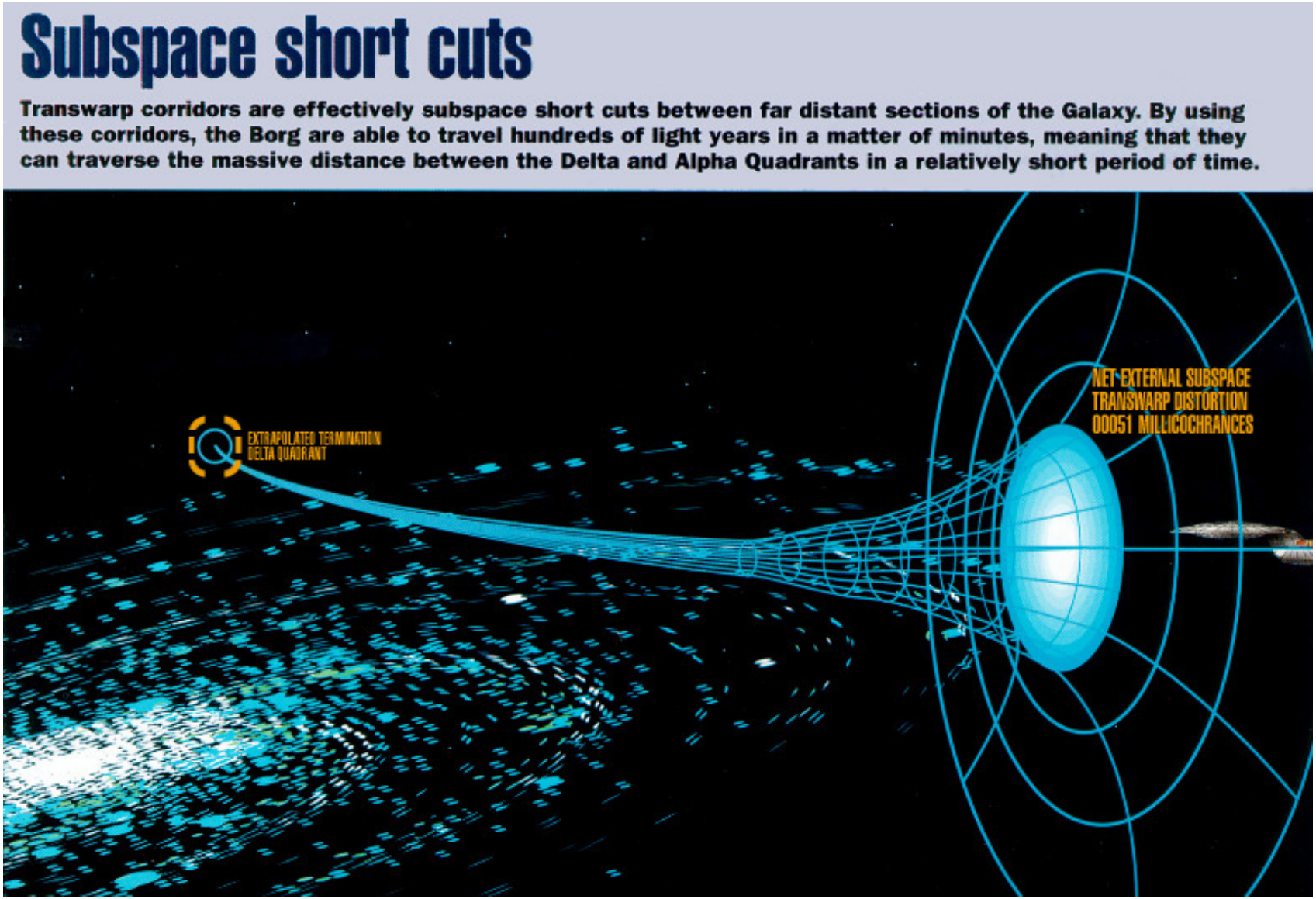
transwarp

A header-only C++ library for task concurrency

Christian Blume

May 17, 2017

transwarp drive



transwarp library

A header-only C++ library for task concurrency

`https://github.com/bloomen/transwarp`

- Written in C++11
- Currently in version 0.1.0
- Tested using GCC and Clang
- MIT license

Let's chat if you want to contribute!

The problem

- we have a bunch of operations that depend on each other
- those operations are run repeatetly with varying input
- some of those operations are independent

Now, we want to:

- understand the dependencies between operations
- run the independent operations in parallel

Typical solution

- launch a thread per operation that should run concurrently
- somehow notify a given operation that it is ready to run
- collect the results and move on to the next operation

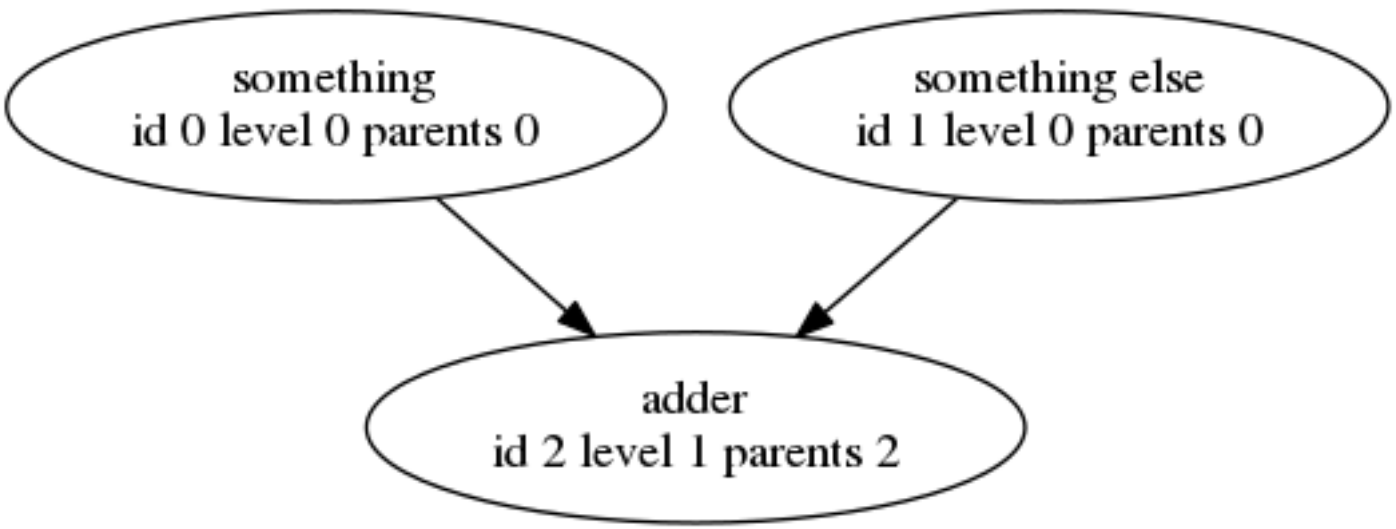
Drawbacks

- explicit thread handling
- a lot of independent operations may lead to thread contention
- error prone notification mechanism
- dependencies are hard to reason about

We can do better!

Solving it with transwarp

- directed acyclic graph of tasks
- a task can have any number of parent tasks
- graph must contain a single final task without children
- underlying thread pool to actually run the tasks
- a task operation may run anywhere (CPU, GPU, etc)

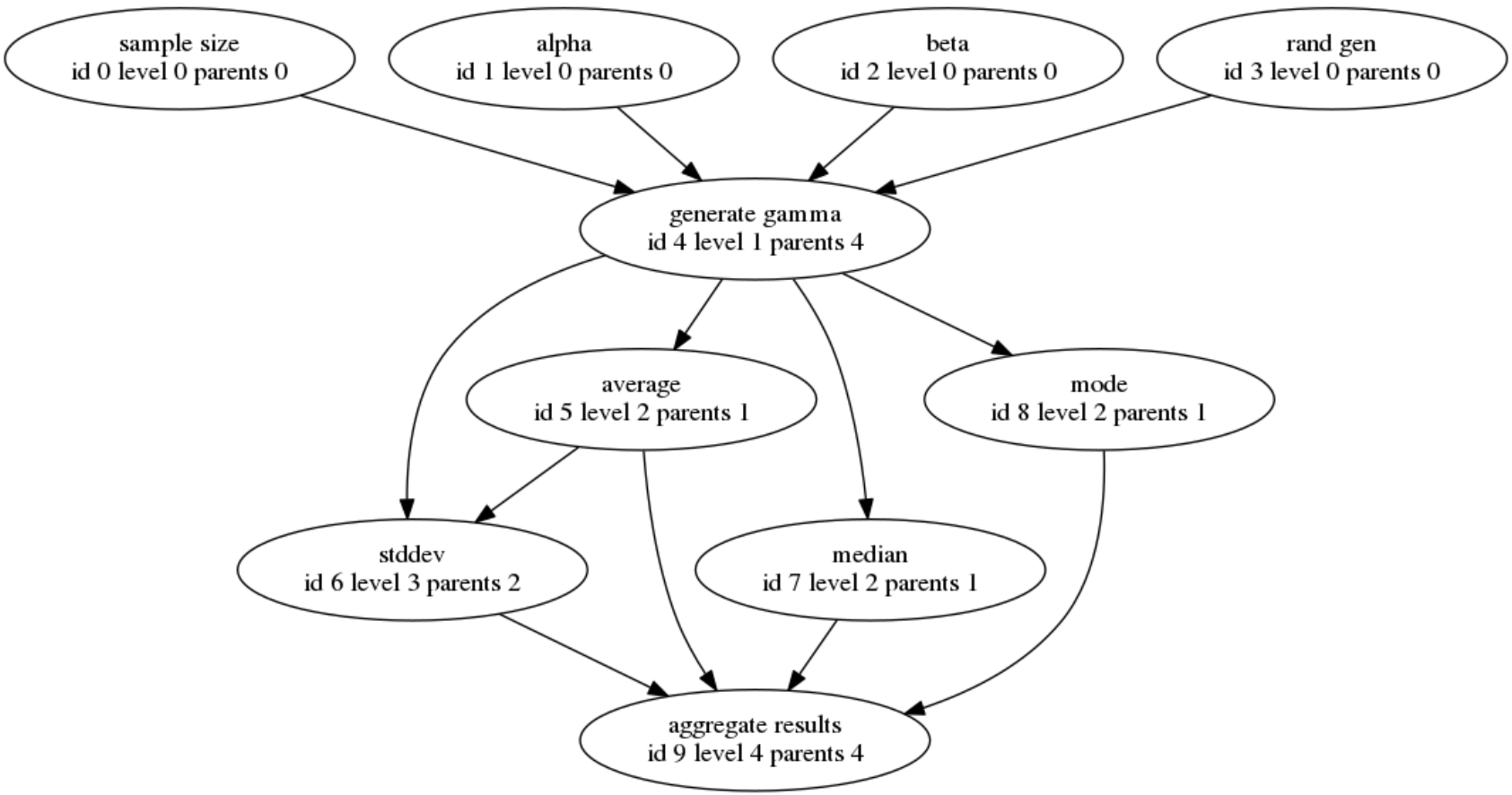


Simple example

```
 1  double add_em_up(double x, int y) {
 2    return x + y;
 3  }
 4
 5  int main() {
 6    double value1 = 13.3;
 7    int value2 = 42;
 8    auto something = [&value1] { return value1; };
 9    auto something_else = [&value2] { return value2; };
10
11    auto t1 = make_task("something", something);
12    auto t2 = make_task("something else", something_else);
13    auto t3 = make_final_task("adder", parallel{4},
        add_em_up, t1, t2);
14
15    t3->schedule();
16    std::cout << t3->get_future().get(); // 55.3
17  }
```

A slightly more evolved example

- generate data from a gamma distribution
- compute statistical measures such as average and median
- check out the code on github



The task class

```
 1  template<typename Functor, typename... Tasks>
 2  class task {
 3  public:
 4
 5    task(std::string name, Functor functor,
 6         std::shared_ptr<Tasks>... parents);
 7
 8    std::shared_future<result_type> get_future() const;
 9
10    const node& get_node() const;
11
12    template<typename Pre, typename Post>
13    void visit(Pre& pre_visitor, Post& post_visitor);
14
15    void unvisit();
16  };
```

Visiting tasks

```
 1  template<typename Pre, typename Post>
 2  void task::visit(Pre& pre, Post& post) {
 3    if (!visited_) {
 4      pre(*this);
 5      detail::visit<Pre, Post> visit_task(pre, post);
 6      detail::call_with_each(visit_task, parents_);
 7      post(*this);
 8      visited_ = true;
 9    }
10  }
11
12  void task::unvisit() {
13    if (visited_) {
14      visited_ = false;
15      detail::call_with_each(detail::unvisit(), parents_);
16    }
17  }
```

Supporting classes

```
 1  struct node {
 2      std::size_t id;
 3      std::size_t level;
 4      std::string name;
 5      std::vector<const node*> parents;
 6  };
 7
 8
 9
10  struct edge {
11      const transwarp::node* child;
12      const transwarp::node* parent;
13  };
```

The final task class

```
 1  template<typename Functor, typename... Tasks>
 2  class final_task : public task<Functor, Tasks...> {
 3  public:
 4
 5    final_task(std::string name, sequenced seq, Functor f,
 6               std::shared_ptr<Tasks>... parents);
 7
 8    final_task(std::string name, parallel par, Functor f,
 9               std::shared_ptr<Tasks>... parents);
10
11    void schedule();
12
13    void set_pause(bool enabled);
14
15    void set_cancel(bool enabled);
16
17    const std::vector<edge>& get_graph() const;
18  };
19
20  std::string make_dot(const std::vector<edge>& graph);
```

Execution policies

```
 1  class sequenced {};
 2
 3
 4
 5  class parallel {
 6  public:
 7      parallel(std::size_t n_threads);
 8
 9      std::size_t n_threads() const;
10  };
```

Scheduling tasks

```
 1  void final_task::schedule() {
 2    if (!*canceled_) {
 3      prepare_callbacks();
 4      if (pool_) { // parallel execution
 5        for (const auto& callback : callbacks_) {
 6          pool_->push(callback);
 7        }
 8      } else { // sequential execution
 9        for (const auto& callback : callbacks_) {
10          while (paused_) {};
11          callback();
12        }
13      }
14    }
15  }
```

Creating packagers

```
 1  detail::wrapped_packager task::make_packager() {
 2    auto packager = [this] {
 3      auto futures = detail::get_futures(parents_);
 4      auto pack_task = std::make_shared<
 5        std::packaged_task<result_type()>>(std::bind(
 6        &task::evaluate, std::ref(*this), futures));
 7      future_ = pack_task->get_future();
 8      return [pack_task] { (*pack_task)(); };
 9    };
10    return {packager, &node_};
11  }
12
13  static
14  result_type evaluate(task& t, std::tuple<...> futures) {
15    if (*t.canceled_)
16      throw task_canceled(t.get_node());
17    return detail::call_with_futures<result_type>(
18      t.functor_, futures);
19  }
```

Scheduling tasks (recap)

```
 1  void final_task::schedule() {
 2    if (!*canceled_) {
 3      prepare_callbacks();
 4      if (pool_) { // parallel execution
 5        for (const auto& callback : callbacks_) {
 6          pool_->push(callback);
 7        }
 8      } else { // sequential execution
 9        for (const auto& callback : callbacks_) {
10          while (paused_) {};
11          callback();
12        }
13      }
14    }
15  }
```

The thread pool

```
 1  std::queue<std::function<void()>> functors_;
 2  std::condition_variable cond_var_;
 3  std::mutex mutex_;
 4
 5
 6  void thread_pool::push(std::function<void()> func) {
 7    {
 8      std::lock_guard<std::mutex> lock(mutex_);
 9      functors_.push(func);
10    }
11    cond_var_.notify_one();
12  }
```

The thread pool (cont.)

```
 1  void thread_pool::worker() {
 2    for (;;) {
 3      std::function<void()> func;
 4      {
 5        std::unique_lock<std::mutex> lock(mutex_);
 6        cond_var_.wait(lock, [this]{
 7          return !paused_ && (done_ || !functors_.empty());
 8        });
 9        if (done_ && functors_.empty())
10          break;
11        func = functors_.front();
12        functors_.pop();
13      }
14      func();
15    }
16  }
```

Summary

Use transwarp when

- you have dependent operations
- the operations are executed repeatetly
- you consider to run certain operations in parallel

What you get is

- parallelism by design
- a task graph that is easy to reason about
- focus on getting stuff done

Outlook

- consider notifications for when futures have been replaced
- test with C++17 enabled
- test with Visual Studio
- add some more real-life examples
- lift the library to version 1.0
- ...

Thank You