NMath Performance Benchmarks

CenterSpace™

Software

Introduction

CenterSpace Software's **NMath**™ numerical library provides object-oriented components for mathematical, engineering, scientific, and financial applications on the .NET platform. **NMath** contains vector classes, matrix classes, complex number classes, random number generators, and other high-performance functions for object-oriented numerics.

NMath provides a modern, easy to use, object-oriented interface, including a very rich set of matrix and vector manipulation semantics. Fully compliant with the Microsoft Common Language Specification (CLS), all **NMath** routines are callable from any .NET language, including C#, Visual Basic.NET, and F#.

For most computations, **NMath** uses the Intel® Math Kernel Library (MKL), which contains highly-optimized, extensively-threaded versions of the C and FORTRAN public domain computing packages known as the BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage). This gives **NMath** classes performance levels comparable to C, and often results in performance an order of magnitude faster than non-platform-optimized implementations. There is, however, a small overhead relative to C or C++ associated with making function calls from managed **NMath** code to unmanaged native code.

This paper compares the performance of **NMath** to both straight C/C++ and straight C# in a series of single-threaded and multithreaded benchmarks that carry out matrix multiplication for matrices filled with random numbers. Tests were performed on square matrices of varying sizes, and with varying numbers of repetitions.

Design

We compared the performance of **NMath** to both straight C/C++ and straight C# in a series of single-threaded and multithreaded benchmarks that carry out matrix multiplication for matrices filled with random numbers.

NMath

We used the following **NMath** test code, which uses the
`CenterSpace.NMath.Core.DoubleMatrix` class:

```
double MatrixInnerProduct(int size, int reps) {
  int rows = size;
  int cols = size;
  double cumTime = 0;

  DoubleMatrix A = new DoubleMatrix(rows, cols);
  DoubleMatrix B = new DoubleMatrix(rows, cols) ;
  DoubleMatrix C = new DoubleMatrix(rows, cols);

  Random r = new Random(0x124);
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
      A[i, j] = r.NextDouble();
      B[i, j] = r.NextDouble();
    }
  }

  DateTime start;
  DateTime finish;
  TimeSpan elapsedTime;

  for (int i = 0; i < numRepeats; ++i) {
    start = DateTime.Now;
    for (int j = 0; j < reps; ++j) {
      NMathFunctions.Product(A, B, C);
    }
    finish = DateTime.Now;
    elapsedTime = finish - start;
    cumTime += elapsedTime.TotalSeconds;
  }
  double avgTime = cumTime / (double)numRepeats;
  return avgTime;
}
```

For single-threaded tests, we referenced the single-threaded **NMath** assemblies;
for multithreaded tests, we referenced the multithreaded **NMath** assemblies.

Straight C/C++

We compared the performance of **NMath** against the following C++ code which calls the MKL implementation of the `dgemm` BLAS routine directly:

```
double MatrixInnerProductTest( int size, int reps ) {
  int rows = size;
  int cols = size;
  int numRepeats = 10;
  double cumTime = 0;

  double* A = new double[rows*cols];
  double* B = new double[rows*cols];
  double* C = new double[rows*cols];

  srand( 0x124 );
  for ( int i = 0; i < rows; ++i ) {
    for ( int j = 0; j < cols; ++j ) {
      A[i*cols + j] = (double(rand())/RAND_MAX);
      B[i*cols + j] = (double(rand())/RAND_MAX);
    }
  }

  double alpha = 1.0;
  double beta = 1.0;
  char transa = 'N';
  char transb = 'N';

  clock_t start, finish;
  double duration;

  for ( int j = 0; j < numRepeats; ++j ) {
    start = clock();
    for ( int i = 0; i < reps; ++i ) {
      dgemm( &transa, &transb, &rows, &cols, &cols, &alpha, A,
             &cols, B, &cols, &beta, C, &cols );
    }
    finish = clock();
    duration = (double)(finish - start)/CLOCKS_PER_SEC;
    cumTime += duration;
  }
  double avgTime = cumTime/(double)numRepeats;
  return avgTime;
}
```

NOTE—Though compiled with a C++ compiler, this is essentially just C code, since no virtual functions or classes are used.

For single-threaded tests, we linked in the single-threaded MKL libraries; for multithreaded tests, we linked in the multithreaded MKL libraries.

Straight C#

Finally, to compare the performance of **NMath** to straight C#, we used the following simple method for performing matrix/matrix multiplication:

```
static void MatrixMult(int size, double[][] m1, double[][] m2,
                       double[][] result) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      result[i][j] = 0;
      for (int k = 0; k < size; k++)
      {
        result[i][j] += m1[i][k] * m2[k][j];
      }
    }
  }
}
```

For multithreaded tests, we used the `Parallel.For` construct from the Task Parallel Library (TPL), the next generation of concurrency support for .NET.[1] (TPL requires .NET Framework 3.5 and higher.)

```
static void MatrixMult( int size, double[][] m1, double[][] m2,
                        double[][] result ) {
  Parallel.For( 0, size, delegate( int i ) {
    for (int j = 0; j < size; j++) {
      result[i][j] = 0;
      for (int k = 0; k < size; k++) {
        result[i][j] += m1[i][k] * m2[k][j];
      }
    }
  } );
}
```

The same timing harness was used as for the **NMath** tests code.

Results

Tests were performed on square matrices of varying sizes, and with varying numbers of repetitions. Each test was run 10 times and the average time was computed.

The machine used was a 2.8GHz Intel Core i7-930 quad core, with 8GB PC3 8500 DDR3 SDRAM, running 64-bit Microsoft Windows 7 Ultimate.

---

[1]Leijen, Daan and Hall, Judd. "Parallel Performance: Optimize Managed Code For Multi-Core Machines". http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

Table 1 summarizes the results for single-threaded tests; Table 2 for multithreaded tests.

Table 1 – Single-threaded matrix/matrix multiplication

Label	Size	Reps	C++ (sec)	NMath (sec)	C# (sec)
XS	2	500,000	0.0748	0.0671	0.0272
S	5	100,000	0.0250	0.0515	0.0538
M	10	50,000	0.0437	0.0468	0.1806
L	100	1,000	0.1919	0.2044	3.4780
XL	1000	1	0.1810	0.1825	7.9482

Table 2 – Multithreaded matrix/matrix multiplication

Label	Size	Reps	C++ (sec)	NMath (sec)	C# (sec)
XS	2	500,000	0.0780	0.0671	1.9484
S	5	100,000	0.0280	0.0515	0.4976
M	10	50,000	0.0422	0.0515	0.5850
L	100	1,000	0.0702	0.0858	1.1638
XL	1000	1	0.0577	0.0671	2.0436

Discussion

The data in Table 1 and Table 2 demonstrate that by using the highly-optimized MKL for most computations, **NMath** offers significantly higher performance than a straight C# implementation, especially for larger matrices. For example, the average time for multiplying 1000x1000 matrices using the C# matrix code averaged over **43 times slower** than **NMath** running single-threaded, and **30 times slower** than **NMath** running multithreaded. Figure 1 plots the results of the single-threaded tests as a function of matrix size.

Figure 1 – Single-threaded matrix/matrix multiplication

Single-threaded Matrix/Matrix Multiplication
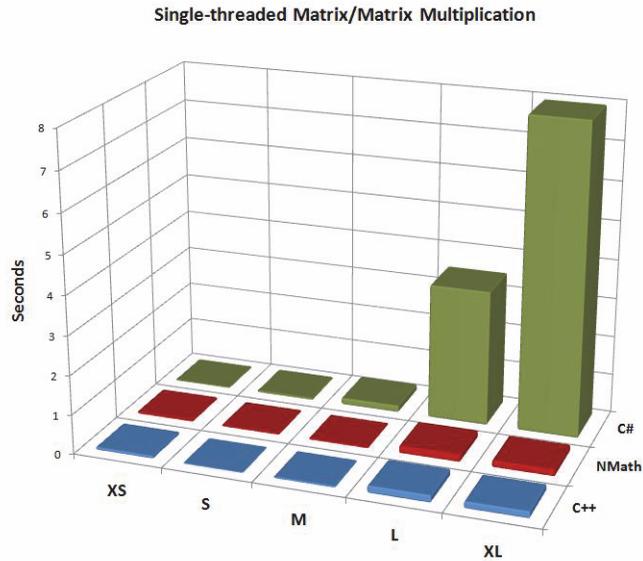
The data also show the negligible overhead relative to straight C or C++ involved in invoking MKL from managed .NET code.

Conclusion

NMath provides a powerful, easy to use, object-oriented interface to standard numerical routines. Fully CLS-compliant, all **NMath** routines are callable from any .NET language, including C#, Visual Basic.NET, and F#. By using MKL for most computations, **NMath** offers performance levels comparable to straight C or C++, and often an order of magnitude faster than non-platform-optimized implementations.