Optimizing ITK's Registration Methods for Multi-processor, Shared-memory Systems

Release 3.0

Stephen Aylward, Julien Jomier, Sebastien Barre, Brad Davis, and Luis Ibanez

September 10, 2007

Kitware, Inc.
http://www.kitware.com

Abstract

This document describes work-in-progress for refactoring ITK's registration methods to exploit the parallel, computation power of multi-processor, shared-memory systems. Refactoring includes making the methods multi-threaded as well as optimizing the algorithms. API backward compatibility is being maintained. Helper classes that solve common registration tasks are also being developed.

The refactoring has reduced computation times by factors of 2 to 200 for metrics, interpolators, and transforms computed on multi-processor systems. Extensive sets of tests are being developed to further test operation and backward compatibility.

More information on this project is available at:
http://www.na-mic.org/Wiki/index.php/ITK_Registration_Optimization

NOTE #1: Recent changes to ITK and BatchMake in preparation for the release of ITK v3.4 and in support of Slicer have caused the build of this project to fail. Please visit this project's dashboard (available via the link above) prior to downloading the code to ensure that the code is working.

NOTE #2: The software is being incorporated directly into ITK. It should be available via ITK's CVS approximately one month after the release of version 3.4 of the Insight Toolkit (October/November 2007). The above wiki page will contain up-to-date information.

Contents

1 Introduction

The work-in-progress described in this paper focuses on refactoring ITK's registration framework [Ibanez 2002] to take advantage of the parallel, computation capabilities of multi-processor, shared-memory systems. The refactoring consists of multi-threading the methods as well as optimizing the algorithm implementations. Simultaneously, backward compatibility with ITK's existing classes must be maintained.

Regarding multi-threading, while ITK's execution pipeline for filters supports multi-threading, ITK's registration framework is not multi-threaded and several of its methods are not thread safe. Many of the registration modules presented in this paper, on the other hand, make heavy use of multi-threading, and all are thread safe.

Regarding algorithm optimization, it should be noted that ITK's registration methods were built to support research, so their implementations emphasized easy-to-understand code and a plug-and-play framework. ITK's registration framework is illustrated in Fig. 1. Modules can be substituted into the transform, interpolator, metric, and solver components. In contrast, the work presented in this paper emphasizes computational speed, i.e., certain implementations have been specialized to speed specific combinations of modules. For example, all image-to-image metrics now detect if they are being combined with a cubic b-spline transform, and in those situations, appropriate pre-computations and algorithmic shortcuts are taken. Nevertheless, in keeping with the policies of the Insight Software Consortium, backward compatibility with ITK's existing API was maintained.

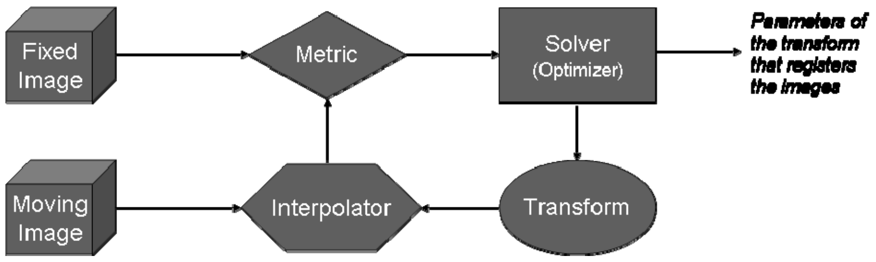


***Figure 1**. ITK's registration framework. Modularity promotes research and experimentation. For optimization, specific combinations of modules are being integrated to speed specific applications.*

ITK's registration methods needed to be refactored because of the growing role of registration in medical image analysis and the extensive registration computation time (over 15 hours) that has been anecdotally reported when using ITK. As driving problems, we selected registration for atlas formation and registration for atlas-based segmentation, particularly as featured in the EMSegmenter algorithm developed by Dr. Kilian Pohl [Pohl 2006]. Dr. Pohl's algorithm is widely used as the EMSegmenter module in Slicer (http://www.slicer.org). His algorithm relies on the inter-subject mapping of cortical MRIs. An analysis of this driving problem led us to focus on the following ITK modules and their combinations:

Transforms:	Rigid, affine, and BSplines
Interpolators:	Linear and cubic-BSplines
Metrics:	Mattes mutual information [Mattes 2003] and mean-squares error

Solvers:	One-plus-one evolutionary optimization [Styner 2000] and LBFGSB (Limited memory Broyden Fletcher Goldfarb Shannon minimization with simple Bounds) optimization [Zhu 1997]

Beyond the above driving problem, the work presented in this paper includes updates to every transform, interpolator, and image-to-image metric in ITK due to base-class modifications needed to support thread safety. Ongoing work is focusing on further speed improvements and extensive testing.

Further details on the background and significance for the work are given next. Then the optimization techniques and the measures used to quantify their performance are presented. The performance improvements gained on two different multi-processor platforms are then presented. The conclusion discusses future work. Additional and updated details are available on the web at the following link:

http://www.na-mic.org/Wiki/index.php/ITK_Registration_Optimization.

2 Background and Significance

By focusing on multi-processor, shared-memory systems, the work presented herein addresses the current trend in computer hardware for laptops, desktops, workstations, computer servers, and grid nodes. In those systems, the trend is to use multi-core processors such as Intel's Core2Duo chip. Intel estimates that over 85% of the processors it is producing are multi-core processors. Today, nearly every new desktop contains a multi-core processor, and mid-range ($4,000) compute servers typically have eight dual-core processors. On a server with eight dual-core processors, 16 independent tasks can simultaneously process a common dataset – thereby potentially generating a result 16 times faster. Intel predicts that within three years, 32-core processors will be the norm – thereby suggesting desktops with 32 cores and compute servers with 256+ cores.

Distributing a task across the processors and cores of a system is accomplished using multi-threading. Typically, threads are sections of a program that can be computed independently and simultaneously. Defining a multi-threaded implementation of an algorithm requires consideration of how information will be allocated to, computed on, and recombined from the threads. Some algorithms are easy to implement in a multi-threaded manner while other algorithms cannot be effectively multi-threaded. Given the multi-core future of computer hardware, algorithm researchers as well as algorithm implementers need to understand multi-threading.

The techniques used to develop optimized registration modules for ITK are summarized next.

3 Techniques Used in Development and Testing

This ongoing project involved code optimization and threading, process prioritization and timing, and comparative quantification of performance gains.

3.1 Optimizations and Threads

The work began by focusing on non-rigid registration and then spread to include other registration challenges. Numerous small changes were made to the code. The major changes are as follows:

1) Developed multi-threaded versions of ITK's image-to-image metrics. Mattes mutual information metric, the mean-squared error metric, and similar voxel-matching metrics now derive from a common image-to-image-metric base class that provides:

    a. Sampling strategies for computing the metric using a (sub-)sample of points in the images: (i) random sub-sampling, (ii) user-specification of the sub-sample points, or (iii) using every voxel in the images. Furthermore, (iv) a mask can be used to limit the domain of (i) random sub-sampling and (iii) using every voxel.

    b. Distributing the (sub-)sampled points across threads when computing the metric,

    c. Threading the pre-computation of a gradient image to speed metric derivative computation,

    d. Caching the affine pre-transform of the (sub-)sampled points when a metric is combined with a BSpline transform,

    e. Pre-computing the BSpline coefficients for the (sub-)sampled points when a metric is combined with a BSpline transform,

    f. Providing "hooks" so that derived classes can conduct threaded computations in three synchronized phases. This approach was chosen to avoid the use of mutex locks – additional details are given below.

2) Made ITK transforms thread safe. While most ITK transforms do not need to use member variables to transform a point, kernel-based transforms (such as the b-spline and thin-plate-spline trasnforms) benefit from using member variables to avoid having to repeatedly allocate and free large data structures that hold intermediate values. ITK's original implementation of select transforms, such as many of the kernel-based transforms, were not thread safe because the use of those intermediate-value member variables was not thread-safe. In the presented implementations, the base itkTransform class was modified so that all transforms may have thread-specific data.

3) Identified and optimized frequently used code segments. Several commonly used and computationally expensive routines were identified using Lightweight Technologies' LTProf (http://www.lw-tech.com/), Linux's Valgrind (http://valgrind.org) and Intel's VTune (http://www.intel.com) profiling tools. For example, ITK's image Fill() method uses a pixel-by-pixel iterative assignment technique. In registration, images are often used to store intermediate values such as joint histograms. As a result, every value and derivative computation was making multiple calls to Fill(), typically to set those values/images to zero. Given that derivatives of joint histograms equate to images with 30,000+ pixels, the Fill() method was accounting for nearly 10% of the time spent on a metric value computation. Replacing Fill() with a single system-level call (memset) reduced computation time by nearly 10%. Similar savings were discovered for other frequently called routines.

Mutex lock checking was determined to be extremely time consuming on SunOS and other platforms. The initial implementations made use of mutex locks. Experiments revealed, however, that even when mutex collisions did not occur, the computation time of a single mutex check per sample exceeds the per voxel time for computing a Mattes Mutual Information metric value.

Instead of using mutex locks, the current implementations create unique copies of relevant variables for each thread and then join their results after thread completion. The challenge is to manage the allocation of data to those variables and the integration of the results from those variables. If that allocation and integration is conducted in the main thread, those costs (i.e., computation times) can exceed the benefits of threading. To alleviate those costs, the implementations provide the "hooks" for three, synchronized sets of threads, as discussed above in Optimization, 3.1.1f. Typically, first one set of threads is run to initialize the thread specific variables. The second set of threads is then run to process the samples allocated to each thread and accrue their results in their thread-specific variables. The third set of threads is then run to combine the results from the thread-specific variables into the final result. A set of threads is not begun until the prior set of threads completes. Consider, for example, that for Mattes Mutual Information computation there is a joint histogram variable. To partition the samples to different threads, each thread is given its own copy of the joint histogram variable. During the first set of threads, each thread initializes its joint histogram to zero. During the second set of threads, the each thread's samples are inserted into that thread's histogram. During the third set of threads, the summation of the total joint histograms is threaded – that is, each thread is responsible from summing a different region of the total joint histogram from the thread-specific histogram variables. Different initialization, allocation, and integration strategies are used for different variables.

Our work also revealed that certain changes and parameterizations could improve performance on one platform and degrade performance on others. For example, one surprising observation was that using more threads than processors had negligible effect on computation speed on certain machines, while on other platforms using more threads than available processes resulted in significant degradation in performance. Additionally, the cost to initiate a thread on certain platforms is much greater than on other platforms.

The complexity of the optimization task necessitated establishing an infrastructure that verified backward compatibility and monitored the effects of the day-to-day code and parameter changes for multiple platforms. That infrastructure is discussed next.

3.2 Backward Compatibility and Performance Monitoring

The registration methods presented were implemented to maximize their backward compatibility with and speed relative to the standard ITK implementations. Backward compatibility was judged by providing consistent results and maintaining a consistent API with respect to the standard ITK implementations. Speed was quantified using real-world ("wall clock") time. These criteria were tested as follows.

Backward Compatibility: To ensure consistent software, i.e., having a backward compatible API and generating similar results, the infrastructure included iterative, interleaved testing of standard ITK methods and the optimized methods. The consistency of these tests was controlled by writing them as C++ frameworks that used symbolic placeholders where the registration methods were left unspecified. CMake's CONFIGURE_FILE command was then used to substitute calls to the standard and optimized methods at the symbolic placeholders. Tests of the standard ITK methods were interleaved with tests of the optimized methods to quantify and thereby enable compensation for workload fluctuations on the testing platforms. The standard tests also served to verify that the optimized methods produced similar results, as explained next.

The optimized methods, for any given number of threads, produce consistent results, and in most cases, when run using a single thread, produce the same results as the standard methods. However the optimized methods' results may vary as the number of threads is varied. Result variation by number of threads arises from across-thread problem partitioning. As mentioned previously, threading often necessitates the use of additional intermediate values for each additional threads used. Such changes in intermediate processing may have an effect on the accumulated values produced, in part due to machine precision. Such variations are likely to have a negligible effect on the result from a single registration optimization iteration, but their cumulative effect over many iterations may be greater. Nevertheless, the implementations, computations, and final outcomes are still valid.

For reasons of consistency as explained above, and to ensure backward compatibility for user-derived (potentially not thread-safe) code that is not distributed with ITK, the optimized methods default to running single threaded. The user must call `metric->RunMultiThreaded(void)` to run with the ITK defined default number of threads. The ITK defined default number of threads is the lesser of 8 or the number processors on the system, see Insight/Code/Common/itkMultiThreader.h. The user may also set the number of threads by calling `metric->SetNumberOfThreads(int)`. Either call must be made prior to calling `metric->Initialize(void)`.

Performance Monitoring: There were two components to process monitoring: speed quantification and centralized reporting. This project made significant developments in each area.

For this project, speed quantification involves code instrumentation, process prioritization, and workload compensation. Code instrumentation refers to the explicit use of start and stop timing calls in the tests. Time for loading/creating the fixed and moving images and for initializing the registration framework is not included in the timing results. The timers are provided by ITK's high performance, cross-platform timing routines (Insight/Code/Common/itkRealTimeClock.h). Since the tests used real-world timers and since the workload placed by other users on the test platforms could not be controlled and varied day to day, process prioritization and workload compensation are used. Specifically, to mitigate some of the effects of concurrent users, an itkHighPriorityRealTimeClock class (located in BWHITKOptimization/Code/Utilities) is defined. When an instance of that class is instantiated in a program, the process' priority is increased. When destructed, the process' priority is returned to normal. Furthermore, as previously mentioned, the standard ITK methods' tests are interleaved with the optimized methods' tests. The speed of the standard ITK methods is thus available to normalize the speed of the optimized methods. Thereby, three measures of performance are reported.

The first measure is the absolute (real-world) run time of the method, Equ. 1. In addition to the number of threads, if a metric is being measured then its time is typically parameterized by the number of samples used to compute the metric, the type of interpolation used, the transform used, and the number of metric value or derivative computations performed. If a transform is being measured, then its time is typically parameterized by the type of interpolation used and the number of point transformations performed.

$$T_n = A + B/n \tag{1}$$

In Equ. 1, A is the portion of the method that cannot be threaded and B is portion of the method that can be threaded.

The second measure is often called *speedup* or *Amdahl's law* [Amdahl 1967].

$$C_n = T_1 / T_n = (A + B) / (A + B/n) \tag{2}$$

Amdahl's law states that the speedup of a threaded algorithm will eventually begin to decrease as additional threads are used, since the serial component remains constant and since the above equation does not account for the cost of distributing data to and integrating data from the parallel processes. Such behavior is revealed in the results presented in this paper.

The third measure is called *optimization ratio*, Equ. 3. It compares the absolute run time of the unoptimized (single-threaded) version of a method with its optimized version running in *n* threads. This metric compensates for workload variations and partially compensates for cross-platform differences in processor type and speed.

$$R_n = T_1(\text{unoptimized}) / T_n(\text{optimized}) \qquad (3)$$

The above three metrics only require the computation of the real-world run times of the unoptimized and optimized methods on the testing platforms. The challenge is managing the variety of tests and their parameters. We developed a novel system for managing these tests, as explained next.

Centralized Performance Reporting: To compute and collect the above measures on the testing platforms, BatchMake (http://www.batchmake.org) and CMake (http://www.cmake.org) are integrated. BatchMake is a cross-platform system for scripting parameter space explorations and batch image processing. BatchMake also includes a web-based reporting system, where results from BatchMake scripts can be collected, monitored, graphed, and compared over the web. By integrating BatchMake with CMake, a cross-platform testing system with centralized reporting is established. Using this system, a cmake build/test sequence now automatically initiates the following:

a. BatchMake's implementation of the Whetstones benchmark is run to compute the speed, i.e., MFLOPS and MIPS ratings, of a single core on each testing platform.

b. BatchMake's system information library is used to determine the number of cores as well as the physical and virtual memory space of each testing platform.

c. Based on the *n* cores on the testing platform as well as its total physical memory, ctest is used to run a set of relevant tests, e.g., using *n* or fewer threads and image sizes less than a pre-defined limit based on total physical memory.

d. Results from each test are automatically sent to a central BatchMake server. The server then generates a multitude of web-based graphs and links those graphs with the CMake dashboard. These graphs reveal the complexities of the high-dimensional parameter space of algorithm parallelization. Linking with CMake dashboards reveals how changes in performance are tied to changes in the code. New graphs can be interactively specified.

A subset of those tests, platforms, and results are presented next.

4 Results

Space and time limit the breadth and depth of the results that can be presented herein. This paper instead focuses on a popular set of registration modules and two mid-range compute servers. The registration module configurations reported in this paper are the following:

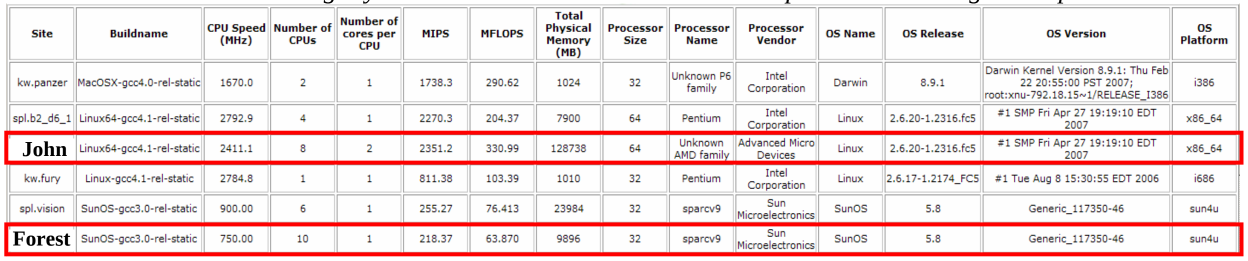
1. Mattes mutual information metric with cubic B-Spline transform and cubic B-Spline interpolation

2. Mattes mutual information metric with linear transform and linear interpolation

3. Mean squared error metric with linear transform and linear interpolation

Table 1 lists the machines on which the tests are computed nightly. The tests that are run on each machine vary based on the number of simultaneous threads supported, i.e., number of CPUs x number of cores per CPU = number of simultaneous threads supported. Tbl. 1 reveals that the code works on Apple Mac OSX, Linux 32bit, Linux 64bit, and SunsOS platforms. Furthermore, experimental builds routinely (albeit not nightly) test the code on Microsoft Windows.

Table 1. *Machines on which nightly tests are conducted to track the ITK optimizations being developed.*



Site	Buildname	CPU Speed (MHz)	Number of CPUs	Number of cores per CPU	MIPS	MFLOPS	Total Physical Memory (MB)	Processor Size	Processor Name	Processor Vendor	OS Name	OS Release	OS Version	OS Platform
kw.panzer	MacOSX-gcc4.0-rel-static	1670.0	2	1	1738.3	290.62	1024	32	Unknown P6 family	Intel Corporation	Darwin	8.9.1	Darwin Kernel Version 8.9.1: Thu Feb 22 20:55:00 PST 2007; root:xnu-792.18.15~1/RELEASE_I386	i386
spl.b2_d6_1	Linux64-gcc4.1-rel-static	2792.9	4	1	2270.3	204.37	7900	64	Pentium	Intel Corporation	Linux	2.6.20-1.2316.fc5	#1 SMP Fri Apr 27 19:19:10 EDT 2007	x86_64
John	Linux64-gcc4.1-rel-static	2411.1	8	2	2351.2	330.99	128738	64	Unknown AMD family	Advanced Micro Devices	Linux	2.6.20-1.2316.fc5	#1 SMP Fri Apr 27 19:19:10 EDT 2007	x86_64
kw.fury	Linux-gcc4.1-rel-static	2784.8	1	1	811.38	103.39	1010	32	Pentium	Intel Corporation	Linux	2.6.17-1.2174_FC5	#1 Tue Aug 8 15:30:55 EDT 2006	i686
spl.vision	SunOS-gcc3.0-rel-static	900.00	6	1	255.27	76.413	23984	32	sparcv9	Sun Microelectronics	SunOS	5.8	Generic_117350-46	sun4u
Forest	SunOS-gcc3.0-rel-static	750.00	10	1	218.37	63.870	9896	32	sparcv9	Sun Microelectronics	SunOS	5.8	Generic_117350-46	sun4u

The two machines focused upon in this paper are highlighted: John and Forest. These machines reside within the Surgical Planning Laboratory of the Brigham and Women's Hospital.

- John is a 64-bit machine with 8 dual-core, 2.4, GHz AMD processors; 128 GB of shared memory; and 64-bit Linux kernel 2.6. The tests were compiled using GCC version 4.1 and CMake's "Release" build options. Tests were performed using 1, 2, 4, 8, and 16 threads on this platform.

- Forest is a 32-bit machine with 10 single-core, 750 MHz, Sparc processors; 9 GB of shared memory; and SunOS 5.8. the tests were compiled using GCC 3.0 and CMake's "Release" build options. Tests were performed using 1, 2, 4, and 8 threads on this platform.

Select results from these two test platforms are summarized next. The graphs shown below are taken directly from the nightly "Batchboards" for this work. Additional Batchboard results can be viewed at

http://www.insight-journal.org/batchmake/
(click on the "BWH-ITK Optimization" public project link)

4.1 Mattes Mutual Information Metric with Cubic B-Spline Transform and Cubic B-Spline Interpolation

One view of Mattes mutual information metric run times (Equ. 1) is given in Fig. 2. For these tests, the metric is paired with a B-Spline transform consisting of a 12x12x12 grid of control points and a cubic B-Spline interpolator. These run times are for 5 calls to the `metric->GetValue()` function using 50,000 (red line) and 100,000 (blue line) samples per metric evaluation.

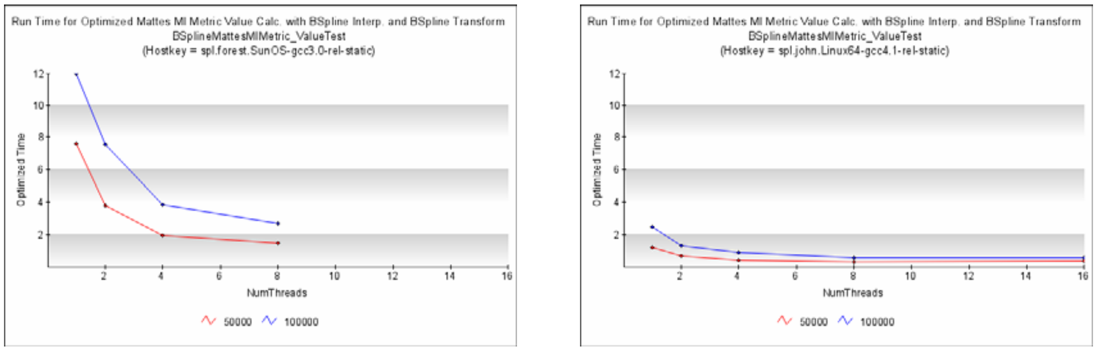


Figure 2. *Mattes mutual information metric evaluation run times for 5 calls to* `metric->GetValue()` *using a B-Spline transform and B-Spline interpolation, for various number of threads. Results for the machine Forest are given on the left and for the machine John on the right.*

The corresponding speedup (Equ. 2) for the optimized metric with 100,000 samples using 8 threads on Forest is C_8 = 12.02 / 3.10 = 3.99. The speedup for the optimized metric using 16 threads on John is C_{16} = 2.22 / 0.65 = 3.42. By Amdahl's law we can expect speedup to eventually decrease as more threads are added.

In Fig.3 the optimization ratios (Equ 3), that compare the optimized method with the standard ITK method, are given. Values greater than one, when one thread is used, indicate that the optimized method is faster than the standard ITK method when run single threaded. On both machines the optimized method is on average approximately 6 times faster than the standard ITK method when 8 threads are used.

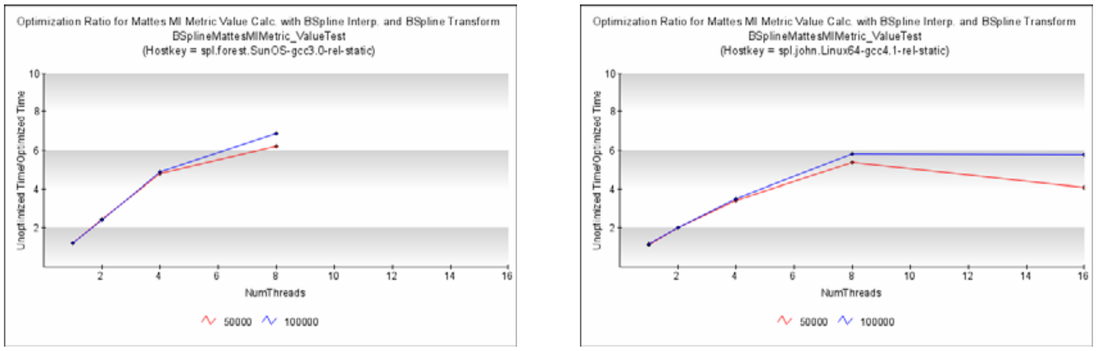


Figure 3*. Optimization ratios (Equ. 3) for Mattes mutual information metric with a B-Spline transform and B-Spline interpolation. Improvement compared to the standard ITK method is better than linear on Forest (left) and only slightly less than linear on John (right).*

A view of Mattes mutual information metric run times and optimization ratios, for five calls to the `metric->GetDerivative()` function, are given in Fig. 4. In these tests, as in the previous set, a 16x16x16 grid of control points is used with the B-Spline transform and linear interpolation is used. These graphs reveal the challenges associated with metric optimization as well as the costs associated with distributing data to and integrating results from multiple threads. Specifically, they show that run times may actually increase as more threads are used.

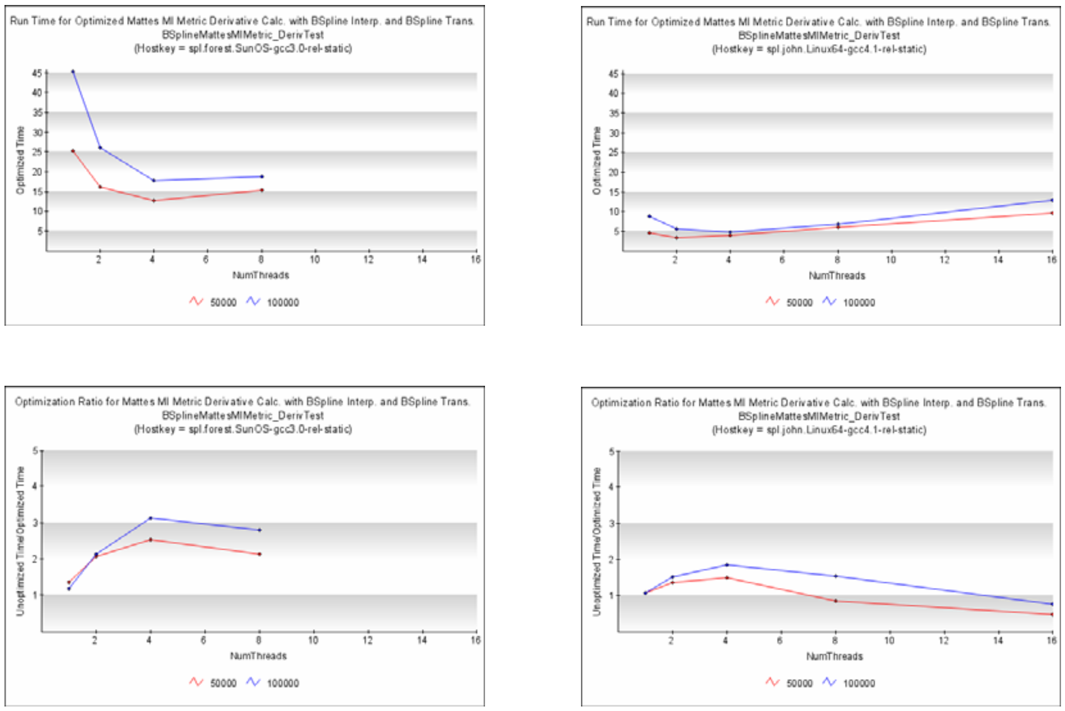


Figure 4. *Run times (top row) and optimization ratios (bottom row) for the* `metric->GetDerivative()` *function of the optimized Mattes mutual information metric using B-Splines for the transform and the interpolator. On both testing platforms (left = Forest, right = John) the best speedup for 100,000 samples was achieved using four threads, e.g., C_4 = 47.0 / 17.8 = 2.6 for Forest. Using additional threads increases run times.*

4.2 Mattes Mutual Information Metric with Linear Transform and Linear Interpolation

The run times in Fig. 5 correspond to tests using the Mattes mutual information metric using a linear (matrix and offset) transform and linear interpolation. The optimization ratios (Equ. 3) were on average approximately 4.0 when 8 threads were used on either platform.

In these tests, unlike the previously reported tests, 10 calls were made to each function, i.e., run times are for conducting twice as much work. While changing the number of calls per test confounds the comparison across methods, that changed was deemed necessary. The motivation is that the runtimes were excessive when using Mattes Mutual Information metric with BSpline transforms and interpolators on certain platforms. Therefore, to reduce testing time, we reduced the number of calls for that setup. However, we also determined that using fewer than 10 calls for the other tests caused those tests to be very susceptible to minor changes in platform workload since those tests using 10 calls already completed in less than one second on most platforms. We regret the confusion, but judged the presented situation to offer an acceptable balance.

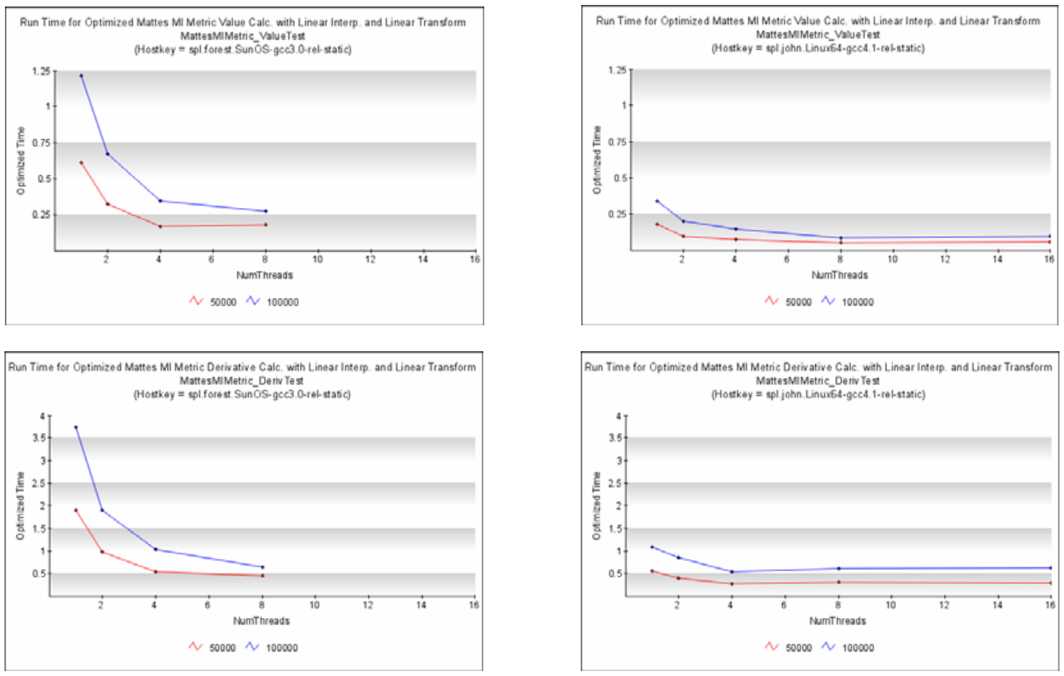


Figure 5. *Run times for ten calls to* `metric->GetValue()` *(top row) and ten calls to* `metric->GetDerivative()` *(bottom row) for Forest (left column) and John (right column). These tests paired Mattes mutual information metric with a linear transform and a linear interpolator.*

4.3 Mean Squared Error Metric with Linear Transform and Linear Interpolation

Fig. 6 presents the run-times for ten calls to `metric->GetValue()` and `metric->GetDerivative()` for the Mean Squared Error metric using a linear transform and linear interpolation.

The standard ITK implementation of the Mean Squared Error metric required the use of every sample in the image when computing the `GetValue()`and the `GetDerivative()` functions. The optimizations performed and the use of a subset of the voxels (sub-sampling) enabled the new metric's optimization ratio (Equ. 3) to typically exceed 300 or more, based on the image size. That is, while ITK's standard implementation prohibited the use of this metric in most situations, the optimized version now makes this metric a viable alternative.

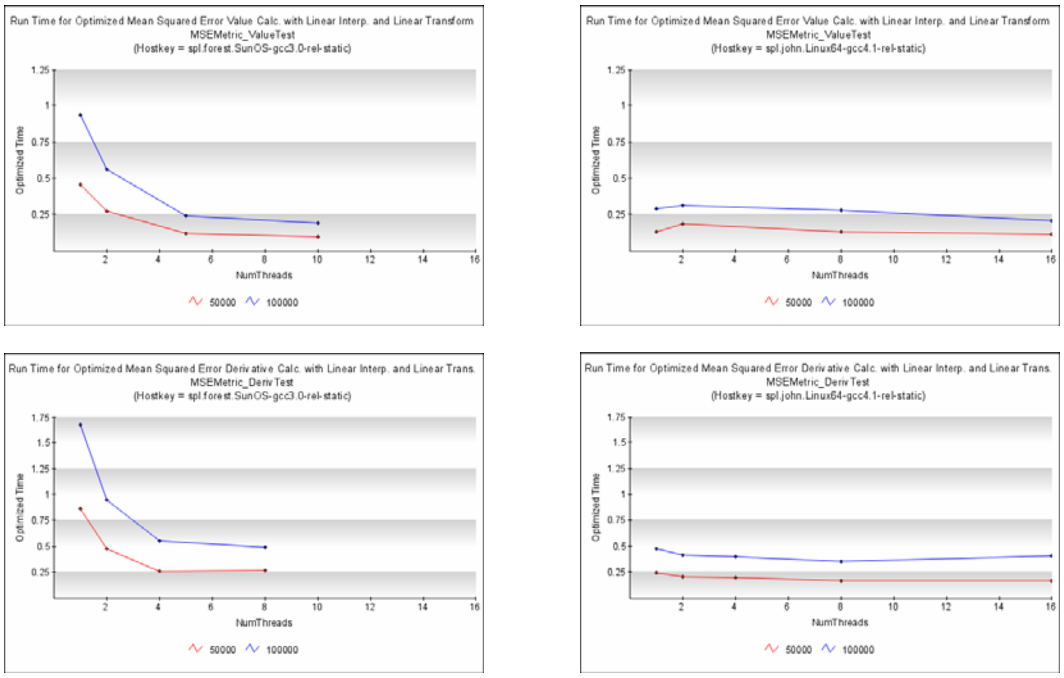


Figure 6*. Mean Squared Error metric run times for ten calls to the* `metric->GetValue()` *(top row) and* `metric->GetDerivative()` *(bottom row) for Forest (left column) and John (right column). Linear interpolation and linear transforms were used with the metric. 50,000 (red line) and 100,000 (blue line) samples were used for the metric computations.*

5 Discussion

The work-in-progress presented in this paper spans a multitude of registration modules in ITK. Only a subset of the results could be presented in this paper. Much additional work remains.

The results indicate that the optimizations employed can decrease the run-time of select existing ITK registration programs by a factor of 6 or more. Run-time improvement ratios will vary significantly if the registration optimizer uses derivatives more heavily than value calls, if the number of threads used isn't optimal, or if the problem size is too small.

The problem sizes used in the tests presented in this paper are small compared to real-world problems and thereby the potential speedup offered by the optimized methods is under estimated. Typically when using 512x512x400 images, more that 100,000 samples should be used to drive the deformable registration optimization process. Problem sizes were kept small in the tests presented in this paper to allow for the nightly testing of multiple parameterizations. Using larger sample sizes will increase the parallelizable component of the method (i.e., the term *B* in Equ 1) – thereby resulting in more speedup for any given number of threads and continued speedup as more threads are used.

Future work will borrow from several of the parallelization techniques presented in [Rohlfing 2003]. In particular, we are extending the metrics and the B-Spline transform to produce sparse Jacobian matrices that exclude B-Spline control points that do not containing meaningful image information (e.g., contain only background voxels). This will reduce the computation time per iteration and should simplify the solution space considered during registration optimization.

6 Acknowledgements

This work is funded in-part by the National Alliance for Medical Image Computing (NAMIC) with the National Institutes of Health through the NIH Roadmap for Medical Research, Grant U54 EB005149. Information on the National Centers for Biomedical Computing can be obtained from http://nihroadmap.nih.gov/bioinformatics.

This work is also funded in-part by a supplemental grant from NAC P41 RR013218-09 for ITK parallelization and grid computing.

Reference

[Amdahl 1967] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," in Proc AFIPS Conf., Vol 30, Reston , VA, 1967, pp. 483-485

[Ibanez 2002] L. Ibanez, L. Ng, J. Gee, and S. Aylward, "Registration patterns: the generic framework for image registration of the Insight toolkit." IEEE International Symposium on Biomedical Imagine, pp. 345-348, 2002

[Matsumoto 1998] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3-30.

[Mattes 2003] D. Mattes, D. R. Haynor, H. Vesselle, T. Lewellen and W. Eubank, "PET-CT Image Registration in the Chest Using Free-form Deformations." IEEE Transactions in Medical Imaging. Vol.22, No.1, January 2003. pp.120-128.

[Pohl 2007] K.M. Pohl, S. Bouix, M. Nakamura, T. Rohlfing, R.W. McCarley, R. Kikinis, W.E.L. Grimson, M.E. Shenton, and W.M. Wells. A hierarchical algorithm for mr brain image parcellation. IEEE Transactions on Medical Imaging, 2007. In Press.

[Rohlfing 2003] T. Rohlfing, D. B. Russakoff, C. R. Maurer Jr., "Expectation Maximization Strategies for Multi-atlas Multi-label Segmentation." IPMI 2003, pp 210-221

[Styner 2000] M. Styner, G. Gerig, C. Brechbuehler, and G. Szekely, "Parametric estimate of intensity inhomogeneities applied to MRI," IEEE Transactions in Medical Imaging.; 19(3), 2000, pp. 153-165

[Wells 1996] W. M. Wells, P. Viola, H. Atsumi, S. Nakajima, R. Kikinis, "Multi-modal volume registration by maximization of mutual information," Medical Image Analysis, 1:53-51, 1996.

[Zhu 1997] C. Zhu, R.H. Byrd, and J. Nocedal, "L-BFGS-B, Fortran routines for large scale bound constrained optimization." ACM Transactions on Mathematical Software, 23(4):550-560, 1995