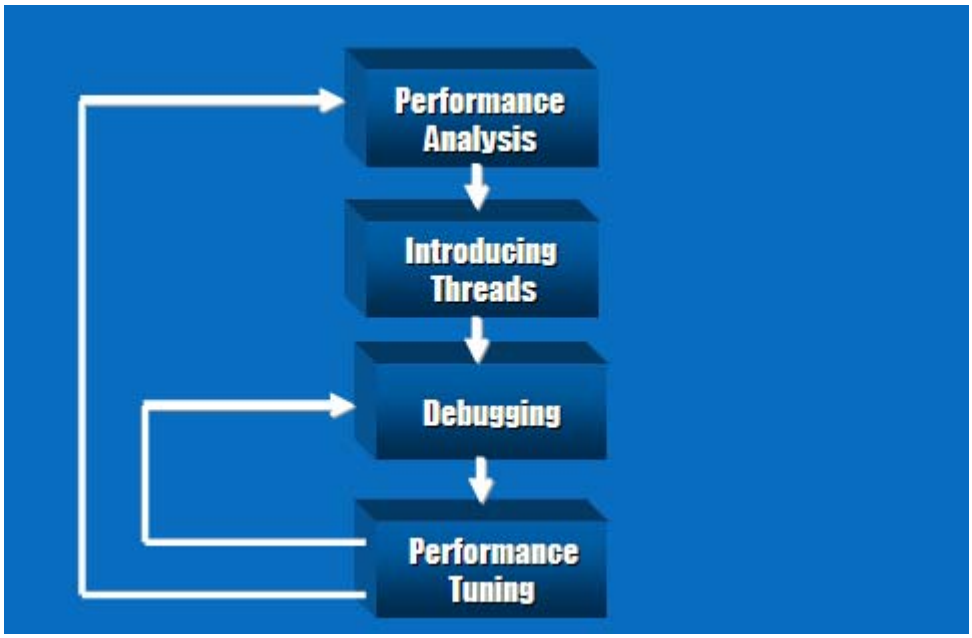

Figure 1. Illustrates the Steps in Threading Methodology

3.1 Performance Analysis
The performance analysis phase is the first step of this threading methodology and deals with optimizing the serial application before introducing threads. There are several tools, both commercial and open source, that are available that can help you understand the application's performance.

Establishing the performance baseline is a critical first step in the performance analysis phase. It includes determining which workload(s) will be used to measure application performance, and create build configurations, which include compiler optimizations and possibly third-party libraries. While establishing a performance baseline sounds obvious and trivial, we find in practice developers without experience in performance optimization can encounter problems at this stage, which can compromise the performance analysis. These potential problems are more common on large application development teams and can occur when workload(s) change and when previous studies made with different workload(s) are not reevaluated. Another common mistake is to use updated source code without determining if source code changes altered the performance of the application. It is best to finish the performance evaluation with a consistent source release, and then investigate the changes a new source release causes. Another potential problem is selecting a workload that isn't representative of how the application is typically used, so wrong parts of the application are tuned. In addition, workloads need to be repeatable. Once a performance baseline is obtained, the next step in the performance analysis is to profile the application to identify where the application is spending the most time. An important use of profile data is to determine the subset of the application that is performance critical and thus focus efforts on optimizing these parts of the application. One can improve the serial performance through source code modifications, compiler optimizations and by using high performance libraries.

The initial serial application performance used in our study is 12.9 seconds using g++ with -O2.

3.2 Black Scholes Performance Analysis
The first step in performance analysis phase is to measure and understand the serial application performance. The Black Scholes application is a "number crunching" application operating on floating point data. For small applications, it is possible to look at the source to determine the nature of the application and where the computational bottlenecks are expected. While programmers may have intricate knowledge of the application, it can be difficult to predict performance bottlenecks, as they often depend on the memory system and in particular the various memory caches. A good profiling tool will accurately measure exactly where the

system is spending time. For larger applications, using a profiling tool is essential to find the performance bottlenecks.

Intel® VTune™ Performance Analyzer and Oprofile* helped identify the hotspots in the application and where we can introduce threads. Both support sampling technology, which allows profiling measurement with low overhead so as to not perturb the system. With respect to the Black Scholes application, `random_uniform_0_1` and `random_normal_mean_sigma` are the hot functions, which would benefit from performance optimization and the introduction of threads.

We established a performance baseline after we finished writing the application, and agreed on a sample workload to measure performance. Next, we took performance data with both VTune™ Performance Analyzer and Oprofile, although we could have used either profiler for this measurement. The profiling tools identified two bottlenecks, the first in the generation of simulated data using standard Monte Carlo techniques and the second the Black Scholes calculation itself.

Since the bottlenecks were manipulating floating point numbers, we tried compiler optimizations that improve floating point performance. We used the GNU g++ compiler version 4.1.0 and the Intel® C++ Compiler for Linux*, version 9.1. Table 1 shows the results of using different compilers and compiler optimizations to improve the application performance. For this application, the Intel® C++ compiler generates faster code than g++. This study points out that by using more aggressive optimizations that help optimize floating point calculations we are able to improve the g++ performance by 24% compared to g++ -O2, while for the Intel ® Compiler we improved performance by 12% compared to the icpc –O2. The purpose of this phase of the performance analysis is to generate the fastest serial version of the application.

An important point is that a compiler optimization can either help or hurt performance, don't assume a given optimization increases performance. It is best to evaluate the performance when changing compiler options. For example, the last g++ build added `'-funroll-all-loops –ftree -msse3'`. A previous study done on a different processor found slightly improved performance when adding these options, but on this processor the performance decreased slightly. It is common for all compilers to generate code with different performance benefits on different processors; in general the changes are small but can be substantial when running on processors with different micro-architectures.

Table 1. Performance with different compilers and compiler optimizations for g++ and icpc.

Compiler and Optimizations	Total Time (s)	Create Data Time (s)	Black Scholes Time (s)
`g++ -O2`	12.9	9.6	3.3
`g++ -O3`	12.7	9.4	3.3
`g++ -O3 -finline-functions -ffast-math`	10.4	7.6	2.8
`g++ -O3 -finline-functions -ffast-math -funroll-all-loops –ftree -msse3`	10.9	8.1	2.8
`icpc -O2`	10.4	8.7	1.7
`icpc -O2 -prof-use -ipo –xW`	9.5	8.1	1.4
`icpc -O3 -prof-use -ipo –xW`	9.3	7.9	1.4

One of the features of the VTune™ Performance Analyzer is Call Graph Profiling which easily indicates the critical path in the application. This can be used to determine if it would be better to introduce threads further up the call tree, instead of threading the child functions. In this case, we used the VTune Call Graph Profiling data to determine it would be best to thread `generateBSdataRand`. This allows each thread to do more work, that is, at a coarser grain level of parallelism and more efficiently use the available processor resources.

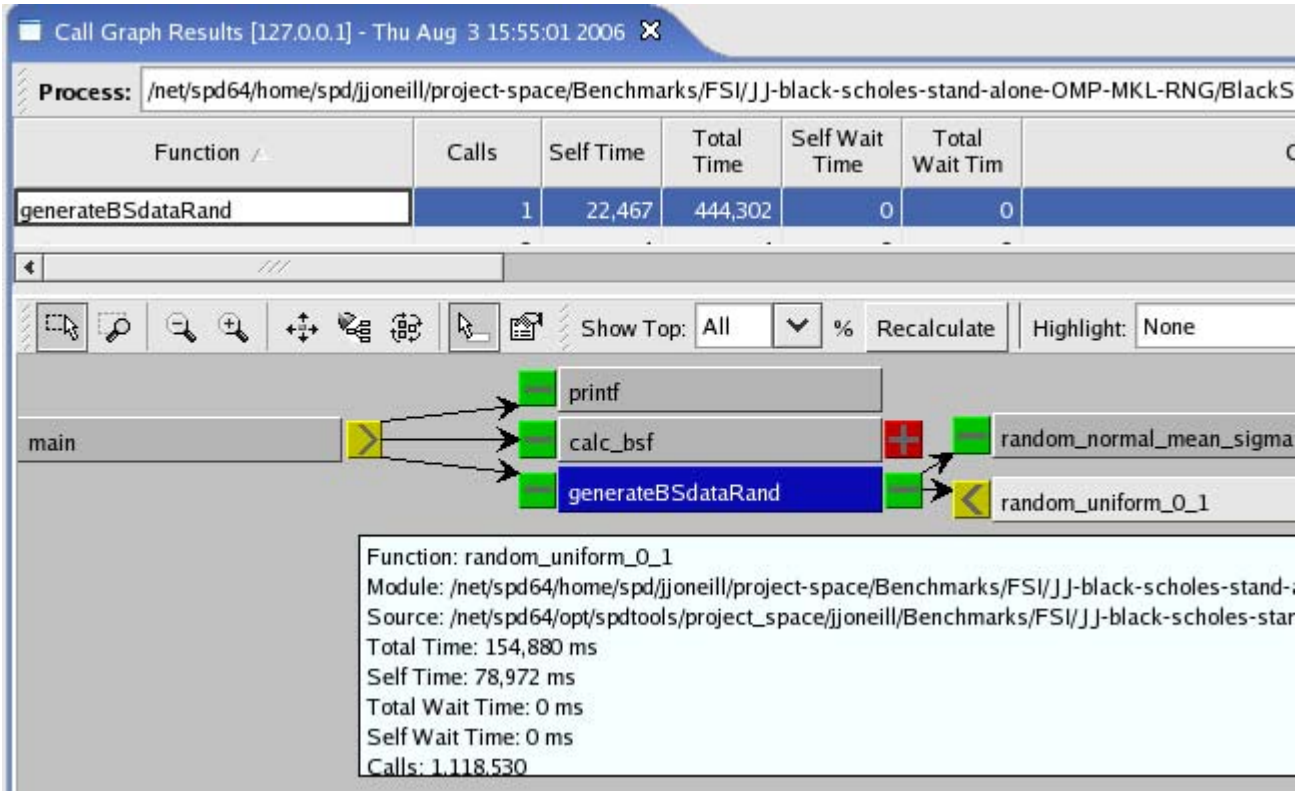

Figure 2. Intel® VTune™ Performance Analyzer Call Graph Profiler which suggests it would be beneficial to thread the `generateBSdataRand( )` function.

The performance analysis phase allowed us to optimize the serial application and suggest where to introduce threads. We use `'icpc -O3 -prof-use -ipo -xW'` as our optimized serial baseline, resulting in a runtime of 9.3 seconds, a speedup of 39% over the gcc –O2 performance.

3.3 Introduce Threads

The second step in the threading methodology is to introduce threads based on the performance analysis results. There are different options that be used to thread your application. You can use already threaded third-party libraries, Intel® Threading Building Blocks, Pthreads*, OpenMP* or Windows* threads.

Each of these threading models has its own strengths and weakness that should be carefully considered by the developer before threading the application.

For our experiments we used OpenMP* threads which also can be used as a rapid way of prototyping multithreaded applications. Since OpenMP* is suited for coarse grain parallelism, we would get a better scaling if we thread one level above the hotspot pointed in the call graph data. So, we threaded the `generateBSdataRand( )` function.

```
void generateBSdataRand(int Iter, double* x, double* c,
          double* t, double* variance, double* rate, int Max)
{

#pragma omp parallel for
  for (int i=0; i<Iter; i++)
          for (int j=0; j<Max; j++)
          {
             x[i*Max+j]=random_normal_mean_sigma( 75.0, 10.0 );
             c[i*Max+j]=random_normal_mean_sigma( 80.0, 10.0);
             t[i*Max+j]=random_uniform_0_1();
             variance[i*Max+j]=random_normal_mean_sigma(0.50,0.20 );
             rate[i*Max+j]=random_normal_mean_sigma(0.05 ,0.015 );
      }
}
```

Listing 1.

In addition, we also threaded the function that does the Black Scholes calculation with OpenMP. Figure 3 shows the results of introducing OpenMP threads on the application; however, instead of showing a performance gain after introducing threads, it shows performance degradation. It is well know that the system `rand ( )` function isn't reentrant or thread-safe (view the `rand` man pages for additional information), so it isn't surprising to see poor scaling when using the `rand ( )` random number generator. In addition to measuring the scaling of the entire application, we measured the scaling of the generation of the data and the Black Scholes calculation separately. As expected, we find very poor scaling for the generation of the data, and excellent scaling on the Black Scholes calculation (3.99x scaling on 4 processors). The fact that we measure excellent scaling on part of the application indicates we have threaded part of the application effectively, and we need to determine how to thread the entire application effectively.

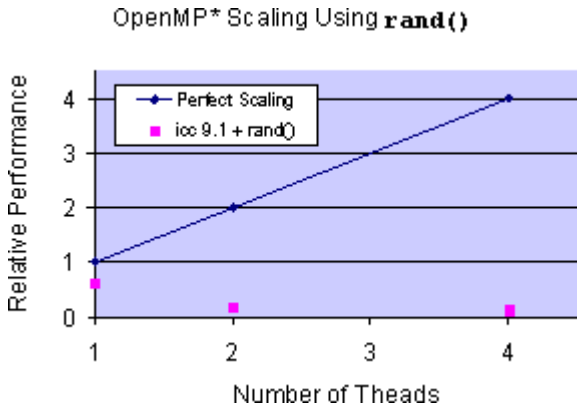

Figure 3. Initial poor scaling results using `rand` for random number generation, illustrating the well known performance problems encountered when using non-thread safe functions..

3.4 Debugging

After threading the Black Scholes application, the next obvious question to ask would be if any bugs were introduced. There are several software development tools available to help debug threaded applications, including Intel® Thread Checker, Valgrind's Helgrind*, and Frysk*. For this application, we used Intel® Thread Checker and Helgrind, both of which pointed out diagnostic warnings related to the `rand()` function.

ID	Short Description	Severity	Count	Context[Best]	Description	1st Access[Best]	2nd Access[Best]
1	Non-reentrant API function	Error	2	[BlackScholes_RAND_OMP, 0x1358]	A non-reentrant API function at [BlackScholes_RAND_OMP, 0x135b] conflicts with the API function call at [BlackScholes_RAND_OMP, 0x135b]	[BlackScholes_RAND_OMP, 0x135b]	[BlackScholes_RAND_OMP, 0x135b]
2	Thread termination	Information	1	Whole Program 1	Thread termination at [BlackScholes_RAND_OMP, 0xcda] - include stack allocation of 1.971 MB and use of 5.246 KB	[BlackScholes_RAND_OMP, 0xcda]	[BlackScholes_RAND_OMP, 0xcda]

Figure 4. Intel® Thread Checker output, showing the error diagnostics in the `rand ( )` function.

Looking closely at the source code of the `rand` function we see that there is a mutex lock used inside the `rand` code. So, this is an example of false positive, due to both tools not recognizing the locking primitives written in assembly. However, the mutex lock in the `rand` creates contention between threads on this lock preventing the application from effectively scaling on the multi-core processors. So another technique to generate random numbers is required.

3.5 Performance Tuning

To avoid the well known problems using `rand()` for threaded applications, we explored using the random number generators provided with the Intel® Math Kernel Library (MKL). The Intel® MKL library provides very high performance random number generators for a large number of random number distributions with various algorithms available for generating the random numbers. In addition, the Intel® MKL random number generators are thread safe, therefore we expect the threaded application performance to be much greater.

After we introduced the Intel® MKL random number generator instead of using `rand`, we measured a large increase in performance. The total application time for the non-threaded application is 2.5 seconds using Intel® MKL, compared to 10.4 seconds when using icpc –O2 with `rand` – a speedup of over 4x just by using an optimized random number generator. We then introduced OpenMP* threads to the Intel® MKL random number version of the application, and we found much improved scaling, which will be discussed following the debugging section below. The following code introduces OpenMP with Intel® MKL random number generator.

```
void generateBSdataMKL(int Iter, double* x, double* c, double* t,
                       double* variance, double* rate, int Max)
{

#pragma omp parallel
{
          vslNewStream( &stream_x, VSL_BRNG_MCG31, 1 );
          …
          #pragma omp for
          for (int i=0; i<Iter; i++)
          {

          vdRngGaussian( VSL_METHOD_SGAUSSIAN_BOXMULLER2,
          stream_x, Max, &x[i*Max], 75.0f, 10.0f );
        …

          }
          …
}
}
```

Listing 2.

3.6 Debugging

After iterating back to the debugging phase and running the application threaded with Black Scholes*, Intel® Thread Checker pointed out a read-write data race for the variables used in `vslNewStream`, which was resolved by making these variables private. A future paper will discuss the techniques to identify and solve the diagnostics displayed by Intel® Thread Checker.

```
#pragma omp parallel private
(stream_x, stream_c, stream_t, stream_variance, stream_rate)
{
          vslNewStream( &stream_x, VSL_BRNG_MCG31, 1 );
  …
          #pragma omp for
          for (int i=0; i<Iter; i++)      {
          vdRngGaussian( VSL_METHOD_SGAUSSIAN_BOXMULLER2,
          stream_x, Max, &x[i*Max], 75.0f, 10.0f );
```

Listing 3.

4. Final Performance Analysis

After fixing the bug in our OpenMP* code, we show the final scaling results in Figure 5 when using the Intel® MKL random number generator. This demonstrates good scaling on 2 processors/cores (1.9x), and 4 processors/cores (3.3x). The 4 thread version of the application ran in 0.76 seconds, which is ~14x faster than the initial application built with g++ -O2 using the rand function to generate random numbers. To further improve the scaling on 4 or more processors, the threading methodology presented in this paper can be used to understand what is causing the bottlenecks, and help identify solutions. We expect improved scaling could be attained by adjusting the algorithm to more efficiently take advantage of the large cache available on

Intel® Xeon® processor 5160-based systems, as well studying micro-architecture performance counters available in Intel® VTune™ Performance Analyzer that we will discuss in a future paper.

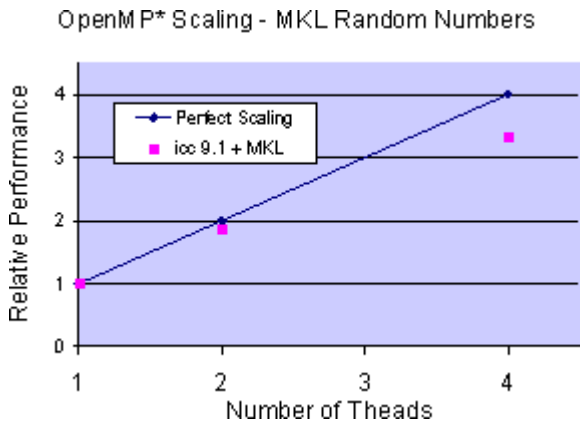

Figure 5. Scaling results when using Intel® MKL thread safe high performance random number generators, showing good scaling on 2 and 4 processors.

5. CONCLUSION

In this paper, we demonstrated a software development methodology to effectively thread applications and used a Black Scholes application to demonstrate the methodology. The final application built with the Intel Compiler and Intel MKL random number generator was ~14x faster than the single threaded g++ -O2 version. The threading methodology is generic and can be applied to any application or library. The software industry provides a wide range of tools that are useful in different stages of the software development cycle to help effectively thread applications. Explore them to effectively and efficiently thread your application to take advantage of multi-core processors. Future articles will discuss each phase of the threading methodology in detail.

6. REFERENCES

1. Intel® VTune™ Performance Analyzer
2. Intel® Thread Checker
3. Intel® Math Kernel Library
4. Intel® Compilers
5. GCC, the GNU Compiler Collection
6. Additional information on OpenMP
7. Additional information on Oprofile* is available online
8. Additional information on Valgrind* is available online
9. Fischer Black and Myron Scholes, 'The Pricing of Options and Corporate Liabilities', 1973, Journal of Political Economy, volume 81, issue 3, pp. 637-654.
10. "Financial Recipes" contains a wealth of information on financial engineering including sample source code.

192.55.52.2 devxweb01