A new implementation of itk::ImageToImageFilter for efficient parallelization of image processing algorithms using Intel® Threading Building Blocks

Release 0.00

Amir Jaberzadeh[1], Benoit Scherrer[1], Simon K. Warfield, Ph.D. [1]

July 21, 2016

[1]Computational Radiology Lab (CRL), Boston Children's Hospital, Harvard Medical School, 300 Longwood Ave, Boston MA 02115

Abstract

Modern medical imaging makes use of high performance computing to accelerate image acquisition, image reconstruction, image visualization and image analysis. Software libraries that provide implementations of key medical imaging algorithms need to efficiently exploit modern CPU architectures. In particular, workstations with small numbers of cores are being replaced by very high core count architectures, and by many integrated core architectures, which offer acceleration by vectorization and multithreading. The Insight Toolkit (ITK) is the premier open source implementation of medical imaging algorithms, with a generic design for image processing filters that allows for many developers to rapidly incorporate these algorithms in to new applications. While ITK filters benefit from a generic, platform independent multithreading capability, the current implementation is difficult to exploit to achieve very high performance. Specifically, ITK relies on a static decomposition of the image into subsets of equal size which can be highly inefficient. Threads that terminate early due to uneven work throughout the image finish early and do not contribute further to the processing of more complex regions, leading to idle computational resources and longer execution times. Performance is also difficult to coordinate across multiple algorithms, as the ITK filter assumes each filter operates independently but the global implementation has an impact across filters. In this work, we propose a novel, simple to use, high performance multithreading capability for ITK that accelerates the itk::ImageToImageFilter. We utilise a workpile data decomposition strategy, and leave the task of optimal job scheduling on CPU cores to the Intel®library called Threading Building Blocks (TBB). We demonstrate the efficacy of multi-threading with TBB in comparison to the itk::Multithreader class, through three simple example image analysis algorithms. Our implementation provides a new multi-threaded itk::ImageToImageFilter that can be conveniently reused to provide simple and efficient multi-threaded code across applications and algorithm libraries. Our new implementation is distributed as open-source software to the community and is straightforward to adopt.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/1338]

Contents

1 Introduction

As medical imaging technology evolves, researchers and clinicians have access to higher resolution images. Moreover, reseachers are developing novel, more complex algorithms to reconstruct and automatically analyze those images. As a result, the computational burden of medical image analysis has substantially increased in the last decade. While in the 2000's, processor vendors have focused on increasing the processor frequency to accelerate computation, this approach became limited due to power efficiency and reliability issues at high frequencies. The new focus is now on increasing processor performance by developing multi-core architectures, in which multiple processing units (*i.e.*, cores) are placed on a single die together with low-level cache memory and high performance buses for shared-memory inter-core communication.

Image processing algorithms can take advantage of multi-core architectures by running computations on multiple cores at the same time. In a number of image analysis algorithms, the series of operations required to create one output pixel are often independent from other output pixels. These algorithms represent the perfect scenario for efficient parallelism since the same series of operations may be executed at every pixel of the image simultaneously without any data race and without requiring data synchronization. A naive implementation may be to create one thread for each pixel and rely on the operating system (OS) scheduler to synchronize the execution of threads onto cores. This solution, however, leads to catastrophic overhead because of the cost of context switching when switching between threads, which ruins both the cache and the instruction pipelining implemented in modern processors. Moreover, the number of threads is generally limited by the OS.

A classic solution, instead, is to decompose the image into subdomains and concurrently process each subdomains using a limited number of threads, the number of which being generally chosen equal to the number of processor cores to minimize overhead.

An object-oriented programming model that follows this strategy is implemented in ITK. The available high-

level programming interface for multithreaded filters in ITK enables developers to easily implement shared-memory algorithms for image processing in a platform-independent manner. The ITK implementation, however, relies on a *static decomposition* of the image into subdomains of equal size, and on the static instantiation of one thread for each subdomain. First, the ITK implementation creates a new set of threads at each filter execution, which leads to unnecessary overhead. More importantly, the static decomposition strategy used in ITK is highly inefficient when the computational complexity varies between subdomains. It commonly leads to threads that terminate early and hibernate while the values at other voxels remain to be computed, leading to a waste of computational resources and longer computational times.

A more efficient strategy is *dynamic image decomposition*, which aims at dynamically distributing the processing load of image analysis among all available threads. It is typically implemented by considering a set of smaller-scale tasks to complete (e.g., each slice of an image to compute) and a pool of threads that concurrently "consume" and complete the next available task. When the granularity of each task is well calibrated, this strategy leads to minimal overhead and enables *continuous and even* distribution of the workload on the processing resources. Determining the optimal granularity for each task, however, is not trivial - it is problem dependent and, moreover, may evolve same problem.

The Intel® Threading Building Blocks (TBB) library provides a direct, high-level, open-source, platform-independent solution to this problem for shared-memory systems. Specifically, TBB provides a high level abstraction of the concept of thread; it enables the developer to expose parallelism and share opportunities for parallelism by defining *tasks*, without explicitly encoding the mapping of tasks onto actual threads. Instead, the mapping of each task onto threads is performed automatically by TBB's scheduler, taking into account the system workload and automatically adjusting the processing granularity of each thread by achieving a real-time, light-weight profiling of each task. This abstraction enables the developer to focus on the implementation of each task instead of manually implementing the complex machinery to execute the tasks. The TBB scheduler has been shown to cause limited overhead, leading to more efficient parallelization on average and, in addition, ensuring improved scaling on future hardware.

In this work, we propose a new, generic ITK class named `itk::TBBImageToImageFilter` that achieves efficient dynamic image decomposition for parallel image algorithms using Intel® TBB. Importantly, our new class is mostly compatible with the original ITK multithreading capability implemented in `itk::ImageToImageFilter`, making it straightforward to adopt for the community. In aggregate, any existing ITK filter can take advantage of our multithreading advances by changing the parent class to TBBImageToImageFilter and renaming ThreadedGenerateData() method.

The paper is organized as follows. First, we describe in Section 2 the implementation of our new `itk::TBBImageToImageFilter` class. Second, we describe in Section 3 two test applications especially suited to parallelism in the context of diffusion-weighted imaging processing, and evaluate the efficacy of our implementation compared to the built-in ITK multithreading mechanism. Finally, we conclude in Section 6 how the ITK community can straightforwardly take advantage of our novel efficient multithreading capability.

2 ITK Implementation

2.1 The original static decomposition multithreading capability of ITK filters

ITK is based on a generic representation of data objects and *process objects*, and on a generic way to connect them together to build processing pipelines for image processing. Processing objects, which are also referred to as *ITK filters*, typically operate on data objects to produce new data objects. Specifically, the base class for all image algorithms producing a new image as output is `itk::ImageToImageFilter`. More than 185 algorithms inherits from this class in ITK 4.8.

The traditional, object-oriented way to implement a multithreaded image algorithm in ITK is to 1) implement a new class that inherits `itk::ImageToImageFilter`; and 2) overload the following virtual protected methods:

```
void BeforeThreadedGenerateData()
void ThreadedGenerateData (const OutputImageRegionType &outputRegionForThread,
                           ThreadIdType threadId)
void AfterThreadedGenerateData()
```

`ThreadedGenerateData` is the function called concurrently in each thread that contains the code to be executed in parallel, with `outputRegionForThread` describing the portion of the output data the current thread is responsible for generating. `ThreadedGenerateData` also requires a `threadId` dedicated to the output data region in which thread is executing and `threadId` can be used before or after the parallel part referring a specific region.

`BeforeThreadedGenerateData` and `AfterThreadedGenerateData` are single-threaded methods typically used to prepare (e.g., allocate memory, pre-compute some values, etc.) or finalize (e.g., free memory, post-compute some values over the entire image, etc.) the filter before and after the multithreaded processing, respectively.

The actual multithreading capability of ITK filters (and execution of the methods mentioned above) is implemented in `itk::ImageSource::GenerateData()` which is responsible for:

- Allocating the output data,
- Calling `BeforeThreadedGenerateData()`,
- Performing the static image decomposition,
- Creating and spawning the threads, each running the virtual method `ThreadedGenerateData()` on one statically defined region,
- and Calling `AfterThreadedGenerateData()`.

2.2 Our new TBB implementation

We propose a new class named `itk::TBBImageToImageFilter` that achieves efficient dynamic task decomposition using Intel® TBB while being mostly compatible with the original ITK multithreading programming model described above.

Our new class `itk::TBBImageToImageFilter` inherits from `itk::ImageToImageFilter` so that all default behaviors for several important aspects (e.g., allocation of the output image) are

compatible with the original multithreaded ITK filters. Our main contribution is a new implementation of `itk::TBBImageToImageFilter::GenerateData()` that overrides the original `itk::ImageSource::GenerateData()` to achieve dynamic task decomposition using TBB instead of the static decomposition of ITK.

Using TBB requires to conceptualize what is the smallest possible task for the application. In this work we considered it to be the computation of the values for one slice of the output image but it is possible to use a line of data or a single voxel as the smallest possible task.

We used the high level `parallel_for` template function of TBB to achieve *parallel iterations* over all the slices of the image. The syntax for `parallel_for` is as the following:

```
parallel_for(blocked_range<int>(start, end, increment), Functor );
```

where `blocked_range<int>(start, end, increment)` defines the range of values for the 'parallel_for' loop and `Functor` is a C++ class that defines the evaluation operator `()` that will be called concurrently by each thread. Below is an example of functor that can be used with `parallel_for` in TBB:

```
class MyFunctor {
public:
  void operator() ( const blocked_range<int>& r ) const;
};
```

Specifically, the code inside `operator()` is automatically called by TBB's scheduler with the parameter `r` describing a collection of tasks identified by a subrange of the for-loop range.

Dynamic decomposition of the image was achieved by using `parallel_for` that ranges from 0 to the number of slices N_S. We implemented a TBB functor that converts a given set of slices (identified by their indexes in $[0, N_S]$) to a `itk::ImageRegion`. Then this region is passed to a virtual method called `TBBGenerateData()` where parallel portion of filter computations are implemented. It is important to note that in this API implementation `ThreadedGenerateData()` is replaced by `TBBGenerateData()` which eliminates the need for passing `threadId` as an argument. In contrast to ITK threading strategy, TBBImageToImageFilter spawns more regions on image space and TBB scheduler dynamically decides the best chunk size at runtime. So the number of times that `TBBGenerateData()` is called inside `parallel_for` is unknown beforehand. On the other hand, depending on the choice of smallest possible task, TBB can generate a workpile with large number of tasks that makes tracking of all those tasks difficult and not efficient. To highlight this difference and avoid further ambiguities, usage of `ThreadedGenerateData()` is deprecated and all derived classes should override `TBBGenerateData()`. Existing filters inherited from `itk::ImageToImageFilter`, can benefit from this API by renaming `ThreadedGenerateData()` to `TBBGenerateData()`. The most important part of our implementation is included below:

```

namespace itk {

        template< typename TInputImage, typename TOutputImage >
        class TBBFunctor
        {
        public:
                typedef TBBFunctor                  Self;
                typedef TOutputImage     OutputImageType;
                typedef typename OutputImageType::ConstPointer  OutputImageConstPointer;
```

```
        typedef typename TOutputImage::SizeType OutputImageSizeType;
        typedef typename OutputImageType::RegionType OutputImageRegionType;

        itkStaticConstMacro(InputImageDimension, unsigned int,
        TInputImage::ImageDimension);
        itkStaticConstMacro(OutputImageDimension, unsigned int,
        TOutputImage::ImageDimension);

        typedef TBBImageToImageFilter<TInputImage,TOutputImage>
        TbbImageFilterType;

        TBBFunctor(TbbImageFilterType *tbbFilter, const OutputImageSizeType&
        outputSize): m_TbbFilter(tbbFilter), m_OutputSize(outputSize) {}

        void operator() ( const tbb::blocked_range<int>& r ) const
        {
                // Setup the size of the jobs to be done
                typename TOutputImage::SizeType size = m_OutputSize;
                size[OutputImageDimension - 1] = r.end() - r.begin();

                // Setup the starting index
                typename TOutputImage::IndexType index;
                index.Fill(0);
                index[OutputImageDimension - 1] = r.begin();

                // Construct an itk::ImageRegion
                OutputImageRegionType myRegion(index, size);

                // Run the ThreadedGenerateData method!
                m_TbbFilter->TBBGenerateData(myRegion);

        }

private:
        TbbImageFilterType *m_TbbFilter;
        OutputImageSizeType m_OutputSize;
};


// Constructor
template< typename TInputImage, typename TOutputImage >
TBBImageToImageFilter< TInputImage, TOutputImage >::TBBImageToImageFilter()
{
        // By default, do not define the number of threads.
        // Let TBB doing that.
        this->SetNumberOfThreads(0);
        m_NumberOfThreads = false;
}

// Destructor
template< typename TInputImage, typename TOutputImage >
TBBImageToImageFilter< TInputImage, TOutputImage >::~TBBImageToImageFilter()
{ }

template< typename TInputImage, typename TOutputImage >
void TBBImageToImageFilter< TInputImage, TOutputImage >::GenerateData()
{
        // Get the size of the requested region
        typename TOutputImage::ConstPointer output =
        static_cast<TOutputImage *>(this->ProcessObject::GetOutput(0));
        typename TOutputImage::SizeType outputSize =
        output->GetRequestedRegion().GetSize();
        this->m_NumberOfJobs = outputSize[OutputImageDimension - 1];

        // Call a method that can be overriden by a subclass to allocate
        // memory for the filter's outputs
```

```
78                this->AllocateOutputs();

80                // Call a method that can be overridden by a subclass to perform
                  // some calculations prior to splitting the main computations into
82                // separate threads
                  this->BeforeThreadedGenerateData();
84
                  // Set up the number of threads. Only for testing purposes. Should
86                // not be used in practice.
                  tbb::task_scheduler_init init(-2);
88                if (m_NumberOfThreads)
                          init.initialize(this->GetNumberOfThreads());
90                else
                          init.initialize();
92
                  // Do the task decomposition using parallel_for
94                tbb::parallel_for( tbb::blocked_range<int>(0, this->m_NumberOfJobs,1)
                  , TBBFunctor<TInputImage, TOutputImage>(this, outputSize));
96
                  // Call a method that can be overridden by a subclass to perform
98                // some calculations after all the threads have completed
                  this->AfterThreadedGenerateData();
100       }
          // This method replaces ThreadedGenerateData() and should be overridden by a subclass
102       // to perform parallel computations. This method is called iteratively inside parallel_for
          template< typename TInputImage, typename TOutputImage >
104       void TBBImageToImageFilter< TInputImage, TOutputImage >
          ::TBBGenerateData(const OutputImageRegionType&)
106       {
                  std::ostringstream message;
108               message << "itk::ERROR: " << this->GetNameOfClass()
                  << "(" << this << "): " << "Subclass should override this method!!!";
110               ExceptionObject e_(__FILE__, __LINE__, message.str().c_str(),ITK_LOCATION);
                  throw e_;
112       }
          // Get maximum number of jobs
114       template< typename TInputImage, typename TOutputImage >
          unsigned int TBBImageToImageFilter< TInputImage,
116       TOutputImage >::GetNumberOfJobs() const
          {
118               return m_NumberOfJobs;
          }
120
} //namespace itk
```

The resulting `itk::TBBImageToImageFilter` is compatible with the original multithreading capability implemented in ITK. Specifically, any existing filter can take advantage of our new implementation by changing the parent class shown below and renaming the `ThreadedGenerateData()` to `TBBGenerateData()`.

Using the old multithreading ITK capability:

```
namespace itk{
2 template< typename TInputImage, typename TOutputImage >
  class itkMyITKFilter : public ImageToImageFilter< TInputImage, TOutputImage>
4 {
  public:
6   typedef ImageToImageFilter< TInputImage, TOutpuImage > Superclass;
```

Using our new generic multithreading capability:

```
namespace itk{
2 template< typename TInputImage, typename TOutputImage >
  class itkMyITKFilter : public TBBImageToImageFilter< TInputImage, TOutputImage>
4 {
  public:
```

```
6  typedef TBBImageToImageFilter< TInputImage , TOutpuImage > Superclass;
```

3 Application and evaluation

In this Section, we describe three data-parallel algorithms in the context of diffusion-weighted image processing. These example algorithms having different workload balance and granularity levels create an environment to evaluate our new multithreading API in comparison to the original ITK multithreading strategy. For comparison purposes we have included a run-time option to choose between TBB, ITK or both multithreading frameworks using a "-u" flag in all commands.

3.1 Background

Diffusion-Weighted Imaging (DWI) is a MRI technique which is sensitive to orientation anisotropy of water molecules in restricted structures. It allows us to study the geometry of human brain-like fiber tracts or can be applied to compute clinical biomarkers usable for diagnosing diseases. Diffusion tensor estimation is the process of fitting pre-defined diffusion models in the brain DWI images. This relationship is expressed by the Stejskal-Tanner formula [2]

$$S_i = S_0 e^{-b g_i^T D g_i} \tag{1}$$

where S_i is the observed signal for ith direction, S_0 is the reference signal, b is a signal attenuation constant called the "b_value", g_i is gradient vector of ith DW image, and D is matrix of unknown tensors. Having at least six non co-linear gradient directions and at least one reference image, one can use least squares method to find the best model fitted into the measured data. In practice, reference signal is also prone to noise and artefact thus adding S_0 signal into the estimation parameters can lead to a better estimation of true tensors. For this purpose matrices of g and D are reshaped as below to compute 7 unknown parameters. Hence g is a $m \times 7$ with m available diffusion directions and reference images.

$$g = \begin{pmatrix} 1 & -b_1 g_{1x}^2 & -b_1 g_{1y}^2 & -b_1 g_{1z}^2 & -2b_1 g_{1x}^2 g_{1y}^2 & -2b_1 g_{1y}^2 g_{1z}^2 & -2b_1 g_{1x}^2 g_{1z}^2 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ 1 & -b_m g_{mx}^2 & -b_m g_{my}^2 & -b_m g_{mz}^2 & -2b_m g_{mx}^2 g_{my}^2 & -2b_m g_{my}^2 g_{mz}^2 & -2b_m g_{mx}^2 g_{mz}^2 \end{pmatrix} \tag{2}$$

and D is a 1×7 vector,

$$D = \begin{bmatrix} ln(S_0) & D_{xx} & D_{yy} & D_{zz} D_{xy} & D_{yy} & D_{yz} \end{bmatrix} \tag{3}$$

In order to solve Eq 1 for unknown parameters D, some DTI estimation methods like linear least squares (LLS) and weighted linear least squares (WLLS) employs a linearized form of Eq 1 by considering its logarithm. On the other hand nonlinear least squares (NLS) methods minimize sum of squared errors between measured and estimated signals which are directly computed from Eq 1. In practice, due to existence of noise or artefacts, estimators happen to find tensors with negative eigen values which has no physical realization. Constraining estimators to positive semi definite solutions is proposed in many studies while a Cholesky factorization method is used here for this purpose [4]. In total, estimation methods developed in this package are listed as LLS, WLLS, CLLS, CWLLS, NLS, WNLS, CNLS and CWNLS which C stands for constrained and W is the weighted version. Also it is known that, at low signal-to-noise ratio (SNR), pixel intensity values have a Rician distribution because of the interference of noise and separation of real and imaginary components of the acquired images [3] so this package also includes a Rician noise correction filter.

3.2 Tested algorithms

Here we describe three algorithms specific to diffusion-weighted image processing. As mentioned in Section 2, `itk::TBBImageToImageFilter` is compatible with the multithreading programming model of `itk::ImageToImageFilter`, which enables straightforward comparison of performance between the two classes.

Rician noise correction filter: In order to correct for the Rician noise, this filter called RicianNoiseCorrectionFilter follows the correction scheme presented in [1]. In this scheme, voxel intensity is replaced with zero if it is lower than the Rician noise standard deviation or $\sqrt{A^2 - \sigma^2}$ is substituted otherwise, where A and σ^2 are voxel intensity and Rician noise variance respectively. For computation of the Rician noise variance, this filter receives a mask of background region or a value set by user. While using multiple threads, several instances of `TBBGenerateData()` iterate over their allocated image regions and generate corrected signals for all voxels. Since this filter tests a simple condition with predictable computation size thereby creates a balanced workload with a low granularity level. RicianNoiseCorrectionFilter recieves a NHDR file and writes it back to a NHDR file with corrected voxel intensities for the use in next steps of this package.

RicianNoiseCorrectionFilter inputImage.nhdr correctedImage.nhdr -m background_mask.nrrd -v variance -c cores -u TBB/ITK/Both

Linear tensor estimation filter: This filter estimates diffusion tensors of Eq 1 using Linear least squares (LLS) and weighted linear least squares algorithm (WLLS) based on the study by Koay [5]. This filter estimates b0 and diffusion tensors formulated in D matrix for each voxel as described in section 3.1. Tensor estimations are limited to a mask region if a brain mask is provided which creates an unbalanced workload for different threads but granularity level is still low due to simple computations required in linear methods. TensorReconstructionFilter is capable of working with images acquired both in multi-shell or single-shell diffusion MRI data by reading and employing corresponding b-values from the input. Both 4D NHDR and multiple 3D NRRD file formats are accepted in this implementation. Besides main tensor image output, other optional outputs generated in this filter are estimated: b0 image, mean squared residuals image and ADC map computed for each voxel which requires optional command line arguments provided as below.

TensorReconstructionFilter inputImage.nhdr -f mask.nrrd tensorImage.nrrd ADCImage.nrrd -r meanResidualImage.nrrd -b b0Image.nrrd -m method -c cores -u TBB/ITK/Both

Constrained non-linear tensor estimation filter: In order to estimate physically viable and more accurate tensors constrained non-linear least squares solutions to Eq 1 are implemented in this filter to minimize cost functions expressing CNLS, CWNLS, CLLS and CWLLS problems as described in section 3.1. Unknown parameters computed using TensorReconstructionFilter are then passed to this filter to initialize a minimization step which aims to converge to a better set of tensor parameters. In each voxel the algorithm computes new parameters if the initial b0 value is higher than a user defined threshold and whether that voxel is located within the brain mask if provided. This example presents an algorithm with higher granularity relative to number of iterations in the optimization process for each voxel. This makes it harder to predict the required computations leading to a more unbalanced workload. This filter optionally receives pre-computed tens and b0 images provided by user and if not available it calls TensorReconstructionFilter for the initialization. In the end, it generates mean squared residuals image in addition to output tensor image.

PSDTensorEstimationFilter inputImage.nhdr -p initialTensor.nrrd -b b0Image.nrrd -f mask.nrrd -r nu-

mOfIterarionsImage tensorImage.nrrd -o meanResidualImage.nrrd -m method -c cores -t thresh -u TBB/ITK/Both

4 Results

4.1 Acceleration over ITK

In this section acceleration performance is compared between ITK and TBB implementations of test filters for different number of cores. For a fair comparison, each filter is executed 30 times on different machines and the minimum time is used for comparisons as there is always some background tasks which increase computation time. Acceleration is defined as ratio of execution time for single threaded divided by its multi-threaded version. Time measurements are based on absolute wall clock time computed for parallel portions of the code using tbb/tick_count.h
Acceleration scalability graphs is shown in Figs 1 - 3 for three test filters respectively. The results show the amount of acceleration by exploiting more computational resources.

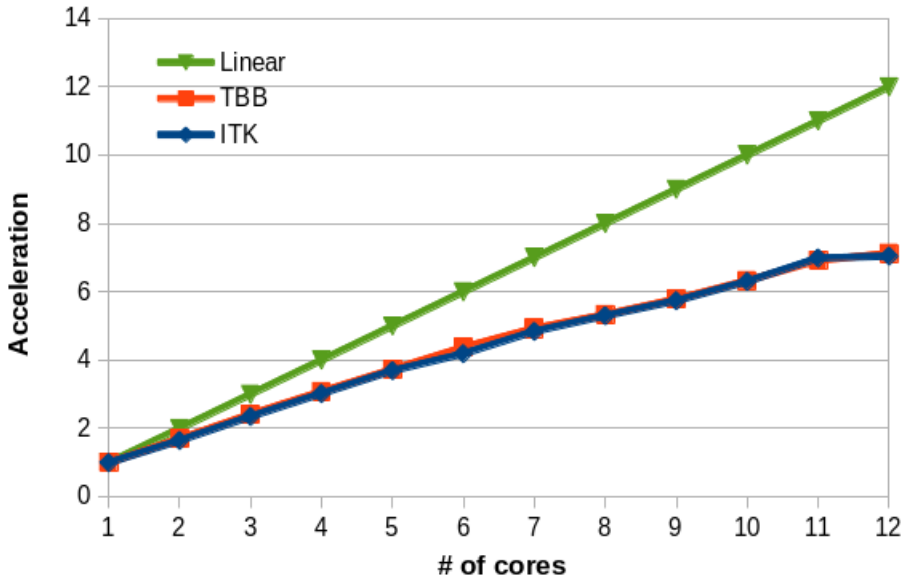

Figure 1: Scalability graph of acceleration in Rician noise correction filter comparing ITK vs TBB performance. Linear acceleration is added as a reference

In Fig 1 Rician noise correction filter shows ITK and TBB have equal performance while dealing with balanced workload and low level of granularity. Linear tensor estimation filter is computed in presence of the brain mask and as demonstrated in Fig 2, TBB outperforms ITK by smart utilization of resources and dynamic scheduling of tasks in presence of workload imbalance. This pattern is more pronounced in non-linear tensor estimation, Fig 3, in which the higher burden of computations leads to a larger performance difference between ITK and TBB. In this case a growing performance gap is visible reaching to 66% improvement for 12 cores. In the last two filters, ITK showed several acceleration jumps in presence of work imbalance. This is caused by the way ITK splits image space and some newly divided regions working on the background image, with few computations, do not contribute into a higher acceleration. Modifying number of threads changes image partitions and the jumps occurs when image space is decomposed in a

more balanced state.

4.2 Filter outputs

Fig 4 illustrates available outputs for TensorReconstructionFilter including tensor, b0 and ADC image. Both tensor estimation filters generate mean squared residual images which demonstrates fitting error of DTI model to the input data. In Fig 5 a comparison between mean squared residuals computed with LLS and CNLS methods are presented. These residuals are normalized respective to maximum value available in both images. As shown in Fig 5, CNLS method can provide a model with a better fit to the acquired images due to lower mean residual values in general. This improvement is achieved in expense of computational time, as linear estimator is computed in 0.83 seconds on a Xeon cpu with 12 cores, non-linear estimator requires 8.15 second on a same machine.

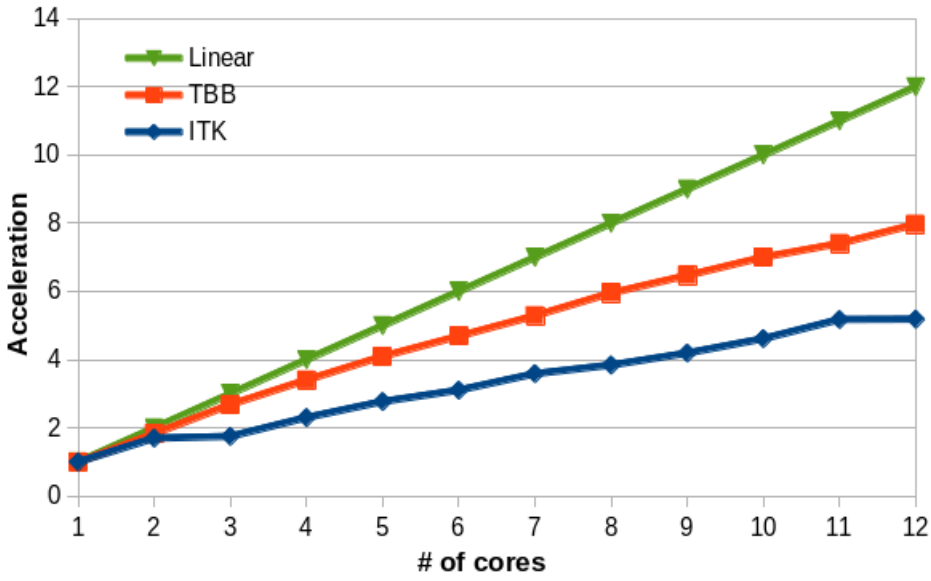

Figure 2: Scalability graph of acceleration in tensor reconstruction filter comparing ITK vs TBB performance. Linear acceleration is added as a reference

5 Conclusion

We developed a new, generic multithreading capability in ITK that achieves efficient dynamic task decomposition using Intel® TBB. We demonstrated improved performance compared to the multithreading programming model implemented in ITK. Our new abstract class for multithreaded ITK filters is compatible with the original multithreading model of ITK, making it straightforward to adopt.

6 Practical notes

Alongside this paper, a sample dataset and a package is provided containing the source code and build instructions that were used in this paper for an easy reproduction of the results. Outputs of the test filters are in NHDR or NRRD extensions and can be visualized by any software displaying diffusion tensors, an example application is MisterI [6]. By executing each filter computational time for the parallel part of the code will be printed out in the console for the chosen multi-threading framework. An automatic bash script parsing method is included in the package which generates speedup values for TBB and ITK threading programming models.

References

[1] Anders H Andersen. On the rician distribution of noisy mri data. *Magnetic resonance in medicine*, 36(2):331–332, 1996. 3.2

[2] Peter J Basser and Derek K Jones. Diffusion-tensor mri: theory, experimental design and data analysis–a technical review. *NMR in Biomedicine*, 15(7-8):456–467, 2002. 3.1

[3] Hákon Gudbjartsson and Samuel Patz. The rician distribution of noisy mri data. *Magnetic resonance in medicine*, 34(6):910–914, 1995. 3.1

[4] Cheng Guan Koay, John D Carew, Andrew L Alexander, Peter J Basser, and M Elizabeth Meyerand. Investigation of anomalous estimates of tensor-derived quantities in diffusion tensor imaging. *Magnetic Resonance in Medicine*, 55(4):930–936, 2006. 3.1

[5] Cheng Guan Koay, Lin-Ching Chang, John D Carew, Carlo Pierpaoli, and Peter J Basser. A unifying theoretical and algorithmic framework for least squares methods of estimation in diffusion tensor imaging. *Journal of Magnetic Resonance*, 182(1):115–125, 2006. 3.2

[6] Benoit Scherrer. http://www.benoitscherrer.com/misteri. 6

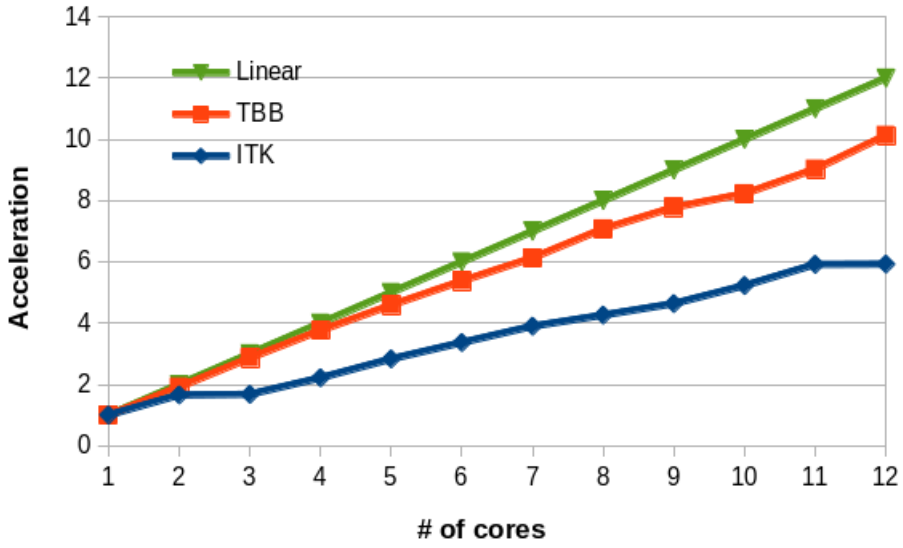

Figure 3: Scalability graph of acceleration in PSD tensor estimation filter comparing ITK vs TBB performance. Linear acceleration is added as a reference

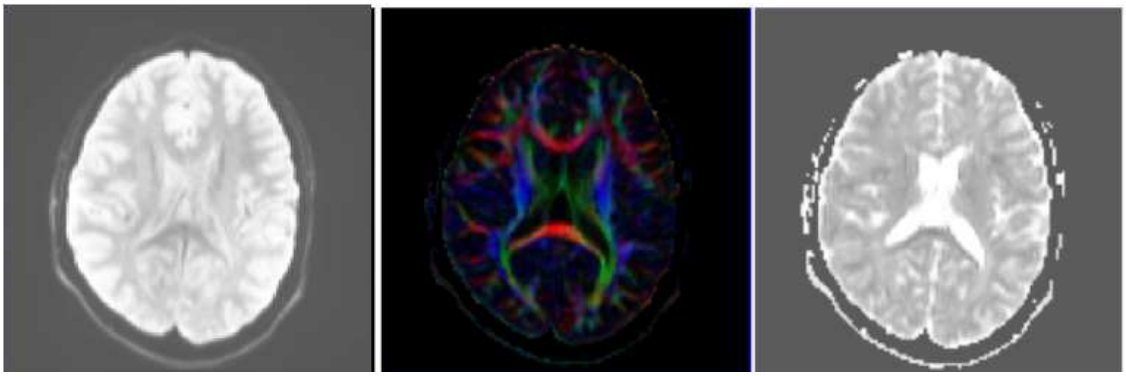

Figure 4: Outputs of TensorReconstructionFilter, b0Image, tensorImage and ADCImage.

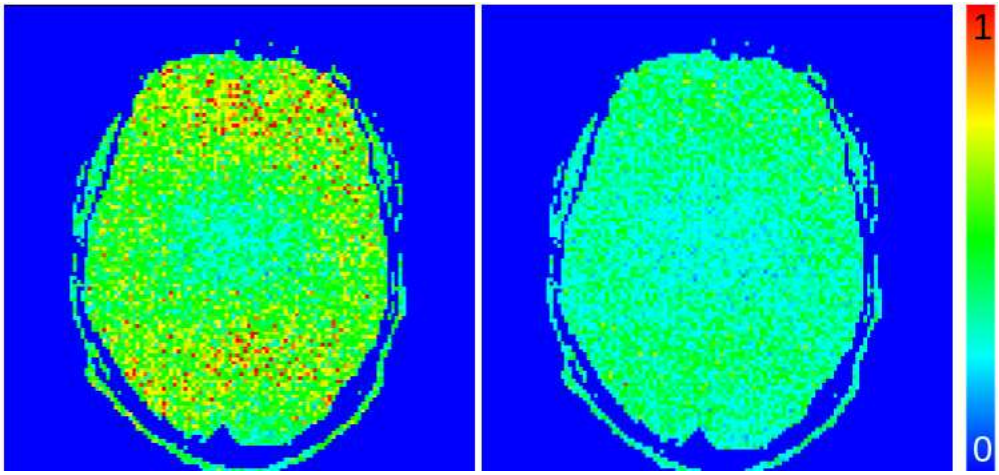

Figure 5: Comparison of mean squared residuals for LLS vs CNLS methods while residuals are normalized to the largest value in two images.