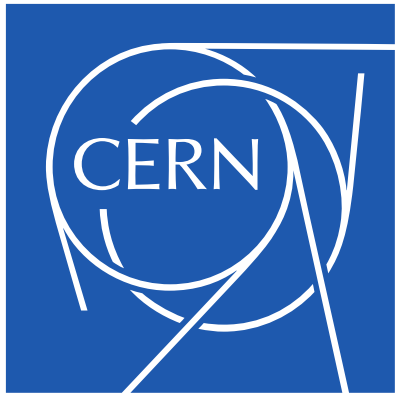
Enable Automatic Differentiation of OpenMP Programs with Clad



Jiayang Li

Mentors: Vassil Vassilev, Martin Vassilev

Background

- What is Automatic Differentiation (AD)?

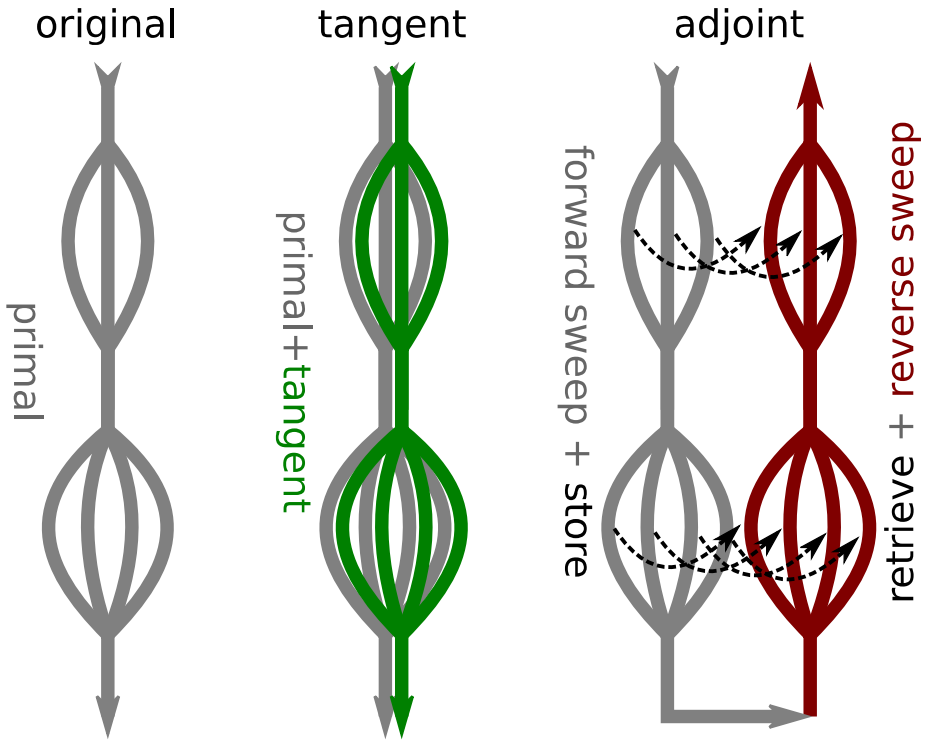
  Exact derivatives by transforming programs; no finite-difference error; supports forward and reverse.

- What is OpenMP?

  Pragmas for shared-memory parallelism (parallel, for, reduction, etc.).

- What is Clad?

  Source-transformation AD library for C++ implemented as a plugin for Clang



Case Study

Start with a representative example: compute the gradient of a simple parallel loop.

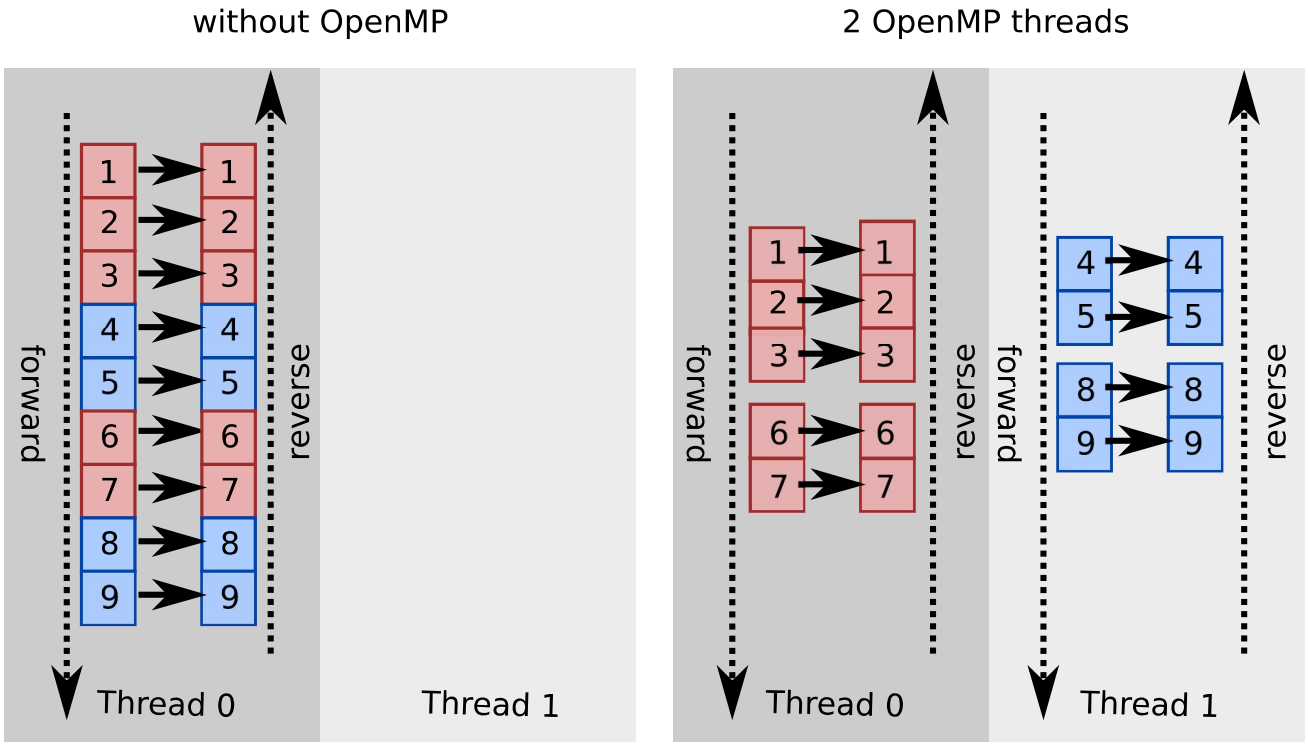
```
1  double parallel_sum_of_squares(const double* x, int n) {
2      double total = 0.;
3      #pragma omp parallel for reduction(+: total)
4      for (int i = 0; i < n; i++) {
5          total += x[i] * x[i];
6      }
7      return total;
8  }
```

Case Study

What might the reverse-mode gradient look like conceptually?

```
 1 void parallel_sum_of_squares_grad(const double *x, int n, double *_d_x, int *_d_n)
 2     double _d_total = 0.;
 3     double total = 0.;
 4     #pragma omp parallel for reduction(+: total)
 5     for (int i = 0; i < n; i++) {
 6         total += x[i] * x[i];
 7     }
 8     _d_total += 1;
 9     #pragma omp parallel private(total) firstprivate(_d_total)
10     for (int i = n - 1; i >= 0; i -= 1) {
11         double _r_d0 = _d_total;
12         _d_x[i] += _r_d0 * x[i];
13         _d_x[i] += x[i] * _r_d0;
14     }
15 }
```

Technical Challenges



- Clang AST specifics for OpenMP (building AST nodes, capture variables, clause handling)
- Variable scoping across OpenMP regions and clauses
- Thread-safe storage for tapes/intermediates
- Deterministic schedule reversal for the reverse pass

Implementation Overview

- The theory is based on the paper from Tapenade, which provides detailed proofs and demonstrates the implementation of OpenMP automatic differentiation on the Fortran platform using Tapenade.
- Since OpenMP constructs are transformed into AST nodes during the Clang AST phase, we can override the corresponding OpenMP-related AST nodes Visit methods—just as we do with other Visit methods—to build new function ASTs.

Forward Mode (#1491)

- Since the execution order of the forward-mode derivatives is consistent with that of the original code, the scopes of the differentiated variables can be directly inherited from the original variables, and the parallel structure of the entire program can also be reused.
- For the previous example, the actually generated code looks like this:

```
 1  // auto d_fn_arr = clad::differentiate(parallel_sum_of_squares, "x[1]");
 2  // d_fn_arr.dump();
 3  double parallel_sum_of_squares_darg0_1(const double *x, int n) {
 4      int _d_n = 0;
 5      double _d_total = 0.;
 6      double total = 0.;
 7      #pragma omp parallel for reduction(+: _d_total,total)
 8          for (int i = 0; i < n; i++) {
 9              _d_total += (i == 1) * x[i] + x[i] * (i == 1);
10              total += x[i] * x[i];
11          }
12      return _d_total;
13  }
```

Reverse Mode — Clang AST

- The previous VisitForStmt method was too invasive for `omp for`; we implement a dedicated `DifferentiateCanonicalLoop` for OpenMP loops.

```
1  // auto fn_grad = clad::gradient(sum_of_squares);
2  // fn_grad.dump();
3  void sum_of_squares_grad(const double *x, int n, double *_d_x, int *_d_n) {
4      int _d_i = 0;
5      int i = 0;
6      double _d_total = 0.;
7      double total = 0.;
8      unsigned long _t0 = 0;
9      for (i = 0; i < n; i++) {
10         _t0++;
11         total += x[i] * x[i];
12     }
13     _d_total += 1;
14     for (; _t0; _t0--) {
15         i--;
16         {
17             double _r_d0 = _d_total;
18             _d_x[i] += _r_d0 * x[i];
19             _d_x[i] += x[i] * _r_d0;
20         }
21     }
22 }
```

Reverse Mode — Clang AST

- The previous VisitForStmt method was too invasive for `omp for`; we implement a dedicated `DifferentiateCanonicalLoop` for OpenMP loops.

```
1  // auto fn_grad = clad::gradient(sum_of_squares);
2  // fn_grad.dump();
3  void sum_of_squares_grad(const double *x, int n, double *_d_x, int *_d_n) {
4      int _d_i = 0;
5      int i = 0;
6      double _d_total = 0.;
7      double total = 0.;
8      unsigned long _t0 = 0;
9      for (i = 0; i < n; i++) {
10         _t0++;
11         total += x[i] * x[i];
12     }
13     _d_total += 1;
14     for (; _t0; _t0--) {
15         i--;
16         {
17             double _r_d0 = _d_total;
18             _d_x[i] += _r_d0 * x[i];
19             _d_x[i] += x[i] * _r_d0;
20         }
21     }
22 }
```

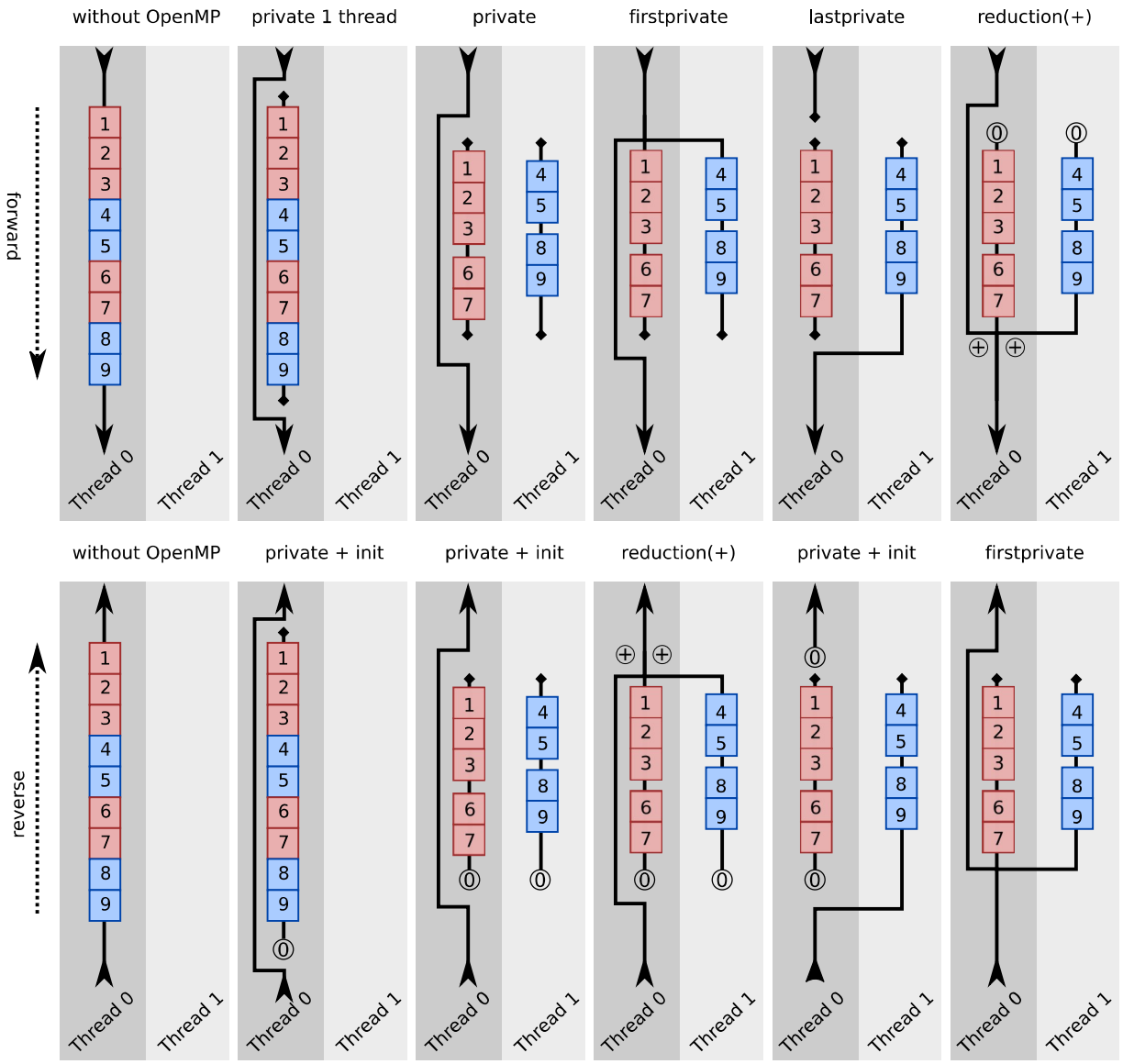
Reverse Mode — Clang AST

- It should be noted that, for Clang's OpenMP implementation, we need to capture the related variables when building `DeclRefExpr` (which corresponds to the `CaptureStmt`'s subnodes in the Clang AST).

  So we also need visit twice to generate two OpenMP regions and their capture lists: one for the forward pass, one for the reverse pass.

```
 1  CLAD_COMPAT_CLANG19_SemaOpenMP(m_Sema).ActOnOpenMPRegionStart(OMPD_parallel, nullptr);
 2  StmtDiff BodyDiff;
 3  {
 4      Sema::CompoundScopeRAII CompoundScope(m_Sema);
 5      if (isOpenMPLoopDirective(D->getDirectiveKind())) {
 6          const auto* FS = cast<ForStmt>(CS);
 7          BodyDiff = DifferentiateCanonicalLoop(FS);
 8      } else {
 9          BodyDiff = Visit(CS);
10      }
11  }
12  Stmt* Forward = CLAD_COMPAT_CLANG19_SemaOpenMP(m_Sema)
13                      .ActOnOpenMPRegionEnd(BodyDiff.getStmt(), OrigClauses)
14                      .get();
15
16  CLAD_COMPAT_CLANG19_SemaOpenMP(m_Sema).ActOnOpenMPRegionStart(OMPD_parallel, nullptr);
17  // Visit twice, but use only the result of the first visit, for capture variables only.
18  {
19      Sema::CompoundScopeRAII CompoundScope(m_Sema);
20      Stmts temp;
21      m_Globals.swap(temp);
22      if (isOpenMPLoopDirective(D->getDirectiveKind())) {
23          const auto* FS = cast<ForStmt>(CS);
24          DifferentiateCanonicalLoop(FS);
25      } else {
```

Reverse Mode — Scope conversion



Reverse Mode — Intermediates and schedule replay

- Intermediates require tapes; the tape itself must become private to each thread; so we make these threadprivate.
- For data consistency, we must ensure that the parallel scheduling of the backward pass strictly reproduces the thread–task allocation of the forward pass. To achieve this, we reverse the schedule while preserving chunk boundaries by a small runtime helper:
  - Forward pass: each thread computes its static chunk from thread ID and executes normally.
  - Reverse pass: the same helper returns identical chunks, then we iterate the chunk in reverse order.

```
1  void clad::GetStaticSchedule(int lo, int hi, int stride, int* threadlo, int* threadhi)
```

Reverse Mode (#1641)

The resulting generated code:

```
1  // auto fn_grad = clad::gradient(parallel_sum_of_squares);
2  // fn_grad.dump();
3  void parallel_sum_of_squares_grad(const double *x, int n, double *_d_x, int *_d_n) {
4      double _d_total = 0.;
5      double total = 0.;
6      #pragma omp parallel reduction(+: total)
7          {
8              int _t_chunklo0 = 0;
9              int _t_chunkhi0 = 0;
10             clad::GetStaticSchedule(0, n - 1, 1, &_t_chunklo0, &_t_chunkhi0);
11             for (int i = _t_chunklo0; i <= _t_chunkhi0; i += 1) {
12                 total += x[i] * x[i];
13             }
14         }
15     _d_total += 1;
16     #pragma omp parallel private(total) firstprivate(_d_total)
17         {
18             int _t_chunklo1 = 0;
19             int _t_chunkhi1 = 0;
20             clad::GetStaticSchedule(0, n - 1, 1, &_t_chunklo1, &_t_chunkhi1);
```

Future Work

- Support dynamic schedules (record executed chunks at runtime and replay).
- Support more OpenMP clauses and directives (atomic, simd, etc.).

  We can even try to enable OpenMP offloading to target accelerators (e.g., GPUs).

Thank You!

Q & A