Parallel Performance of the IMSL C# Numerical Library

A White Paper by Rogue Wave Software.

April 2010
Parallel Performance of the IMSL C # Numerical Library

by Rogue Wave Software

© 2012 by Rogue Wave Software. All Rights Reserved

Printed in the United States of America

Publishing History:

April 2010

Trademark Information

The Rogue Wave Software name and logo, SourcePro, Stingray, HostAccess, IMSL and PV-WAVE are registered trademarks of Rogue Wave Software, Inc. or its subsidiaries in the US and other countries. JMSL, JWAVE, TS-WAVE, PyIMSL and Knowledge in Motion are trademarks of Rogue Wave Software, Inc. or its subsidiaries. All other company, product or brand names are the property of their respective owners.

IMPORTANT NOTICE: The information contained in this document is subject to change without notice. Rogue Wave Software, Inc. makes no warranty of any kind with regards to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Rogue Wave Software, Inc. shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.
# TABLE OF CONTENTS

Parallel Performance of the IMSL C# Numerical Library ................................................................. 4  
  Thread-Safe User Functions ........................................................................................................ 4  
  Amdahl’s Law .......................................................................................................................... 4  
Measurements ................................................................................................................................ 5  
  Microsoft Windows Server 2008 R2 Standard (64-bit)........................................................ 5  
  Microsoft Windows Server 2003 R2 Standard (32-bit)....................................................... 10  
Test Problems ............................................................................................................................... 15  
  Quadrature Functions ............................................................................................................. 15  
  Optimization Neural Network Problem ............................................................................... 15  
  Full Set of Benchmark Problems ......................................................................................... 15  
Sample Benchmarking Code ....................................................................................................... 18  
Conclusion ..................................................................................................................................... 21
Parallel Performance of the IMSL C# Numerical Library

The IMSL C# Numerical Library uses the Task Parallel Library (TPL) in .NET 4.0 to enable parallelism on shared memory systems, especially multi-core systems. Starting with Version 6.5 of the IMSL C# Numerical Library, released in April 2010, codes using TPL were added to a variety of classes in the library. The goal for the release was to take advantage of multi-core systems while minimizing impact on existing user code that references the IMSL C# Numerical Library and also minimizing the engineering resources in developing the release. Under these constraints, existing algorithms in the library were not re-written, but instead parallelized by adding TPL codes wherever sensible.

The performance of selected IMSL C# Numerical Library classes was measured on multi-core systems. Each class was used to solve a large enough problem to allow for parallelism. Each test case was run with a varying number of processors allowed. The maximum number of processors allowed was set using the `NumberOfProcessors` property in the appropriate class. The specific problems run are described at the end of this paper.

**Thread-Safe User Functions**

Many of the parallelized IMSL C# Numerical Library classes can evaluate user-defined methods in parallel. This requires that the user’s methods be thread-safe. By default, the IMSL C# Numerical Library assumes user-defined methods are thread-safe and will evaluate such methods in parallel. If the user-defined methods are not thread-safe, they should be flagged by setting the `Parallel` property in the appropriate class to `false`. Otherwise, the behavior is unpredictable.

For example, to flag the user-defined method is not thread-safe for the `MinUnconMultiVar` class, use:

```csharp
MinUnconMultiVar.Parallel = false;
```

**Amdahl’s Law**

Amdahl’s Law relates the speedup using parallel processors versus using only one serial processor. The speedup, $S$, using $N$ processors is

$$S_n = \frac{1}{(1 - P) + \frac{P}{N}}$$

where $P$ is the fraction of the code which is parallel. Even if an unlimited number of processors were available, the maximum speedup is limited to

$$S_\infty = \frac{1}{(1 - P)}$$
For example, if 80% of the code is parallelized, then the maximum speedup is 5.

The IMSL C# Numerical Library nonlinear regression class, `NonlinearRegression`, can be used to estimate the fraction of the code running in parallel, $P$, from the measured speedups at various values of $N$. This is only an approximation because the fraction of the code which is parallelized can be a function of $N$, but it allows for easy evaluation of the performance gained for each function.

**Measurements**

Each benchmark was run five times, and the reported results are the average elapsed wall clock time over all the five runs.

**Microsoft Windows Server 2008 R2 Standard (64-bit)**

Hardware: 2 Quad Core Intel Xeon CPU (8 cores in total) 2.33GHz, 8 GB RAM
Compiler: Microsoft Visual Studio 2010 (Any CPU project)

<table>
<thead>
<tr>
<th>Name</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>Fraction Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterHierarchical</td>
<td>5.952</td>
<td>3.959</td>
<td>2.811</td>
<td>2.523</td>
<td>2.428</td>
<td>0.69</td>
</tr>
<tr>
<td>Dissimilarities</td>
<td>54.632</td>
<td>27.74</td>
<td>14.939</td>
<td>10.755</td>
<td>8.575</td>
<td>0.96</td>
</tr>
<tr>
<td>FactorAnalysis</td>
<td>111.317</td>
<td>92.832</td>
<td>84.064</td>
<td>81.508</td>
<td>80.259</td>
<td>0.32</td>
</tr>
<tr>
<td>AutoCorrelation</td>
<td>11.423</td>
<td>5.713</td>
<td>2.855</td>
<td>1.91</td>
<td>1.458</td>
<td>1</td>
</tr>
<tr>
<td>CrossCorrelation</td>
<td>42.406</td>
<td>21.213</td>
<td>10.534</td>
<td>7.07</td>
<td>5.275</td>
<td>1</td>
</tr>
<tr>
<td>MultiCrossCorrelation</td>
<td>47.291</td>
<td>23.712</td>
<td>12.349</td>
<td>9.151</td>
<td>6.802</td>
<td>0.98</td>
</tr>
<tr>
<td>ARAutoUnivariate</td>
<td>13.204</td>
<td>7.819</td>
<td>5.158</td>
<td>4.003</td>
<td>3.412</td>
<td>0.84</td>
</tr>
<tr>
<td>ARMA</td>
<td>21.563</td>
<td>12.387</td>
<td>7.495</td>
<td>5.816</td>
<td>4.773</td>
<td>0.88</td>
</tr>
<tr>
<td>NextGaussianCopula</td>
<td>0.518</td>
<td>0.421</td>
<td>0.449</td>
<td>0.427</td>
<td>0.409</td>
<td>0.23</td>
</tr>
<tr>
<td>NextStudentsTCopula</td>
<td>0.468</td>
<td>0.427</td>
<td>0.449</td>
<td>0.424</td>
<td>0.412</td>
<td>0.11</td>
</tr>
<tr>
<td>NextMultivariateNormal</td>
<td>0.471</td>
<td>0.424</td>
<td>0.408</td>
<td>0.405</td>
<td>0.43</td>
<td>0.15</td>
</tr>
<tr>
<td>CanonicalCorrelation</td>
<td>507.301</td>
<td>259.258</td>
<td>135.717</td>
<td>94.174</td>
<td>70.11</td>
<td>0.98</td>
</tr>
<tr>
<td>FaureSequence</td>
<td>5.451</td>
<td>2.727</td>
<td>1.367</td>
<td>0.91</td>
<td>0.684</td>
<td>1</td>
</tr>
<tr>
<td>Covariances</td>
<td>70.092</td>
<td>35.35</td>
<td>18.043</td>
<td>12.308</td>
<td>9.465</td>
<td>0.99</td>
</tr>
<tr>
<td>PartialCovariances</td>
<td>2.986</td>
<td>1.941</td>
<td>1.358</td>
<td>1.22</td>
<td>1.179</td>
<td>0.7</td>
</tr>
<tr>
<td>ANOVA</td>
<td>3.456</td>
<td>2.493</td>
<td>2.009</td>
<td>1.903</td>
<td>1.862</td>
<td>0.54</td>
</tr>
<tr>
<td>ContingencyTable</td>
<td>437.801</td>
<td>229.409</td>
<td>133.92</td>
<td>108.823</td>
<td>92.874</td>
<td>0.9</td>
</tr>
<tr>
<td>EpochTrainer</td>
<td>43.277</td>
<td>37.397</td>
<td>34.311</td>
<td>33.844</td>
<td>33.339</td>
<td>0.27</td>
</tr>
<tr>
<td>IntFcnAlgLog</td>
<td>534.213</td>
<td>292.953</td>
<td>155.78</td>
<td>155.928</td>
<td>87.49</td>
<td>0.93</td>
</tr>
<tr>
<td>Function</td>
<td>Time (ms)</td>
<td>Time (ms)</td>
<td>Time (ms)</td>
<td>Time (ms)</td>
<td>Time (ms)</td>
<td>Speedup</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>IntFcnCauchy</td>
<td>533.36</td>
<td>292.32</td>
<td>154.54</td>
<td>154.48</td>
<td>87.16</td>
<td>0.93</td>
</tr>
<tr>
<td>IntFcnInf</td>
<td>30.57</td>
<td>16.66</td>
<td>9.37</td>
<td>8.20</td>
<td>6.90</td>
<td>0.89</td>
</tr>
<tr>
<td>IntFcnSing</td>
<td>28.12</td>
<td>15.42</td>
<td>8.16</td>
<td>8.16</td>
<td>4.65</td>
<td>0.93</td>
</tr>
<tr>
<td>IntFcnSingPts</td>
<td>83.45</td>
<td>45.39</td>
<td>23.62</td>
<td>23.60</td>
<td>12.97</td>
<td>0.94</td>
</tr>
<tr>
<td>IntFcnSmooth</td>
<td>28.11</td>
<td>15.41</td>
<td>8.16</td>
<td>8.16</td>
<td>4.76</td>
<td>0.93</td>
</tr>
<tr>
<td>IntFcnTrig</td>
<td>28.13</td>
<td>15.43</td>
<td>8.17</td>
<td>8.17</td>
<td>4.58</td>
<td>0.93</td>
</tr>
<tr>
<td>HyperRectangleQuadrature</td>
<td>19.88</td>
<td>9.95</td>
<td>4.98</td>
<td>4.38</td>
<td>2.49</td>
<td>1</td>
</tr>
<tr>
<td>MinConNLP</td>
<td>27.07</td>
<td>19.03</td>
<td>12.46</td>
<td>10.52</td>
<td>9.86</td>
<td>0.73</td>
</tr>
<tr>
<td>MinConGenLin</td>
<td>245.79</td>
<td>169.92</td>
<td>106.71</td>
<td>89.08</td>
<td>81.52</td>
<td>0.76</td>
</tr>
<tr>
<td>MinUnconMultiVar</td>
<td>25.13</td>
<td>17.43</td>
<td>10.80</td>
<td>9.03</td>
<td>8.36</td>
<td>0.76</td>
</tr>
<tr>
<td>NonlinLeastSquares</td>
<td>12.92</td>
<td>9.02</td>
<td>5.64</td>
<td>4.80</td>
<td>4.41</td>
<td>0.75</td>
</tr>
<tr>
<td>BoundedLeastSquares</td>
<td>13.02</td>
<td>9.10</td>
<td>5.79</td>
<td>4.85</td>
<td>4.55</td>
<td>0.74</td>
</tr>
<tr>
<td>LU</td>
<td>10.87</td>
<td>6.45</td>
<td>6.82</td>
<td>5.73</td>
<td>5.13</td>
<td>0.58</td>
</tr>
<tr>
<td>ComplexLU</td>
<td>32.16</td>
<td>17.05</td>
<td>13.76</td>
<td>12.41</td>
<td>11.99</td>
<td>0.74</td>
</tr>
<tr>
<td>QR</td>
<td>19.33</td>
<td>11.39</td>
<td>9.40</td>
<td>7.44</td>
<td>6.42</td>
<td>0.75</td>
</tr>
<tr>
<td>SVD</td>
<td>46.88</td>
<td>37.19</td>
<td>33.16</td>
<td>31.43</td>
<td>30.84</td>
<td>0.39</td>
</tr>
<tr>
<td>Eigen</td>
<td>754.44</td>
<td>702.67</td>
<td>686.22</td>
<td>674.69</td>
<td>669.4</td>
<td>0.13</td>
</tr>
<tr>
<td>Matrix.Multiply</td>
<td>28.85</td>
<td>15.48</td>
<td>7.90</td>
<td>6.05</td>
<td>4.21</td>
<td>0.97</td>
</tr>
<tr>
<td>ComplexMatrix.Multiply</td>
<td>110.28</td>
<td>59.23</td>
<td>29.99</td>
<td>23.06</td>
<td>16.11</td>
<td>0.97</td>
</tr>
<tr>
<td>Vector*Matrix</td>
<td>0.3</td>
<td>0.16</td>
<td>0.09</td>
<td>0.07</td>
<td>0.05</td>
<td>0.94</td>
</tr>
<tr>
<td>ComplexVector*Matrix</td>
<td>0.39</td>
<td>0.21</td>
<td>0.12</td>
<td>0.10</td>
<td>0.08</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Microsoft Windows Server 2008 R2 Standard 64-bit Multivariate Analysis

![Graph showing speedup for different processing scenarios.](image)
Microsoft Windows Server 2008 R2 Standard 64-bit

Microsoft Windows Server 2008 R2 Standard 64-bit
Miscellaneous

Microsoft Windows Server 2008 R2 Standard 64-bit
Quadrature

www.roguewave.com
**Microsoft Windows Server 2003 R2 Standard (32-bit)**

Hardware: 2 Quad Core Intel Xeon CPU (8 cores in total) 2.33GHz, 8 GB RAM


Compiler: Microsoft Visual Studio 2010 (Any CPU project)

<table>
<thead>
<tr>
<th>Name</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>Fraction Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ClusterHierarchical</td>
<td>6.181</td>
<td>4.231</td>
<td>3.175</td>
<td>2.91</td>
<td>2.869</td>
<td>0.63</td>
</tr>
<tr>
<td>Dissimilarities</td>
<td>62.794</td>
<td>32.147</td>
<td>17.185</td>
<td>12.075</td>
<td>9.516</td>
<td>0.97</td>
</tr>
<tr>
<td>FactorAnalysis</td>
<td>103.486</td>
<td>84.348</td>
<td>75.604</td>
<td>72.298</td>
<td>71.02</td>
<td>0.36</td>
</tr>
<tr>
<td>AutoCorrelation</td>
<td>11.884</td>
<td>5.938</td>
<td>2.985</td>
<td>2.056</td>
<td>1.534</td>
<td>0.99</td>
</tr>
<tr>
<td>CrossCorrelation</td>
<td>41.513</td>
<td>21.269</td>
<td>10.425</td>
<td>7.269</td>
<td>5.291</td>
<td>1</td>
</tr>
<tr>
<td>MultiCrossCorrelation</td>
<td>69.766</td>
<td>34.901</td>
<td>18.166</td>
<td>13.434</td>
<td>9.322</td>
<td>0.99</td>
</tr>
<tr>
<td>ARAutoUnivariate</td>
<td>13.522</td>
<td>8.354</td>
<td>5.387</td>
<td>4.191</td>
<td>3.678</td>
<td>0.83</td>
</tr>
<tr>
<td>ARMA</td>
<td>23.494</td>
<td>13.925</td>
<td>8.581</td>
<td>6.716</td>
<td>5.488</td>
<td>0.87</td>
</tr>
<tr>
<td>NextGaussianCopula</td>
<td>0.672</td>
<td>0.588</td>
<td>0.566</td>
<td>0.566</td>
<td>0.566</td>
<td>0.2</td>
</tr>
<tr>
<td>NextStudentsTCopula</td>
<td>0.691</td>
<td>0.594</td>
<td>0.578</td>
<td>0.581</td>
<td>0.578</td>
<td>0.2</td>
</tr>
<tr>
<td>NextMultivariateNormal</td>
<td>0.653</td>
<td>0.584</td>
<td>0.566</td>
<td>0.562</td>
<td>0.562</td>
<td>0.17</td>
</tr>
<tr>
<td>CanonicalCorrelation</td>
<td>481.075</td>
<td>249.203</td>
<td>129.675</td>
<td>91.051</td>
<td>69.257</td>
<td>0.98</td>
</tr>
<tr>
<td>FaureSequence</td>
<td>5.5</td>
<td>2.76</td>
<td>1.375</td>
<td>0.915</td>
<td>0.694</td>
<td>1</td>
</tr>
<tr>
<td>Covariances</td>
<td>84.985</td>
<td>43.207</td>
<td>21.766</td>
<td>14.722</td>
<td>11.247</td>
<td>0.99</td>
</tr>
<tr>
<td>PartialCovariances</td>
<td>3.078</td>
<td>2.046</td>
<td>1.484</td>
<td>1.316</td>
<td>1.237</td>
<td>0.69</td>
</tr>
<tr>
<td>ANOVA</td>
<td>4.669</td>
<td>4.193</td>
<td>3.919</td>
<td>3.894</td>
<td>3.981</td>
<td>0.19</td>
</tr>
<tr>
<td>ContingencyTable</td>
<td>411.252</td>
<td>218.435</td>
<td>125.074</td>
<td>100.676</td>
<td>86.267</td>
<td>0.91</td>
</tr>
<tr>
<td>EpochTrainer</td>
<td>39.457</td>
<td>35.072</td>
<td>32.635</td>
<td>31.857</td>
<td>31.61</td>
<td>0.23</td>
</tr>
<tr>
<td>IntFcnAlgLog</td>
<td>620.34</td>
<td>339.751</td>
<td>179.805</td>
<td>178.943</td>
<td>100.529</td>
<td>0.94</td>
</tr>
<tr>
<td>IntFcnCauchy</td>
<td>611.701</td>
<td>374.27</td>
<td>184.296</td>
<td>183.905</td>
<td>114.027</td>
<td>0.91</td>
</tr>
<tr>
<td>IntFcnInf</td>
<td>33.274</td>
<td>20.613</td>
<td>10.572</td>
<td>9.666</td>
<td>6.438</td>
<td>0.9</td>
</tr>
<tr>
<td>IntFcnSing</td>
<td>32.412</td>
<td>18.828</td>
<td>9.825</td>
<td>9.785</td>
<td>5.447</td>
<td>0.93</td>
</tr>
<tr>
<td>IntFcnSingPts</td>
<td>95.96</td>
<td>58.148</td>
<td>28.297</td>
<td>28.181</td>
<td>15.341</td>
<td>0.94</td>
</tr>
<tr>
<td>IntFcnSmooth</td>
<td>32.344</td>
<td>18.75</td>
<td>9.778</td>
<td>9.719</td>
<td>5.531</td>
<td>0.92</td>
</tr>
<tr>
<td>IntFcnTrig</td>
<td>32.338</td>
<td>19.682</td>
<td>9.775</td>
<td>9.734</td>
<td>5.428</td>
<td>0.93</td>
</tr>
<tr>
<td>HyperRectangleQuadrature</td>
<td>22.078</td>
<td>11.85</td>
<td>5.963</td>
<td>4.003</td>
<td>3.056</td>
<td>0.98</td>
</tr>
<tr>
<td>MinConNLP</td>
<td>45.341</td>
<td>27.641</td>
<td>15.722</td>
<td>11.622</td>
<td>9.781</td>
<td>0.89</td>
</tr>
<tr>
<td>MinConGenLin</td>
<td>407.924</td>
<td>253.319</td>
<td>143.205</td>
<td>103.623</td>
<td>85.482</td>
<td>0.9</td>
</tr>
<tr>
<td>MinUnconMultiVar</td>
<td>44.51</td>
<td>26.307</td>
<td>14.666</td>
<td>10.559</td>
<td>8.712</td>
<td>0.92</td>
</tr>
</tbody>
</table>
Microsoft Windows Server 2003 R2 Standard 32-bit
Time Series and Forecasting

Microsoft Windows Server 2003 R2 Standard 32-bit
Random Number Generation
The benchmarks include some problems used to benchmark several different IMSL C# Numerical Library classes. This section provides details on the various test problems in the benchmark suite.

### Quadrature Functions

\[
f_1(x) = \sum_{i=3}^{M-1} \sum_{j=3}^{N-1} (i + j/100)^x \quad \text{with} \quad M=100,000 \quad \text{and} \quad N=100.
\]

\[
f_2(x) = \sum_{i=3}^{M-1} \sum_{j=3}^{N-1} (i + j/100)^{-x} \quad \text{with} \quad M=100,000 \quad \text{and} \quad N=100.
\]

### Optimization Neural Network Problem

A simple neural network problem was used to benchmark the optimization classes. The network has 20 inputs, 20 perceptrons in a single hidden layer, and a single output. All of the input nodes are connected to all of the perceptrons in the hidden layer. All of the hidden layer perceptrons are connected to the single output node. The network has 441 weights, which are the unknowns in the optimization problem. Each optimization class was used to determine the value of the weights such that the network best fit 10,000 observations; i.e., the network was trained with 10,000 patterns. If the network evaluation function is \( N(x;w) \), the optimization problem is

\[
\min_{w} \sum_{i=0}^{N-1} [N(x_i;w) - y_i]^2
\]

where \( N = 10,000 \), \( x \) and \( y \) are the observations (“training patterns”) and \( w \) is the weights. The observations are

\[
x_i = \cos(301i + 401j)
\]

\[
y_i = \sum_{j=0}^{N-1} \cos(301i + 401j)
\]

### Full Set of Benchmark Problems

- **ClusterHierarchical**
  The hierarchical clustering was done on a uniform random 6,000 by 6,000 distance matrix.

- **Dissimilarities**
  The dissimilarities matrix was computed from an array of 2,000 by 2,000 uniform random numbers, with one added to the diagonal. The correlation coefficient distance method was used.
• **FactorAnalysis**
  
  Factor analysis was performed on the 1,000 by 1,000 covariance matrix
  
  \[ a_{ij} = \frac{1}{i + j + 1} + 0.2\delta_{ij}, \text{where } \delta_{ij} \text{ is one if } i \text{ equals } j, \text{ and is zero otherwise.} \]

• **AutoCorrelation**
  
  The standard errors of autocorrelation using the Bartlett’s formula were computed from an array of 30,000 stationary time series.

• **CrossCorrelation**
  
  The standard errors of cross-correlation using the Bartlett’s formula were computed from two arrays of 30,000 stationary time series.

• **MultiCrossCorrelation**
  
  The cross-covariances were computed from two 300 by 300 channels.

• **ARAutoUnivariate**
  
  The final autoregressive parameter estimates were computed from an array of 3,000 time series with the maximum number of autoregressive lags set to 750.

• **ARMA**
  
  Forecasts and their associated probability limits for an ARMA(2500, 20) model for 5,000 observations were computed with the maximum backward origin set to 1,000 and the maximum lead time for forecasts set to 500.

• **NextGaussianCopula**
  
  6,000 random numbers from a Gaussian Copula distribution were computed.

• **NextStudentsTCopula**
  
  6,000 random numbers from a Student’s t Copula distribution were computed with 5 degrees of freedom.

• **NextMultivariateNormal**
  
  6,000 random numbers from a multivariate normal distribution were computed.

• **CanonicalCorrelation**
  
  The canonical correlation matrix was computed from a 3,000 by 3,000 array of deviate values.

• **FaureSequence**
  
  The next point in the Faure sequence with a 50,000 dimension was computed.

• **Covariances**
  
  The variance-covariance matrix was computed from an array of 100 observations by 4,000 variables. This array contained random numbers with about 1% set to NaN (missing value indicator). Variances used are computed from the valid pairs of data. MissingValueMethod = 3 was used.

• **PartialCovariances**
  
  The partial correlation matrix was computed from a 5,000 by 5,000 covariance matrix with 30 independent variables and 30 degrees of freedom.

• **ANOVA**
  
  The F statistic was computed from a 15,000 by 15,000 array containing the responses.
• **ContingencyTable**
  The Pearson chi-squared test statistic was computed from a 500 by 500 two-way contingency table.

• **EpochTrainer**
  The optimization neural network problem was used.

• **IntFcnAlgLog**
  \[
  \int_{0}^{1} f_{1}(x)x^{1/4}(1 - x)^{3/4} \log(x)dx
  \]

• **IntFcnCauchy**
  \[
  \int_{0}^{1} \frac{f_{1}(x)}{x - 0.5}dx
  \]

• **IntFcnInf**
  \[
  \int_{0}^{\infty} f_{1}(x)dx
  \]

• **IntFcnSing**
  \[
  \int_{0}^{1} f_{1}(x)dx
  \]

• **IntFcnSingPts**
  \[
  \int_{0}^{1} f_{1}(x)dx \text{ with singular points given as 0.3, 0.5 and 0.7.}
  \]

• **IntFcnSmooth**
  \[
  \int_{0}^{1} f_{1}(x)dx
  \]

• **IntFcnTrig**
  \[
  \int_{0}^{1} f_{1}(x)\cos(\omega x)dx \text{ where } \omega = 2.5.
  \]

• **HyperRectangleQuadrature**
  \[
  \int_{0}^{1} \cdots \int_{0}^{1} f_{1}(x_{i})dx_{i} \cdots dx_{0} \text{ with } M = 1000 \text{ and } N = 10.
  \]

• **MinConNLP**
  The optimization neural network problem was used with the additional requirement that the weights are bounded to be in [-3,3].

• **MinConGenLin**
  The optimization neural network problem was used with the additional requirement that the weights are bounded to be in [-3,3].

• **MinUnconMultiVar**
  The optimization neural network problem was used.

• **NonlinLeastSquares**
  The optimization neural network problem was used.
• **BoundedLeastSquares**  
The optimization neural network problem was used with the additional requirement that the weights are bounded to be in \([-3,3]\).

• **LU**  
\[ Ax = b \] was solved using LU factorization, where \(A\) contained a 2,000 by 2,000 matrix.

• **ComplexLU**  
\[ Ax = b \] was solved using LU factorization, where \(A\) contained a 2,000 by 2,000 complex matrix.

• **QR**  
\[ Ax = b \] was solved using QR decomposition, where \(A\) contained a 3,000 by 1,500 matrix.

• **SVD**  
The Moore-Penrose generalized inverse of a 1,500 by 750 matrix was computed using singular value decomposition.

• **Eigen**  
The eigenvalues and eigenvectors were computed from a 2,000 by 2,000 matrix.

• **Matrix.Multiply**  
The matrix multiply, \(C = A \times B\), was computed, where \(A\) and \(B\) contained 2,000 by 2,000 matrices.

• **ComplexMatrix.Multiply**  
The complex matrix multiply, \(C = A \times B\), was computed, where \(A\) and \(B\) contained 2,000 by 2,000 complex matrices.

• **Vector*Multiply**  
The product of a 1 by 5,000 row matrix and a 5,000 by 5,000 rectangular matrix was computed.

• **ComplexVector*Multiply**  
The product of a 1 by 5,000 complex row matrix and a 5,000 by 5,000 complex rectangular matrix was computed.

### Sample Benchmarking Code

The code that follows benchmarks IntFcnInf and Covariances. The test cases described above are used.

The property, **NumberOfProcessors**, is used to set the maximum number of threads allowed during each phase of the benchmarking. This code assumes the use of a maximum of eight processors.

The user-defined method for IntFcnInf is thread-safe. By default, such user-defined method is evaluated in parallel.

Each benchmark is run five times to reduce errors in the timings.
namespace Benchmark
{
    using System;
    using Imsl.Math;
    using Imsl.Stat;

    public class Benchmark : Quadrature.IFunction
    {
        // User-defined method for BenchIntFcnInf.
        public double F(double x)
        {
            double sum = 0.0;
            int i, j;

            for (i = 3; i < 100000; i++)
            {
                for (j = 3; j < 100; j++)
                {
                    sum += Math.Pow(i + 0.01 * j, -x);
                }
            }
            return sum;
        }

        // IntFcnInf benchmark.
        public static double BenchIntFcnInf(int CPU)
        {
            Quadrature q = new Quadrature();
            q.NumberOfProcessors = CPU;

            long time = (DateTime.Now.Ticks - 621355968000000000) / 10000;
            double result = q.Eval(new Benchmark(), 0.0, Double.PositiveInfinity);
            time = (DateTime.Now.Ticks - 621355968000000000) / 10000 - time;
            return ((double)time / 1000.0);
        }

        // Covariances benchmark.
        public static double BenchCovariances(int CPU)
        {
            int nobs = 100, nvar = 4000;
            double[,] x = new double[nobs, nvar];

            // setup x.
            for (int i = 0; i < nobs; i++)
            {
                for (int j = 0; j < nvar; j++)
                {
                    x[i, j] = random.NextDouble();
                }
            }
        }
    }
}
for (int i = 0; i < nobs * nvar; i += 123)
{
    int row = i / nvar;
    int col = i - row * nvar;
    x[row, col] = Double.NaN;
}

Covariances co = new Covariances(x);
co.MissingValueMethod = 3;
co.NumberOfProcessors = CPU;

long time = (DateTime.Now.Ticks - 621355968000000000) / 10000;
double[,] result =
    co.Compute(Covariances.MatrixType.VarianceCovariance);
time = (DateTime.Now.Ticks - 621355968000000000) / 10000 - time;
return ((double)time / 1000.0);

public static void Main(string[] args)
{
    int[] cpu = { 1, 2, 4, 6, 8 };
    String[] names = { "IntFcnInf", "Covariances" };
    int CPU, repeat = 5, tests = names.Length;
    double[,] time = new double[tests, cpu.Length];

    for (int index = 0; index < cpu.Length; index++)
    {
        CPU = cpu[index];
        for (int j = 0; j < tests; j++)
        {
            for (int i = 0; i < repeat; i++)
            {
                switch (j)
                {
                    case 0:
                        time[j, index] += BenchIntFcnInf(CPU);
                        break;
                    case 1:
                        time[j, index] += BenchCovariances(CPU);
                        break;
                    default:
                        break;
                }
            }
            time[j, index] /= (double)repeat;
        }
    }

    Console.WriteLine("\t\t1\t2\t4\t6\t8");
    for (int i = 0; i < tests; i++)
    {
        Console.WriteLine("\t\t{0}", time[0, i]);
        Console.WriteLine("\t\t{0}", time[1, i]);
    }
}
Console.Write(names[i]);
for (int j = 0; j < cpu.Length; j++)
{
    Console.Write("\t{0, 0:F2}", time[i, j]);
}
Console.WriteLine();

Conclusion

The use of the TPL in .NET 4.0 within the IMSL C# Numerical Library is an effective means of achieving scalable parallelism. Some classes show very good, nearly linear, scaling up to 8 processors, which is quite good considering the algorithms were not re-written, but rather just had codes using the TPL added in strategic places. Finally, Rogue Wave has performed these benchmarks on standard systems using the publically available version of the software, and while we expect you should get similar results, it is always best to evaluate the algorithms you use on your deployment hardware for truly accurate results.