An alternative threading model for the Insight Toolkit

Release 1.00

Daniel Blezek[1]

December 21, 2008

[1]blezek.daniel@mayo.edu
Mayo Clinic, Rochester, MN, USA

Abstract

This technical note presents an alternative threading model for the Insight Toolkit. The existing ITK threading model is based on a "scatter / gather" model and divides work evenly amongst all threads. Though suitable for many filters, considerations such as memory allocation per thread are important in some classes of filters. We propose to use the ZThread library to explore a threading model based on an execution pool. The ZThread library is a cross platform, open source thread abstraction library loosely based on Java's threading model.

Contents

1 Introduction and Background

The basic model underlying the `MultiThreader` is "scatter / gather" of threads. A multi-threaded filter implements `ThreadedGenerateData()` and processes the region passed a region to operate on (`OutputImageRegionType`). Inside `ImageSource::GenerateData`, the entire output region is divided into N subregions, where N is the number of allowed threads (`MultiThreader::GetNumberOfThreads()`). Then all N threads are created, and handed their region to process. `ImageSource::GenerateData` then waits until all N threads have completed and returns. The algorithm for `ImageSource::GenerateData` is shown in Algorithm 1.

Algorithm 1 Threading model for ImageSource::GenerateData.

AllocateOutputs() { Allocate the output of this filter }
BeforeThreadedGenerateData() { Allow the sub class to perform any necessary setup}
for all Each thread is given an ID from 0..N-1 **do** {Execute N threads in parallel}
  { Get the threadID'th region of N output sub-regions}
  SplitRegion = SplitRequestedRegion (threadID, N)
  { Have the subclass execute this portion of the output}
  ThreadedGenerateData (SplitRegion, threadID)
end for
AfterThreadedGenerateData() { Allow the subclass to perform any cleanup }

`MultiThreader` instance in `ImageSource` is responsible for starting up N threads, calling `ImageSource::ThreadedGenerateData` in each one, then waiting for all N threads to exit.

This model, though extremely useful for exploiting modern multi-core processors, has several important drawbacks for certain classes of filters. Filters requiring large amounts of intermediate memory suffer in the current implementation. Regardless of what N is, all intermediate results are required to be allocated all at once. Consider a Hessian based filter such as the `HessianToObjectnessMeasureImageFilter` recently submitted to the Insight Journal [1]. If one thread is used, the entire output will be requested, necessitating the generation of the entire Hessian. A Hessian calculation requires, at minimum, 6 times the memory of the input volume. This memory requirement is prohibitively large for otherwise practical images. If many threads are used, the memory overhead is not changed, in addition, the filter fails to achieve full speedup from multiple cores because it is forced to recalculate pixels on the border between regions. Thus, this class of filters is often limited to a single thread. For the purposes of this discussion, I call these filters Large Memory Filters (LMF).

A second important limitation of `MultiThreader` is the lack of flexibility in scheduling threads. The implementation of `ImageSource::SplitRequestedRegion` simply divides the image along the last dimension, i.e., by slices in 3D. Though the user can override the default behavior of `ImageSource::SplitRequestedRegion` to divide the image up different, he cannot, for instance, divide the image into M small regions to be processed by N threads (where M >> N). This concept is know as a work pool (see `http://en.wikipedia.org/wiki/Thread_pool_pattern` for instance). The two limitations are related.

2 Fine Grain Thread Control

The limitations of ITK's threading scheme can be rectified through the use of several different parallel constructs. Consider Algorithm 2. In this algorithm, the processing of a LMF is divided into small parcels based on memory requirements, rather than simply splitting the image into N parts. Allowing filter designers the flexibility to divide jobs by memory leads to fine grain control of parallel processing. The number of threads executing may still be the number of processors, as before, but each processor will have more jobs of smaller size.

Algorithm 2 Threading model for ImageSource::GenerateData

```
BeforeThreadedGenerateData()
queue = new PoolExecutor ( N ) {create a queue with N threads }
J = GetMemoryRequired() / 10 {Split the requested region into 10 MB jobs, J >> N}
for i = 1..J do
  SplitRegion = SplitRequestedRegion ( i, J );
  queue.push ( new Job ( SplitRegion ) )
end for
queue.wait() {Wait until all the jobs have been completed}
AfterThreadedGenerateData() { Allow the subclass to perform any cleanup }
```

Algorithm 2 uses a JobPool (`PoolExecutor` from the ZThread library). JobPools maintain a queue of jobs to process and one or more worker threads. Each worker pops a job off the queue and processes it, repeating until the queue is empty. The JobPool class grants greater flexibility to the filter writer. This algorithm is shown in `itk::BilateralZThreadImageFilter` included with this document. Rather than allowing the default implementation of `GenerateData` located in `itk::ImageSource` to use the standard ITK `itk::MultiThreader`, `itk::BilateralZThreadImageFilter` uses the `PoolExecutor` from the ZThread library. The code is shown here:

```
//----------------------------------------------------------------------------
template< class TInputImage, class TOutputImage >
void
BilateralZThreadImageFilter<TInputImage, TOutputImage>
::GenerateData()
{
  // Call a method that can be overriden by a subclass to allocate
  // memory for the filter's outputs
  this->AllocateOutputs();

  // Call a method that can be overridden by a subclass to perform
  // some calculations prior to splitting the main computations into
  // separate threads
  this->BeforeThreadedGenerateData();

  // Do this with ZThread's PoolExecutor
  ZThread::PoolExecutor executor(this->GetMultiThreader()->GetNumberOfThreads());
  typename TOutputImage::RegionType splitRegion;
  try
```

```
    {
    int NumberOfRegions = 20;
    for ( int i = 0; i < NumberOfRegions; i++ )
      {
      ZThreadStruct* s = new ZThreadStruct();
      s->threadId = i;
      s->Filter = this;
      this->SplitRequestedRegion(s->threadId, NumberOfRegions, splitRegion);
      s->region = splitRegion;
      executor.execute ( s );
      }
    // Wait for all jobs to finish
    executor.wait();
    }
  catch ( ZThread::Synchronization_Exception &e )
    {
    itkGenericExceptionMacro ( << "Error adding runnable to executor: " << e.what() );
    }

  // Call a method that can be overridden by a subclass to perform
  // some calculations after all the threads have completed
  this->AfterThreadedGenerateData();
}
```

In this example, we create `NumberOfRegions` jobs in the `ZThread::PoolExecutor` represented by `ZThreadStruct`, a simple class that calls the filter's `ThreadedGenerateData`. The `PoolExecutor` ensures each `ZThreadStruct` is `delete`d after being processes.

Ideally, this code would migrate out of an individual filter and into a subclass of `MultiThreader`, however, the design of `MultiThreader` discourages sub-classing. None of the methods are virtual, and many useful variables are private. Use of alternative threading schemes in ITK would require redesign of this class.

3 ZThread Library

The ZThread library abstracts many important thread concepts including barriers, mutex locks, semaphores, and thread pools. ZThreads is hosted on SourceForge (http://sourceforge.net/projects/zthread/) and documented using Doxygen (ZThreads is implemented using platform specific primitives. The abstractions present a uniform API and many higher level constructs such as PoolExecutors (http://zthread.sourceforge.net/html/classZThread_1_1PoolExecutor.html).

4 Software Requirements

You need to have the following software installed to compile this code:

- Insight Toolkit 3.0 or greater

- CMake 2.4

The code described in this paper is in the Subversion repository at

```
http://svn.na-mic.org/NAMICSandBox/trunk/ThreadIT
```

and may be anonymously checked out using the command:

```
svn co http://svn.na-mic.org/NAMICSandBox/trunk/ThreadIT ThreadIT
```

The code should build on all reasonable platforms (ZThreads may not support SGI's sproc).

Note that other versions of the Insight Toolkit are also available in the testing framework of the Insight Journal. Please refere to the following page for details

http://www.insightsoftwareconsortium.org/wiki/index.php/IJ-Testing-Environment

5 ZThread License

References

References

[1] Luca Antiga. Generalizing vesselness with respect to dimensionality and shape. *Insight Journal*, 2007. 1