Parallel Processing with the SMP Framework in VTK

Berk Geveci
Kitware, Inc.

November 3, 2013

Introduction

The main objective of the SMP (symmetric multiprocessing) framework is to provide an infrastructure for easy development of shared memory parallel algorithms in VTK. In order to achieve this objective, we have developed core classes that support basic SMP functionality:

- Atomic integers and associated operations
- Thread local storage
- Parallel building blocks

That is mainly it! And not many building blocks are necessary to parallelize even more complex algorithms. At this point, we only have a parallel for loop implementation and we have demonstrated that complex algorithms can be parallelized with just these tools. As a basic reference for the work we have built upon, see `http://hal.inria.fr/hal-00789814`.

In the sections below, we describe the framework in more detail and provide examples of how it can be used. The main purpose of this document is to provide a background for developers that are interested in using the framework to parallelize existing algorithms or to develop parallel algorithms from ground up.

Fine- and Coarse-Grained Parallelism

When parallelizing an algorithm, it is important to first consider the dimension over which to parallelize it. For example, the Imaging modules parallelize many algorithms by assigning subsets of the input image (VOIs) to a thread safe function which processes them in parallel. Another example is parallelizing over blocks of a composite dataset (such as an AMR dataset). We refer to these as coarse-grained parallelism here. On the other hand, we can choose points or cells as a dimension over which to parallelize. Many algorithms simply loop over cells or points and are relatively trivial to parallelize this way. We refer to this approach as fine-grained parallelism here. Note that some algorithms fall into a gray area. For example, if we parallelize streamline generation over seeds, is it fine- or coarse-grained parallelism?

Backends

The SMP framework is essentially is a thin abstraction over a number of backends. Currently, we support 4 backends: Sequential (serial execution), Simple (uses vtkMultiThreader for simple parallelization), TBB (based on Intel's TBB) and Kaapi (based on Inria's XKaapi (`http://kaapi.gforge.inria.fr/`)). Note that only the TBB and Kaapi backends are suitable for parallel production use. The Simple backend is useful for debugging but should not be used unless no other backend can be made to work on the target platform. Sequential is the default backend.

Note that TBB and XKaapi use thread pools. They will create a number of threads when they are initialized and use those threads throughout the program until they are finalized. This minimizes the overhead of creating threads. The Simple backend currently creates a new set of threads in each parallel section. We will probably address this in the future.

Normally, initialization happens automatically. However, it is possible to control how many threads TBB and Simple will create by manually initializing the SMP framework as follows:

```
vtkSMPTools::Initialize(nThreads);
```

XKaapi does not provide an API for setting the number of threads. You need to either use a system level command (such as taskset on Linux) or set the KAAPI_CPUCOUNT environment variable.

Thread Safety

Probably, the most important thing in parallelizing an algorithms is to make sure that all operations that occur in a parallel region are performed in a thread-safe way. Note that there is much in the VTK core functionality that is not thread-safe. We are starting to work towards cleaning this up and marking APIs that are thread-safe to use. At this point, the best approach is to double check by looking at the implementation. Also, we highly recommend using an analysis tool such as Valgrind (with the Helgrind tool) to look for race conditions in your code. The Simple backend produces the best results with Helgrind.

Functors and Parallel For

Currently, the only parallel building block algorithm implemented is a parallel for loop. It looks like this:

```
class vtkSMPTools
{
public:

  template <typename Functor>
  static void For(vtkIdType first, vtkIdType last, vtkIdType grain, Functor& f);

  template <typename Functor>
  static void For(vtkIdType first, vtkIdType last, Functor& f);
};
```

Given a range defined by [first, last) and a functor, For() will call the functor's operator(), usually in parallel, over a number of subranges of [first, last). TBB and Kaapi backends will actually use task stealing to make sure that these subranges are assigned to threads in a way that is load balanced. Below is an example based on Filters/SMP/Testing/Cxx/TestSMPWarp.cxx.

```
class vtkSetFunctor
{
public:
  float* pts;
  float* disp;

  // Note that the parallelization of happening over
  // the outer dimension (the z dimension).
  void  operator()(vtkIdType begin, vtkIdType end)
  {
    vtkIdType offset = 3 * begin * resolution * resolution;
    float* itr = pts + offset;
    float* ditr = disp + offset;

    // Process the given subrange of k and all of i and j
    for (int k=begin; k<end; k++)
      for (int j=0; j<resolution; j++)
        for (int i=0; i<resolution; i++)
          {
          *itr = i*spacing;
          itr++;
          // ...

          *ditr = 10;
          ditr++;
          // ...
         }
  }
};

vtkNew<vtkStructuredGrid> sg;
sg->SetDimensions(resolution, resolution, resolution);

// ...

vtkSetFunctor func;
func.pts = (float*)pts->GetVoidPointer(0);
func.disp = (float*)disp->GetVoidPointer(0);
vtkSMPTools::For(0, resolution, func);
```

This will execute operator() in parallel. Note: don't expect this parallelization to scale well beyond a few cores given that it does a fair amount of data movement between the memory and the cores but very little actual math.

When using the first signature of For(), it is possible to provide a grain parameter. Grain is a hint to the underlying backend about the coarseness of the typical range when parallelizing a for loop. If you don't know what grain will work best for a partical problem, use the second signature and let the backend find a suitable grain. TBB specially does a good job with this. Somethimes, you can eek out a little bit more performance by setting the grain just right. Too small, the task queuing overload will be too much. Too little, load balancing will suffer.

Thread Local Storage

Thread local storage is generally referred to memory that is accessed by one thread only. In the SMP framework, vtkSMPThreadLocal and vtkSMPThreadLocalObject enable the creation objects local to executing threads. The main difference between the two is that vtkSMPThreadLocalObject makes it easy to manage vtkObject and subclasses by allocating and deleting them appropriately. Below is an example of thread local objects in use:

```
typedef vtkSMPThreadLocal<std::vector<double> > TLS;

class vtkBoundsFunctor
{
public:
  vtkFloatArray* pts;
  TLS LocalBounds;

  vtkBoundsFunctor(const std::vector<double>& exemplar) : LocalBounds(exemplar)
  {
  }

  void operator()(vtkIdType begin, vtkIdType end)
  {
    // Returns a std::vector<double> local to this
    // thread. The first call will create the vector,
    // all others will return the same.
    std::vector<double>& bds = this->LocalBounds.Local();
    double* lbounds = &bds[0];

    float* x = pts->GetPointer(3*begin);
    for (vtkIdType i=begin; i<end; i++)
      {
      lbounds[0] = x[0] < lbounds[0] ? x[0] : lbounds[0];
      // ...
      x += 3;
      }
  }
};

static const double defaults[] = { VTK_DOUBLE_MAX, - VTK_DOUBLE_MAX,
                                   VTK_DOUBLE_MAX, - VTK_DOUBLE_MAX,
                                   VTK_DOUBLE_MAX, - VTK_DOUBLE_MAX};
std::vector<std::vector<double> > bounds(defaults, defaults + 6);
vtkBoundsFunctor calcBounds(bounds);
vtkSMPTools::For(0, resolution*resolution*resolution, calcBounds);
```

A few things to note here:

- LocalBounds.Local() will return a new instance of a std::vector<std::vector<double> > per thread the first time it is called by that thread. All calls afterwards will return the same instance for that thread. Therefore, threads can safely access the local object over and over again without worrying about race conditions.

- The first call to Local() with initialize the new instance of the vector with the exemplar argument, which in this case is invalid bound values. This will use the copy constructor so if you use your own class, make sure that the copy constructor does the right thing.

So at the end of the For() call, the LocalBounds will contain a number of vectors, each that was populated by one thread during the parallel execution. These still need to be reduced to a single value. This can be achieved by iterating over all values and reducing them in the main thread as follows.

```
typedef TLS::iterator TLSIter;

TLSIter end = calcBounds.LocalBounds.end();
for (TLSIter itr = calcBounds.LocalBounds.begin(); itr != end; ++itr)
  {
  std::vector<double>& aBounds = *itr;
  bounds[0] = bounds[0] < aBounds[0] ? bounds[0] : aBounds[0];
  // ...
 }
```

Very important note: if you use more than one thread local storage object, don't assume that the iterators will traverse them in the same order. The iterator for one may return the value from thread i with begin() whereas the other may return the value form thread j. If you need to store and access values together, make sure to use a struct or class to group them.

Thread local objects are immensely useful. Often, visualization algorithms want to accumulate their output by appending to a data structure. For example, the contour filter iterates over cells and produces polygons that it adds to an output vtkPolyData. This is usually not a thread safe operation. One way to address this is to use locks that serialize writing to the output data structure. However, mutexes have a major impact on the scalability of parallel operations. Another solution is to produce a different vtkPolyData for each execution of the functor. However, this can lead to hundreds if not thousands of outputs that need to be merged, which is a difficult operation to scale. The best option is to use one vtkPolyData per thread using thread local objects. Since it is guaranteed that thread local objects are accessed by one thread at a time (but possibly in many consecutive functor invocations), it is thread safe for functors to keep adding polygons to these objects. The result is that the parallel section will produce only a few vtkPolyData, usually the same as the number of threads in the pool. It is much easier to efficiently merge these vtkPolyData.

Functors That Initialize

What would happen if we wanted a more complex initialization of the thread local object? Note that this can't be done before the parallel section starts as the objects local to other threads than main can't be created from the main thread. Neither can they be initialized as follows.

```
void operator()(vtkIdType begin, vtkIdType end)
{
  std::vector<double>& bds = this->LocalBounds.Local();
  bds.resize(6);
```

```
  memcpy(&bds[0], default_values, 6*sizeof(double));
  // ...
}
```

This is because after the first call, Local() will return the same object and this code will overwrite previously computed values. One way of getting around this is to do something like the following.

```
void operator()(vtkIdType begin, vtkIdType end)
{
  std::vector<double>& bds = this->LocalBounds.Local();
  int& inited = this->BoundsInitialized.Local()
  if (!inited)
    {
    bds.resize(6);
    memcpy(&bds[0], default_values, 6*sizeof(double));
    inited = 1;
    }
    // ...
}
```

This is such a common design pattern that we added the capability of doing it within a specialized vtkSMPTools::For() implementation. To use it, simply use a functor that has Initialize() and Reduce() functions as follows.

```
static const double defaults[] = { VTK_DOUBLE_MAX, - VTK_DOUBLE_MAX,
                                   VTK_DOUBLE_MAX, - VTK_DOUBLE_MAX,
                                   VTK_DOUBLE_MAX, - VTK_DOUBLE_MAX};
class vtkBoundsFunctor
{
  typedef vtkSMPThreadLocal<std::vector<double> > TLS;
  typedef TLS::iterator TLSIter;

public:
  vtkFloatArray* pts;
  TLS LocalBounds;
  double Bounds[6];

  vtkBoundsFunctor()
  {
    memcpy(this->Bounds, default, 6*sizeof(double));
  }
  void Initialize()
  {
    std::vector<double>& bds = this->LocalBounds.Local();
    bds.resize(6);
    memcpy(&bds[0], defaults, 6*sizeof(double));
  }

  void operator()(vtkIdType begin, vtkIdType end)
  {
    std::vector<double>& bds = LocalBounds.Local();
    double* lbounds = &bds[0];
```

```
    float* x = pts->GetPointer(3*begin);
    for (vtkIdType i=begin; i<end; i++)
      {
      lbounds[0] = x[0] < lbounds[0] ? x[0] : lbounds[0];
      // ...
      x += 3;
      }
  }

  void Reduce()
  {
    double* bounds = this->Bounds;

    TLSIter end = this->LocalBounds.end();
    for (TLSIter itr = this->LocalBounds.begin(); itr != end; ++itr)
      {
      std::vector<double>& aBounds = *itr;
      bounds[0] = bounds[0] < aBounds[0] ? bounds[0] : aBounds[0];
      // ...
      }
  }
};

vtkBoundsFunctor calcBounds();
vtkSMPTools::For(0, resolution*resolution*resolution, calcBounds);
```

Note that this encapsulates all of the previous code into the functor. The user can simply get the final bounds by accessing calcBounds.Bounds at the end of For().

Atomic Integers

Another very useful tool when developing shared memory parallel algorithms is atomic integers. Atomic integers provide the ability to manipulate integer values in a way that can't be interrupted by other threads. A very common use case for atomic integers is implementing global counters. For example, in VTK, the modified time (MTime) global counter and vtkObject's reference count are implemented as atomic integers.

VTK supports a subset of the C++11 atomic API; mainly ++, –, +=, -=, load and store. Note that all operations use full fencing (e.g. memory_order_seq_cst, if you don't know what this means, you can ignore it for now). Consider the following example.

```
static int Total = 0;
static vtkAtomicInt<vtkTypeInt32> TotalAtomic(0);
static const int Target = 1000000;
static const int NumThreads = 2;


VTK_THREAD_RETURN_TYPE MyFunction(void *)
{
```

```
  for (int i=0; i<Target/NumThreads; i++)
    {
    ++Total;
    ++TotalAtomic;
    }

  return VTK_THREAD_RETURN_VALUE;
}

vtkNew<vtkMultiThreader> mt;
mt->SetSingleMethod(MyFunction, NULL);
mt->SetNumberOfThreads(NumThreads);
mt->SingleMethodExecute();

cout << Total << " " << TotalAtomic.load() << endl;
```

When this program is executed, most of the time Total will be different (smaller) than Target whereas TotalAtomic will be exactly the same as Target. For example, on my Mac, one run prints: 999982 1000000. This is because when the integer is not atomic, both threads can read the same value of Total, increment and write out the same value, which leads to loosing one increment operation. Whereas, when ++ happens atomically, it is guaranteed that it will read, increment and write out Total all in one uninterruptable operation.

When atomic operations are supported at hardware level, they are very fast. Note that when the processor does not support atomic operations, we use locking to simulate this functionality, which can have a very negative impact on performance. For best performace, make sure that the atomic type that you would like to use is supported by the hardware on your platform as well as VTK's atomic implementation. As a rule of thumb, 64 bit atomic integers are not supported by 32 bit hardware (or 32 bit binaries most of the time).

Tips

In this section, we provide some tips that we hope will be useful to those that want to develop shared memory parallel algorithms.

Think about safety

First things first, think about thread safety. If the parallel section does not produce correct results consistently, there is not a lot of point in the performance improvement it produces. Read on common parallel design patterns. Check and double check your code for potential thread issues. Verify that the API you are using is thread safe under your particular application.

Helgrind is your friend

Valgrind's Helgrind is a wonderful tool. Use it often. We developed the Simple backend mainly to use with Helgrind. TBB and Kaapi produce many false positives within Helgrind so it is harder to instrument those backends. There are commercial tools with similar functionality also. Intel's Parallel Studio has static and dynamic checking.

Avoid locks

Mutexes are expensive. Avoid them as much as possible. Mutexes are usually implemented as a table of locks by the kernel. They take a lot of CPU cycles to acquire. Specially, if multiple threads want to acquire them in the same parallel section. Use atomic integers if necessary. Try your best to design your algorithm without modifying the same data concurrently.

Use atomics sparingly

Atomics are very useful. They are definitely much better than mutexes. However, overusing them may lead to performance issues. Try to design your algorithm in a way that you avoid locks and atomics. This also applies to using VTK classes that manipulate atomic integers such as MTime and reference count. Try to minimize operations that cause MTime or shared reference counts to change in parallel sections.

Grain can be important

In some situation, setting the right value for grain may be important. TBB does a decent job with this but there are situations where it can't do the optimal thing. Kaapi does not seem to set the grain automatically so we set the grain to be $\sqrt{n}$ where n is the number of items in the parallel for. There are a number of documents on setting the grain size with TBB on the Web. If you are interested in tuning your code further, we recommend taking a look at some of them.

Minimize data movement over work

This is true for serial parts of your code too but it is specially important when there are bunch of threads all accessing the main memory. This can really push the limits of the memory bus. Code that is not very intensive computationally compared to how much memory it consumes is unlikely to scale well. Good cache use helps of course but may not be always sufficient. Try to group work together in tighter loops.