Efficient multithreading for manycore processor: Multidimensional domain decomposition using Intel® TBB

Release 0.00

Etienne St-Onge[1], Benoit Scherrer[1], Simon K. Warfield [1]

July 18, 2017

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

Abstract

The Insight Toolkit (ITK) utilizes a generic design for image processing filters that allows many developers to rapidly implement new algorithms. While ITK filters benefit from a platform-independent and versatile multithreading capability, the current implementation does not easily achieve high performance. First, ITK relies on a static decomposition of the image into subsets of equal size which is highly inefficient when the computational complexity varies between subsets (unbalanced workloads). Second, the current domain decomposition is limited to subdivide the input domain along a single dimension (typically the slice dimension in a 3-D volume), which causes a multithreading under-utilization when the number of threads is larger than the size of this dimension when using massively parallel compute systems. We previously presented a new itk::TBBImageToImageFilter [2] class that replaced the static task decomposition by a dynamic task decomposition for improved workload balancing, in which the job scheduling task was optimized using the Intel® Threading Building Blocks (TBB) library [4]. In this work, we propose a new multidimensional dynamic image decomposition approach that allows decomposition over an arbitrary number of dimensions. This new generic multithreading capability, combined with the TBB dynamic task scheduler, substantially improves multithreading performance when using massively parallel processors.

Contents

1 Introduction

Many image processing algorithms show a high degree of data locality and can efficiently utilize multi-core architectures to run computations simultaneously and reduce the computational duration time. While using graphics processing units (GPUs) has been popular to accelerate various image processing algorithms in the last decade, it requires re-writing algorithms using specialized programming languages (*e.g.*, CUDA) which is very often not possible in terms of resources and competences. In contrast, the emergence of massively parallel x86 architectures such as the Intel® Xeon Phi™ has enabled unprecedented opportunities to parallelize conventional code (C/C++) and substantially reduce the computational time.

The current multithreading capability of ITK relies on a static decomposition of the image into subsets of equal size which is highly inefficient when the processing time is not equal for all subsets. The uneven computation among threads (unbalanced workloads) can be caused by local conditional computation, masked regions or other background process, for example. In addition, the current partitioning system (*itk::ImageRegionSplitter*) only decomposes the image domain along a single dimension which leads to a multithreading under-utilization when the number of threads is larger than the reduced dimension. This limitation is critical when using massively parallel processor such as Intel® Xeon Phi™ Knight Landing (64 cores / 256 hardware threads).

We previously proposed the TBBImageToImageFilter [2] abstract class with dynamic task decomposition and scheduling, using the Intel® Threading Building Blocks (TBB) library, to improve the performance with unbalanced workloads (when processing time is not evenly distributed across regions). In this work, we propose a novel multidimensional image decomposition strategy that allows decomposition over an arbitrary number of dimensions (*e.g.*, slice-by-slice, line-by-line, voxel-by-voxel for a 3-D volume). It effectively generates a larger number of tasks and significantly enhances the multithreading performance when using massively parallel processors. We first describe the implementation of the current ITK multithreading partitioner and our proposed domain decomposition. Second, we detail the context in which we evaluated our new class. In the results section, we compare this new multithreading approach with three different algorithms on three platforms. Finally, we explain how the ITK multithreading can take advantage of using a multidimensional domain decomposition with a thread and work pool system.

2 Implementation

In the section, we first present the current ITK multithreading implementation and limitations. Secondly, we describe our proposed decomposition based on a multidimensional decomposition. Finally, we discuss about the advantages of our new algorithm for manycore processors.

2.1 Original decomposition

The current ITK multithreading image filter is based on a static partitioning, subdivided equally to each thread, along a single dimension. This algorithm separates the domain of the chosen dimension in equal sized partition to feed each thread. By default, the last dimension is used for faster memory access (contiguous memory readout). This decomposition is optimal when the workload is equal for each thread, because it requires minimal thread management and interaction. Unfortunately, in most cases, the workload is not equal (for example due to a variable complexity at each voxel) and some threads finish earlier than others and stay idle. In addition, when the number of threads is larger than the size of the partitioned dimension, the multithreading is limited to this dimension size.

2.2 Proposed decomposition

In this paper, we propose a method to avoid the current limitations of the current ITK multithreading implementation. Our first goal is to allow the domain decomposition over an arbitrary number of dimension. This multidimensional domain decomposition is able to create a high or a low number of tasks depending on the number of cores and to balance the workload on the available number of cores.

For example, the proposed multidimensional decomposition can partition the domain of a 2D image by lines or pixels. A 3D volume can be decomposed by slices, lines or voxels. With this process of reducing the dimensionality, we are able to generalize this splitting process to any number of dimensions. This is particularly helpful when the size of the last dimension is small compared to the number of thread available. Like previously said, generating too many task might cause an overhead, but having too few cause a bottleneck.

We introduce a class variable *m_TBBNbReduceDimension* used as a parameter that describes the subdivision's granularity. While the number of dimension reduction may be set manually (using the *SetNbReduceDimension()* function), we also implemented an heuristic to automatically determine the default number of dimensions that should be decomposed (*GenerateNumberOfJobs()*). Based on our result, we determined and defined its default value in Section Results. This algorithm is based on the shape of the image, the size of each dimension and the number of threads. With this feature, our multithreading filter is transparent to the user and doesn't require to manually set parameters. The Job per thread ratio has been determined with our scalability benchmark results, the current ratio value is 15 and can easily be modified.

The filter computes the number of jobs (*m_TBBNumberOfJobs*) based on this number and the image dimensions sizes:

```
template< typename TInputImage , typename TOutputImage >
void TBBImageToImageFilter< TInputImage , TOutputImage >::GenerateNumberOfJobs()
{
    // 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();
    const int imgDim = OutputImageDimension;
    const int nbReduceDim = GetNbReduceDimensions();

    // Generate the number of jobs
    if (nbReduceDim < 0)
    {
        // Heuristic to automatically determines m_TBBNbReduceDimensions
        m_TBBNbReduceDimensions = 0;
        m_TBBNumberOfJobs = 1;
        int currentDim = imgDim - 1;

        // Minimum Number of Jobs, based on the Number of thread
        unsigned int minNbJobs = JOB_PER_THREAD_RATIO * GetNumberOfThreads();
        while( currentDim >= 0 && m_TBBNumberOfJobs < minNbJobs )
        {
            ++m_TBBNbReduceDimensions;
            m_TBBNumberOfJobs *= outputSize[currentDim];
            --currentDim;
        }
    }
    else
    {   // If manually chosen m_TBBNbReduceDimensions
        assert(nbReduceDim <= imgDim);

        m_TBBNumberOfJobs = 1;
        for (int i = imgDim - nbReduceDim; i < imgDim; ++i)
        {
            m_TBBNumberOfJobs *= outputSize[i];
        }
    }
}
```

2.3 Dynamic Pooling

2.3.1 TBB pooling system

We used the Threading Building Blocks (TBB) pooling system to handle the Threads and Jobs pool [4]. An efficient job queue or pooling system improves multithreading performance when managing a large amount of threads and jobs, more importantly when workloads is not equally distributed.

The dynamic TBB dispatch is not straightforward to use with a multidimensional approach, because it could aggregate multiple regions and exceed the current dimension size. Therefore, we didn't used the TBB dynamic domain decomposition and generated each task with a static multidimensional decomposition.

We used the TBB scheduler and pooling system with the *tbb::parallel_for* and *tbb::blocked_range*. Using the *tbb::simple_partitioner()*, we disabled the dynamic decomposition by forcing the granularity to be equal to one:

```
// Do the task decomposition using parallel_for
tbb::parallel_for(
  tbb::blocked_range<int>(0, this->GetNumberOfJobs()),
  TBBFunctor<TInputImage, TOutputImage>(this, outputSize),
  tbb::simple_partitioner());
```

```
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
  {
    typename TOutputImage::SizeType size = m_OutputSize;
    typename TOutputImage::IndexType index;
    index.Fill(0);

    // Compute the current index from NbReduceDimensions
    if (m_TBBFilter->GetNbReduceDimensions() > 0)
    {
      unsigned int i = OutputImageDimension - (unsigned int)m_TBBFilter->GetNbReduceDimensions();

      index[i] = r.begin();
      size[i] = 1;
      while (i < OutputImageDimension - 1)
      {
        index[i+1] = index[i] / m_OutputSize[i];
        index[i] = index[i] % m_OutputSize[i];
        size[i+1] = 1;
        i++;
      }
    }

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

    // Run the TBBGenerateData method (equivalent of ThreadedGenerateData)
    m_TBBFilter->TBBGenerateData(myRegion);
  }

private:
    TbbImageFilterType *m_TBBFilter;
    OutputImageSizeType m_OutputSize;
};
```

2.3.2 *ThreadID* incompatibility

The original implementation of the ITK multi-threaded filter provided a *ThreadID* parameter (in *ThreadedGenerateData()*) to identify which thread was running. Providing this parameter was possible because of the static decomposition and static distribution of threads used by ITK, leading to a one-to-one thread-to-partition correspondence. Some image filters used this property to combine the results of parallel calculations achieved in the various threads into a single final result (process also known as reduction) in *AfterThreadedGenerateData()*.

When using dynamic pooling, it must be understood that a same thread may be used to do computation on various image sub-domains. Consequently, the *ThreadID* cannot represent the computation identifier anymore. We therefore decided not to expose the *ThreadID* variable in our new TBBImageToImageFilter. This was achieved by marking the original *ThreadedGenerateData(const OutputImageRegionType&, ThreadIdType)* as final and by defining a new *TBBGenerateData(const OutputImageRegionType&)* virtual function:

```
virtual void ThreadedGenerateData(const OutputImageRegionType&, ThreadIdType ) ITK_FINAL;
virtual void TBBGenerateData(const OutputImageRegionType& );
```

3 Application and evaluation

We applied and tested our new filter in the analysis of diffusion-weighted magnetic resonance images, which typically requires complex computations and especially benefits from parallel implementations.

3.1 Background

Magnetic resonance imaging (MRI) is a medical imaging technique used to reconstruct a 3D volume of the anatomy. The contrast and intensity at each voxel (pixel in three dimensions), is based on the tissue properties and acquisition parameters. Diffusion-Weighted Imaging (DWI), or diffusion MRI (dMRI), is a MRI technique which is sensitive to the water diffusion [1]. For each voxel, we measure the diffusion intensity and decay along multiple directions. From the diffusion profile and anisotropy, we can infer a physical model and reconstruct the underlying structure. The Diffusion Tensor Imaging (DTI) is a 3D Gaussian diffusion model used to measure local characteristic from the dMRI acquisition, such as the Fractional Anisotropy (FA) and Mean Diffusivity (MD) [3]. The DTI map can also be used to do tractography, a computational reconstruction of the white matter architecture. Even though, this Gaussian approximation is not able to reconstruct the full complexity of the underlying structure, the tensor model is still widely used for it ease to use and fast MRI acquisition.

For the comparison, we used a standard dMRI acquisition. The volume resolution is $128x128x70$ where each voxel represent an isotropic $2cm^3$ with 85 diffusion directions. For more details about the acquisition, refer to to CUSP90 described in the DCI model from Scherrer et al. [6].

3.2 Tested algorithms

In this work, we used the same diffusion MRI filters than Jaberzadeh et al. [2] to benchmark and compare results. The goal of using several algorithms is to see in which environment the presented method is preferable, and useful to outperform the standard decomposition and multithreading. This section only present the performance characteristics and multithreading potential of each test.

- The *Rician Noise Correction* is a simple dMRI filter with balanced workloads and fast computation.
- The *Tensor Estimation* is linear least-squares fitting. This filter compute the estimation at each voxel inside the region of interest. This test will be used as a non-computationally intensive with unbalanced workloads, because of the mask.
- The constrained *Non-Linear Tensor Estimation* reconstruct the diffusion tensor using a optimization approach. This method also have an unbalanced workloads, but with a more intensive computation.

3.3 Tested processors

For the results (benchmarks and scalability), three different processors were used: a high end computer and two manycores processors. The high end processor we used is the Intel® Xeon® E5-2697 v2 with 24 cores at 2.70GHz. We also benchmark two Intel® Xeon Phi™, from the Many Integrated Core (MIC) family: the Knights Corner (KNC #3120), a 1.10GHz co-processor with 57 cores (228 hardware threads), and the Knights Landing (KNL #7210), a 1.30GHz host processor with 64 cores (256 hardware threads).

4 Results

4.1 Domain decomposition

The image domain decomposition with the original ITK filter (itk::ImageRegionSplitterSlowDimension), over our 128x128x70 test data, generate a maximum of 70 partitions. This could be extended to 128 by using another dimension to partition the image, but the memory access might be slower because of the non-contiguous access.

Our multidimensional approach with a single dimension reduction will create 70 jobs, the same as the original ITK filter. It can also decompose the two last dimensions into 8960 jobs (128*70) or reduce it by three and generate a total of 1146880 jobs (128*128*70). We compared our previous dynamic TBB decomposition [2] with our new multidimensional static approach, where both of them use the integrated TBB tasks scheduler.

4.2 Acceleration on the Intel® Xeon® architecture (24 cores)

Figure 1 presents the multithreading scaling performance on the Intel® Xeon® E5-2697 v2 (24 cores at 2.70GHz) for each method. It shows results for the current ITK version (itk), our previous dynamic approach (tbb) and our new multidimensional decomposition (tbb_nd). We included the three possible multidimensional partitioning of the volume: in slices (tbb_nd 1), in lines (tbb_nd 2) and in voxels (tbb_nd 3). We compared these methods, using previously presented test programs, with balanced or unbalanced workloads and intensive computation or not.

Rician Noise Correction filter (balanced workloads, simple computation, Figure 1-a): most multithreading strategies perform the same. The voxel-wise decomposition (tbb_nd 3) performs worst due to the large amount of jobs created for this non-intensive computation.

Linear Tensor Reconstruction filter (unbalanced workloads, simple computation, Figure 1-b): all approaches with the last dimension decomposition (itk, tbb, tbb_nd 1) have the same acceleration shape. Because workloads are unbalanced, the lines decomposition (tbb_nd 2) outperforms the other strategies. In contrast, the voxel-wise decomposition (tbb_nd 3) leads to a large number of jobs that causes substantial overhead and, ultimately, a slower computation.

Non-Linear Tensor Estimation filter (unbalanced workloads, complex computation, Figure 1-c): the proposed method (tbb_nd) and previous dynamic approach (tbb) have better scaling than the static decomposition (itk). The lines decomposition (tbb_nd 2) is 1.5 time faster at 24 threads.

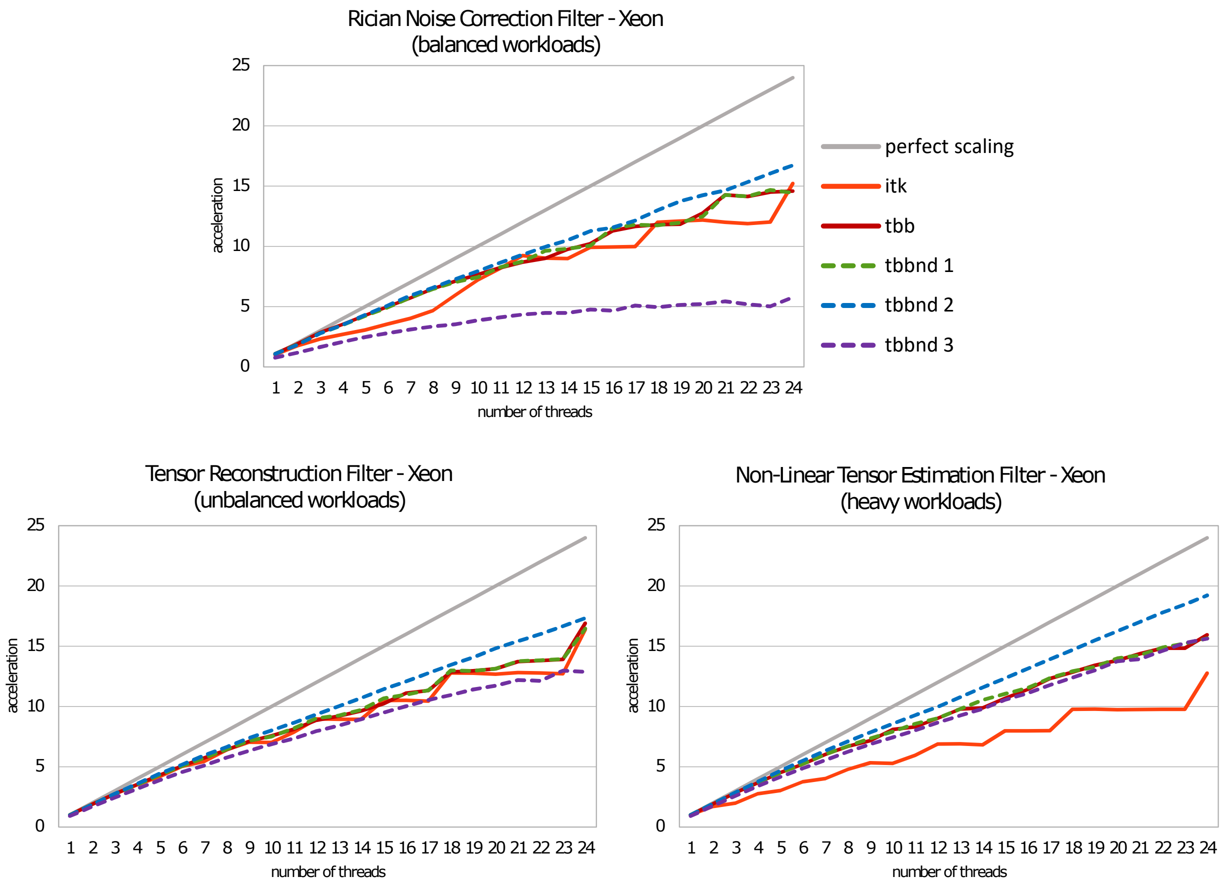

Figure 1: Multithreading scaling for the three test algorithm with the Xeon processor (24 cores). Comparing the current ITK multithreading (itk), our previous tbb dynamic scheduler [2] (tbb) and our proposed multidimensional decomposition (tbb_nd).

4.3 Acceleration on the Intel® Xeon Phi™ massively parallel architecture (57-64 cores)

Figure 2 shows that both the current ITK and our previous implementation of TBBImageToImageFilter stop scaling above 70 threads. This is because the provided input image has 70 slices, and illustrates how the unidimensional partitioning limits the multithreading scaling capability when using massively parallel architectures. In contrast, our new decomposition strategy is not limited when using more than one reduced dimension (lines or voxels).

For KNC and KNL, using tbb_nd 2 (line decomposition) is ultimately 2x faster than the current ITK version when using all available threads. We observe in Figure 2-b that the KNL platform (but not KNC) results in thread throttling when using both a large number of threads and a large number of jobs (tbb_nd 3). This phenomenon happens when the task scheduler overhead stops being negligible, compared to the processing time of each task. We hypothesize that we don't observe this phenomenon in Figure 2-a because of the slower processing speed of KNC.

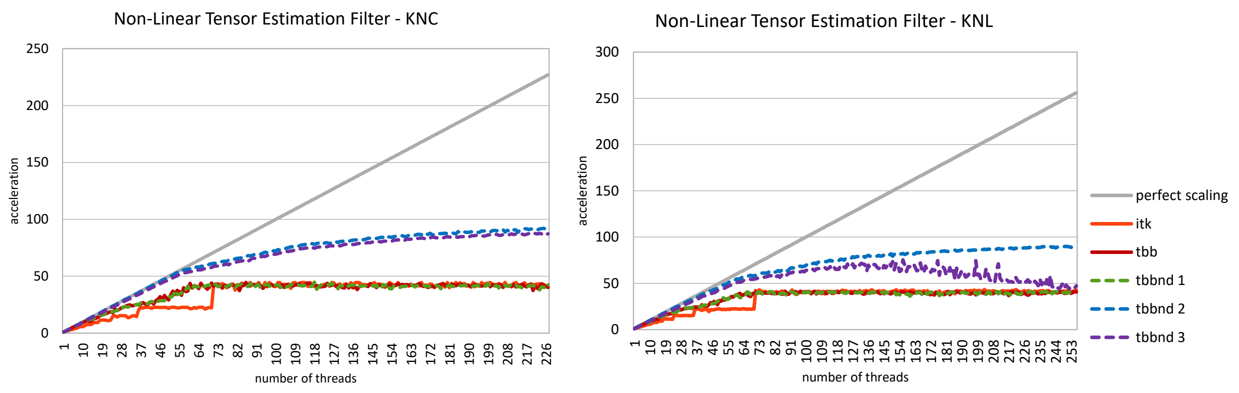

Figure 2: Scaling of the *Non-Linear Tensor Estimation* (heavy workloads) on manycore processors (KNC and KNL) with the current ITK multithreading (itk), our previous tbb dynamic scheduler [2] (tbb) and our proposed multidimensional decomposition (tbb_nd).

5 Discussion

From the result in Figure 1-a, we see that the current ITK static decomposition perform well in a balanced workloads situation and when the images are large enough compared to the number of threads. However, in unbalanced workloads application (Figure 1-b,c) or with highly parallel architectures (Figure 2-a,b), using a dynamic pooling system with a larger number of jobs (tbb_nd 2) allows substantial increase in multithreading performance and scaling. This is true only to certain granularity: generating an excessive amount of partition, i.e. using the voxel decomposition (tbb_nd 3), can cause an overhead and decrease the filter overall performance (Figure 1-a, Figure 2-b).

From the multithreading scaling result with unbalanced workloads applications, we determined that the job per thread ratio should be around 15 (Figure 1, Figure 2). In unbalanced workloads test (Figure 1-b,c), when using more than 5 threads, the multidimensional approach with line decomposition (tbb_nd 2) outperform other algorithms. The ratio value should be high enough to utilize lines partition after a certain amount of threads, but shouldn't utilize voxel partition (tbb_nd 3) without a really large number of threads. With our dMRI image last dimension size equal to 70, the heuristic use slice decomposition before 5 threads and then line decomposition up to 598 threads.

6 Conclusion

The ITK multithreading implementation can be improved by including a multidimensional domain decomposition utilizing a thread and work pool system. This new strategy not only improves the performance when there is more thread than the size of the last dimension, but also when the image processing algorithm has unbalanced workloads. The proposed decomposition heuristic for the multidimensional partitioning, based on the image size and number of threads, enables the filter to be transparent for the user.

7 Practical notes

Alongside this paper, a package containing the source code, build instructions and a sample dataset is provided. We also included the CMake Toolchain file for the Knights Corner cross-compilation. Filtered anatomical and diffusion images, can be visualized with Mister I [5] or any software displaying diffusion tensors supporting NHDR or NRRD extensions.

An ITK remote module ITKTBBImageToImageFilter with improved code readability and improved consistency with the ITK coding guidelines is also available at:
`https://github.com/InsightSoftwareConsortium/ITKTBBImageToImageFilter`.

References

[1] Maxime Descoteaux and Cyril Poupon. Diffusion-weighted mri. *Comprehensive Biomedical Physics*, 3(6):81–97, 2012. 3.1

[2] Amir Jaberzadeh, Benoit Scherrer, and Simon Warfield. A new implementation of itk::imagetoimagefilter for efficient parallelization of image processing algorithms using intel threading building blocks. 07 2016. (document), 1, 3.2, 4.1, 1, 2

[3] Denis Le Bihan, Jean-François Mangin, Cyril Poupon, Chris A Clark, Sabina Pappata, Nicolas Molko, and Hughes Chabriat. Diffusion tensor imaging: concepts and applications. *Journal of magnetic resonance imaging*, 13(4):534–546, 2001. 3.1

[4] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007. (document), 2.3.1

[5] Benoit Scherrer. http://www.benoitscherrer.com/misteri. 7

[6] Benoit Scherrer, Armin Schwartzman, Maxime Taquet, Sanjay P Prabhu, Mustafa Sahin, Alireza Akhondi-Asl, and Simon K Warfield. Characterizing the distribution of anisotropic micro-structural environments with diffusion-weighted imaging (diamond). In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 518–526. Springer, 2013. 3.1