DASH - Data Access and Sharing

Stefan Eilemann[*]
Blue Brain Project, EPFL

Abstract

Increased demand for parallelism, both in shared memory and distributed memory systems, calls for new software libraries to support the development of race-free parallel applications. The decreasing memory size and bandwidth per CPU instruction requires efficient memory usage in multithreaded applications.

In this poster we present DASH, a C++ library addressing generic, efficient multithreaded data access, sharing and synchronization in heterogeneous environments, focusing on the main use case of task-parallel analysis and visualization applications.

Index Terms: D.1.3 [Software]: Programming Techniques/Concurrent Programming—Parallel Programming; I.3.4 [Computer Graphics]: Graphics Utilities—Software Support; I.3.m [Computer Graphics]: Miscellaneous—Dataflow Programming

1 Introduction

The continuing validity of Moore's law, coupled with the clock rate wall, leads to processors with an increasing number of cores. To exploit this increased hardware parallelism in the form of multi-core CPU workstations, application software has to be parallelized. Data parallel and task parallel algorithms are the most prominent approaches to achieve parallelism within a task, and between tasks.

For task (pipeline) parallelism, downstream tasks read input data while upstream tasks produce output data for the next iteration concurrently. Implementing thread-safety for this access pattern leads to either increased memory usage by creating a full copy of the data, or to decreased parallelism due to locking of shared data. Established pipeline frameworks such as VTK [4] provide limited support for multi-threaded parallel task execution.

In this poster we present DASH, providing thread-safe data sharing through a generic data storage layer using an initial 'shared-all' state with lazy copy-on-write and explicit change propagation. This model allows the implementation of memory-efficient, highly parallel multithreaded programs by providing selective distributed memory with safe data synchronization to multithreaded programs. Typically DASH is used as a data backend in scene graphs and visualization libraries to implement thread-safe modifications.

2 Concepts and Programming Interface

The core of the DASH API consists of four classes: `Attribute`, `Node`, `Context` and `Commit`. An attribute holds any C++ object or POD type. A node has an arbitrary number of attributes, parent and child nodes. Nodes form a directed acyclic graph and store the actual data in attributes. A context provides an isolated view on the data stored in nodes and attributes. A commit is a set of changes emitted by and applied to contexts. Figure 1 depicts the core UML class hierarchy of DASH.

2.1 Attribute and Node

Nodes form directed acyclic graphs (DAG), in which attributes store user data. All data access to attributes and nodes is thread-safe, as long as they are accessed from different contexts. Modifications are not visible to other contexts until the change is synchronized by applying the respective commit to the other context.

[*]e-mail: stefan.eilemann@epfl.ch

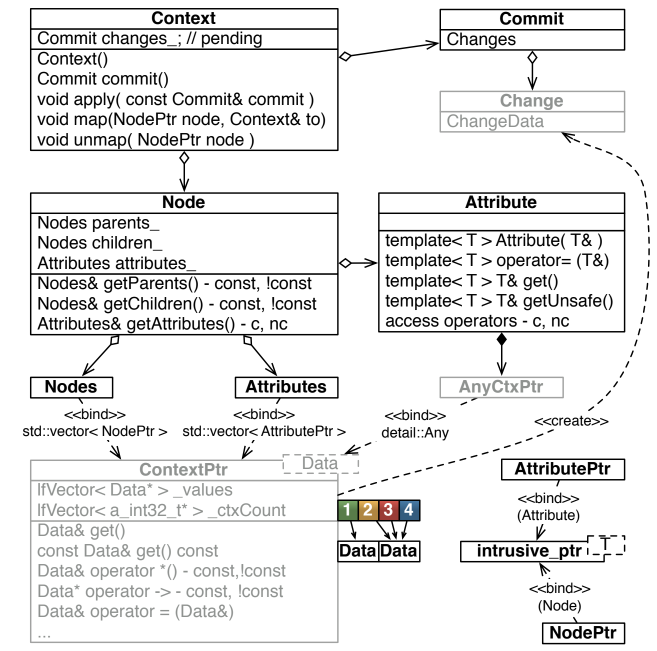
Context
Commit changes_; // pending
Context()
Commit commit()
void apply(const Commit& commit)
void map(NodePtr node, Context& to)
void unmap(NodePtr node)
Commit
Changes
Change
ChangeData
Node
Nodes parents_
Nodes children_
Attributes attributes_
Nodes& getParents() - const, !const
Nodes& getChildren() - const, !const
Attributes& getAttributes() - c, nc
Attribute
template< T > Attribute(T&)
template< T > operator= (T&)
template< T > T& get()
template< T > T& getUnsafe()
access operators - c, nc
Nodes
Attributes
AnyCtxPtr
<<bind>>
std::vector< NodePtr >
<<bind>>
std::vector< AttributePtr >
<<bind>>
detail::Any
<<create>>
ContextPtr
Data
lfVector< Data* > _values
lfVector< a_int32_t* > _ctxCount
Data& get()
const Data& get() const
Data& operator *() - const,!const
Data* operator -> - const, !const
Data& operator = (Data&)
...
1
2
3
4
Data
Data
AttributePtr
<<bind>>
(Attribute)
intrusive_ptr
T
<<bind>>
(Node)
NodePtr

Figure 1: DASH UML class diagram with public (black) and internal (grey) classes.

The attribute uses an internal implementation extended from boost::any [3] by boost::serialization support. The companion library CoDASH uses this serialization to implement data synchronization between different processes and machines using the Collage network library [2].

2.2 Context and Commit

A context provides a view on the data stored in nodes and attributes. Nodes and attributes need to be explicitly mapped to contexts to be visible in them. A commit is a set of changes. It is generated by `Context::commit`, which returns and resets the change set of the context. Commits are light-weight objects meant to be communicated from one thread to another, e.g., by using a thread-safe queue.

A context is a handle to a separate address space for all DASH nodes and attributes. The current context is a thread-local variable, that is, `Context::setCurrent` sets the context current in the calling thread. Each thread can have a different context and multiple threads may use the same context. A thread always has a context and by default a main context is created and used. The loose coupling between threads and contexts allows different use, e.g., employing two contexts from the same thread to switch between two views of data (fast undo/redo) or to use the same context in multiple threads for data-parallel operations (with additional locking).

Concurrent read and write access to distinct contexts is thread-safe, while access from multiple threads to one node or attribute instance from a single context is not. The first write to a shared node or attribute creates a copy and a change in the current context. Writes are only visible to the current context until they are propagated using `Context::commit()` and `Context::apply()`. The commit and apply operations are thread-safe with any operation in all other contexts, but not thread-safe with operations on the respective context.

3 Task-Parallel Pipelines with DASH

The primary use case for DASH is to implement parallel dataflow pipelines efficiently. Each pipeline filter may run in a separate thread, and can have different synchronization models between them. Figure 2 shows one exemplary processing pipeline.

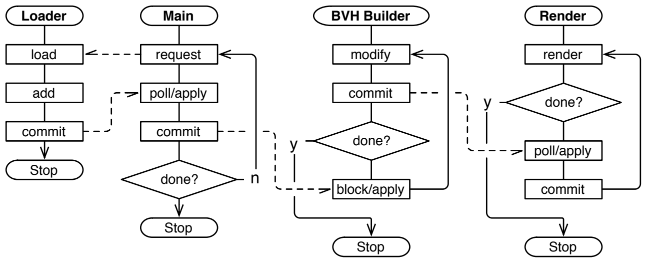

Figure 2: Exemplary DASH pipeline.

The main loop implements the application logic and has the main context. During startup, it spawns two threads: one for updating an acceleration structure and one for rendering using the acceleration structure, each having its own context. When the user requests a file load, it spawns a loader thread.

The loader thread creates its own context and makes it current. It maps the relevant application data to this context, and starts loading the requested data. When it is finished, it commits its context, posts a message to the application containing the commit and exits.

Meanwhile, the application thread continued to run. It regularly polls its message queue, and eventually applies the commit from the loader thread. This will add all loaded data to the application view, and generates a new set of changes. On the next frame, it commits its context, posts the commit to the acceleration builder thread and continues operations.

The acceleration builder thread blocks on new commits. It applies all queued commits, rebuilds the acceleration structure, commits the context, pushes the commit to the rendering thread and then blocks again. It uses the geometry read by the loader in a read-only fashion (no copy) and updates the acceleration structure (copies, propagated by commit).

The render thread, as the application thread, runs at an interactive framerate and polls the queue from the acceleration builder each frame. Updates are applied to its context as they are received, and the new data is used from thereon.

This example illustrates the power of DASH as a building block in task-parallel applications. Separate threads can run asynchronously (builder – render) and synchronously (application – render), operate on the same objects and only incur temporary copies for the changed data. Neither the API, nor the design of DASH restricts it the usage to this use case. We took special care to remove a priori constraints to make room for creative use of the DASH API.

4 Implementation

The main requirements for the design of DASH are memory efficiency, very low overhead data access, fast synchronization between threads and extensibility to data distribution in clusters.

The implementation relies on atomic variables and lock-free algorithms to guarantee that concurrent read accesses are lock-free and wait-free, uncontended writes are fast and data updates between contexts do not require copies.

The core of the implementation is a lock-free vector using the algorithm described by [1]. An internal `ContextPtr` implements the copy-on-write and associated change notifications. It behaves like a `shared_ptr` with added functionality. The context pointer uses a slot index stored in the current context to access the actual data pointer stored in a lock-free vector.

A const read on the data retrieves the slot index from the current context and looks up the object stored in the given slot. A non-const read retrieves the slot, checks if the object is shared, creates and notifies the copy if needed, and returns the unique copy. Figure 3 illustrates this algorithm, where context A modifies an object while context B reads it from a different thread. When the commit of context A is applied to context B, the original object will be released and both contexts will share the new copy.

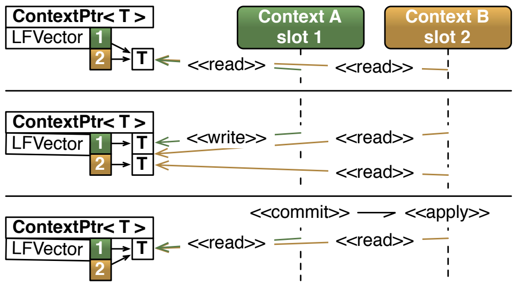

Figure 3: Context pointer before (top) and after (middle) copy-on-write, after change propagation (bottom).

The overhead of reading and writing through a context pointer over a `boost::shared_ptr` is one thread-local storage (TLS) read and one array lookup into the lock-free vector, plus a shallow copy of the object on the first write only. Microbenchmarks reading or writing a single four-byte value show that the context pointer is about five times slower reading and two times slower writing data. Removing the TLS lookup by explicitely passing the current context in the API makes the context pointer less than two times slower than a `boost::shared_ptr` in all cases.

5 Conclusion

In this poster, we present a novel approach for thread-safe data access in multithreaded applications. Using state of the art algorithms, we derive a simple, potent and efficient open source implementation. The design is driven by the KISS principle to create a simple, yet powerful building block for parallel applications.

We demonstrate how DASH can be used in task-parallel applications, which is one of the important use cases influencing the design. CoDASH, a prototype implementation, validates that DASH can easily be extended to share data between processes on distributed memory systems.

Acknowledgements

This work was supported in part by the Blue Brain Project and the Swiss National Science Foundation under Grant 200020-129525.

References

[1] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In *OPODIS*, pages 142–156, 2006.
[2] Eyescale Software GmbH. Collage Network Library. http://www.libcollage.net/, 2012.
[3] K. Henney. Boost.Any C++ Library. http://www.boost.org/libs/any/index.html, 2001.
[4] W. J. Schroeder, L. S. Avila, and W. Hoffman. Visualizing with VTK: A Tutorial. *IEEE Computer Graphics and Applications*, 20:20–27, 2000.