RedGrapes Documentation

Michael Sippel

Dec 15, 2019

OVERVIEW

RedGrapes is a C++14 framework for declaratively creating and scheduling task-graphs, based on high-level resource descriptions.

CHAPTER

ONE

MOTIVATION

Writing scalable software using bare threads is hard and error-prone, especially if the workload depends on input parameters and asynchronous operations further complicating the program flow. For this reason the decoupling of processing stages from their execution is useful because it allows to dynamically schedule them. This is typically done with task-graphs, which are directed acyclic graphs (DAGs), whose vertices are some sort of computation and the edges denote the execution precedence order. This execution precedence results from the dataflow between the tasks, which gets complex pretty fast and may also be dynamic which makes it nearly impossible to manually write explicit task dependencies. So ideally these would be derived from some sort of high-level description of the dataflow. The goal of this project is to provide a task-based programming framework, where the task-graph gets created declaratively.

CHAPTER

TWO

EXAMPLE

TODO

CHAPTER

THREE

REQUIREMENTS

- C++14
- Boost >= 1.62

CHAPTER

FOUR

BUILD A PROJECT USING REDGRAPES

RedGrapes is a C++ header-only library so you only need to set the include path. If you are using CMake, the following is sufficient:

```
find_package(redGrapes REQUIRED CONFIG PATHS "[path to redGrapes]")
include_directories(SYSTEM ${redGrapes_INCLUDE_DIRS})
```

CHAPTER

FIVE

EXAMPLES & TESTS

In order to build the examples and tests, do the typical cmake procedure:

```
mkdir build
cd build
cmake ..
make -j
```

CHAPTER

SIX

GETTING STARTED

This chapter shows exemplary how to use RedGrapes. The features are introduced in the order of the sections from basic to more advanced. Most code snippets are compilable and you may be able to understand the basic usage by looking through them.

6.1 Task Creation

6.1.1 The Manager

The very first thing to do in every application using RedGrapes is to create a *manager*. It combines all required components and provides us with an interface for creating tasks. You also might want to create a namespace alias.

```
#include <redGrapes/manager.hpp>

namespace rg = redGrapes;

int main()
{
    rg::Manager<> mgr;

    return 0;
}
```

Its template arguments allow an application specific configuration and are discussed in the following sections (see also *Extending Task Properties* as well as *Writing Custom Schedulers*), but it is also usable with defaults. The runtime parameter is the number of worker threads which are created additionally to the main thread. By default, it uses the result of `std::hardware_concurrency()`.

```
rg::Manager<TaskProperties, EnqueuePolicy, Scheduler> mgr( n_threads );
```

By the the manager-object's destructor, the thread (which is the main thread) will behave as additional worker thread until all tasks are consumed. Only then the destruction of the manager returns.

6.1.2 Starting a Task

To create a task, the manager method `emplace_task()` is used. The first parameter is any nullary callable. By using `emplace_task()` the scheduler is automatically activated and the task will get scheduled and executed in one of the worker threads.

```
#include <iostream>
#include <redGrapes/manager.hpp>

namespace rg = redGrapes;

int main()
{
    rg::Manager<> mgr;

    mgr.emplace_task(
        []
        {
            std::cout << "Hello World!" << std::endl;
        }
    );

    return 0;
}
```

Caution: Tasks are executed asynchronously, so be sure that all captures outlive the tasks execution. For best practice see *Lifetimes of Captured Variables*.

6.1.3 Return Values

The callable passed to `emplace_task()` can have any return type. The result can be retrieved through a future object which is returned by `emplace_task()`.

```
auto result = mgr.emplace_task( []{ return 123; } );
assert( result.get() == 123 );
```

Caution: Always use `auto` on task results. Do not cast them to `std::future`, deadlocks might occur! (See *Task-Results*)

6.2 Task Properties

Every task has *properties*, which contain additional scheduling or debug information about that task. What these task-properties are, must be configured by the user. This is typically done by combining multiple predefined and custom property classes, each providing a *builder*. RedGrapes provides the means for combining such independent property definitions accordingly from a variadic template:

```
redGrapes::TaskProperties< Property1, Property2, ... >
```

When creating a task using `emplace_task()`, the second parameter is the task properties. Each individual property class should have sensible defaults and provide builder functions for creating property-configurations nicely.

Here is a full example using the predefined `LabelProperty`:

```
#include <iostream>
#include <redGrapes/manager.hpp>
#include <redGrapes/property/label.hpp>

namespace rg = redGrapes;

using TaskProperties = rg::TaskProperties< rg::LabelProperty >;

int main()
{
    rg::Manager< TaskProperties > mgr;

    mgr.emplace_task(
        [] { std::cout << "Hello World!" << std::endl; },
        TaskProperties::Builder().label( "Example Task" )
    );

    return 0;
}
```

Another essential predefined property is the `ResourceProperty`, which will be discussed in the next section!

6.3 Describing Dataflows

Dataflows occour whenever tasks share any kind of data, i.e. one task outputs data which is used as input for the next. Dataflows between tasks determine their dependencies, i.e. which tasks must be absolutely kept in order and serial. In RedGrapes this is expressed using *resources*. Each resource represents shared data. Their possible usage by tasks is modelled by an *access policy*, which defines all possible access modes for a task on this resource, e.g. *read/write*. An specific configuration of a resource and its access mode is called *resource access*. Tasks can now store a list of resoruce accesses in their properties which is then used to derive the task precedence.

6.3.1 Task Dependencies

When creating a new task, it is inserted into the precedence graph based on an *EqueuePolicy*, which compares the properties of two tasks and decides whether they are dependent. This is done in reverse with all previously inserted tasks to calculate the task dependencies. The manager must be configured with an enqueue policy. `redGrapes::ResourceEnqueuePolicy` is predefined and uses the resource properties which are defined with `redGrapes::ResourceProperty`.

```
using TaskProperties =
    rg::TaskProperties<
        redGrapes::ResourceProperty,
        /* other properties ... */
    >;

rg::Manager<
    TaskProperties,
    rg::ResourceEnqueuePolicy
> mgr;
```

6.3.2 Resources

The next thing to do is to represent the resources in your code. Any data that is shared between tasks should be represented as resource. Generally resources are just identifiers but there are also wrappers which are memory managed to make resource usage more safe. A very simple, predefined access policy is `IOAccess`. It supports the access modes *read* and *write*, where reads can be executed independently.

```
#include <redGrapes/resource/resource.hpp>
#include <redGrapes/access/io.hpp>

// just an identifier, no association with actual data
rg::Resource< rg::access::IOAccess > r1;
```

Resource Access

Resource accesses are created with the method `Resource::make_access(AccessPolicy)` and can be added to tasks like normal properties. This is the information used by the enqueue policy.

```
mgr.emplace_task(
    []{ /* ... */ },
    TaskProperties::Builder().resources({ r1.make_access( rg::access::IOAccess::read␣
↪) })
);
```

Shared Resource Objects

Using just the previously described mechanisms would require for each shared object an additional resource object and doesn't give any guarantees about what is actually done in the task. So we could just get the resource accesses wrong and don't know about it. Furthermore the data must absolutely outlive the execution of all tasks.

`rg::SharedResourceObject< T, AccessPolicy >` is an `Resource<AccessPolicy>` and additionally stores an `shared_ptr<T>`. So we firstly have the data and the resource identifier united into one object and secondly all lifetime issues are solved through reference counting.

Tip: To avoid lifetime issues, be strict and never capture anything by reference. Only allow copy and move captures.

Access Guards

By manually adding the resource accesses to the task properties we still cannot check if all operations inside the task are correctly represented by them. The solution to this problem are *access guards*: Wrappers around a *shared resource object*, for each possible access mode one, that only allows the operations corresponding to the access. For *read*/*write* this would be an dereference to `T const&` or `T&` respectively.

Additionally we need to create both the guard object and the task property together with one expression. This is done with so called *property building parameters*. These are function parameters which are bound to the task immediately at creation (to make it ultimately nullary again), but additionally implement a trait in which they can use the property-builder to modify the task properties. Each access-guard simply implements this trait and so by taking all resources by parameter instead of capture we can use the correct wrapper.

See also *Creating new Resource-Types*.

For convenience the guard objects also provide methods to create new guard objects with a subset of the access.

```
#include <redGrapes/resource/ioresource.hpp>

rg::IOResource< int > r1;

mgr.emplace_task(
    []( auto r1 )
    {
        // ok.
        std::cout << *r1 << std::endl;

        // compile-time error!
        *r1 = 123;
    },
    r1.read()
);
```

Tip: Altough it is possible to capture resources and add their properties via builders, it is recommended to access them through the parameters, because then the resource usage in the task is checked at compile time.

6.3.3 Full Example

In this example *Task 2* and *Task 3* will be executed after *Task 1*. When enough threads are available, *Task 2* and *Task 3* will run in parallel.

```
#include <redGrapes/manager.hpp>
#include <redGrapes/resource/ioresource.hpp>
#include <redGrapes/property/inherit.hpp>
#include <redGrapes/property/resource.hpp>
#include <redGrapes/property/label.hpp>

namespace rg = redGrapes;

using TaskProperties =
    rg::TaskProperties<
        rg::ResourceProperty,
        rg::LabelProperty
    >;

int main()
{
    rg::Manager< TaskProperties, rg::ResourceEnqueuePolicy > mgr;

    rg::IOResource< int > a;

    mgr.emplace_task(
        []( auto a ){ *a = 123; },
        TaskProperties::Builder().label("Task 1"),
        a.write()
    );

    mgr.emplace_task(
        []( auto a ){ int x = *a; },
        TaskProperties::Builder().label("Task 2"),
        a.read()
```

(continues on next page)

(continued from previous page)

```
    );

    mgr.emplace_task(
        []( auto a ){ int x = *a; },
        TaskProperties::Builder().label("Task 3"),
        a.read()
    );

    return 0;
}
```

6.4 Refining Tasks

It is possible to create a sub-graph inside a task during its execution. This is done without further thought by just calling `emplace_task()` inside another task. Either you always capture the manager by reference or create a singleton (See *Singleton for Manager*).

```
mgr.emplace_task(
    [&mgr]
    {
        mgr.emplace_task(
            []{ /* ... */ },
            TaskProperties::Builder().label("Child Task")
        );
    },
    TaskProperties::Builder().label("Parent Task")
);
```

6.4.1 Property Constraints

Because the properties of the parent task already made decisions about the scheduling, any child tasks are not allowed to revert these assumptions. So the properties of child tasks are constrained and assertet at task creation. This is implemented by the EnqueuePolicy. In case of using the predefined *ResourceEnqueuePolicy*, it asserts the resource accesses of the parent task to be supersets of its child tasks. That means firstly no new resources should be introduced and secondly all access modes must be less or equally "mutable", e.g. a child task cannot write a resource that is only read by the parent task.

Note: Not meeting the resource constraint will throw an exception when calling `emplace_task()`. This is only possible because we don't use access guards in this example.

```
 rg::Resource< rg::access::IOAccess > r1;

 mgr.emplace_task(
    [&mgr, r1]
    {
        // OK.
        mgr.emplace_task(
            []{ /* ... */ },
            TaskProperties::Builder()
                .label("good child")
```

(continues on next page)

(continued from previous page)

```
                .resources({ r1.make_access(rg::access::IOAccess::read) })
        );

        // throws runtime error
        mgr.emplace_task(
            []{ /* ... */ },
            TaskProperties::Builder()
                .label("bad child")
                .resources({ r1.make_access(rg::access::IOAccess::write) })
        );
    },
    TaskProperties::Builder()
        .label("Parent Task")
        .resources({ r1.make_access(rg::access::IOAccess::read) })
);
```

6.4.2 Resource Scopes

It is also possible to create resources which exist locally inside a task and are only relevant for sub-tasks.

```
rg::IOResource< int > r1;

mgr.emplace_task(
    [&mgr]( auto r1 )
    {
        rg::IOResource< int > local_resource;

        mgr.emplace_task(
            []( auto r1, auto r2 ){ /* ... */ },
            TaskProperties::Builder().label("Child Task 1"),
            r1.read(),
            // use local_resource here without violating the subset constraint
            local_resource.write(),
        );

        mgr.emplace_task(
            []( auto r ){ /* ... */ },
            TaskProperties::Builder().label("Child Task 2"),
            local_resource.read()
        );
    },
    TaskProperties::Builder().label("Parent Task")

    // can't and doesn't need local_resource
    r1.read()
);
```

Note: The context in which the constructor of a resource is called determines its *scope-level*. Local resources should therefore be constructed inside of the parent task.

6.5 Access Demotion

A very cool feature is that functors can modify their own properties while runnig. This allows us for example to demote resource accesses so that other functors can start earlier. Of course the possible changes at runtime have to be well constrained, similarly to creating sub-tasks.

This is done by creating a **patch** in the same manner with builders as the initial properties. This patch is then applied to the current task by the manager method `update_properties()`. This method must be called inside of a task and applies for exactly the task it is called in. This call also automatically triggers the scheduler to reevaluate outgoing edges in the task-graph.

The builder ResourceProperty provides in its builder interface the methods `add_resources()` and `remove_resources` for changing the resource access information.

Note: In the case of ResourceProperty it is only possible to **demote** the access, i.e. the new access has to be a subset of the old (e.g. we can change a write to read).

Caution: When using access demotion, it is possible again to mess up the actual resource usage and properties, despite access guards, because we can't "delete" a symbol inside a scope.

```
rg::IOResource< int > r1;

mgr.emplace_task(
    [&mgr]( auto r1 )
    {
        // OK.
        mgr.update_properties(
            TaskProperties::Patch::Builder()
                .remove_resources({ r1.write() })
                .add_resources({ r1.read() })
        );

        // compiles, but is wrong
        // be sure to avoid this
        *r1 = 123;

        // throws runtime error, only demotion allowed
        mgr.update_properties(
            TaskProperties::Patch::Builder()
                .add_resources({ r1.write() })
        );
    },
    r1.write()
);
```

CHAPTER

SEVEN

BEST PRACTICES

7.1 Singleton for Manager

An easy way to make the manager globally accessible is to create a singleton:

```
using TaskProperties = rg::TaskProperties< /*...*/ >;
static auto & mgr()
{
    static rg::Manager<
        TaskProperties,
        rg::ResourceEnqueuePolicy,
        MyScheduler
    > m;

    return m;
}

void foo()
{
    mgr().emplace_task( []{ /* ... */ } );
}
```

7.2 Lifetimes of Captured Variables

use `shared_ptr`

7.3 Task-Results

always use `auto`

7.4 Writing Container Classes

If you implement a structure which should be used **as** resource, then just derive from the corresponding resource type:

```
struct MyContainer : rg::IOResource {
    // ...
}
```

TODO: Access Guards

CHAPTER

EIGHT

DEBUGGING

8.1 Task Backtraces

Sometimes it is useful to create a backtrace of tasks. This can be done with the manager method `backtrace()`. It returns a `std::vector<TaskProperties>`.

```
mgr().emplace_task(
    []
    {
        mgr().emplace_task(
            []
            {
                int depth = 0;
                for( auto t : mgr().backtrace() )
                    std::cout << "[" << depth++ << "]" << t.label << std::endl;
            },
            TaskProperties::Builder().label("Child Task")
        );
    },
    TaskProperties::Builder().label("Parent Task")
);
```

This will give us the output:

```
[0] Child Task
[1] Parent Task
```

8.2 Writing out the Task-Graph

TODO

CHAPTER

NINE

DOMAIN SPECIFIC EXTENSIONS

9.1 Creating new Resource-Types

Lets suppose your own resource does more than only read/write. Then you want to define your own AccessPolicy which encodes the possible accesses to your resource type. This implementation must satisfy the *AccessPolicy concept*.

Consider an array where you can specify, which element you want to access. Two accesses have to be executed sequential, if they use the same index.

```
struct MyArrayAccess {
    int index;

    static bool is_serial(MyArrayAccess a, MyArrayAccess b) {
        return (a.index == b.index);
    }
    static bool is_superset_of(MyArrayAccess a, MyArrayAccesss b) {
        return (a.index == b.index);
    }
}

struct MyArray : rmngr::Resource<MyArrayAccess> {
    std::array<...> data;

    rmngr::ResourceAccess access_index( int index ) const {
        return this->make_access( MyArrayAccess{ index } );
    }
}
```

9.1.1 Combining Access Types

TODO

9.2 Extending Task Properties

TODO

CHAPTER

TEN

ASYNCHRONOUS OPERATIONS

e.g. Compute Kernels, MPI calls need to be represented as tasks, but their execution only consists of starting an asynchronous process. The task however should not finish until the asynchronous operation is done, but not through blocking inside the task. So we need to delay the removal of the task from the graph. This is done with *events*, which can be registered inside a task and then can be triggered by some polling loop.

10.1 Creating Events

`Manager::create_event()` creates an event object, on which the current task now depends. That means it will not be removed from the graph before the event is reached, even if the task itself is done executing. The removal of the task from the graph can then be triggerd with `Manager::reach_event( EventID )`. If there are multiple events, the task will disappear when all events are reached.

See examples/8_event.cpp

10.2 Polling

Instead of blocking, a worker thread can be configured to use a polling function when no tasks are available for this thread.

CHAPTER

ELEVEN

WRITING CUSTOM SCHEDULERS

TODO

CHAPTER

TWELVE

COMPONENTS

12.1 Resources & Dependency-Description

12.1.1 Resource

group **AccessPolicy** **Description**

An implementation of the concept AccessPolicy creates a new resource-type (*Resource*<AccessPolicy>) and should define the possible access modes / configurations for this resource-type (e.g. read/write)

Required public member functions

- `static bool is_serial(AccessPolicy, AccessPolicy)` check if the two accesses have to be **in order**. (e.g. two reads return false, an occuring write always true)
- `static bool is_superset(AccessPolicy a, AccessPolicy b)` check if access `a` is a superset of access `b` (e.g. accessing [0,3] is a superset of accessing [1,2])

template<typename **AccessPolicy** = DefaultAccessPolicy>
class Resource : **public** redGrapes::ResourceBase

Represents a concrete resource. Copied objects represent the same resource.

Template Parameters

- `AccessPolicy`: Defines the access-modes (e.g. read/write) that are possible with this resource. Required to implement the concept *AccessPolicy*

Subclassed by redGrapes::SharedResourceObject< T, AccessPolicy >

Public Functions

ResourceAccess **make_access**(AccessPolicy *pol*) **const**

Create an *ResourceAccess*, which represents an concrete access configuration associated with this resource.

Return *ResourceAccess* on this resource

Parameters

- `pol`: AccessPolicy object, containing all access information

12.1.2 ResourceAccess

class ResourceAccess

Public Functions

bool **is_same_resource**(*ResourceAccess* **const** &*a*) **const**
Check if the associated resource is the same

Return true if `a` is associated with the same resource as `this`

Parameters

- `a`: another *ResourceAccess*

12.1.3 ResourceUser

class ResourceUser
Subclassed by redGrapes::ResourceProperty

12.2 Tasks

template<typename **NullaryCallable**>
class DelayedFunctor

12.3 Scheduler

12.3.1 Precedence Graph

template<typename **T**, typename **EnqueuePolicy**, template<class> typename **Graph** = DefaultGraph>
class QueuedPrecedenceGraph : **public** redGrapes::PrecedenceGraph<*T*, *Graph*>
Precedence-graph generated from a queue using an enqueue-policy

Public Types

template<>
using VertexID = **typename** PrecedenceGraph::VertexID

Public Functions

QueuedPrecedenceGraph()

QueuedPrecedenceGraph(std::weak_ptr<RecursiveGraph<T, Graph>> *parent_graph*, VertexID *parent_vertex*)

auto **push**(T *a*)

auto **update_vertex**(VertexID *a*)

void **finish**(VertexID *vertex*)

12.3.2 Recursive Graph

template<typename **T**, template<class> typename **T_Graph** = DefaultGraph>
class RecursiveGraph

Boost-Graph adaptor storing a tree of subgraphs which refine a node. Every vertex of a refinement has an edge to the refinements root node.

Public Types

template<>
using Graph = T_Graph<std::pair<T, std::shared_ptr<RecursiveGraph>>>

template<>
using VertexID = **typename** boost::graph_traits<*Graph*>::vertex_descriptor

Public Functions

virtual ~RecursiveGraph()

auto **shared_lock**()

auto **unique_lock**()

Graph &**graph**(void)
get graph object

bool **empty**()

void **add_subgraph**(VertexID *vertex*, std::shared_ptr<RecursiveGraph> *subgraph*)

void **remove_vertex**(VertexID *vertex*)

std::pair<*Iterator*, *Iterator*> **vertices**()

template<typename **Result**>
void **collect_vertices**(std::vector<*Result*> &*collection*, std::function<std::experimental::optional<*Result*>(T **const**&)> **const** &*filter_map*, size_t *limit* = std::numeric_limits<size_t>::max()

void **write_dot**(std::ostream &*out*, std::function<unsigned int(T **const**&)> **const** &*id*, std::function<std::string(T **const**&)> **const** &*label*, std::function<std::string(T **const**&)> **const** &*color*

void **write_refinement_dot**(std::ostream &*out*, std::function<unsigned int(T **const**&)> **const** &*id*, std::function<std::string(T **const**&)> **const** &*label*, std::function<std::string(T **const**&)> **const** &*color*

struct Iterator

Public Functions

template<>
T **const** &**operator***()

template<>
bool **operator==**(Iterator **const** &*other*)

template<>
bool **operator!=**(Iterator **const** &*other*)

template<>
void **operator++**()

Public Members

template<>
RecursiveGraph &**r**

template<>
boost::graph_traits<*Graph*>::vertex_iterator **g_it**

template<>
std::unique_ptr<std::pair<Iterator, Iterator>> **sub**

template<>
std::shared_lock<std::shared_mutex> **lock**

12.3.3 Building the Graph with Policies

12.3.4 Thread Pool

group **Thread** **Required public**

- constructor: `Thread(Callable, Args&&...)` spawns a new thread which executes the callable with the given arguments
- `void join()`

template<typename **Scheduler**, typename **Thread** = std::thread>
class ThreadDispatcher

Manages a thread pool. Worker-threads request jobs from scheduler and execute them, until the *ThreadDispatcher* gets destroyed and all workers finished.

Template Parameters

- `JobSelector`: must implement `bool empty()` and `void consume_job()`
- `Thread`: must satisfy *Thread*

INDEX

R