Coarse Grained Parallelism in VTK

The goal of this article is to answer the question: Should we adopt more coarse grained multithreading in VTK? In order to understand the potential benefits and work involved, we implement a few new coarse grained multithreading features for VTK execution pipeline.

For all experiments, we use the machine kadoop1 at KRS. This machine has two quad-core processors—making up a total of eight cores.

Code References

- Changes to VTK code can be checked out from the branch parallel-composite-pipeline in `git://github.com/leo1978/berks-vtk.git`. If you do not want to use some experimental performance enhancement code, you can also check out the topic "Towards a generic multithreading framework" from the branch.
- Test program code can be found at the master branch of `git://github.com/leo1978/tbb.git`. There is a python script "./testall.py" that tests the most important programs.

1 Intel TBB library

We rely on the Intel library Thread Building Block (TBB) for thread management. The reason for using such an external library is that wider adaptation of multithreading will almost certainly require some type of thread pool and dynamic load balancing, which VTK does not currently provide.

The main benefit of using thread pool is that the overhead of creating and destroying threads is minimized. In our experience, during a single program run, TBB never creates more threads than what is necessary to achieve the maximum parallelism, e.g. four threads on a quad-core machine. In contrast, `vtkMultiThreader` will create as many threads as tasks.

TBB performs load balancing automatically. To test whether the automatic load balancing works as advertised, we create a simple test, where a single "unbalance" parameter k controls how a fixed amount of work w is distributed among a set of n tasks: For task $0 \leqslant i < n$, its work is $w\left(k(\frac{2i}{n} - 1) + \frac{1}{n}\right)$.

Figure 1 shows that TBB load balancing performs well, resulting almost constant running time, while not load balancing results in running time that degrades with the unbalance parameter.

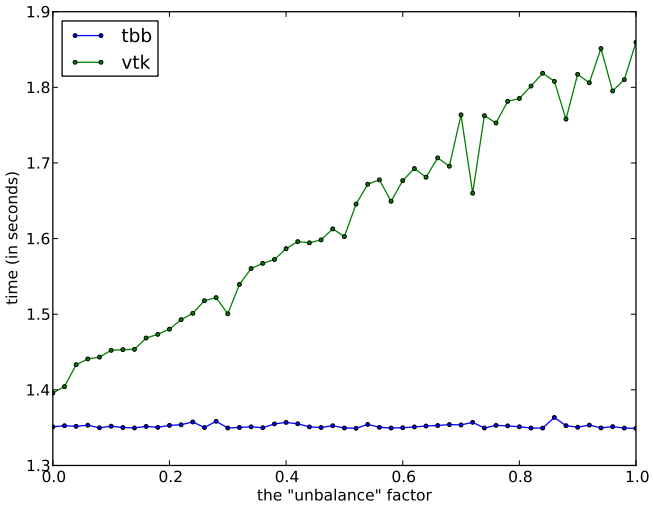

Figure 1: Running time as a function of the "unbalance" factor. Each data measurement is averaged over 10 runs. The "tbb" plot gives the result using `tbb::parallel_for`, and the "vtk" plot gives the results by first partitioning the tasks evenly in indices and running the threads using `vtkMultiThreader`

Code References

- Build configuration: Debug.
- Run `./testparallelfor [balance factor between 0 and 1] [-vtk|-tbb|-1]`, where the option "-vtk" uses vtkParallelFor and "-1" means single threaded execution.
- To generate the plots, run `python ./analyze-vtk-tbb.py [output without -noTbb] [output with -noTbb]`.

2 Testing correctness using `vtkParallelFor`

One drawback of using TBB is that it does not work well with helgrind—a popular open source tool for detecting race conditions. Running helgrind on TBB code generates many false positives even for extremely simple codes, and it is not clear how the false positives can be suppressed correctly.

For the purpose of correctness check, and also for comparing performance against a naive multithreading scheme, we implemented a template algorithm `vtkParallelFor` on top of existing VTK threads that have the same input and output as `tbb::parallel_for`, so switching between the two is easy.

3 Multithreaded Composite Data Executive

The current composite data pipeline iterates through the data blocks in a composite data and executes the algorithm for each block. We replace the serial execution by parallel execution. For each parallel task, we construct input and output vtkInformation objects before passing them to the filter. The code changes involves a small amount of refactoring of the class `vtkCompositeDataPipeline` for the purpose of introducing a multithreaded child `vtkThreadedCompositeDataPipeline`, which handles all the multithreading work.

3.1 Contouring AMR

For this experiment, we form a processing pipeline with three filters:

`vtkAMRFlashReader` → `vtkCellDataToPointData` → `vtkSynchronizedTemplates3D`.

The input is an AMR flash data set consists of 20K blocks of image data. The output is approximately 1M polygons. We only time the processing time of the last two filters, since only their times are affected by multithreading. We use `vtkParallelFor` for comparison.

The running time from both the Release and Debug builds are shown in the figure 2. We suspect that the difference in scaling behavior is due to the fact that individual images are very small (142 pixels on average). This causes the total running time to be sensitive to the overhead per image.

We can also see that the performance of TBB is very similar to vtkParallelFor. This can be read in two ways:

- The work load happens to be pretty evenly distributed across the AMR blocks so that TBB's load balancing cannot help much here.
- TBB did a good job choosing big enough task sizes so that the overhead per task (constructing vtkInformation objects) becomes neglible after amotizing over all the blocks. We found that, in the quad core case, TBB created only 354 tasks in total, which implies an average task size of 54. Had we forced the task size to be 1, the processing time would have tripled.

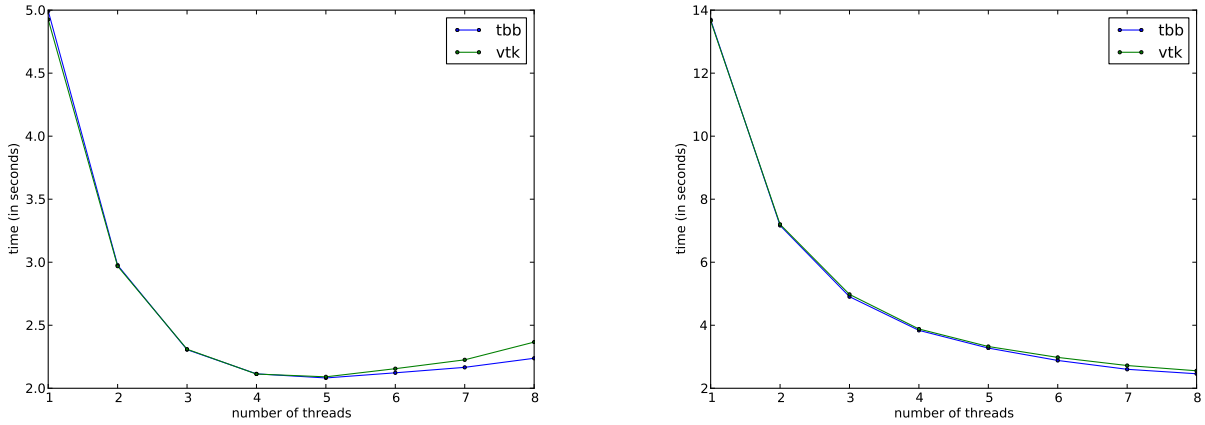

Figure 2: Running time as a function of the number of parallel threads. Left: Release; Right: Debug. Each data measurement is averaged over 10 runs. The "tbb" plot gives the result using `tbb::parallel_for`, and the "vtk" plot gives the results by using `vtkParallelFor`.

Code References

- Build configuration: `Release and Debug`.
- To generate the plots, run
  - `python run.py "./contouramr -bigdata" > pcontour-tbb.log`
  - `python run.py "./contouramr -bigdata -noTbb" > pcontour-vtk.log`
  - `python ./analyze-vtk-tbb.py pcontour-tbb.log pcontour-vtk.log`.

3.2 Stream tracing

For this experiment, we have a pipeline:

`vtkRTAnalyticSource` → `vtkImageGradient` → `vtkStreamTracer`.

The input is is an image of size $[-10, \ldots, 10]^3$, together with 1000 seeds uniformly sampled on a plane. The output has 2.6M points.

We made the following code changes:

- The input seed points are created as a multiblock data set, where each block is a `vtkPolyData` object with one point.
- The output traces are merged after running the stream tracer using `vtkAppendPolyData`.
- Refactoring out the member variables `InputData`, `HasMatchingPointAttributes`, and `LastStepSize`.
- In `vtkStreamTracer`, replace call to `vtkIdType* vtkImageData::GetIncrements()` by `vtkImageData::GetIncrement`

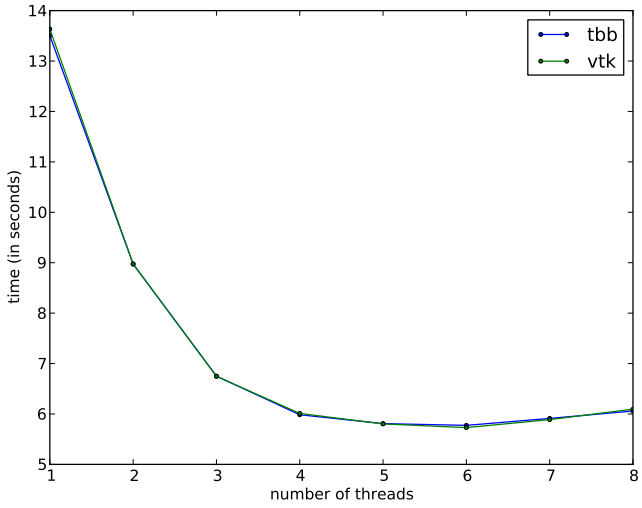

Figure 3: Running time of stream tracing as a function of the number of parallel threads. Each data measurement is averaged over 10 runs. The "tbb" plot gives the result using `tbb::parallel_for`, and the "vtk" plot gives the results by using `vtkParallelFor`

The running times are shown in figure 3. Surprisingly, even with what seems to be significant overhead of initializing the stream tracer, the running time decreases nicely as the number of threads increases.

Code References

- Build configuration: Release.
- To generate the plots, run

  1. `python run.py "./threaded_streamtrace -extent 100 -numTraces 1000" > stream-trace-tbb.log`
  2. `python run.py "./threaded_streamtrace -extent 100 -numTraces 1000 -noTbb" > stream-trace-vtk.lo`
  3. `python ./analyze-vtk-tbb.py stream-trace-vtk.log stream-trace-tbb.log`.

3.3 Contouring and Cutting Images

We multithreaded two algorithms for contouring and cutting: vtkSynchronizedTemplates3D and vtkSynchronizedTemplates3DCutter. These algorithms already take input image extents, so we divide the tasks according to image extents. The input is an image of size $[-200, 200]^3$, which we divide into $4^3 = 64$ chunks. Contouring produces 1.3M points and 2.7M cells; cutting produces 0.12M points and 0.24M cells.

The most significant part of the work went in speeding up merging the output from the chunks. The initial naive solution is to use vtkAppendPolyData to union the output and then use vtkCleanPolyData to resolve the duplicate points, but this turns out to be way too slow, as vtkCleanPolyData spends more time than contouring. We are able to implement a new algorithm that integrate functionality of both filters but runs 20 times faster. This makes it feasible to take the multithreaded approach.

In figure 4, we can see that the running time scales well, although merging time is still significant, particularly for large number of threads. However, it does have the nice property of being "output sensitive": this overhead directly depends on the output size, rather than the input.

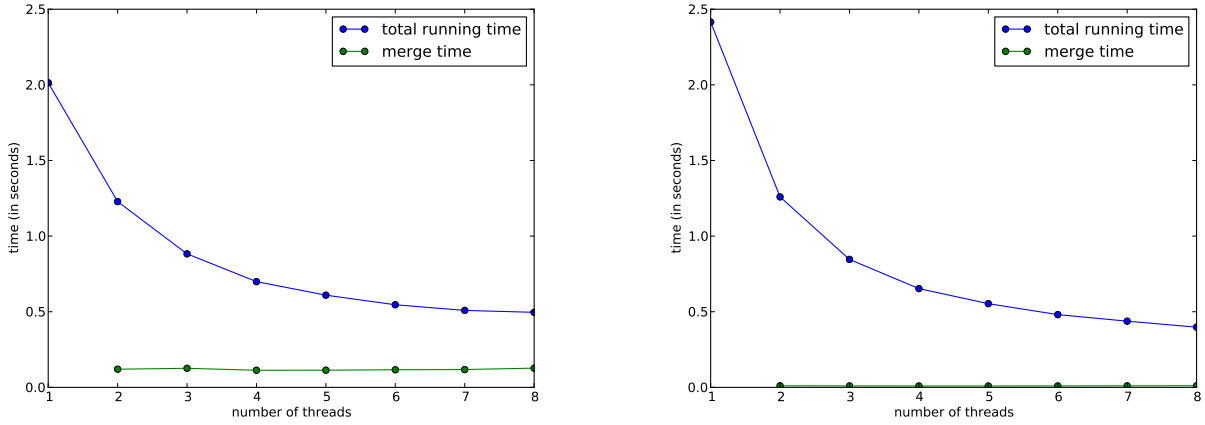

Figure 4: Running time of contouring (left) and cutting (right) as a function of the number of parallel threads. Each data measurement is averaged over 10 runs. The "merge time" plot shows the time to merge the polygons from all chunks.

Code References

- Configuration: Release.
- To generate the plots, run
  1. `python run.py "./pcontour -extent 200 " > pcontour.log`
  2. `python run.py "./pcontour -extent 200 -filter cutter" > pcutter.log`
  3. `python analyze-pcontour.py pcontour.log`.
  4. `python analyze-pcontour.py pcutter.log`.

4 Towards a multithreading framework

A common parallel algorithm pattern is to divide the data into chunks, process each chunk in parallel and finally merge the results. We capture this usage pattern here by adding two extra pipeline passes: `vtkThreadedCompositeDataPipeline::REQUEST_DIVIDE` and `vtkThreadedCompositeDataPipeline::REQUEST_MERGE`.

The results in the previous section are produced using this approach, where image splitting and surface data merging are handled in the respective passes.

5 Comparison with DAX

We compare the performance of TBB-configured DAX with our coarse grained approach. We contour two data sets: One has a single large image; the other is an AMR data set consisting of very small images. In the first case,the times are similar and, in the second case, our approach performs better by a small margin.

Figure 5 shows the timing details. We can see that the two data sets give different scaling behavior. For the super nova data set, both approaches scale well; for the AMR data set, dax cannot scale at all, probably due to the fact that the images are too small that they fall under the minimal TBB grain size.

We should mention that the comparison here is not entirely apple-to-apple: We use vtkSynchronizedTemplates while DAX has implemented only the marching cube algorithm; also, our time with no merging does not concatenate the output chunks, while DAX produces a single output array. However, considering the similarity of the two algorithms, we do not expect a more accurate comparison to yield very different conclusions.

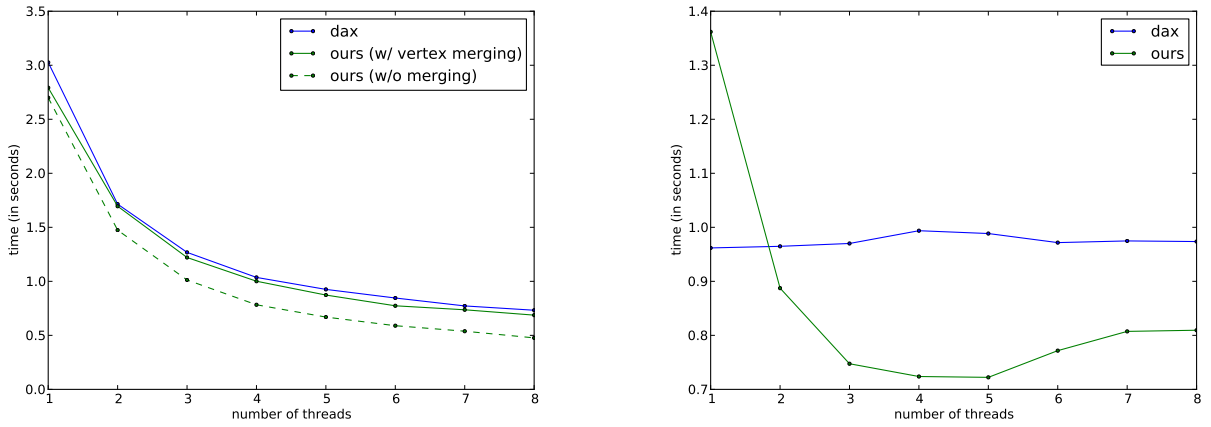

Figure 5: Contouring time (seconds) plotted against number of threads. Left: The input is the "SuperNova" datat set—a single image of size 432^3 (provided by Rob Maynard). Right: an AMR data set 'smooth.flash", which has around 20 thousands images of average size 142 (mostly of dimension $[0..8]^3$). All time measurements are averaged over 10 runs.

Code References

- The Dax tests are derived from code provided by Robert, in the directory MarchingCubeComparison.
- The DaxToolKit needs to be modified so that the user can specifiy the maximum number of threads.
- For the AMR data set, Dax must be configured with `DAX_USE_DOUBLE_PRECISION:BOOL=ON`.
- For the supernova data set, Dax must be configured with `DAX_USE_DOUBLE_PRECISION:BOOL=OFF` , run "./marchingCompare –numThreads [number of threads]".

6 Discussion

We identify patterns of VTK code that are not thread safe and propose changes.

- Various subclasses of vtkDataSet have get methods that modify temporary variables. e.g. `vtkIdType* vtkImageData::GetIncrements()`. The users must call `vtkImageData::GetIncrements(vtkIdType*)`.
- `vtkAlgorithm::UpdateProgress()` is not thread-safe
- The debugging configuration `VTK_DEBUG_LEAKS:BOOL=OFF` is not thread safe.

7 TBB Notes

TBB not only calls the `operator()` on a user defined task class, but also the copy constructor.