

Parallel Performance of the IMSL C Numerical Library

Benchmarking OpenMP

A White Paper by Visual Numerics, Inc.
November, 2008

Visual Numerics[®]

Visual Numerics, Inc.
2500 Wilcrest Drive, Suite 200
Houston, TX 77042
USA
www.vni.com

Parallel Performance of the IMSL C Numerical Library

Benchmarking OpenMP

by **Visual Numerics, Inc.**

Copyright 2008 by Visual Numerics, Inc. All Rights Reserved
Printed in the United States of America

Publishing History:

November 2008

Trademark Information

Visual Numerics, IMSL and PV-WAVE are registered trademarks. JMSL, PyIMSL, TS-WAVE, and JWAVE are trademarks of Visual Numerics, Inc., in the U.S. and other countries. All other product and company names are trademarks or registered trademarks of their respective owners.

The information contained in this document is subject to change without notice. Visual Numerics, Inc. makes no warranty of any kind with regard to this material, included, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Visual Numerics, 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
Windows with Visual Studio 2008 (32-bit)	5
Windows with Visual Studio 2008 (64-bit)	10
Red Hat 5 with Intel C 10.1 (64-bit)	14
SuSE 10 with Intel C 10.1 (64-bit)	18
Sun Solaris Opteron (64-bit)	22
Test Problems	25
Quadrature Functions.....	25
Optimization Neural Network Problem	25
Full Set of Benchmark Problems	26
Sample Benchmarking Code	29
Conclusion.....	32
About the Author	33

Parallel Performance of the IMSL C Numerical Library

The IMSL C Numerical Library uses OpenMP to enable parallelism on shared memory systems, especially multi-core systems. Starting with Version 7.0 of the IMSL C Library, released in November 2008, OpenMP directives were added to a variety of functions 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 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 added OpenMP directives wherever sensible. Because high performance vendor libraries are linked by the IMSL C Library for best performance for many linear algebra functions, the focus was on parallelizing other sections of the library.

The performance of selected IMSL C Numerical Library functions was measured on multi-core systems. Each function was used to solve a large enough problem to allow for parallelism. Each test case was run with a varying number of threads allowed. The number of threads was set using the OpenMP function `omp_set_num_threads`. The specific problems run are described at the end of this paper.

Thread-Safe User Functions

Many of the IMSL C Library functions which use OpenMP can evaluate user-defined functions in parallel. This requires that the user's functions be thread-safe. For backward compatibility with existing user code, the IMSL C Library does not assume user functions are thread-safe and will not evaluate such functions in parallel. They are evaluated in parallel only if they are flagged as thread-safe by calling:

for the IMSL C Math Library,

```
imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
```

and for the IMSL C Stat Library,

```
imsls_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
```

These calls were made for all of the benchmarks described in this paper.

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 Library nonlinear regression function, `imsls_d_nonlinear_regression`, 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. The one exception is `imsls_d_naive_bayes_trainer`, which was run 250 times because the time for each run was small.

Windows with Visual Studio 2008 (32-bit)

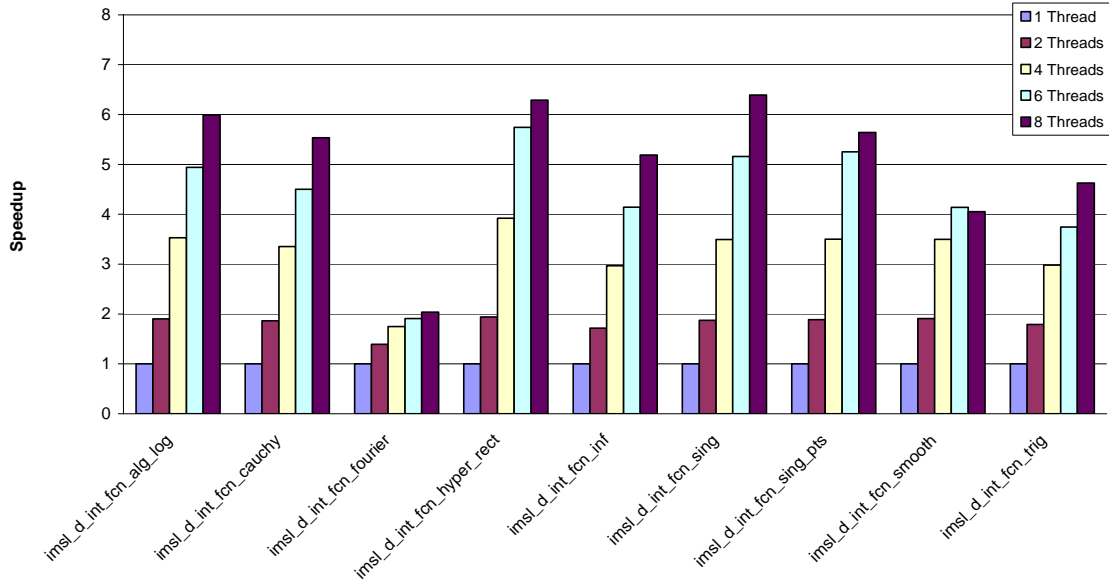
Hardware: Dual Quad Core Xeon E5420 (Harpertown) (8 cores in total) 2.5GHz, 133MHz Front Side Bus

Operating System: Windows Server 2003 R2

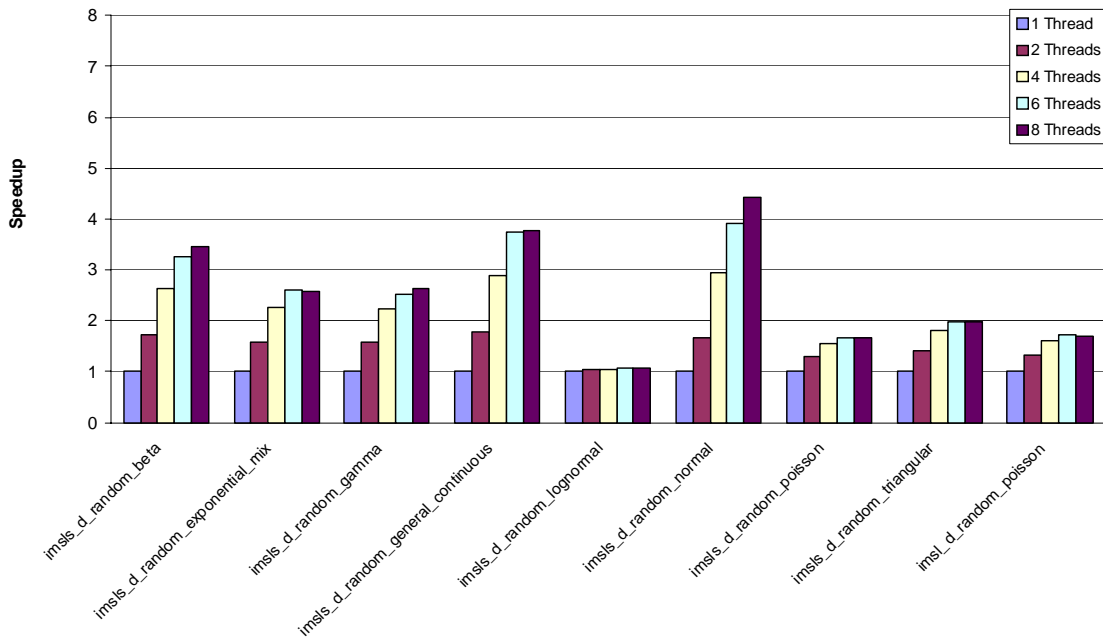
Compiler: Microsoft Visual Studio 2008 (32-bit project)

omp_set_num_threads	1	2	4	6	8	Fraction Parallel
<i>IMSL Function</i>	<i>Time (seconds)</i>					
imsl_d_int_fcn_alg_log	40.87	21.47	11.58	8.27	6.82	0.95
imsl_d_int_fcn_cauchy	87.58	46.96	26.12	19.45	15.82	0.94
imsl_d_int_fcn_fourier	9.90	7.11	5.66	5.19	4.86	0.58
imsl_d_int_fcn_hyper_rect	35.73	18.43	9.11	6.22	5.68	0.97
imsl_d_int_fcn_inf	67.57	39.35	22.76	16.31	13.02	0.92
imsl_d_int_fcn_sing	17.13	9.13	4.90	3.32	2.68	0.96
imsl_d_int_fcn_sing_pts	82.58	43.76	23.59	15.72	14.64	0.95
imsl_d_int_fcn_smooth	17.18	9.00	4.91	4.15	4.24	0.89
imsl_d_int_fcn_trig	39.42	22.02	13.22	10.53	8.52	0.89
imsl_d_fast_poisson_2d	15.57	8.79	7.55	7.18	7.09	0.65
imsl_d_constrained_nlp	15.77	8.03	4.09	2.82	2.64	0.96
imsl_d_min_con_gen_lin	307.18	295.43	149.82	101.20	85.27	0.80
imsl_d_min_uncon_multivar	94.25	47.96	23.90	16.11	13.79	0.98
imsl_d_nonlin_least_squares	33.03	31.69	16.43	11.26	9.75	0.78
imsl_d_covariances	19.52	9.62	5.10	3.49	3.44	0.96
imsl_d_random_poisson	4.61	3.45	2.88	2.69	2.70	0.49
imsls_d_nonlinear_optimization	217.91	119.38	68.89	54.78	61.55	0.86
imsls_d_nonlinear_regression	35.54	18.93	10.59	7.92	7.57	0.91
imsls_d_covariances	126.08	63.09	32.65	22.60	20.51	0.97
imsls_d_dissimilarities	55.56	41.63	24.52	17.41	13.62	0.84
imsls_d_factor_analysis	103.07	67.82	48.26	40.63	36.18	0.73
imsls_d_cluster_hierarchical	21.56	16.55	12.29	11.42	12.68	0.53
imsls_d_multivariate_normal_cdf	64.13	35.11	17.53	17.60	10.39	0.94
imsls_d_random_beta	10.86	6.34	4.13	3.33	3.13	0.82
imsls_d_random_exponential_mix	6.97	4.39	3.07	2.66	2.71	0.72
imsls_d_random_gamma	6.38	4.05	2.87	2.52	2.43	0.72
imsls_d_random_general_continuous	15.07	8.50	5.20	4.03	4.00	0.86
imsls_d_random_lognormal	68.90	66.51	65.30	64.88	64.70	0.07
imsls_d_random_normal	31.18	18.55	10.61	7.99	7.04	0.89
imsls_d_random_poisson	4.42	3.38	2.84	2.66	2.65	0.47
imsls_d_random_triangular	4.41	3.10	2.45	2.23	2.21	0.58
imsls_d_classification_trainer	308.09	192.42	127.86	107.09	101.16	0.77
imsls_d_genetic_algorithm	20.29	14.74	11.44	10.32	10.16	0.58
imsls_d_mlff_network_trainer	46.72	32.18	23.74	21.41	20.88	0.64
imsls_d_naive_bayes_trainer	0.06	0.04	0.02	0.02	0.02	0.87

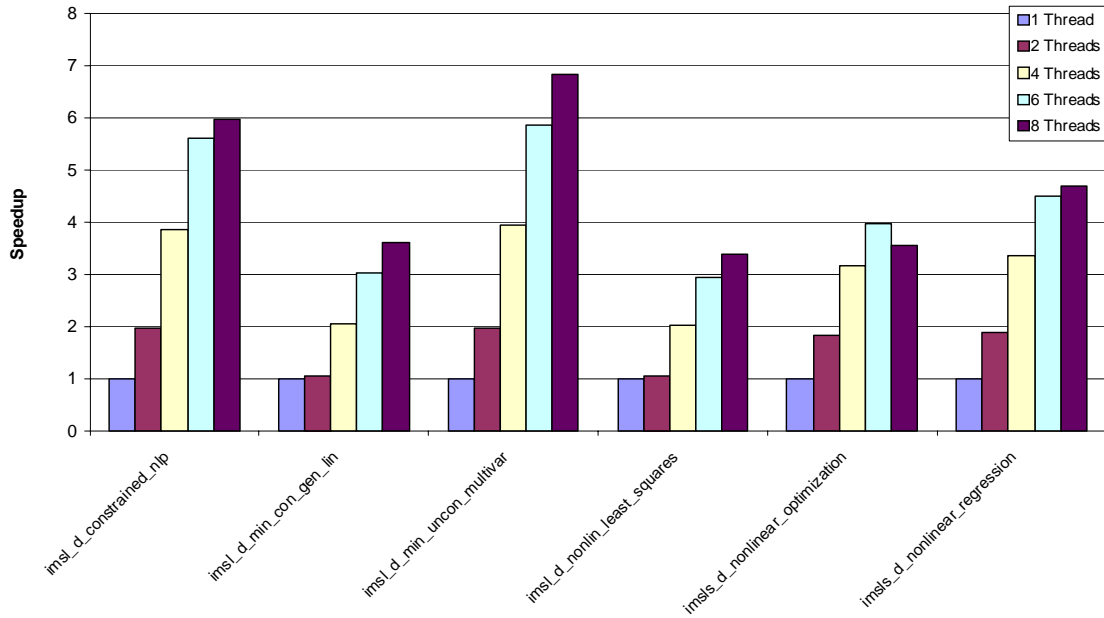
Microsoft Visual Studio 2008 (32 bit)
Quadrature



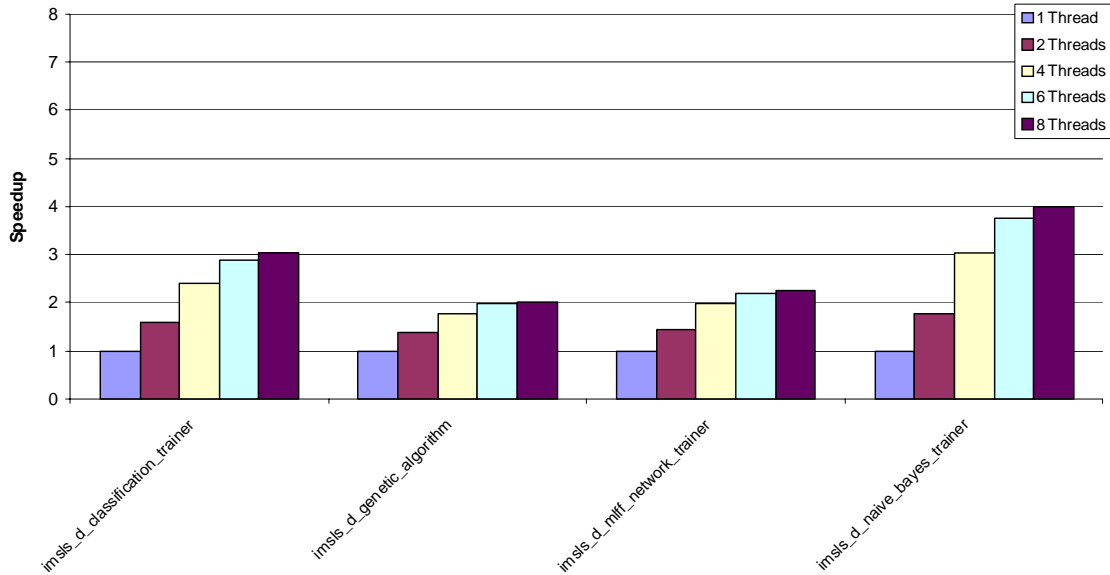
Microsoft Visual Studio 2008 (32 bit)
Random Number Generation



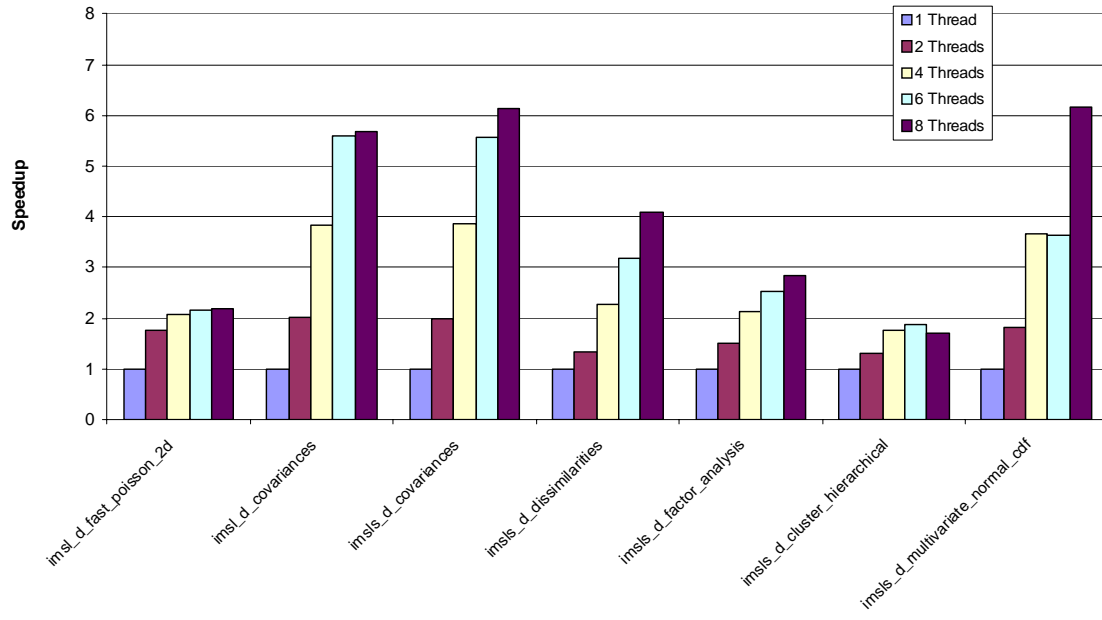
Microsoft Visual Studio 2008 (32 bit)
Optimization



Microsoft Visual Studio 2008 (32 bit)
Data Mining



Microsoft Visual Studio 2008 (32 bit)
Miscellaneous



Windows with Visual Studio 2008 (64-bit)

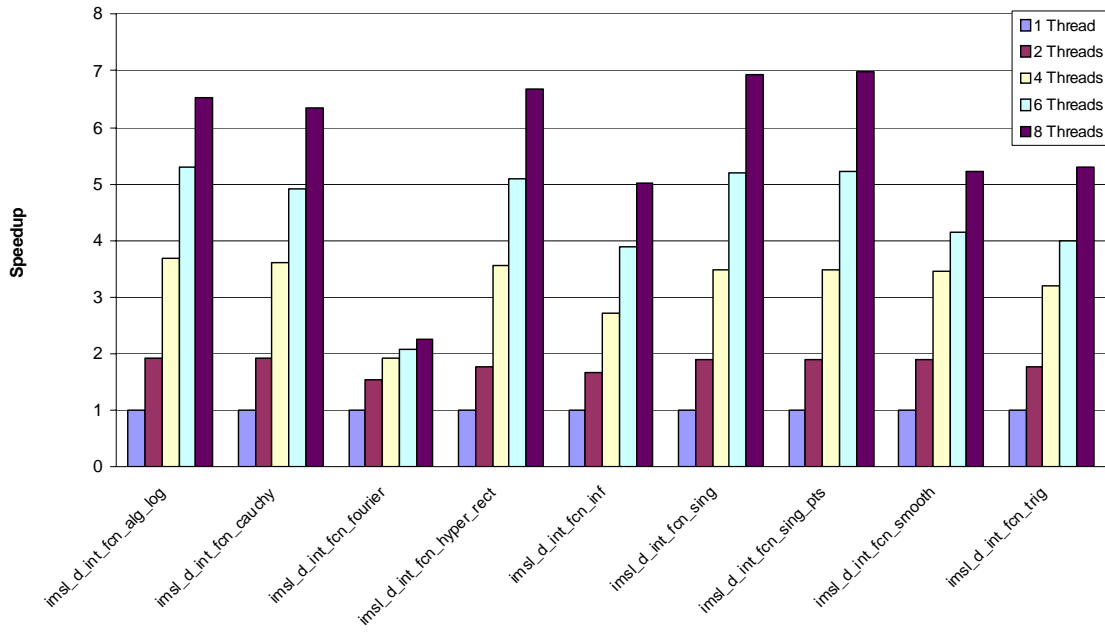
Hardware: Dual Quad Core Xeon E5420 (Harpertown) (8 cores in total) 2.5GHz, 133MHz Front Side Bus

Operating System: Windows Server 2003 R2

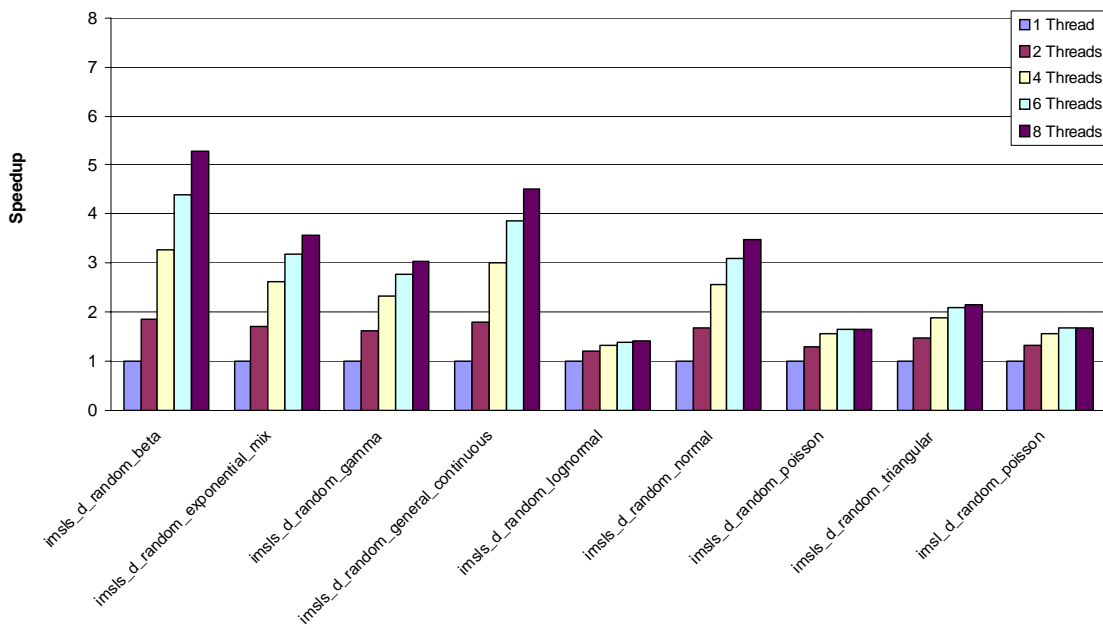
Compiler: Microsoft Visual Studio 2008 (64-bit project)

omp_set_num_threads	1	2	4	6	8	Fraction Parallel
<i>IMSL Function</i>	<i>Time (seconds)</i>					
imsl_d_int_fcn_alg_log	54.69	28.38	14.83	10.32	8.37	0.97
imsl_d_int_fcn_cauchy	123.36	64	34.2	25.1	19.43	0.96
imsl_d_int_fcn_fourier	5.65	3.65	2.92	2.71	2.51	0.64
imsl_d_int_fcn_hyper_rect	1.07	0.6	0.3	0.21	0.16	0.97
imsl_d_int_fcn_inf	100.08	59.86	36.83	25.71	19.99	0.9
imsl_d_int_fcn_sing	23.84	12.48	6.82	4.58	3.44	0.97
imsl_d_int_fcn_sing_pts	95.08	49.79	27.21	18.22	13.63	0.98
imsl_d_int_fcn_smooth	23.84	12.54	6.88	5.75	4.57	0.92
imsl_d_int_fcn_trig	54.41	30.68	17.04	13.66	10.28	0.92
imsl_d_fast_poisson_2d	13.84	12.75	12.06	11.83	11.72	0.17
imsl_d_constrained_nlp	63.36	32.34	16.71	11.47	8.9	0.98
imsl_d_min_con_gen_lin	466.17	233.09	117.61	79.07	59.92	1
imsl_d_min_uncon_multivar	50.28	25.31	12.81	8.63	6.57	0.99
imsl_d_nonlin_least_squares	25.51	12.98	6.75	4.66	3.61	0.98
imsl_d_covariances	5.82	3.01	1.65	1.17	0.94	0.96
imsl_d_random_poisson	2.07	1.56	1.32	1.24	1.23	0.48
imsls_d_nonlinear_optimization	163.61	90.07	53.19	41.96	38.4	0.88
imsls_d_nonlinear_regression	26.19	13.35	7.05	4.92	3.86	0.97
imsls_d_covariances	41.19	20.87	10.66	7.35	5.63	0.99
imsls_d_dissimilarities	15.64	12.15	7.5	6.25	5.8	0.71
imsls_d_factor_analysis	47.71	39.95	29.85	25.33	22.95	0.56
imsls_d_cluster_hierarchical	10.68	7.81	6.3	5.96	5.78	0.53
imsls_d_multivariate_normal_cdf	8.61	4.32	2.17	2.17	1.09	0.98
imsls_d_random_beta	12.91	6.93	3.94	2.94	2.44	0.93
imsls_d_random_exponential_mix	5.34	3.14	2.04	1.68	1.5	0.82
imsls_d_random_gamma	4.03	2.49	1.72	1.46	1.33	0.77
imsls_d_random_general_continuous	8.59	4.77	2.86	2.22	1.9	0.89
imsls_d_random_lognormal	12.12	10.09	9.08	8.73	8.57	0.33
imsls_d_random_normal	6.76	4.01	2.63	2.18	1.95	0.81
imsls_d_random_poisson	2.03	1.56	1.3	1.24	1.22	0.47
imsls_d_random_triangular	2.51	1.72	1.33	1.2	1.17	0.62
imsls_d_classification_trainer	122.05	81.7	58.51	49.76	46.66	0.7
imsls_d_genetic_algorithm	13.11	7.81	4.87	3.92	3.55	0.84
imsls_d_mlff_network_trainer	11.17	10.76	7.74	8.37	7.63	0.34
imsls_d_naive_bayes_trainer	0.025	0.015	0.009	0.007	0.006	0.86

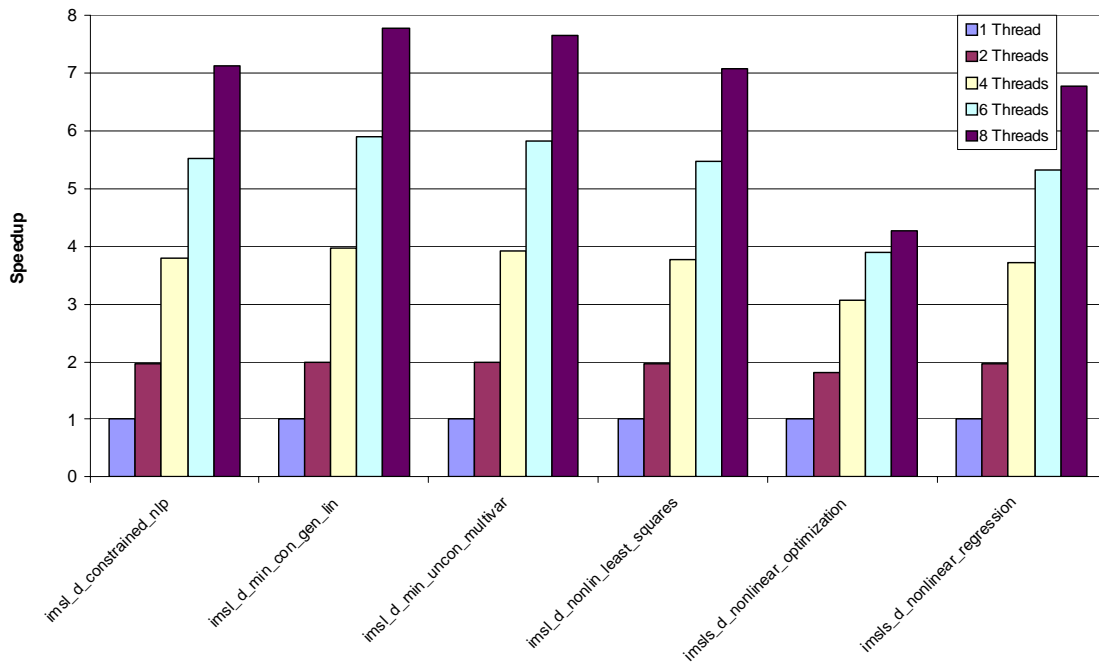
Microsoft Visual Studio 2008 (64 bit)
Quadrature



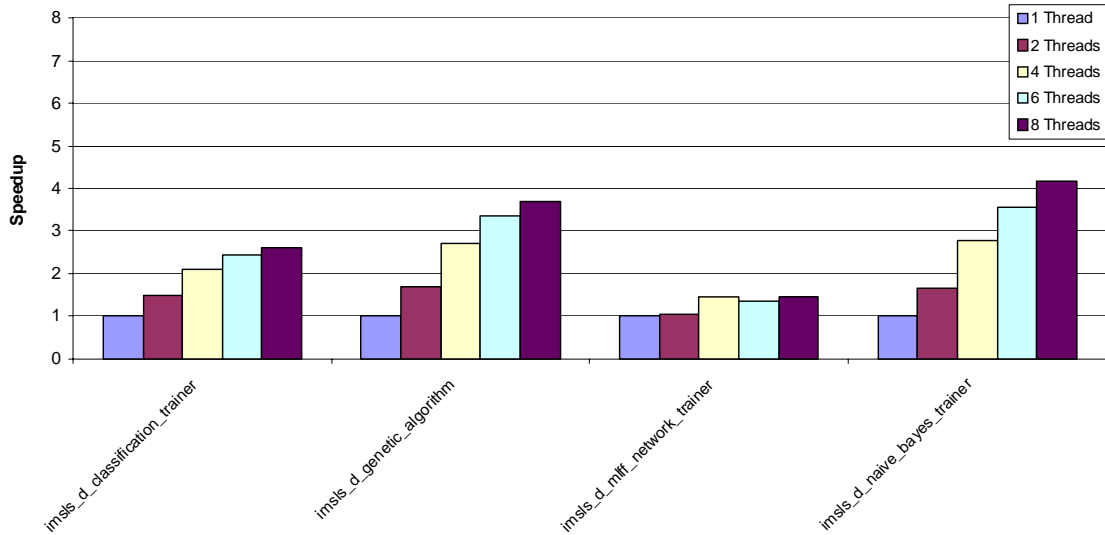
Microsoft Visual Studio 2008 (64 bit)
Random Number Generation



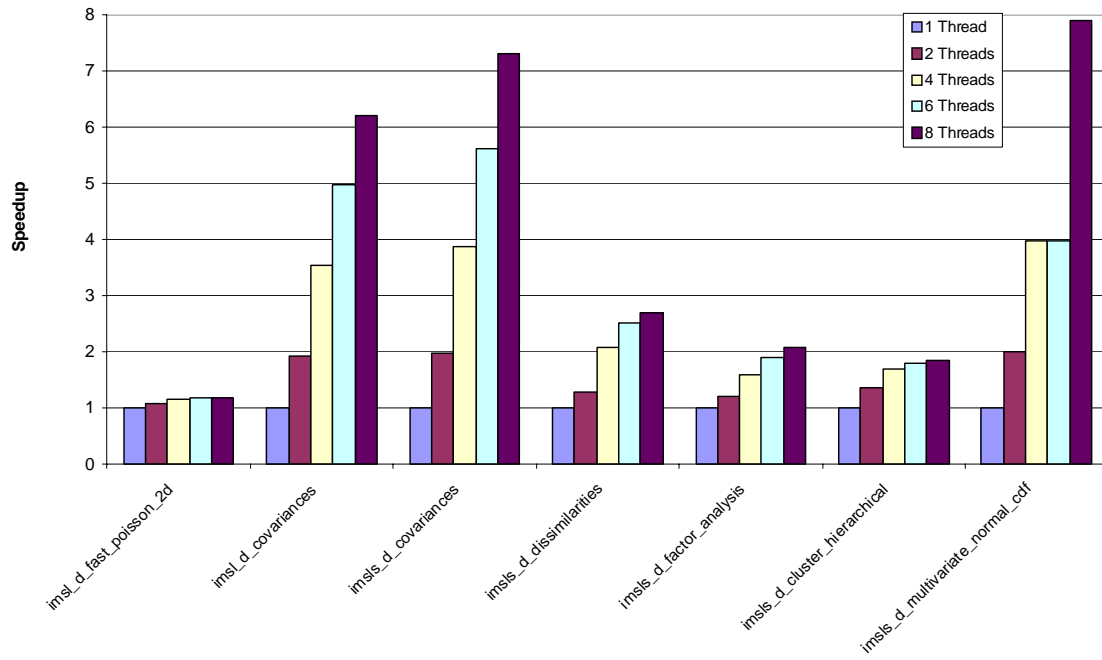
Microsoft Visual Studio 2008 (64 bit)
Optimization



Microsoft Visual Studio 2008 (64 bit)
Data Mining



Microsoft Visual Studio 2008 (64 bit)
Miscellaneous



Red Hat 5 with Intel C 10.1 (64-bit)

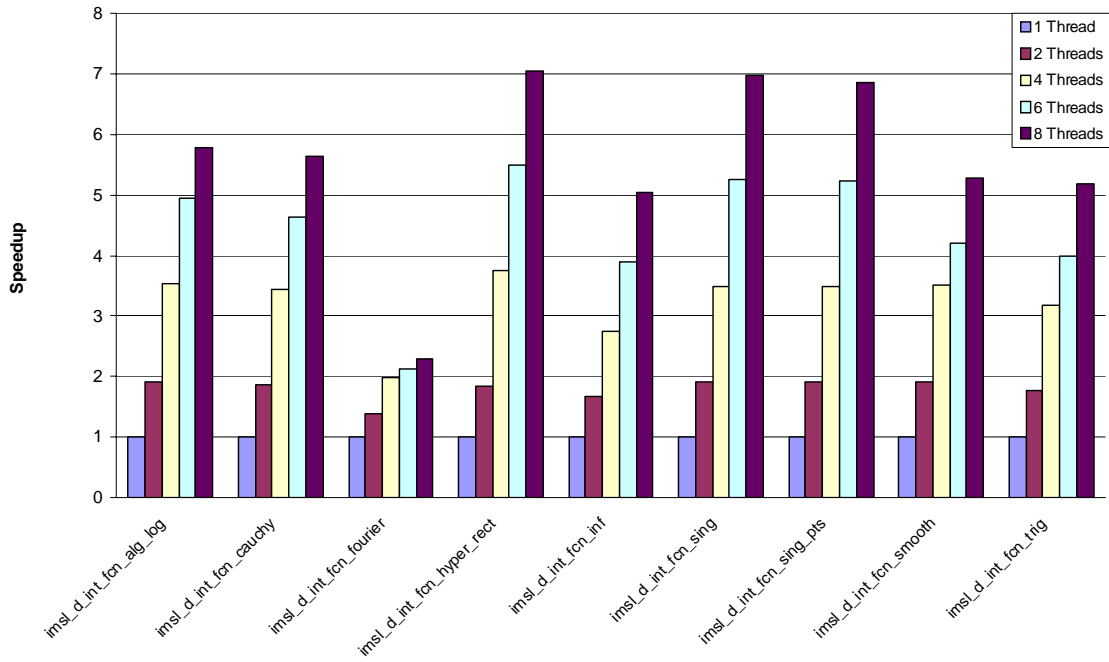
Hardware: Dual Quad Core Xeon E5420 (Harpertown) (8 cores in total) 2.5GHz, 133MHz Front Side Bus

Operating System: Red Hat 5

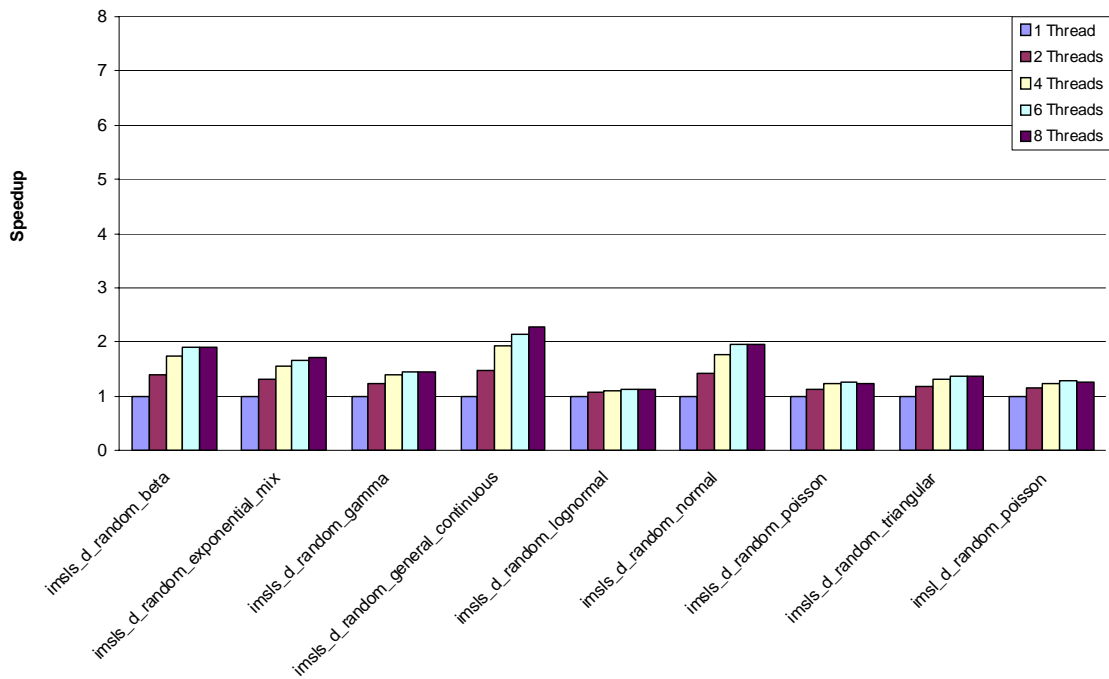
Compiler: Intel C 10.1 (64-bit)

omp_set_num_threads	1	2	4	6	8	Fraction Parallel
<i>IMSL Function</i>	<i>Time (seconds)</i>					
imsl_d_int_fcn_alg_log	22.92	11.99	6.48	4.63	3.96	0.95
imsl_d_int_fcn_cauchy	49.04	26.31	14.3	10.61	8.71	0.94
imsl_d_int_fcn_fourier	4.94	3.57	2.5	2.33	2.15	0.64
imsl_d_int_fcn_hyper_rect	1.76	0.96	0.47	0.32	0.25	0.98
imsl_d_int_fcn_inf	40.42	24.2	14.66	10.37	8.03	0.9
imsl_d_int_fcn_sing	9.54	5.01	2.73	1.82	1.37	0.98
imsl_d_int_fcn_sing_pts	38.19	20.05	10.93	7.29	5.58	0.97
imsl_d_int_fcn_smooth	9.69	5.07	2.76	2.31	1.84	0.92
imsl_d_int_fcn_trig	22.16	12.49	6.97	5.57	4.28	0.91
imsl_d_fast_poisson_2d	59.35	58.23	57.45	56.99	57.11	0.04
imsl_d_constrained_nlp	54.67	27.83	14.45	9.94	7.71	0.98
imsl_d_min_con_gen_lin	277.6	139.66	70.47	47.2	35.91	1
imsl_d_min_uncon_multivar	24.18	12.14	6.12	4.12	3.29	0.99
imsl_d_nonlin_least_squares	12.38	6.28	3.24	2.24	1.78	0.98
imsl_d_covariances	6.04	3.14	1.69	1.27	1.55	0.9
imsl_d_random_poisson	5.11	4.41	4.11	4	4.03	0.25
imsls_d_nonlinear_optimization	127.91	66.42	36.99	27.5	23.82	0.93
imsls_d_nonlinear_regression	12.75	6.62	3.45	2.43	1.91	0.97
imsls_d_covariances	58.54	29.71	15.2	10.38	8	0.99
imsls_d_dissimilarities	25.95	19.85	12.18	8.97	7.37	0.79
imsls_d_factor_analysis	70.16	61.92	51.18	46.46	44.06	0.4
imsls_d_cluster_hierarchical	11.92	8.4	6.15	5.3	5.64	0.63
imsls_d_multivariate_normal_cdf	14.33	7.22	3.63	3.62	1.83	0.98
imsls_d_random_beta	8.02	5.77	4.58	4.2	4.2	0.56
imsls_d_random_exponential_mix	6.57	4.99	4.22	3.92	3.84	0.48
imsls_d_random_gamma	5.42	4.41	3.9	3.75	3.72	0.37
imsls_d_random_general_continuous	9.38	6.38	4.87	4.39	4.12	0.64
imsls_d_random_lognormal	16.04	15	14.5	14.35	14.26	0.13
imsls_d_random_normal	9.71	6.88	5.45	4.97	4.93	0.58
imsls_d_random_poisson	5.02	4.41	4.1	4	4.02	0.24
imsls_d_random_triangular	5.01	4.19	3.79	3.65	3.65	0.32
imsls_d_classification_trainer	161.95	102.1	70.69	59.9	55.56	0.75
imsls_d_genetic_algorithm	13.39	10.91	9.21	8.76	8.53	0.41
imsls_d_mlff_network_trainer	13.94	10.21	7.69	6.91	6.59	0.6
imsls_d_naive_bayes_trainer	0.028	0.016	0.01	0.007	0.007	0.88

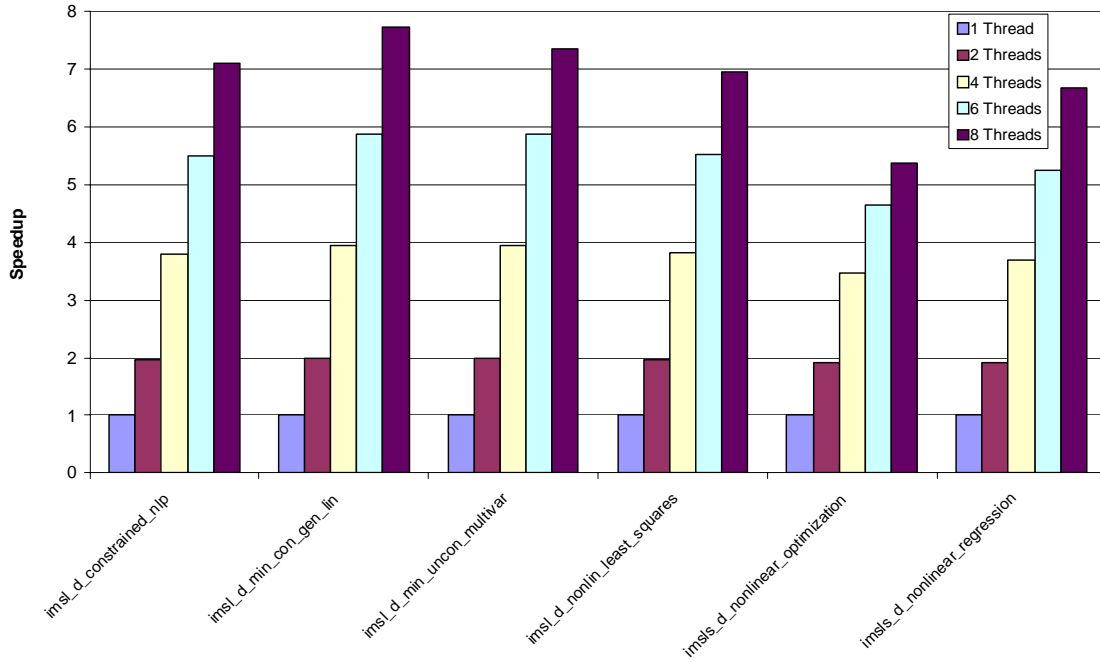
RedHat Linux / Intel 10.1 (64 bit)
Quadrature



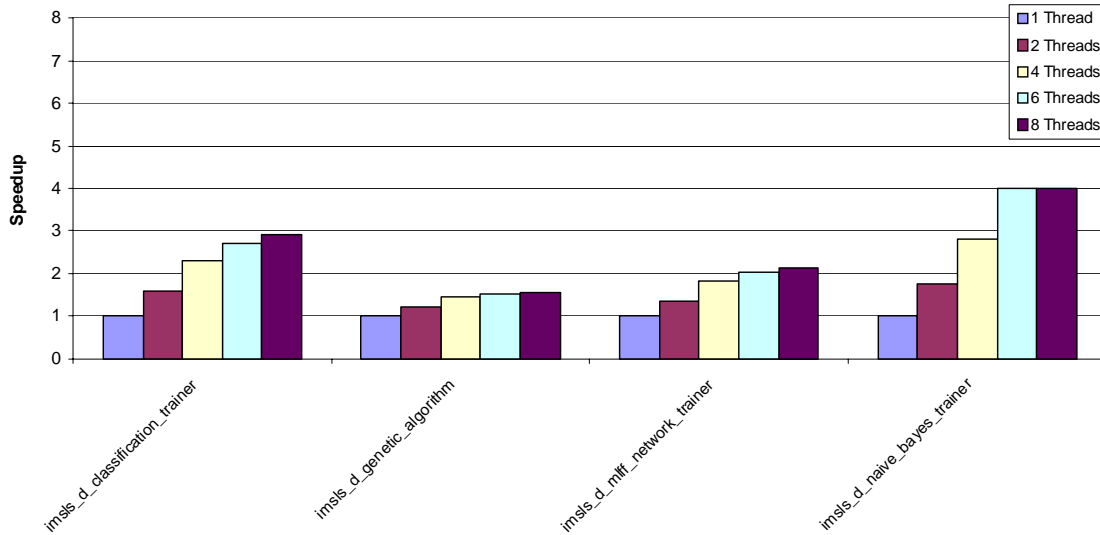
Redhat Linux / Intel 10.1 (64 bit)
Random Number Generation



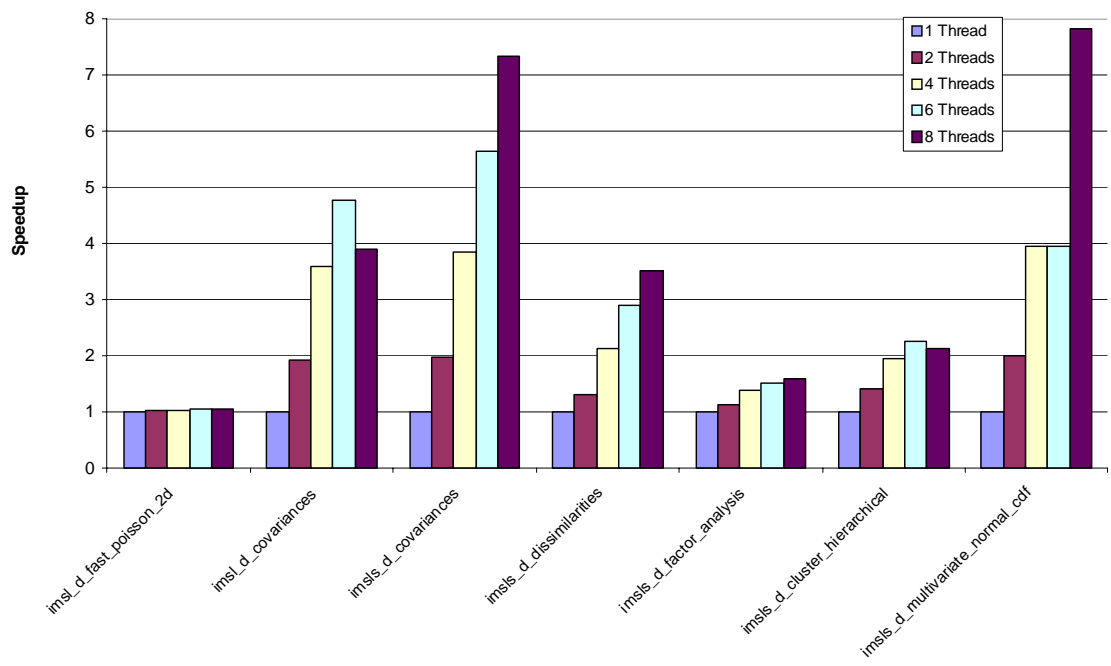
RedHat Linux / Intel 10.1 (64 bit)
Optimization



RedHat Linux / Intel 10.1 (64 bit)
Data Mining



RedHat Linux / Intel 10.1 (64 bit)
Miscellaneous



SuSE 10 with Intel C 10.1 (64-bit)

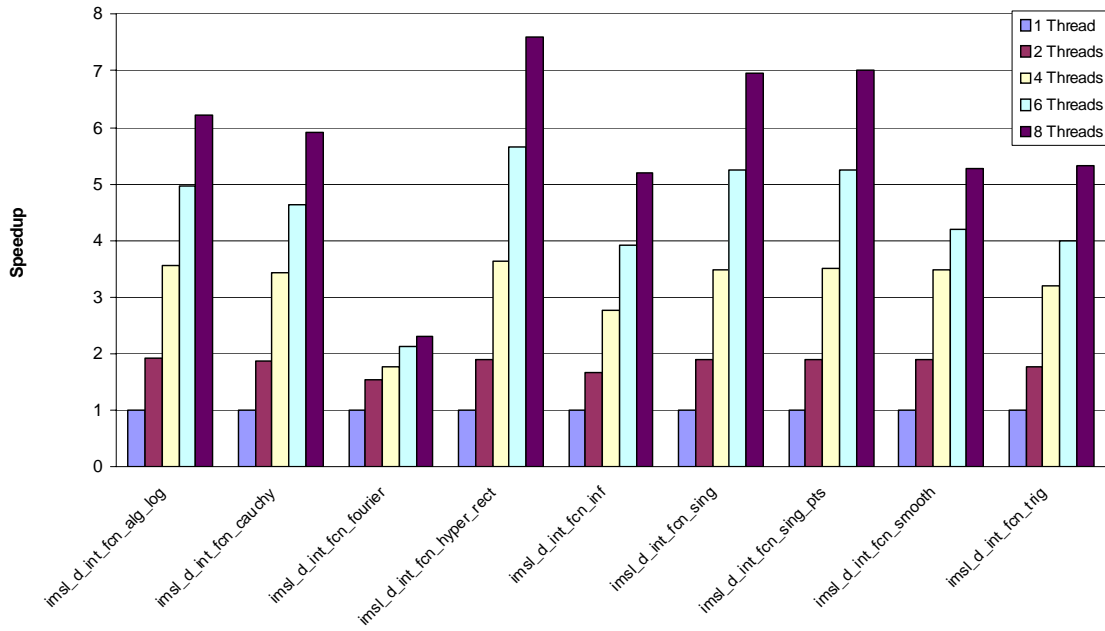
Hardware: Dual Quad Core Xeon E5420 (Harpertown) (8 cores in total) 2.5GHz, 133MHz Front Side Bus

Operating System: SuSE 10

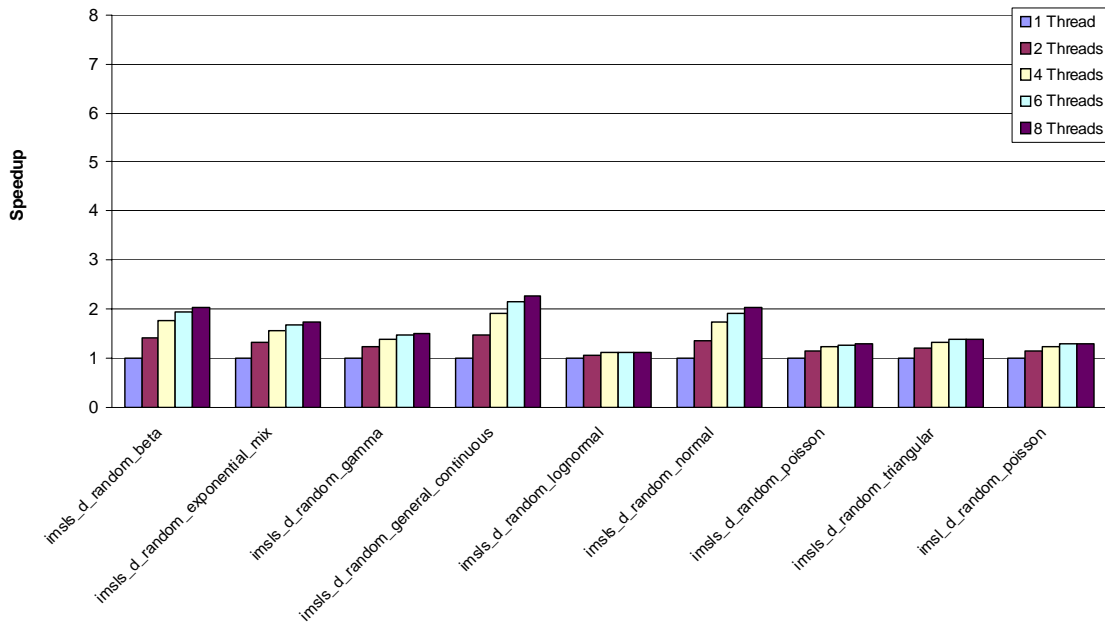
Compiler: Intel C 10.1 (64-bit)

omp_set_num_threads	1	2	4	6	8	Fraction Parallel
<i>IMSL Function</i>	<i>Time (seconds)</i>					
imsl_d_int_fcn_alg_log	22.92	11.98	6.45	4.61	3.69	0.96
imsl_d_int_fcn_cauchy	49.05	26.26	14.27	10.59	8.29	0.95
imsl_d_int_fcn_fourier	4.96	3.19	2.8	2.32	2.15	0.64
imsl_d_int_fcn_hyper_rect	1.75	0.92	0.48	0.31	0.23	0.99
imsl_d_int_fcn_inf	40.3	24.05	14.53	10.26	7.76	0.91
imsl_d_int_fcn_sing	9.54	5	2.73	1.82	1.37	0.98
imsl_d_int_fcn_sing_pts	38.21	19.99	10.9	7.28	5.46	0.98
imsl_d_int_fcn_smooth	9.68	5.07	2.77	2.3	1.84	0.92
imsl_d_int_fcn_trig	22.13	12.45	6.92	5.53	4.15	0.92
imsl_d_fast_poisson_2d	9.06	7.9	7.3	7.1	7	0.26
imsl_d_constrained_nlp	54.47	27.74	16.42	11.32	8.63	0.96
imsl_d_min_con_gen_lin	273.7	137.55	69.33	46.85	35.3	1
imsl_d_min_uncon_multivar	24.16	12.1	6.09	4.12	3.12	1
imsl_d_nonlin_least_squares	12.15	6.19	3.22	2.22	1.73	0.98
imsl_d_covariances	5.81	2.96	1.55	1.16	0.91	0.96
imsl_d_random_poisson	4.88	4.21	3.9	3.8	3.76	0.27
imsls_d_nonlinear_optimization	124.98	66.66	36.54	27.67	23.14	0.93
imsls_d_nonlinear_regression	12.75	6.57	3.43	2.39	1.87	0.98
imsls_d_covariances	58.37	29.24	14.79	10.11	7.71	0.99
imsls_d_dissimilarities	26.04	20.31	12.23	8.98	7.3	0.8
imsls_d_factor_analysis	70.14	61.92	51.14	46.63	44.15	0.39
imsls_d_cluster_hierarchical	11.92	8.41	6.05	5.33	4.89	0.67
imsls_d_multivariate_normal_cdf	10.14	5.1	2.56	2.56	1.29	0.98
imsls_d_random_beta	8.03	5.7	4.54	4.15	3.97	0.58
imsls_d_random_exponential_mix	6.59	4.98	4.18	3.92	3.79	0.49
imsls_d_random_gamma	5.46	4.41	3.9	3.73	3.65	0.38
imsls_d_random_general_continuous	9.42	6.39	4.88	4.38	4.13	0.64
imsls_d_random_lognormal	15.59	14.55	14.03	13.86	13.78	0.13
imsls_d_random_normal	9.73	7.24	5.62	5.08	4.81	0.57
imsls_d_random_poisson	4.82	4.21	3.9	3.8	3.75	0.25
imsls_d_random_triangular	5.03	4.21	3.79	3.66	3.59	0.33
imsls_d_classification_trainer	154.94	96.36	67.22	57.53	52.64	0.75
imsls_d_genetic_algorithm	5.91	4.21	3.19	2.82	2.6	0.63
imsls_d_mlff_network_trainer	13.53	12.05	9.22	8.95	8.22	0.42
imsls_d_naive_bayes_trainer	0.029	0.016	0.01	0.007	0.007	0.88

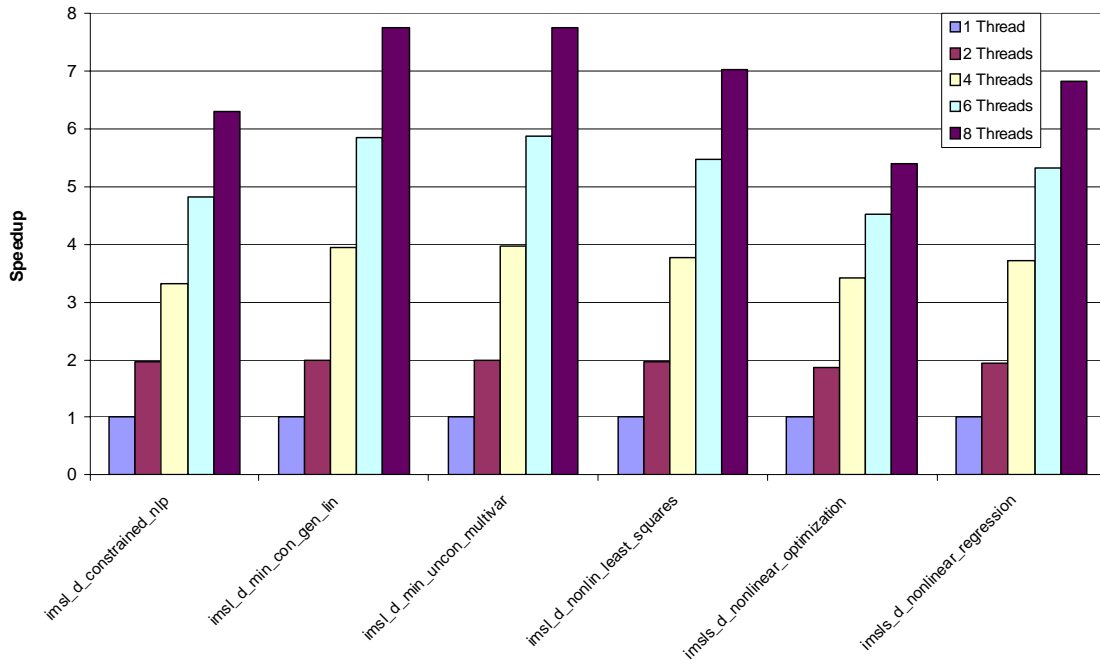
SuSE Linux / Intel C 10.1 (64 bit)
Quadrature



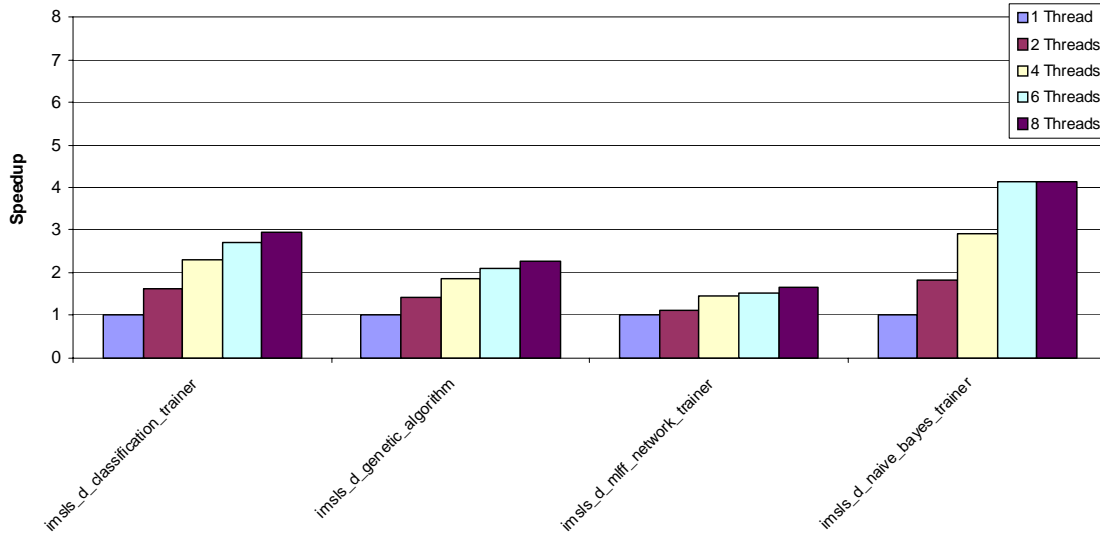
SuSE Linux / Intel C 10.1 (64 bit)
Random Number Generation



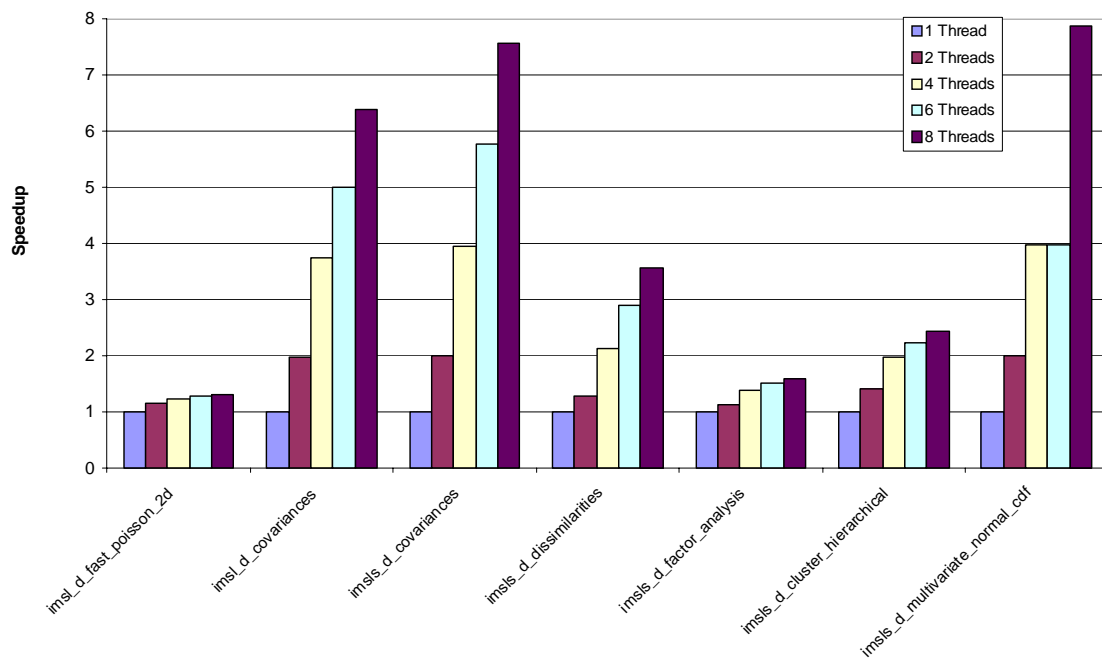
SuSE Linux / Intel C 10.1 (64 bit)
Optimization



SuSE Linux / Intel C 10.1 (64 bit)
Data Mining



SuSE Linux / Intel C 10.1 (64 bit)
Miscellaneous



Sun Solaris Opteron (64-bit)

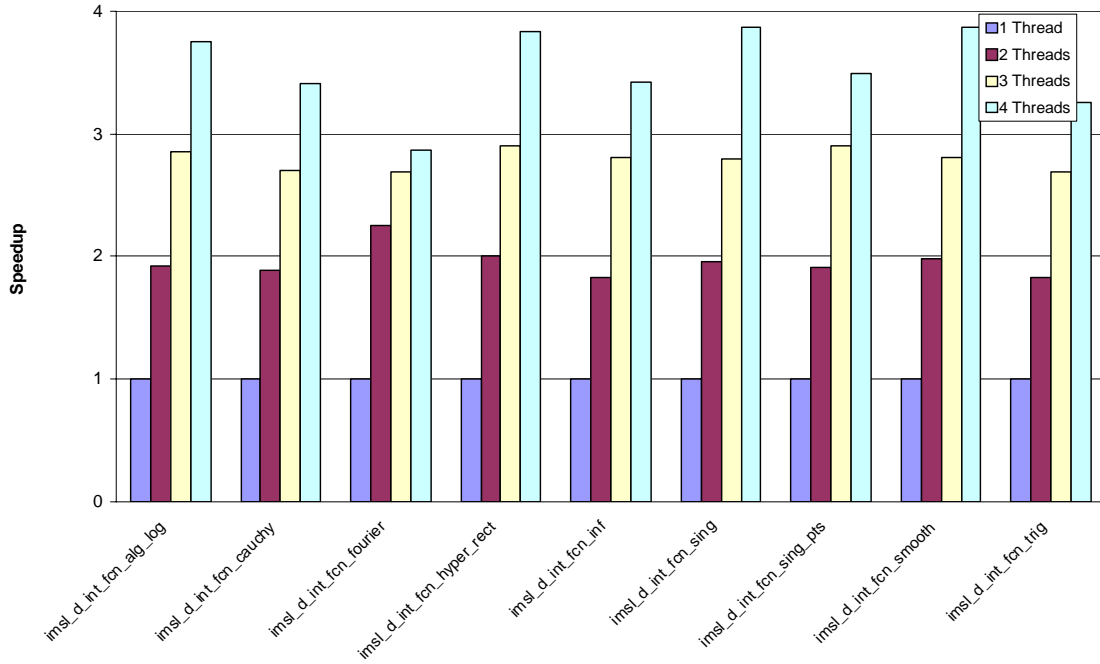
Hardware: Two Dual-Core AMD Opteron Processor 2216 (4 cores in total) 2.4GHz

Operating System: Solaris 10 (SunOS 5.10)

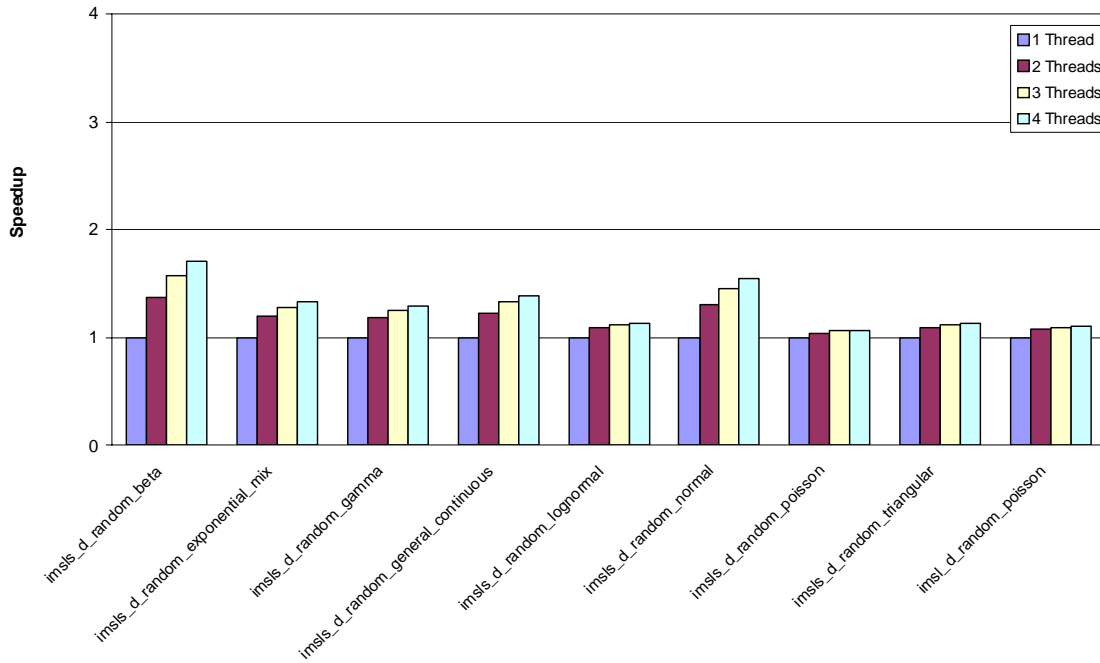
Compiler: Sun Studio 12 (Sun C 5.9) (64-bit)

omp_set_num_threads	1	2	3	4	Fraction Parallel
<i>IMSL Function</i>	<i>Time (seconds)</i>				
imsl_d_int_fcn_alg_log	64.39	33.6	22.6	17.17	0.98
imsl_d_int_fcn_cauchy	143.13	75.89	53.05	42.02	0.94
imsl_d_int_fcn_fourier	13.11	5.82	4.88	4.57	0.9
imsl_d_int_fcn_hyper_rect	1.8	0.9	0.62	0.47	0.98
imsl_d_int_fcn_inf	104.64	57.38	37.3	30.6	0.95
imsl_d_int_fcn_sing	27.55	14.09	9.87	7.12	0.98
imsl_d_int_fcn_sing_pts	114.48	59.96	39.47	32.77	0.96
imsl_d_int_fcn_smooth	28.32	14.31	10.1	7.32	0.98
imsl_d_int_fcn_trig	63.34	34.71	23.53	19.45	0.93
imsl_d_fast_poisson_2d	35.64	33.44	32.78	32.35	0.12
imsl_d_constrained_nlp	31.8	16.55	11.12	8.6	0.97
imsl_d_min_con_gen_lin	141.18	71.28	47.48	35.98	0.99
imsl_d_min_uncon_multivar	20.97	10.57	7.07	5.37	0.99
imsl_d_nonlin_least_squares	11.02	5.56	3.82	2.95	0.98
imsl_d_covariances	8.98	6.3	4.56	4.37	0.7
imsl_d_random_poisson	13.21	12.28	12.1	12	0.13
imsls_d_nonlinear_optimization	136.1	79.21	59.52	52.78	0.83
imsls_d_nonlinear_regression	12.19	7.84	5.66	9.15	0.58
imsls_d_covariances	227.12	160.9	112.27	90.23	0.78
imsls_d_dissimilarities	27.71	21.2	15.87	12.73	0.68
imsls_d_factor_analysis	92.68	82.03	70.2	64.92	0.37
imsls_d_cluster_hierarchical	39.9	36.18	27.73	26.25	0.43
imsls_d_multivariate_normal_cdf	14.24	8.19	6.31	4.62	0.88
imsls_d_random_beta	26.02	18.89	16.48	15.27	0.55
imsls_d_random_exponential_mix	17.47	14.54	13.58	13.09	0.33
imsls_d_random_gamma	16.75	14.2	13.35	12.91	0.31
imsls_d_random_general_continuous	18.63	15.12	13.95	13.37	0.38
imsls_d_random_lognormal	47.81	44.03	42.78	42.19	0.16
imsls_d_random_normal	23.19	17.76	15.93	15.02	0.47
imsls_d_random_poisson	12.83	12.28	12.09	12	0.09
imsls_d_random_triangular	13.8	12.71	12.34	12.16	0.16
imsls_d_classification_trainer	205.01	131.72	105.43	92.09	0.73
imsls_d_genetic_algorithm	10.94	8.5	7.66	7.98	0.4
imsls_d_mlff_network_trainer	17.12	11.65	10.81	10.74	0.53
imsls_d_naive_bayes_trainer	0.037	0.063	0.112	0.129	-2.3

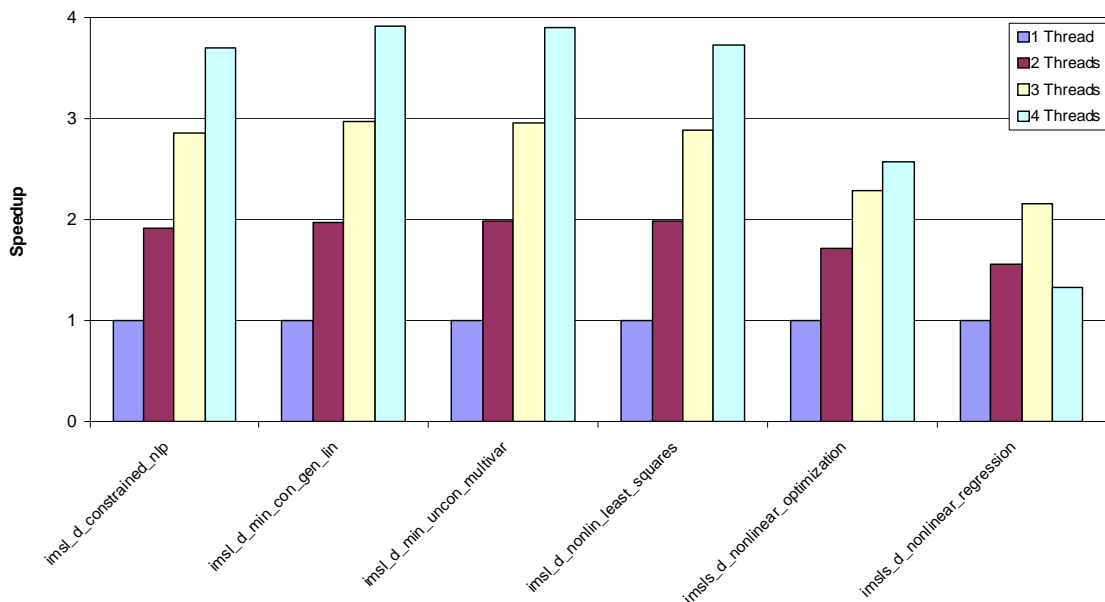
Solaris/AMD (64 bit)
Quadrature



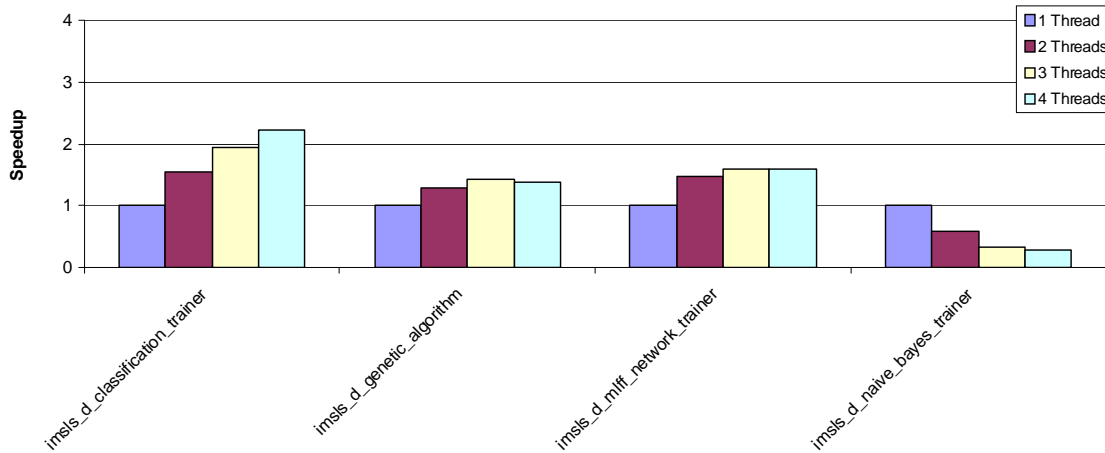
Solaris/AMD (64 bit)
Random Number Generation

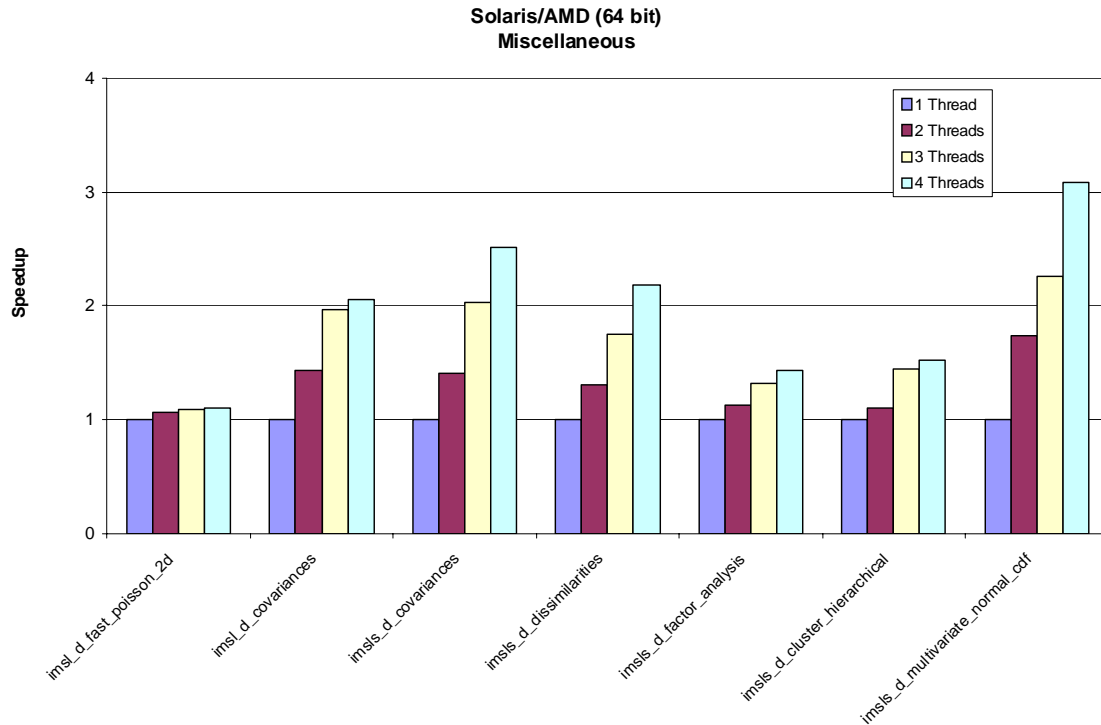


Solaris/AMD (64 bit)
Optimization



Solaris/AMD (64 bit)
Data Mining





Test Problems

The benchmarks include some problems used to benchmark several different IMSL C Library functions. This section provides details on the various test problems in the benchmark suite.

Quadrature Functions

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

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

Optimization Neural Network Problem

A simple neural network problem was used to benchmark the optimization functions. 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 function 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

$$x_{ij} = \cos(301i + 401j)$$

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

Full Set of Benchmark Problems

- `imsl_d_int_fcn_2d`

$$\int_0^{1+\sqrt{x+1}} \int_{-\sqrt{x+1}}^9 \sum_{i=3}^4 \sum_{j=3}^4 (i + j/100)^{x-y} dy dx$$
- `imsl_d_int_fcn_alg_log`

$$\int_0^1 f_1(x) x^{1/4} (1-x)^{1/4} \log(x) dx$$
- `imsl_d_int_fcn_cauchy`

$$\int_0^1 \frac{f_1(x)}{x-0.5} dx$$
- `imsl_d_int_fcn_fourier`

$$\int_a^{\infty} f_1(x) \sin(\omega x) dx \text{ where } a = 2.5 \text{ and } \omega = 3.5.$$
- `imsl_d_int_fcn_hyper_rect`

$$\int_0^1 \cdots \int_0^1 \sum_{i=0}^9 \Phi(x_i) dx_1 \cdots dx_9, \text{ where } \Phi \text{ is the normal cumulative distribution function.}$$
- `imsl_d_int_fcn_inf`

$$\int_0^{\infty} f_2(x) dx$$
- `imsl_d_int_fcn_sing`

$$\int_0^1 f_1(x) dx$$
- `imsl_d_int_fcn_sing_pts`

$$\int_0^1 f_1(x) dx \text{ with singular points given as } 0.3, 0.5 \text{ and } 0.7.$$
- `imsl_d_int_fcn_smooth`

$$\int_0^1 f_1(x) dx$$

- `imsl_d_int_fcn_trig`

$$\int_0^1 f_1(x) \cos(\omega x) dx \text{ where } \omega = 2.5.$$

- `imsl_d_fast_poisson_2d`

The PDE $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + 3u = -2 \sin(x + 2y) + 16e^{2x+3y}$ with the boundary

condition $\frac{\partial u}{\partial y} = 2 \cos(x + 2y) + 3e^{2x+3y}$ on the bottom side and $u = \sin(x + 2y) + e^{2x+3y}$

on the other three sides was solved on a 4,096 by 4,096 grid.

- `imsl_d_constrained_nlp`

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

- `imsl_d_min_con_gen_lin`

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

- `imsl_d_min_uncon_multivar`

The optimization neural network problem was used.

- `imsl_d_nonlin_least_squares`

The optimization neural network problem was used.

- `imsl_d_covariances`

The variance-covariance matrix was computed from a random matrix with 100 observations and 5,000 variables with no missing values.

- `imsl_d_random_poisson`

Generates 10^8 random numbers with a mean value of 0.5.

- `imsls_d_nonlinear_optimization`

The optimization neural network problem was used.

- `imsls_d_nonlinear_regression`

The optimization neural network problem was used.

- `imsls_d_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. The value of the optional argument, `IMSL_MISSING_VALUE_METHOD`, was 3.

- `imsls_d_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 Mahalanobis distance method was used.

- `imsls_d_factor_analysis`

Factor analysis was performed on the 1,000 by 1,000 covariance matrix

$$a_{ij} = \frac{1}{i+j+1} + 0.2\delta_{ij}, \text{ where } \delta_{ij} \text{ is one if } i \text{ equals } j \text{ and is zero otherwise.}$$

- `imsls_d_cluster_hierarchical`
The hierarchical clustering was done on a uniform random 10,000 by 10,000 distance matrix.
- `imsls_d_multivariate_normal_cdf`
The multivariate normal CDF was computed for $k = 200$ variates. The upper bounds were all equal to 5. The means were $\mu_i = \sqrt{i}$, for $i = 0, \dots, k$. The variance-covariance matrix was $\Sigma_{ij} = \frac{1}{i+j+1} + 0.2\delta_{ij}$, where δ_{ij} is one if i equals j and is zero otherwise.
- `imsls_d_random_beta`
Generates 10^8 random numbers with $p = 1$ and $q = 3$.
- `imsls_d_random_exponential_mix`
Generates 10^8 random with $\theta_1 = 2$, $\theta_2 = 1$, and $p = 0.5$.
- `imsls_d_random_gamma`
Generates 10^8 random numbers with shape parameter equal to 1.
- `imsls_d_random_general_continuous`
Generates 10^8 random numbers from a general continuous distribution, here the beta distribution with $p = 3$ and $q = 2$.
- `imsls_d_random_lognormal`
Generates 10^8 random numbers with a mean value of 3 and a standard deviation of 2.
- `imsls_d_random_normal`
Generates 10^8 random numbers.
- `imsls_d_random_poisson`
Generates 10^8 random numbers with a mean value of 0.5.
- `imsls_d_random_triangular_numbers`
Generates 10^8 random numbers.
- `imsls_d_mlff_classification_trainer`
The Poker Hand data set from the [UCI Machine Learning Repository](#)¹ was used. The set is a variation of the set from Robert Cattral and Franz Oppacher². The set contains 25,010 observations. Each observation consists of 10 nominal values resulting in 85 inputs to the classification neural network. The observations are classified into 10 classes. Training was done with 10 Stage I epochs, each with 6,000 patterns randomly chosen from the given 25,010 patterns. No Stage II training was performed.
- `imsls_d_genetic_algorithm`
The n -Queens problem was solved with the number of queens $n = 100$. The problem is to place n queens on an n by n chessboard such that no queen can capture another queen.

¹ Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.

² R. Cattral, F. Oppacher, D. Deugo. Evolutionary Data Mining with Automatic Rule Generalization. Recent Advances in Computers, Computing and Communications, pp.296-300, WSEAS Press, 2002.

- `imsls_d_mlff_network_trainer`
The optimization neural network problem was used.
- `imsls_d_naive_bayes_trainer`
The Poker Hand dataset was used. This is the same dataset as used for benchmarking `imsls_d_mlff_classification_trainer`. The Naïve Bayes trainer used all 25,010 observations.

Sample Benchmarking Code

The following code benchmarks the functions `imsls_d_random_normal`, `imsls_d_genetic_algorithm` and `imsl_d_fcn_inf`. The test cases described above are used.

This code requires OpenMP. The standard OpenMP function, `omp_set_num_threads`, is used to set the number of threads during each phase of the benchmarking. This code assumes the use of a maximum of eight threads.

The functions `imsl_omp_options` and `imsls_omp_options` are used to indicate that the functions passed as arguments to the IMSL C Library functions are thread-safe. In this sample, these are the functions `fcnld_inf` and `queensFitness`. By default, such user supplied functions are not evaluated in parallel.

Each benchmark is run five times to reduce errors in the timings. The code includes a function, `wall_clock_time`, to measure the actual elapsed (“wall clock”) time in seconds.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <imsl.h>
#include <imsls.h>
static double benchmark_random_normal();
static double benchmark_int_fcn_inf();
static double fcnld_inf(double x);
static double benchmark_generic_algorithm();
static double queensFitness(Imsls_d_individual* individual, int* n_queens);
static double wall_clock_time();

int main()
{
    int    num_threads[] = {1, 2, 4, 6, 8};
    int    n_threads = 5;
    int    n_rep = 5;
    int    n_benchmarks = 3;
    int    i_threads, i_rep, i_benchmark;
    double *times;

    imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
    imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    times = (double*)calloc(n_threads*n_benchmarks, sizeof(double));
    for (i_threads = 0; i_threads < n_threads; i_threads++)
```

```

    {
        omp_set_num_threads(num_threads[i_threads]);
        for (i_rep = 0; i_rep < n_rep; i_rep++)
        {
            times[i_threads*n_benchmarks+0] += benchmark_random_normal();
            times[i_threads*n_benchmarks+1] += benchmark_generic_algorithm();
            times[i_threads*n_benchmarks+2] += benchmark_int_fcn_inf();
        }
    }

    printf("%s      %s      %s      %s\n",
           "n_threads", "random_normal", "generic_algorithm", "int_fcn_inf");
    for (i_threads = 0; i_threads < n_threads; i_threads++)
    {
        printf("%5d  ", num_threads[i_threads]);
        for (i_benchmark = 0; i_benchmark < n_benchmarks; i_benchmark++)
        {
            printf("%20.2f",
                   times[i_threads*n_benchmarks+i_benchmark]/n_rep);
        }
        printf("\n");
    }
}

static double benchmark_random_normal()
{
    int      n = (int)1.0e8;
    double   time;
    double   *random;

    time = wall_clock_time();
    random = imsls_d_random_normal(n, 0);
    time = wall_clock_time() - time;
    imsls_free(random);
    return time;
}

static double benchmark_int_fcn_inf()
{
    double time;
    double result;

    time = wall_clock_time();
    result = imsl_d_int_fcn_inf(fcnld_inf, 0.0, IMSL_BOUND_INF,
                               0);
    time = wall_clock_time() - time;
    return time;
}

static double fcnld_inf(double x)
{
    double sum = 0.0;
    int     i, j;

    for (i = 3; i < 100000; i++)

```

```

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

static double benchmark_generic_algorithm()
{
    int      i;                /* index variables          */
    int      n = 500;         /* population size          */
    int      n_generations;   /* number of generations   */
    int      n_queens = 100;  /* number of nominal phenotypes*/
    int      n_categories[100];
    double   maxFit;          /* maximum fitness hurdle  */
    double*  genStats;        /* generation statistics    */
    Imsls_d_chromosome* chromosome; /* chromosome data structure */
    Imsls_d_individual* best_individual; /* optimum                  */
    Imsls_d_population* population; /* population data structure */
    double   time;

    imsls_random_seed_set(12345);

    /*
     * Setup Problem
     */
    time = wall_clock_time();
    maxFit = n_queens - 0.5;
    for(i = 0; i < n_queens; i++) {
        n_categories[i] = n_queens;
    }
    chromosome = imsls_d_ga_chromosome(
        IMSLS_NOMINAL, n_queens, n_categories,
        0);

    population = imsls_d_ga_random_population(n, chromosome,
        IMSLS_PMX_CROSSOVER,
        IMSLS_FITNESS_FCN_WITH_PARS, queensFitness, &n_queens,
        0);

    /*
     * Solve Problem
     */
    best_individual = imsls_d_genetic_algorithm(NULL, population,
        IMSLS_FITNESS_FCN_WITH_PARS, queensFitness, &n_queens,
        IMSLS_PMX_CROSSOVER,
        IMSLS_LINEAR_SCALING, 2.0,
        IMSLS_CROSSOVER_PROB, 0.7,
        IMSLS_MUTATION_PROB, 0.01,
        IMSLS_MAX_GENERATIONS, 10000,
        IMSLS_MAX_FITNESS, maxFit,
        IMSLS_GENERATION_STATS, &genStats,
        IMSLS_N_GENERATIONS, &n_generations,
        0);
    time = wall_clock_time() - time;
}

```

```

    imsls_d_ga_free_individual(best_individual);
    imsls_d_ga_free_population(population);
    imsls_free(chromosome);
    return time;
}

static double queensFitness(Imsls_d_individual* individual, int* n_queens)
{
    int i, j, k, p;    /* Index variables */
    int f = 0;        /* Fitness value */
    int *allele = individual->chromosome->allele;
    for(i = 0; i < *n_queens-1; i++) {
        for(j = i+1; j < *n_queens; j++) {
            k = allele[i] - allele[j];
            p = i - j;
            if (k*k == p*k) f++;
        }
    }
    f = *n_queens - f;
    return (double)f;
}

static double wall_clock_time()
{
#ifdef _WIN32
    return clock() / (double)(CLOCKS_PER_SEC);
#else
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return (tp.tv_sec + 1.0e-6*tp.tv_usec);
#endif
}

```

Conclusion

The use of OpenMP directives within the IMSL C Numerical Library is an effective means of achieving scalable parallelism in a portable fashion across a wide variety of computing platforms. While other technologies exist for SMP parallelization, OpenMP is a stable, well-known and mature choice with broad platform support, which is a key for the IMSL C Numerical Library. Some functions show very good, nearly linear, scaling up to 8 threads, which is quite good considering the algorithms were not re-written, but rather just had OpenMP directives added in strategic places. This paper discusses a wide range of algorithms parallelized in IMSL C Numerical Library version 7.0, but this is not an exhaustive list; please refer to the product documentation for a complete list. Finally, Visual Numerics 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.

About the Author

Dr. John Brophy, Ph.D. has been with Visual Numerics since 1983. Recently he has been leading in the design and implementation of the parallelization of selected functions in the IMSL C Numerical Library of which he was an original designer. Previously he led in the design and implementation of an object-oriented, native C# and Java, library for mathematics, statistics, finance and charting. He also worked with the Java Grande Forum to make Java a better language for numerical computing.