Case Study: Parallelizing a Recursive Problem with Intel® Threading Building Blocks

Recently I have been working closely with DreamWorks Animation engineers to improve the performance of an existing key library in their rendering system. Using a combination of algorithmic improvements and parallelization, we achieved an overall of 35X performance improvement in some cases. One of the requirements for this project was to minimize changes to the library structure to control development cost. In this article, I will share some of the techniques I used to parallelize a recursive problem in the DreamWorks Animation rendering system.

Before I dive into the details, let me give a quick overview of the application that I was trying to analyze and speed up. Artists use digital content creation tools to create scene assets, such as virtual cameras, lights, and 3D models. To render an image using these scene assets files, they first must be parsed and then converted to rendering data structures before executing the renderer. This conversion step is referred to as data conditioning. Scene assets are represented by a graph. The graph has nodes representing objects like cameras, lights, and models. A node could also reference another node as an instance, for example, a forest of trees. The data conditioning step recursively transforms in-memory scene objects into the representation needed for rendering, copying all the necessary data. Data conditioning often involves a large amount of data and a large number of data objects. In practice the data conditioning cost varies widely depending on the number of objects and the complexity of objects in the scene. It’s essential that this conversion operation is done quickly because it happens at the start of each rendering process. Hypothetically, if it takes an hour to render a frame, we don’t want to spend 15 minutes in data conditioning. That would be a 25% overhead. More importantly, part of the interactive workflow, where a few seconds could make a big difference, also has to go through the data conditioning step. The main computation of the data conditioning library involved recursive graph traversal. My task was to figure out how to speed up this step as much as possible.
Data Transformation = ~100

Figure 1. Intel® VTune Amplifier XE Lightweight Hotspot shows the CPU utilization of the data conditioning library. The top area is showing the timeline and two green bars (each representing a thread). The active thread (second green bar) is showing CPU activities. The bottom half of the picture shows the overall CPU utilization.

A few of the Intel tools and technologies, such as Intel® C++ Compiler, Intel® VTune Amplifier XE, Intel® Inspector XE, and Intel® Threading Building Blocks, were essential in achieving the performance improvements. I used Intel® VTune Amplifier XE’s Lightweight Hotspot analysis to profile the library, see Figure 1. This picture is only showing the execution time inside the library, excluding all the system calls using the Module filter in VTune Amplifier XE. As shown in the time line, about 100 seconds are spent on data conditioning and overall CPU utilization is low. After some performance analysis, I found that the data conditioning library is a good candidate for parallel execution because in most cases, data objects can be processed independently.
Let’s take a look at the call stack from the Hotspot analysis shown in Figure 2. `conditionObject()` is the main entry point for the recursive traversal. The call stack goes much deeper, which is not shown in this picture. This type of recursion is called mutual recursion or indirect recursion. The `conditionObject()` method is called from many locations in the library. Almost 90% of the data transformation is spent on `conditionObject()`, so it’s a great target for parallelization.

Intel® VTune Amplifier XE analysis provides valuable information on where the performance issues are. For example, I discovered that during a function call, an object was automatically casted into another object when passed in as a function parameter. Constructing a new instance of that class is fairly expensive and that function was called frequently. It would be difficult to detect such issues without a tool like VTune Amplifier XE. Over the course of the project, DreamWorks Animation engineers made many algorithmic and implementation improvements (such as fixing the object casting issue) which sped up the single thread conditioning library performance by over 4.5X. Using TBB for parallelization, I was able to obtain an average of 6.25X additional speed up on an 8 core Xeon® system.

Figure 2. The conditioning library sequential execution call stack and corresponding computation cost. The middle column shows Self CPU time spent in a particular function. The right column shows Total CPU time which includes Self time and Self time of all functions that were called from that function. Here it is shown as a percentage of the total execution time of the run.
Intel has many technologies for enabling parallelization: TBB, OpenMP*, Cilk Plus*, and many others. I chose the TBB library because the problem we are trying to solve is complex and TBB has solved most of them already. Also DreamWorks has standardized on TBB as their primary programming model for data and task parallelism in their graphics code. The computation kernel (in this case, conditionObject()) can be considered as a TBB task. TBB already has thread-safe data structures and algorithms. We can leverage some of the important features of TBB, such as the high performance memory allocator, work-stealing based task scheduler, and synchronization primitives. One thing to note is that data conditioning involves allocating a large number of small objects. This is important to consider because the memory allocator could be a performance bottleneck in multithreaded applications due to synchronization. As I will show later, the TBB high performance memory allocator can help solve this problem (1).

One great way to learn TBB is to study how it’s used with design patterns (2). When I started working on this project, I considered whether I could simply apply some of the existing TBB design patterns to solve this problem. The recursion is similar to the Agglomeration and Divide and Conquer patterns, but in this case we are dealing with indirect recursion without a clean way to convert to direct recursion requiring a change to the implementation. There are also data dependencies and thread-safety issues that need to be resolved.

To figure out a solution, let’s step back and look at the problem in a more abstract way. From the call stack, you can see that the recursive computation is called from many different locations. The number of nodes we need to transform is unknown ahead of the time because nodes can create instances of other nodes. There are data dependencies between the nodes, which limits parallelism and increases complexity. Additionally, although one of the design goals of the data conditioning library is to be thread-safe, some parts of it are not. For example, the node data are accessed through a cache data structure called Context. This cache data structure is restricted to a single thread. Ideally, we want to enable multithreading while trying to minimize the changes to the library. More changes to the library means increased risks and complexity for the project.
We can actually solve each of the problems independently. To remove the dependencies between the nodes, we do the computation in the following two ways. One is to satisfy the bookkeeping of data objects so that the child node can be processed without blocking on the parent, see Figure 3. For example, we can keep track of all the nodes we have already visited and create an instance of the corresponding data object without actually filling in the data (which is the expensive part). This allows object instances to still reference each

Figure 3. (a) Shows the control flow diagram of the example recursive program. We start by visiting the root node and do some work. For example, the work might involve allocating new objects (e.g. cameras, lights, and 3D models), and then process and compute object data. If this node has children, we visit each child node and do some more work there. This is done recursively. (b) Shows the result of refactoring the code to prepare for parallelization.
other. Another way to remove data dependency is to extract these operations into a post-processing step. For example, when one node object requests data from another node object, this type of operation has to be moved into the post-processing stage after task synchronization.

![Flowchart](image)

**Figure 4.** Adding TBB into the mix. Instead of executing the compute kernel directly, a TBB task is created and spawned.

While we don't know the number of tasks ahead of time, using TBB we can create them recursively on the fly. This may not be the most optimal way of using TBB, but the
flexibility is important to us, see Figure 4. The independent part of the computation can then run in parallel. To work around the thread-safety issues of the cache, we can create an instance of the data structure for each thread. Fortunately, the cache only uses small amount of memory. If it’s unfeasible to create separate cache instances for each thread, I would look into changing the cache data structure to ensure thread-safety.

```cpp
#include "GraphNode.h"

void processNode(const GraphNode* node, const Context* context, const State& state) {
    work(node, context, state);
    for (GraphNode::ChildrenIterator child = node->children_begin(); child != node->children_end(); child++) {
        processNode(*child, context, state);
    }
}

int main(int argc, char *argv[])
{
    ...
    GraphNode* root = new GraphNode(rand());
    buildGraph(root, depthLevel, childLimit);
    State state;
    Context* context = new Context();
    processNode(root, context, state);

    // clean up
    ...
}
```

Figure 5. An example of a recursive program that traverses a graph and does some computation at each node of the graph.
Let's look at the source code of a simplified example program, see Figure 5. This example has a similar structure as the DreamWorks data conditioning library with many details in the original library safely ignored for the purpose of this discussion. We have a simple program which builds a graph, and for each of the node in the graph we want to do some work through the `processNode()` function. A few parameters are used by `processNode()`: the graph node, a context, and the state. `Context` has a cache that’s not thread-safe. The state object has everything else we need to carry around for the computation. Now we are going to make this program run in parallel.

```c
void bookkeeping(const GraphNode* node,
                 const Context* context,
                 const State& state) {
...
}
void computeKernel(const GraphNode* node,
                    const Context* context,
                    const State& state) {
...
}
void postProcess(const GraphNode* node,
                 const Context* context,
                 const State& state) {
...
}
void processNode(const GraphNode* node,
                 const Context* context,
                 const State& state) {
    bookkeeping(node, context, state);
    computeKernel(node, context, state);
}
```

Figure 6. Code refactoring to separate object data dependencies.

If you recall, our solution to remove object data dependencies is by separating work into multiple parts:

- Bookkeeping to manage new objects and inter-object references.
- Main computation kernel that processes and computes independent object data.
• Post-processing on object data that have dependencies. Everything else remains the same. Figure 6 shows the new structure of the code. The key is, with these changes, all the interfaces remained intact. Any external calls to processNode() need not be changed.

```c++
void processNode(GraphNode* node,
                 Context* context,
                 const State& state) {
  bookkeeping(node, context, state);
  ComputeTask* task = new(TASK_ROOT->allocate_additional_child_of(*TASK_ROOT))
                          ComputeTask(node, context, state);
  TASK_ROOT->spawn(*task);
}

int main(int argc, char *argv[]) {
  ...
  State state;
  Context* context = new Context();

  TASK_ROOT = new(tbb::task::allocate_root()) tbb::empty_task;
  TASK_ROOT->set_ref_count(1);
  processNode(root, context, state);
  TASK_ROOT->wait_for_all();
  TASK_ROOT->destroy(*TASK_ROOT);
  postProcess(root, context, state);

  // clean up
  ...
}
```

Figure 7. Added TBB code to spawn tasks for the compute kernel.

Now we are ready to add TBB into the mix. Since the computationKernel() function can be run in parallel safely, I will create a TBB task for it. Figure 7 shows the actual code to do just that. The bold faced lines are new code I have added to the example program. TASK_ROOT is going to be the parent task of the tasks we are going to create later. It’s an
empty_task because it doesn’t actually do anything. It’s important to set the reference count to 1 immediately. It’s used to let TBB know that I am going to call wait_for_all() in a blocking style. Otherwise, wait_for_all() might return before all the tasks complete. For empty_task I also have to destroy it explicitly when it’s no longer needed. Now look at the processNode() function. Instead of calling computeKernel(), I created a ComputeTask as the children of our TASK_ROOT. Allocate_additional_child_of() increases the reference count of the parent task. Then the child task is spawned.

```cpp
class ComputeTask: public tbb::task {
public:
  ComputeTask(GraphNode* node, 
              Context* context, 
              const State& state):
    mNode(node), 
    mContext(context), 
    mState(state)
  {}

  tbb::task* execute() {
    computeKernel(mNode, mContext, mState);
    return NULL;
  }

private:
  GraphNode* mNode;
  Context* mContext;
  const State mState;
};
```

Figure 8. The ComputeTask that is run by the TBB scheduler.

Figure 8 shows the implementation of ComputeTask which inherited from the tbb::task base class. It keeps a copy of all the function parameters we passed to it so that it can continue to run when TBB schedules an instance of this task and runs execute(). In this case, computeKernel() function is called with all the parameters and the proper values. While this code will run in parallel, we still have one remaining problem. Recall that Context is
not thread safe. So far we have one context that’s shared by all the tasks and threads. We need to fix this so that we don’t have race conditions. What we need to do is have an instance of Context for each thread. We don’t want to create a context for each task because that would be too expensive.

To get the per thread data, there are two ways you can do it. One way is for each thread to find out its own unique thread ID at run time. Another way is to use TBB thread local storage (TLS). TBB TLS is essentially a container that stores per thread data. In any given thread that’s running, you can ask for your local instance of the data from this container. Each instance of the data is only created the first time when a thread asks for it. For example, if your machine has 16 threads, and you allocated only 5 threads to run TBB tasks. There will be a maximum of 5 instances of the data created for these threads.

The Intel® TBB team has recommended using TLS rather than using threads ID for a number of reasons (3). TBB advocates task-based parallelism. It wants us to stay away from exposing the underlying threads. If you know the thread ID, then you can do things that TBB may not intend to be used for. TBB allows you get thread ID, but only use it if you have very good reasons. Another benefit of using thread local storage is that you don’t have to worry about how many threads you are working with or what type of system you are running on.
Figure 9. Use TBB thread local storage to access data that cannot be shared between threads.

Figure 9 shows how I used thread local storage. Instead of passing in the context when the task was created, I used the thread local context instance. I have declared a thread local storage type using TBB `enumerable_thread_specific` class. I also created an object called `THREADCONTEXT` initialized with an exemplar context object which will hold our thread local data. When `THREADCONTEXT::local()` is called, first `THREADCONTEXT` will check that for this thread whether a context object has already been allocated. If it has been allocated, `local()` simply returns a reference to it. Otherwise, a new instance of the context object will be created (through copy constructor) then returned.
In summary, I have divided the computation kernel at each node into three parts:

- Bookkeeping to keep track of the objects and allow node objects to reference each other if needed
- I moved data dependency into the post processing step so that the computation kernel can run independently in parallel.
- Thread local storage is used for non-thread-safe data to avoid race conditions.

Using these techniques, I was able to speed up the data conditioning library by 6.25X on average on an 8 core Xeon system without making major changes to the library interface. One of the test data, shot 2, was speed up by 35X comparing to the original ~100 seconds of run time, see Figure 10. Notice the speed improvements when TBB malloc is used to

---

**Figure 10.** RDL library performance comparison. Shot 2 is a medium size scene and our improvements resulted in 35X speed up, from the original 98.56 seconds run time down to 2.79 seconds. Legend: ST = single thread, MT-TBB = multithreaded using TBB, MT-TBB-Malloc = multithreaded using TBB and TBB malloc.
replace the standard malloc. I have done additional tests on shots of various sizes and the performance improvements are consistent. Of course, there is still room for improvement. This library uses mutexes in a couple locations to avoid thread safety issues. I believe we can improve the performance further by reducing the number of synchronizations. Some data which didn’t require locking, achieve over 8X speed up on an 8 core Xeon. In TBB 4.0, flow graph is introduced to solve graph related problems such as the one we discussed here. TBB graph could help simplify and solve some of the object data dependency problems.