HyperFlow and ITK v4 Integration: Exploring the use of a modern parallel dataflow architecture in ITK

Huy T. Vo, Lauro D. Lins, and Cláudio T. Silva

NYU-Poly

1 Introduction

In this document, we report our activities on the subcontract with Kitware where we carefully study the upcoming ITK v4 pipeline design to help the ITK v4 development team to add support of modern dataflow architectures to ITK. The current dataflow architecture in ITK (up to version 3) has been designed primarily to work on either a single CPU or a small collection of CPUs, such as a small SMP workstation. Recent hardware advancements have greatly increased the level of parallelism available in these architectures, and it is expected that this trend is likely to continue in the future. Also, supporting GPUs is becoming increasingly important, as well as having good support for clusters of such machines. One of our main goals is to make sure that the design of the new ITK's pipeline is extensible with scalable constructs, such as those available in the multi-threaded VTK [2] and HyperFlow [1].

In particular, we have performed the following tasks:

1. Created a prototype application that uses ITK filters from a HyperFlow pipeline. Through this application, we uncovered a major bottleneck in the current design of ITK v4 pipeline that will prevent the toolkit to add support for parallel execution. We are suggesting a modification to the processing pipeline of ITK v4 to address this issue.

2. We implemented the modification proposed in (1) to ITK v4 along with a test program that was constructed purely in ITK v4 to show the impact. Both the prototype application in (1) and the test program here (2) are available as open source applications including a simplified version of HyperFlow.

3. We believe that our detailed report on the suggested modification would help future designs of ITK v4 to evolve without hindering the integration process with modern dataflow architectures such as HyperFlow.

We are describing these tasks with more details in the next sections.

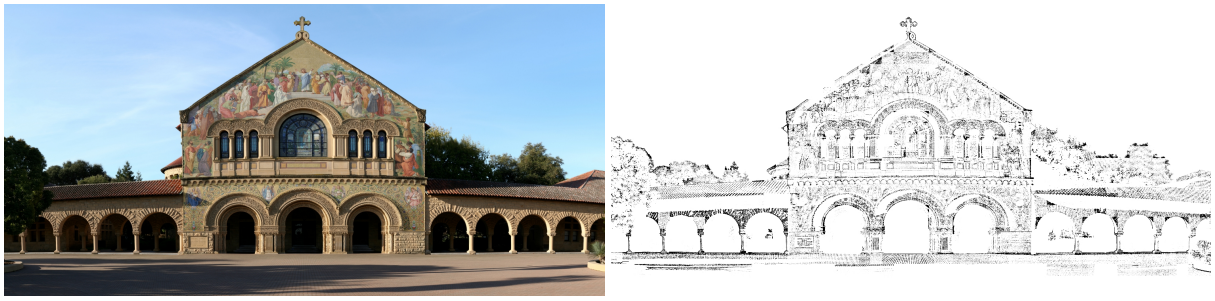


Figure 1: Edge detection of the Stanford church computed using HyperFlow: (left): 512 input images of 3 Mp each (1.5 Gp total); (right): edge detection result computed in 7 seconds using a heterogeneous system composed of 16 CPU threads.

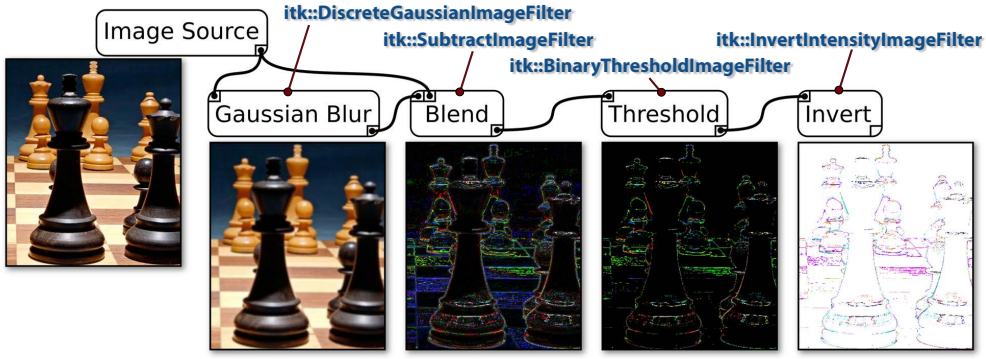


Figure 2: The edge detection pipeline and how ITK v4 modules are used.

2 Using ITK v4 inside HyperFlow

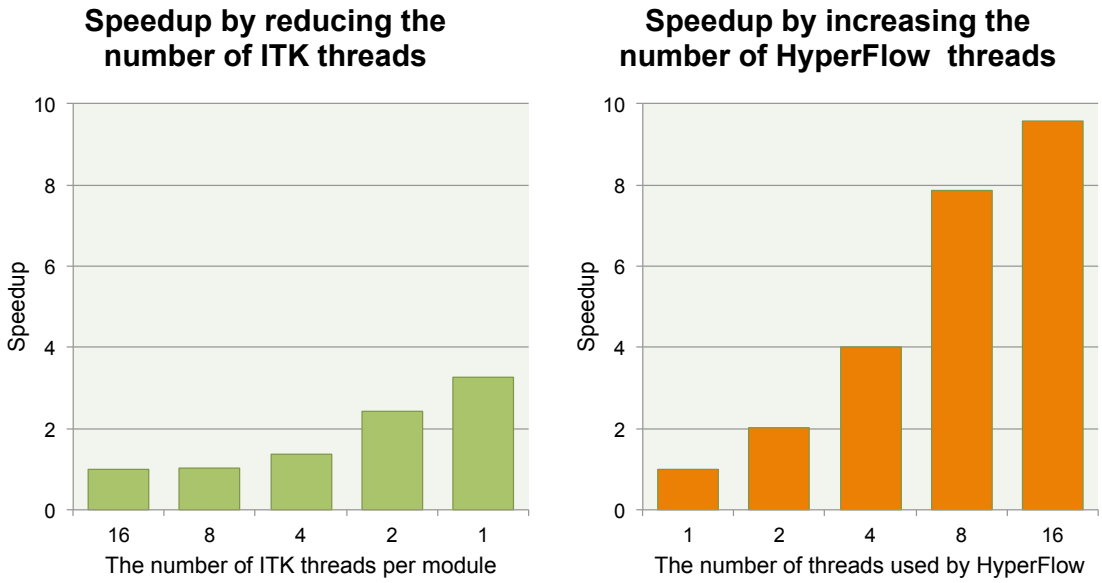
Our first step towards evaluating the pipeline design of the ITK v4 was to test it inside one of the dataflow architectures that are designed for highly parallel machines, in our case, it was HyperFlow. The design of HyperFlow relies on the construction of abstraction layers. At the module level, pipelines are defined as interconnected *Task-Oriented Modules (TOMs)*, and executed as a set of token-based data instances called *flows*. To encapsulate available processing units, a Virtual Processing Element (VPE) forms an abstraction layer over the actual computing resources available in the system. A pipeline in HyperFlow consists of a set of interconnected Task-Oriented Modules (TOMs). Each TOM defines and holds parameters needed for a specific computational task, such as the number of input and output ports. To allow the same pipeline to be executed across a wide range of different computational resources transparently, TOMs do not explicitly store the task implementation. Instead, they store a list of *task implementation* objects, which are dynamically scheduled to perform the actual computation on a given set of inputs. This separation of task specification and implementation is one of the main differences between HyperFlow and similar systems. The requirement for a TOM to be executed at runtime is that it should have a task implementation that matches the system resources (e.g. CPUs or GPUs).

In our prototype, we used the same application that we showed at the first ITK v4 kick-off meeting in June 2010 (see Figure 1) but use ITK v4 filters instead of our own implementation for appropriate CPU tasks. The prototype was an edge detection framework capable of dealing with multiple images concurrently. The pipeline and how ITK v4 modules are used in this pipeline is Figure 2. It starts with a source TOM which decodes a stream of images from disk and send them down to another TOM that performs Gaussian blur (`itk::DiscreteGaussianImageFilter`). The next step in this pipeline consists of a blending operation that returns the difference between the original image and the blurred version (`itk::SubtractImageFilter`). The combined image is streamed to a threshold TOM that computes the image accumulated histogram and discards pixels whose accumulated frequency fall outside a given range (`itk::BinaryThresholdImageFilter`). We set this range to be between 95% and 100% of the total accumulated histogram value. Finally, the pipeline sends images to an inversion TOM for display preparation. The main form of parallelism in this pipeline is the streaming data-parallelism.

Our first result with this pipeline showed that the use of ITK v4 modules had substantially decreased the pipeline performance on an 8-core/16-thread machine. A more careful inspection revealed that since ITK v4 supports filter level multithreading by default, each time a module is executed, it tries to instantiate the maximum number of threads on the machine. This leads to a saturation of computing resources when HyperFlow tries to parallelize the execution of these modules. Figure 3(a) shows how reducing the number of threads per module in ITK v4 may result in better performance. The top performance is reached when HyperFlow was completely in charge of the thread distribution. Figure 3(b) also shows that HyperFlow scales well when it treats each ITK box as a single threaded process and exercises data and pipeline-parallelism onto those. Our main conclusion here is that *allowing each module to use all CPU cores can lead to a substantial performance loss in concurrent execution because of the saturation of computing resources.* Concurrent execution here does not necessarily mean more than one ITK module being executed at the same time: it could be an ITK module plus some other task from the application. This happens very often in scientific applications where data analysis and visualization task are being perform in parallel. Basically, any other processing task happenning during the execution of an ITK module can certainly cause this saturation.

3 Parallelize the ITK v4 pipeline

In the previous experiment, we were able to use ITK as individual modules and use the infrastructure of HyperFlow to coordinate between the modules. Although such integration can really bring additional flexibility and efficiency to ITK, a pure ITK pipeline sometimes might be preferred, for example in legacy codes and applications. Thus, in this experiment, we take the challenge to extend ITK to different types of parallelism in dataflow. As shown in the multithreaded VTK paper [2], it is often more efficient to divide resources among different tasks in a pipeline, also known as task-parallelism. Hence, our next step was to add support of task-parallelism to ITK. However, this is not possible given the current implementation of the ITK v4 execution mechanism. *The shortcoming is coming from the design of `vtkProcessObject` that is only suitable for sequential execution*. ITK uses the demand-driven pipeline, thus, when a module is updated, it will



(a) HyperFlow manages all 16 threads

(b) 1 thread per each ITK v4 module

Figure 3: How computing resource management of ITK and HyperFlow affected performance results: (a) since each module in ITK v4 always tries to use all possible threads on a machine, executing them concurrently may result in performance loss.

ask its upstream modules to update themselves, that in turn will lead to more modules being updated. However, all of these "update" functions are always triggered in a single threaded context, being bundled together in the call stack with its actual computation code. Thus, it is not possible to migrate the call to another thread for concurrent execution. That prevents us from enabling task-parallelism for independent blocks of the pipeline.

Below is the original implementation of `itk::ProcessObject::UpdateOutputData()`:

```
/**
 * Propagate the update call - make sure everything we
 * might rely on is up-to-date
 * Must call PropagateRequestedRegion before UpdateOutputData if multiple
 * inputs since they may lead back to the same data object.
 */
m_Updating = true;
if ( m_Inputs.size() == 1 )
  {
  if ( this->GetPrimaryInput() )
    {
    this->GetPrimaryInput()->UpdateOutputData();
    }
  }
```

```
  else
    {
    for ( DataObjectPointerMap::iterator it=m_Inputs.begin();
          it != m_Inputs.end(); it++ ) {
      if ( it->second )
        {
          it->second->PropagateRequestedRegion();
          it->second->UpdateOutputData();
        }
      }
    }

  // Actual computation
  ...
```

It should be noted that all the upstream `UpdateOutputData()` are being called in a fixed order *and* inside the downstream update function. And there is no way to fork the upstream updates before a module updates itself since the code flow is fixed. To address this issue, we propose to separate the process of updating upstream modules from the actual module computation. This way, customized applications can have the freedom to add their own scheduler (`ProcessScheduler`) to ITK v4 without the hassle of re-writing the `ProcessObject` object. In particular, we propose to have a general "scheduler" class that allows redirection of upstream traversal and module update calls from a single threaded environment to a multi-threaded one. The new logic for `UpdateOutputData` should be:

```
void ProcessObject::UpdateOutputData( DataObject * itkNotUsed(output) )
{
  if ( m_Updating ) { return; }

  // Prepare all the outputs. This may deallocate previous bulk data.
  this->PrepareOutputs();

  // Goes upstream and make sure all modules are up-to-date
  this->m_Scheduler->scheduleTraverseUpstream(this);

  // Actually perform/schedule the computation of this module
  this->m_Scheduler->schedulePerformComputation(this);

  // Mark that we are no longer updating the data in this filter
  m_Updating = false;
}
```

The separation of `scheduleTraversalUpstream()` and `schedulePerformComputation()` is necessary for customized scheduler to control the synchronization point, e.g. joining upstream

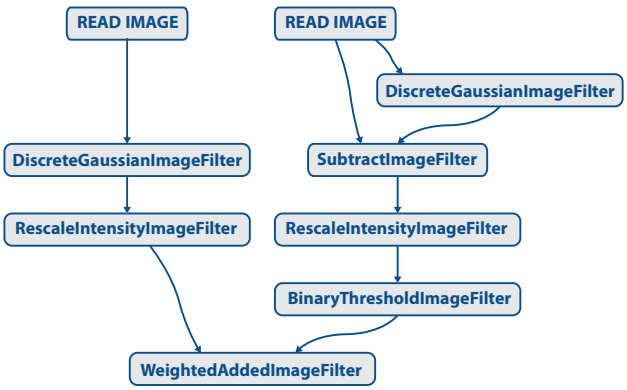


Figure 4: A full ITK pipeline that performs edge detecton and blend the result with another image.

update threads before performing the computation. The default implementation of these two functions are just calling the traversal and compute code, thus, should be fully backward compatible with all current ITK pipelines. The other change that we made to ITK's `ProcessObject` is adding a `SetScheduler` to allow each `ProcessObject` to have its own scheduling strategy besides setting the default scheduler in our new `ProcessScheduler` class.

Next, we are going to show an example to illustrate the new approach in building an advanced scheduler similar to the multithreaded VTK [2]'s.

4 Task-Parallelism Scheduler

Consider the pipeline described in Figure 4. It is similar to the edge detection pipeline in our first prototype but without the streaming part, thus, can be put entirely in the ITK framework. Here, we also make the final image showing the edge detected overlay on another blur image. The branch corresponded to the left "READ IMAGE" module contains the added components. All of the modules in this pipeline can be performed with ITK's default filter level multithreading, except for the reader. Based on our previous work [2] and our learning in Section 1, we believe that adding support for task-parallelism will improve performance for this pipeline, at least the image reading part. Therefore, we are building a custom scheduler for ITK v4 to test the flexibility of our design as well as assessing the performance of current ITK v4 processing pipeline.

From the user point of view, all that we need to add to the regular ITK pipeline creation process is a new class subclassed from `ProcessScheduler`, and set that scheduler to be the default one of ITK at runtime executing:

```
itk::ProcessScheduler::SetDefaultScheduler(MyScheduler::New());
```

For our task-parallel scheduler, we implement a simple task-parallel scheduling on top of ITK's filter level multithreading by making `scheduleTraversalUpstream` to assign "task branch" to modules during update propagation. In this implementation, we would divide the "task blend" into at most the number of sinks of the pipeline. In our `schedulePerformComputation`, we just push

all the compute requests in a queue and browse through them for execution when we have all the requests. The pseudo code for both functions are illustrated below:

```
class MySchedule : public itk::ProcessScheduler
{
public:

    void scheduleTraverseUpstream(itk::ProcessObject *proc)
    {
        if <proc is a source module && proc has not been visited>
            create a new task branch and assign to proc;

        // perform the original task of the function
        itk::ProcessScheduler::scheduleTraverseUpstream(proc);

        if <proc has parent module>
            assign the task rank of proc to its parent;
    }

    void schedulePerformComputation(itk::ProcessObject *proc)
    {
        if <proc has children>
            push proc to the job queue;
            return;

        // now we got a sink module/or the one who called Update()
        initialize runtime stats for all procs in the job queue;

        if <the number of tasks changed (including the first time)>
            assign the same last running time to all tasks;

        allocate the thread count for jobs  based on their last running time;

        call itk::MultiThreader::SetGlobalDefaultNumberOfThreads() -
          for each proc on the queue, then call  PerformComputation();
    }


}
```

For the results, Figure 5 shows the performance of the original ITK v4 pipeline versus the one with a custom task-parallel scheduler in processing 50 high definition photos. For all cases, the pipeline that was processed with the task-parallel scheduler performed considerably faster than a

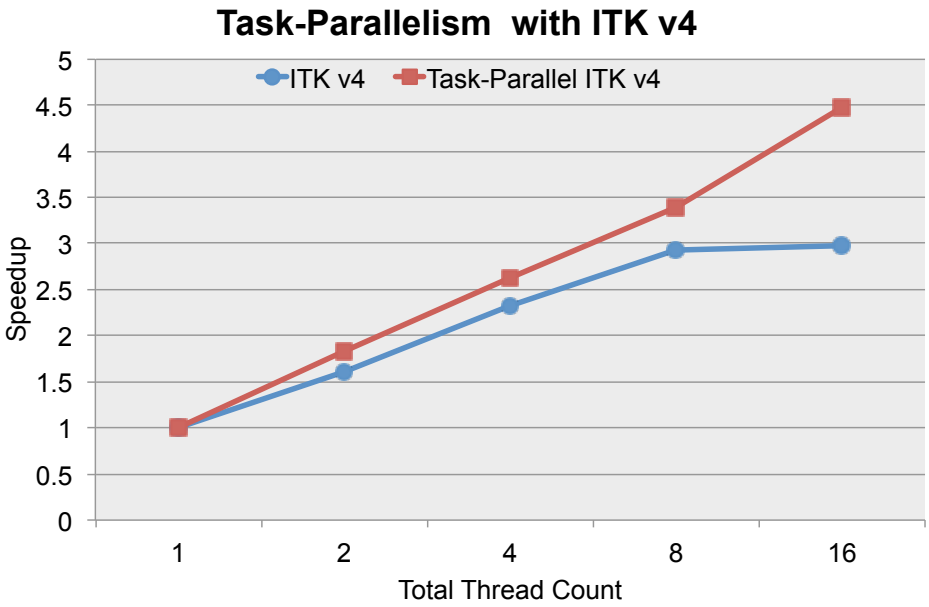


Figure 5: A speedup comparison between the original ITK v4 and one with a user-define task-parallel scheduler. Task-parallelism is shown to always be an additional help to any dataflow system, especially in maximizing the logical core technology.

pure filter level multithreading though both approaches used the same number of threads. The gap is biggest at 16 threads. It seems that when the system gets past the maximum limit of physical core (8), task-parallelism would be more helpful in pushing the performance of the logical ones (HyperThreading). Our conjecture is that task-parallelism focuses on distributing tasks among different modules, thus, can better explore the chance of decreasing the dependant instructions on the pipeline design of HyperThreading. A data-parallelism at filter level will most of the time hit the limit of data bandwidth, thus, will not be much beneficial from task-parallelism.

5 Design Principles

Through our study on the current ITK v4 design, we think that the following design principles need to be kept in place in ITK v4 in order to support highly parallel execution engine:

1. Multithreading at filter level may cause a drop in performance instead of expected gain. This is especially true when ITK v4 is embedded in other systems where there is always additional tasks running on the back. Since filter level multithreading only focuses on data-parallelism most of the time, always running at the maximum number of cores including the logical ones will not help much in increasing the performance. Instead, we suggest that each module should run at the maximum of physical core instead. This would provide otherwise wasted resources to both background inter-module coordination tasks.

2. The current design of processing objects in ITK v4 really limits the concurrent module execution because of the use of recursiveness. It is suggested that unrolling these logics and/or adding a way for general code injection like we did in Section 3 will really make ITK v4 much more flexible.

6 Source Codes

Extension to ITKv4 that are described in Section 2 and 3 can be found online at github.org using the following URL `https://github.com/hvo/ITK.git`, under the `HyperFlow` branch. Users can check out a complete modified ITKv4 using the following commands:

```
git clone https://github.com/hvo/ITK.git
cd ITK
git checkout -t origin/HyperFlow
```

Source codes for all the examples are available at `https://github.com/hvo/ITK_HF_Tests.git`. The example in Section 2 is located under the folder `itk_hf` where it can be compiled using CMake. It should be noted that the example can be executed with or without Qt depending on the value of CMake `USE_QT` variable. When Qt is not used, only timing statistics are reported. Below is the program usage:

```
Usage: ./edge_detection_itk [--itk] [--cpu <int>] filename
```

If `--itk` is specified, the module implementations that use ITK will be selected during execution. The `--cpu` switch can be used to specify how many thread that each module of ITK would run on. If this switch is omitted, the default value of ITK will be used. `filename` specifies a simple image streaming format where block of PNG images are stitched together. The example file can be downloaded at `http://vgc.poly.edu/files/ITK/SLC.streaming`.

Source codes for the the example shown in section 3 and 4 are located under the folder `itk_edge_detect` and can also be configured using CMake. The only dependency of this program is ITK. Below is the program usage:

```
Usage:  itk_edge_detect <-d|-t> <thread_count> <iterations> input output
        -d              : Use the default scheduling strategy
        -t              : Use the task-parallel scheduling strategy
        <thread_count>  : the number of ITK thread per module/global for the
                          default and task-parallel strategy. Specify 0 to
                          use the maximum number of threads.
        <iterations>    : the number of computing iterations. Default is 10.
```

Input and output are refering to image files that are supported in ITK. For example, the timing statistics in Figure 5 can be generated using the following bash script:

```
#!/bin/bash

# For the blue line
for i in 1 2 4 8 16
do
  ./itk_edge_detect -d $i input.jpg output.jpg
done

# For the red line
for i in 1 2 4 8 16
do
  ./itk_edge_detect -t $i input.jpg output.jpg
done
```


Acknowledgements

This work was funded by the National Institutes of Health under the ITKv4 effort with ARRA funds and the National Science Foundation (CCF-08560, CCF-0702817, CNS-0751152, CNS-1153503, IIS-0844572, IIS-0904631, IIS-0906379, IIS-1153728). Our work was performed as a subcontract to Kitware. We thank Luis Ibanez for comments on this report.

References

[1] H. T. Vo, D. K. Osmari, J. L. D. Comba, P. Lindstrom, and C. T. Silva. Hyperflow: A heterogeneous dataflow architecture. In *Eurographics Symposium on Parallel Graphics and Visualization 2012*, page to appear, 2012.

[2] H. T. Vo, D. K. Osmari, B. Summa, J. L. D. Comba, V. Pascucci, and C. T. Silva. Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum*, 29(3), 2010.