Integration of STL concurrency primitives into CLAD

Organization: CERN-HSF

Project size: large (350 hrs.)

Name: Mozil Petro

CLAD is a compiler plugin that enables automatic differentiation of C++ functions. It supports differentiation of user-defined structs, however it struggles with complex structures/classes. It does not account for functions with side effects either. Though this could be aided with the use of custom derivatives, it would still be useful to have built-in support of some of those complex structures and functions with side effects. One of the most useful ones is std::thread, which is the thread class from STL. Utilizing parallelism is very important for modern numerical methods and, as such, having support of std::thread and other STL concurrency primitives would be a useful feature for CLAD.

Problem Statement

Clad recursively iterates over the syntax tree to check whether a given statement should be differentiated. Each function that is called from inside of a differentiated function should be differentiated as well, and so should any object method. The main issue for clad is that std::thread is an object, and thus as a type that should be differentiated However, std::thread shouldn't be differentiated, the function inside of it should. Some of STL's concurrency primitives face the same problem - the methods in them should not be differentiated, and only the location of where they were called should be preserved.

Expected Project Results

This project shall add support for std::thread, std::mutex and std::atomic at least. If time allows, std::condition_variable, std::jthread, std::barrier and std::atomic_thread_fence should also be supported.

Currently, a function as such

```
void worker_fn(double val, double &out) {
      out = val * val;
}

double result(double const *vals, size_t n) {
  std::vector<double> results(n);
  std::vector<std::thread> workers;

  for (size_t i = 0; i < n; i++)
      workers.emplace_back(worker_fn, vals[i], std::ref(results[i]));

  double res = 0.0L;
  for (const auto val : results)
      res += val;

  return res;
}

int main() {
      auto fn = clad::differentiate(result, "vals[0]");

      std::vector<double> vals{1.0, 2.0, 3.0};
      std::vector<double> res;
      fn.execute(vals.data(), vals.size());
}
```

Would not be differentiated. Clad fails in the function which processes function calls..

Support for a type involves adding extensive tests for the object and, if it's required, adding functionality to CLAD. std::mutex and std::barrier should not require any additional functionality beyond the optimization of their derivative, so as to minimize the amount of syscalls used on the derivative of the object.

General Methodology

The overall implementation of each type would involve such steps:

1. Implement a basic code example for the use of a primitive, launch it and check whether any problems occur.
2. Integrate the code example into the project as a test, and add more tests if required.
3. If any problem with the code example did occur, add functionality to cover the case of the code example and other tests.

Expected timeline

Phase 1: std::thread support (2-3 weeks)

During this phase, support for std::thread should be added.
Firstly, visitors should be modified to recognize std::thread constructor calls and handle call expressions of functions / functors in the constructor. Then a set of tests for std::thread should be integrated into the codebase

Phase 2: std::mutex and std::atomic (2-3 weeks)

These types would be more straightforward to implement. They wouldn't require too much effort, however they would require custom derivatives. Similarly, after implementation more tests should be created

Phase 3: std::condition_variable and std::unique_lock (3 - 4 weeks)

These objects would require more attention regarding custom derivatives as well as, very likely, modifying the visitors to account for the condition_variable dataflow.

Phase 4: std::jthread and other primitives (if time allows)

These would be implemented if time allows.

By-Week deliverables

Week	Dates	Activity	
Community bonding period	May	Implement support for template specializations and other features, familiarize myself with the codebase	Merge requests for some issues, mainly ones related to template specialization
Week 1	Jun 1-7	Support for	Create tests for

		std::thread	std::thread use in clad. Start adding support for std::thread
Week 2	Jun 8-14	Support for std::thread	Add support for std::thread
Week 3	Jun 15-21	Support for std::thread	Create merge request for std::thread
Week 4	Jun 21-28	Atomics and mutexes	Create tests for atomics and mutexes. Start adding support for them
Week 5	Jun 29 - Jul 5	Atomics and mutexes	Implement support for atomics and mutexes
Week 6	Jul 6 - 11	Atomics and mutexes	Create merge request for atomics and mutexes
Week 7	Jul 12 - 18	Condition variables and locks	Create code examples for condition variables and locks
Weeks 8-10	Jul 19 - Aug 10	Condition variables and locks	Add support for condition variables and locks
Week 11	Aug 11 - 17	Condition variables and locks	Create merge request for condition variables and threads
Week 12	Aug 18 - 24	Support for std::jthread	Add support for std::jthread if time allows
Week 13	Aug 25 - 31	Cleaning up code, buffer week	If there were any bugs found or some merge requests did not yet merge, the code should be cleaned up.

Availability

My semester ends in the middle of may, i will be able to work 35-40 hours a week during the end of may through summer.

Personal Details

I am a third-year student of computer science at Ukrainian Catholic University. I have experience with parsing and compilers.

Name: Petro Mozil
University: Ukrainian Catholic University (https://ucu.edu.ua/en/)
Email: mozil.petryk@gmail.com
Telephone: +38 095 867 21 48
Instant Messaging: Discord (pmozil) Telegram (@pm0zil), Matrix (@pm0zil:matrix.org), libera.chat
(pmozil)
Country of Residence: Ukraine
Timezone: EET (UTC+2)