Expression templates for primal value taping in the reverse mode of AD

Max Sagebaum, Tim Albring and Nicolas R. Gauger

AG Scientific Computing
TU Kaiserslautern

AD2016 - 7th International Conference on Algorithmic Differentiation

Overview

- Motivation: Jacobi taping vs. primal value taping
- Expression templates for primal value taping
- Handling of constant and passive types
- Memory and time analysis

CoDiPack introduction

Active type structure:

```
struct RealReverse {
  double p; // primal value
  int i;    // index for the adjoint
}
```

Index management

- Linear index manager (e.g. RealReverse)
  - Statements and adjoint variables tightly coupled
  - *i = ++tape.globalStatmentIndex;*
  - Advantage: LHS indices can be computed (no storing of LHS index)
  - Drawback: Big adjoint vector
- Reuse index manager (e.g. RealReverseIndex)
  - No coupling between statements and adjoint variables
  - *tape.indexManager.assignIndex(i);*
  - Advantage: Small adjoint vector
  - Drawback: Additional 4 bytes per statement (index needs to be stored)

CoDiPack introduction

Tape implementation

- Simple (e.g. RealReverseUnchecked)
  - No bounds check performed
  - Memory needs to be preallocated
  - Faster evaluation
- Chunk (e.g. RealReverse)
  - Memory allocated on the fly
  - Slower due to bounds check

Motivation: Jacobi taping vs. primal value taping

Elemental operator

$$\phi : \mathbb{R}^n \to \mathbb{R}, n \in \mathbb{N}$$

Reverse update

$$\bar{x}_i \mathrel{+}= \frac{\partial \phi}{\partial x_i}(x_i)^T \bar{y}, \quad \forall i = 1...n$$

$$\bar{y} = 0$$

Motivation: Jacobi taping vs. primal value taping

Elemental operator

$$\phi : \mathbb{R}^n \to \mathbb{R}, n \in \mathbb{N}$$

Reverse update

$$\bar{x}_i \mathrel{+}= \frac{\partial \phi}{\partial x_i}(x_i)^T \bar{y}, \quad \forall i = 1...n$$

$$\bar{y} = 0$$

Jacobi taping:

- Store $\frac{\partial \phi}{\partial x_i}$, indices for x_i and n
- Bytes per statement: $n * 8 + n * 4 + 1 = 12 * n + 1$
- Assumption: $n < 256 \Rightarrow$ Each elemental operation less than 256 arguments

Motivation: Jacobi taping vs. primal value taping

Elemental operator

$$\phi : \mathbb{R}^n \to \mathbb{R}, n \in \mathbb{N}$$

Reverse update

$$\bar{x}_i \mathrel{+}= \frac{\partial \phi}{\partial x_i}(x_i)^T \bar{y}, \quad \forall i = 1...n$$

$$\bar{y} = 0$$

Jacobi taping:

- Store $\frac{\partial \phi}{\partial x_i}$, indices for x_i and n
- Bytes per statement: $n * 8 + n * 4 + 1 = 12 * n + 1$
- Assumption: $n < 256 \Rightarrow$ Each elemental operation less than 256 arguments

Primal value taping:

- Store y, indices for x_i and handle to function
- Bytes per statement: $8 + n * 4 + 8 = 4 * n + 16$

Expression templates for primal value taping

The equation

$$w = ((a + b) * (c - d))^2$$

Expression templates for primal value taping

The equation

$$w = ((a + b) * (c - d))^2$$

is represented by the structure

POW<MULT<ADD<ActiveReal, ActiveReal>, SUB<ActiveReal, ActiveReal>>>

Expression templates for primal value taping

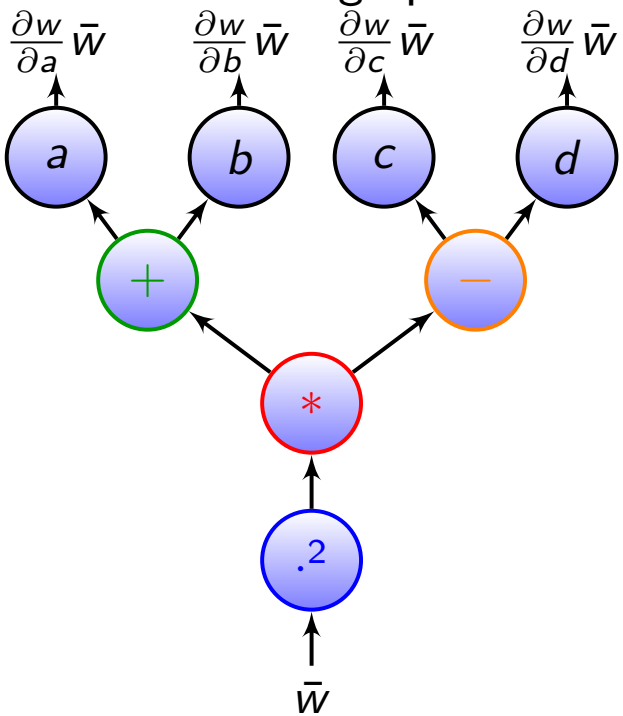
The equation

$$w = ((a + b) * (c - d))^2$$

is represented by the structure

POW<MULT<ADD<ActiveReal, ActiveReal>, SUB<ActiveReal, ActiveReal>>>

The reverse AD graph:



Expression templates for primal value taping

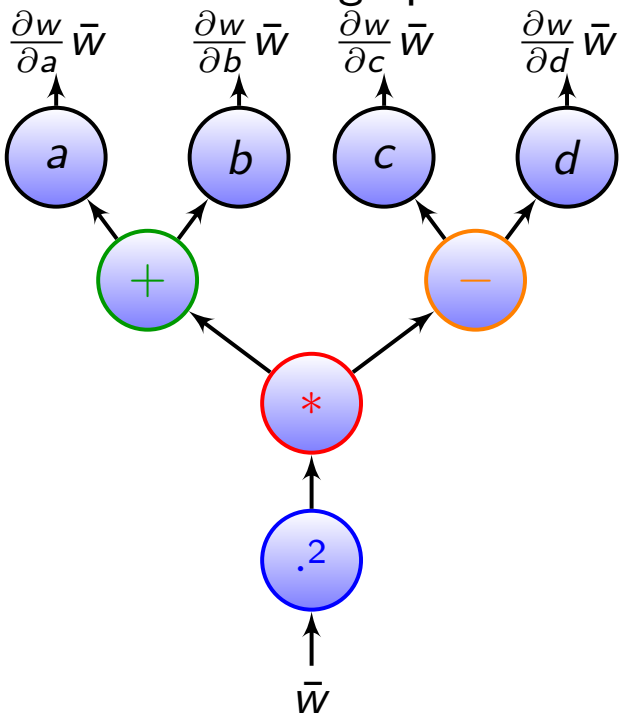
The equation

$$w = ((a + b) * (c - d))^2$$

is represented by the structure

POW<MULT<ADD<ActiveReal, ActiveReal>, SUB<ActiveReal, ActiveReal>>>

The reverse AD graph:



The required code:

```
add = a + b; sub = c - b; mul = add * sub;
w = pow(mul, 2.0);

w_b = tape.getAdjoint(w_i);
mul_b += 2 * mul * w_b;
sub_b += add * mul_b;
add_b += sub * mul_b;
  c_b += sub_b;
  d_b += -sub_b;
  a_b += add_b;
  b_b += add_b;
tape.updateAdjoint(a_b, a_i);
tape.updateAdjoint(b_b, b_i);
tape.updateAdjoint(c_b, c_i);
tape.updateAdjoint(d_b, d_i);
```

Expression templates for primal value taping

Which actions do we need?

Expression templates for primal value taping

Which actions do we need?

1. Gather all indices
2. Evaluate the adjoint
3. Store a function handle

? .. Probably many more

Expression templates for primal value taping - 1. Gather

- Add per value action to expression interface

```
template<typename A>
struct Expression {
  ...
  template<typename Data, typename Func>
  inline void valueAction(Data data, Func func) const;
}
```

Expression templates for primal value taping - 1. Gather

- Add per value action to expression interface

```
template<typename A>
struct Expression {
  ...
  template<typename Data, typename Func>
  inline void valueAction(Data data, Func func) const;
}
```

```
template<typename A, typnname B>
struct BinaryOperator : public Expression<BinaryOperator<A, B> > {
  template<typename Data, typename Func>
  inline void valueAction(Data data, Func func) const {
    a_.valueAction(data, func);
    b_.valueAction(data, func);
  }
}
```

Expression templates for primal value taping - 1. Gather

- Add per value action to expression interface

```
template<typename A>
struct Expression {
  ...
  template<typename Data, typename Func>
  inline void valueAction(Data data, Func func) const;
}
```

```
template<typename A, typname B>
struct BinaryOperator : public Expression<BinaryOperator<A, B> > {
  template<typename Data, typename Func>
  inline void valueAction(Data data, Func func) const {
    a_.valueAction(data, func);
    b_.valueAction(data, func);
  }
}
```

```
struct RealReverse: public Expression<RealReverse> {
  template<typename Data, typename Func>
  inline void valueAction(Data data, Func func) const {
    CALL_MEMBER_FN(globalTape, func)(data, this->p, this->i);
  }
}
```

Expression templates for primal value taping - 1. Gather

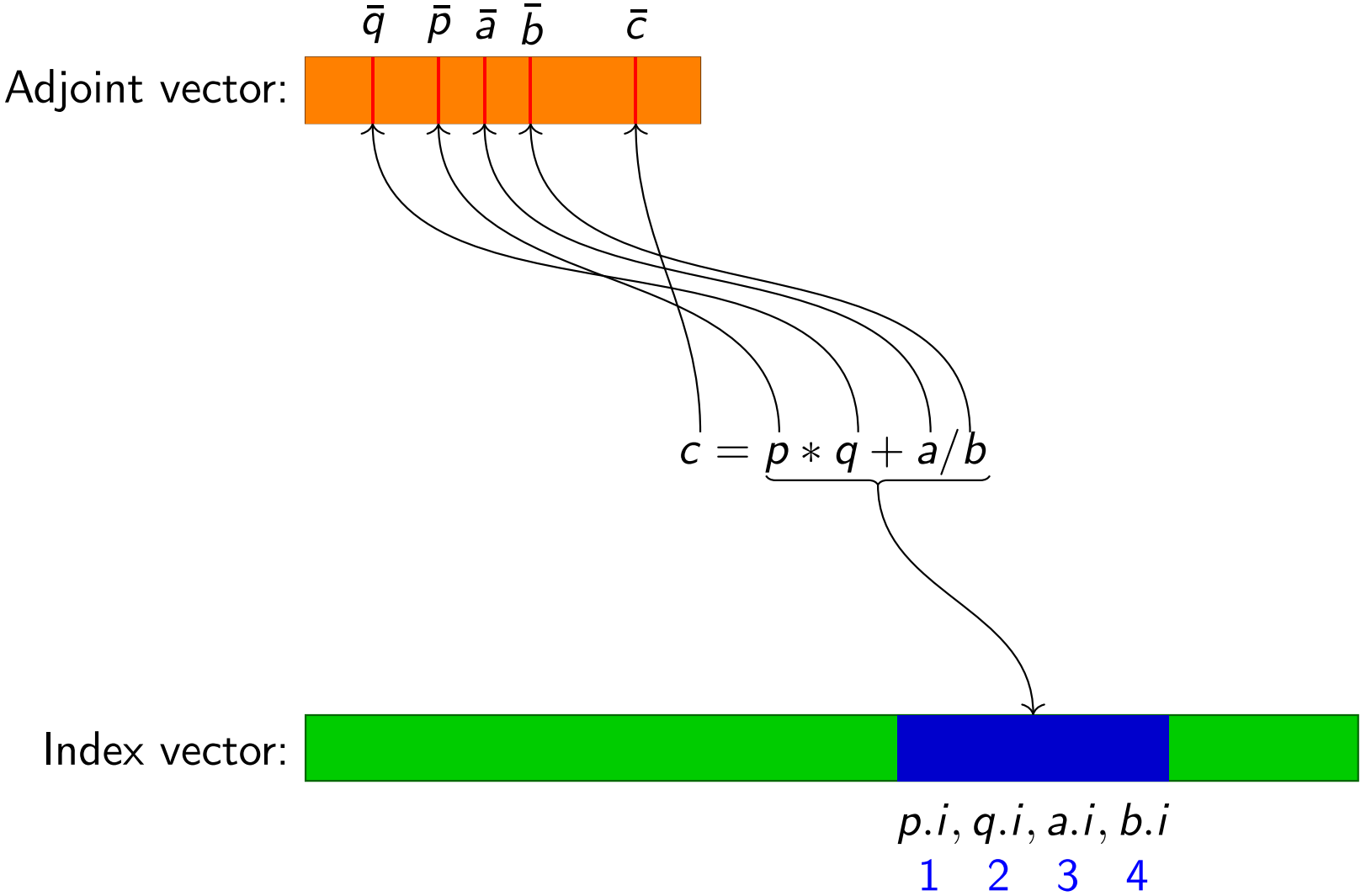
- Use the method in the tape

```
struct PrimalValueTape {
  ...
  template<typename Expr>
  inline void store(double& lhsValue, int& lhsIndex, const Expr& rhs) {
    indexVector.reserveItems(ExpressionTraits<Expr>::maxActiveVariables);
    rhs.valueAction(NULL, &PrimalValueTape::pushIndices);

    ...
  }

  inline void pushIndices(void* data, const double& value, const int& index) {
    indexVector.setDataAndMove(pushIndex);
  }
}
```

Expression templates for primal value taping - 2. Evaluate



Expression templates for primal value taping - 2. Evaluate

```
// BinaryOperator
template<size_t offset>
static inline double getValue(const int* indices, const Real* primalValues) {
  const double aPrimal = A::template getValue<offset>(indices, primalValues);
  const double bPrimal = B::template getValue
      <offset + ExpressionTraits<A>::maxActiveVariables>(indices, primalValues);
  return PRIMAL_CALL(aPrimal, bPrimal);
}
```

Expression templates for primal value taping - 2. Evaluate

```
// BinaryOperator
template<size_t offset>
static inline double getValue(const int* indices, const Real* primalValues) {
  const double aPrimal = A::template getValue<offset>(indices, primalValues);
  const double bPrimal = B::template getValue
      <offset + ExpressionTraits<A>::maxActiveVariables>(indices, primalValues);
  return PRIMAL_CALL(aPrimal, bPrimal);
}
```

```
// RealReverse
template<size_t offset>
static inline double getValue(const int* indices, const Real* primalValues) {
  return primalValues[indices[offset]];
}
```

Expression templates for primal value taping - 2. Evaluate

```
//BinaryOperator
template<size_t offset>
static inline void evalAdjoint(const double& seed, const int* indices,
                               const Real* primalValues, Real* adjointValues) {
  const double aPrimal = A::template getValue<offset>(indices, primalValues);
  const double bPrimal = B::template getValue
      <offset + ExpressionTraits<A>::maxActiveVariables>(indices, primalValues);
  const double resPrimal = PRIMAL_CALL(aPrimal, bPrimal);

  const double aJac = GRADIENT_FUNC_A(aPrimal, bPrimal, resPrimal) * seed;
  const double bJac = GRADIENT_FUNC_B(aPrimal, bPrimal, resPrimal) * seed;
  A::template evalAdjoint<offset>(aJac, indices, primalValues, adjointValues);
  B::template evalAdjoint<offset + ExpressionTraits<A>::maxActiveVariables>
                          (bJac, indices, primalValues, adjointValues);
}
```

Expression templates for primal value taping - 2. Evaluate

```
//BinaryOperator
template<size_t offset>
static inline void evalAdjoint(const double& seed, const int* indices,
                               const Real* primalValues, Real* adjointValues) {
  const double aPrimal = A::template getValue<offset>(indices, primalValues);
  const double bPrimal = B::template getValue
      <offset + ExpressionTraits<A>::maxActiveVariables>(indices, primalValues);
  const double resPrimal = PRIMAL_CALL(aPrimal, bPrimal);

  const double aJac = GRADIENT_FUNC_A(aPrimal, bPrimal, resPrimal) * seed;
  const double bJac = GRADIENT_FUNC_B(aPrimal, bPrimal, resPrimal) * seed;
  A::template evalAdjoint<offset>(aJac, indices, primalValues, adjointValues);
  B::template evalAdjoint<offset + ExpressionTraits<A>::maxActiveVariables>
                          (bJac, indices, primalValues, adjointValues);
}
```

```
//RealReverse
template<size_t offset>
static inline void evalAdjoint(const double& seed, const int* indices,
                               const Real* primalValues, Real* adjointValues) {
  adjointValues[indices[offset]] += seed;
}
```

Expression templates for primal value taping - 3. Store

- Required information:
  - Function pointer (8 byte)
  - Number of arguments (1 byte)
    - ⇒ Static information does not need to be stored in the tape

⇒ Use static handles and store references to them:

```
// the handle
struct ExpressionHandle {
  const StatementFuncPointer adjointFunc;
  const size_t maxActiveVariables;
};

// the store
template<typename Expr>
struct ExpressionHandleStore {
    static const ExpressionHandle handle;
};

// the instantiation
template<typename Expr>
const ExpressionHandle ExpressionHandleStore<Expr>::
    handle(Expr::template evalAdjoint<0>, ExpressionTraits<Expr>::maxActiveVariables);
```

Expression templates for primal value taping

Complete store function:

```
      // PrimalValueTape
template<typename Expr>
inline void store(double& lhsValue, int& lhsIndex, const Expr& rhs) {
  indexVector.reserveItems(ExpressionTraits<Expr>::maxActiveVariables);
  rhs.valueAction(NULL, &PrimalValueTape::pushIndices);

  stmtVector.reserveItems(1);
  stmtVector.setDataAndMove(&ExpressionHandleStore<Rhs>::handle);

  lhsIndex = ++this->globalStatementIndex;
  lhsValue = rhs.getValue();

  checkPrimalsSize();
  this->primals[lhsIndex] = lhsValue;
}
```

Expression templates for primal value taping

How to evaluate in the reverse sweep:

```
// PrimalValueTape
inline void evaluateStack(const size_t& startAdjPos, const size_t& endAdjPos,
                          size_t& stmtPos, Handle* &statements,
                          size_t& indexPos, IndexType* &indices) {
  size_t adjPos = startAdjPos;

  while(adjPos > endAdjPos) {
    const GradientValue& adj = adjoints[adjPos];
    --adjPos;
    --stmtPos;

    indexPos -= statements[stmtPos].maxActiveVariables;
    statements[stmtPos].adjointFunc(adj, &indices[indexPos], primalVector, adjoints);
  }
}
```

Expression templates for primal value taping

Summary expression templates for primal value taping:

- Use general actions to store the indices
- Use static handles to store the static data for each statement
  - Function pointer
  - Number of variables
- Optimized implementation w.r.t. the compiler:
  - Offset into arrays as template parameter
  - Inlining of recursive calls

Expression templates for primal value taping

Summary expression templates for primal value taping:

- Use general actions to store the indices
- Use static handles to store the static data for each statement
  - Function pointer
  - Number of variables
- Optimized implementation w.r.t. the compiler:
  - Offset into arrays as template parameter
  - Inlining of recursive calls

What is missing?

Expression templates for primal value taping

Summary expression templates for primal value taping:

- Use general actions to store the indices
- Use static handles to store the static data for each statement
  - Function pointer
  - Number of variables
- Optimized implementation w.r.t. the compiler:
  - Offset into arrays as template parameter
  - Inlining of recursive calls

What is missing?

- Constant values
- Passive values

Constant value handling

Constant value example:

$$w = \sin(2.0 * a)$$

Required changes:

- Add push passive action

```
inline void pushPassive(void*, const double& value) {
  constantValueVector.setDataAndMove(value);
}
```

- Add constant value count to handle

```
constantPos -= exprHandle->maxConstantVariables;
```

- Add constant offset to evaluation function

```
template<size_t offset, size_t constantOffset>
static inline void evalAdjoint(
  const double& seed, const int* indices, const double* constantValues,
  const double* primalValues, double* adjointValues);
```

Passive value handling

Passive value example:

```
RealReverse two = 2.0; // has index 0
RealReverse three = 3.0; // has idnex 0
RealReverse w = sin(two * a) / three; // which value should be stored in index 0
```

Possible approaches:

Passive value handling

Passive value example:

```
RealReverse two = 2.0; // has index 0
RealReverse three = 3.0; // has idnex 0
RealReverse w = sin(two * a) / three; // which value should be stored in index 0
```

Possible approaches:

1. Make variable active in statement
   - Each use creates an additional 24 bytes

Passive value handling

Passive value example:

```
RealReverse two = 2.0; // has index 0
RealReverse three = 3.0; // has idnex 0
RealReverse w = sin(two * a) / three; // which value should be stored in index 0
```

Possible approaches:

1. Make variable active in statement
   - Each use creates an additional 24 bytes
2. Activate variable on first use
   - Only possible with reuse index manager
   - First use creates an additional 24 bytes
   - Increases *active* section of program
     $\Rightarrow$ more statements recorded

Passive value handling

Passive value example:

```
RealReverse two = 2.0; // has index 0
RealReverse three = 3.0; // has idnex 0
RealReverse w = sin(two * a) / three; // which value should be stored in index 0
```

Possible approaches:

1. Make variable active in statement
   - Each use creates an additional 24 bytes
2. Activate variable on first use
   - Only possible with reuse index manager
   - First use creates an additional 24 bytes
   - Increases *active* section of program
     $\Rightarrow$ more statements recorded
3. Store values in passive vector
   - Global use of an additional 256 * 24 bytes
   - Each use create an additional 8 bytes
   - Each statement used an additional 1 byte

Passive value handling - Option 3

Required changes:

- Store passive value with indices

```
inline void pushIndices(int* passiveVarCount, const double& value, const int& index) {
  IndexType pushIndex = index;
  if(0 == pushIndex) {
    *passiveVarCount += 1;
    pushIndex = *passiveVarCount;
    constantValueVector.setDataAndMove(value);
  }

  indexVector.setDataAndMove(pushIndex);
}
```

- Add number of passives to statement

```
  stmtVector.setDataAndMove(handle, (uint8_t)passiveVariableCount);
```

- Update primal values in reverse sweep

```
  constantPos -= passiveActives;
  for(StatementInt i = 0; i < passiveActives; ++i) {
    primalVector[i + 1] = constants[constantPos + i];
  }
```

Example and memory consumption

- 2D coupled Burgers equation on 601x601 grid with 32 time steps

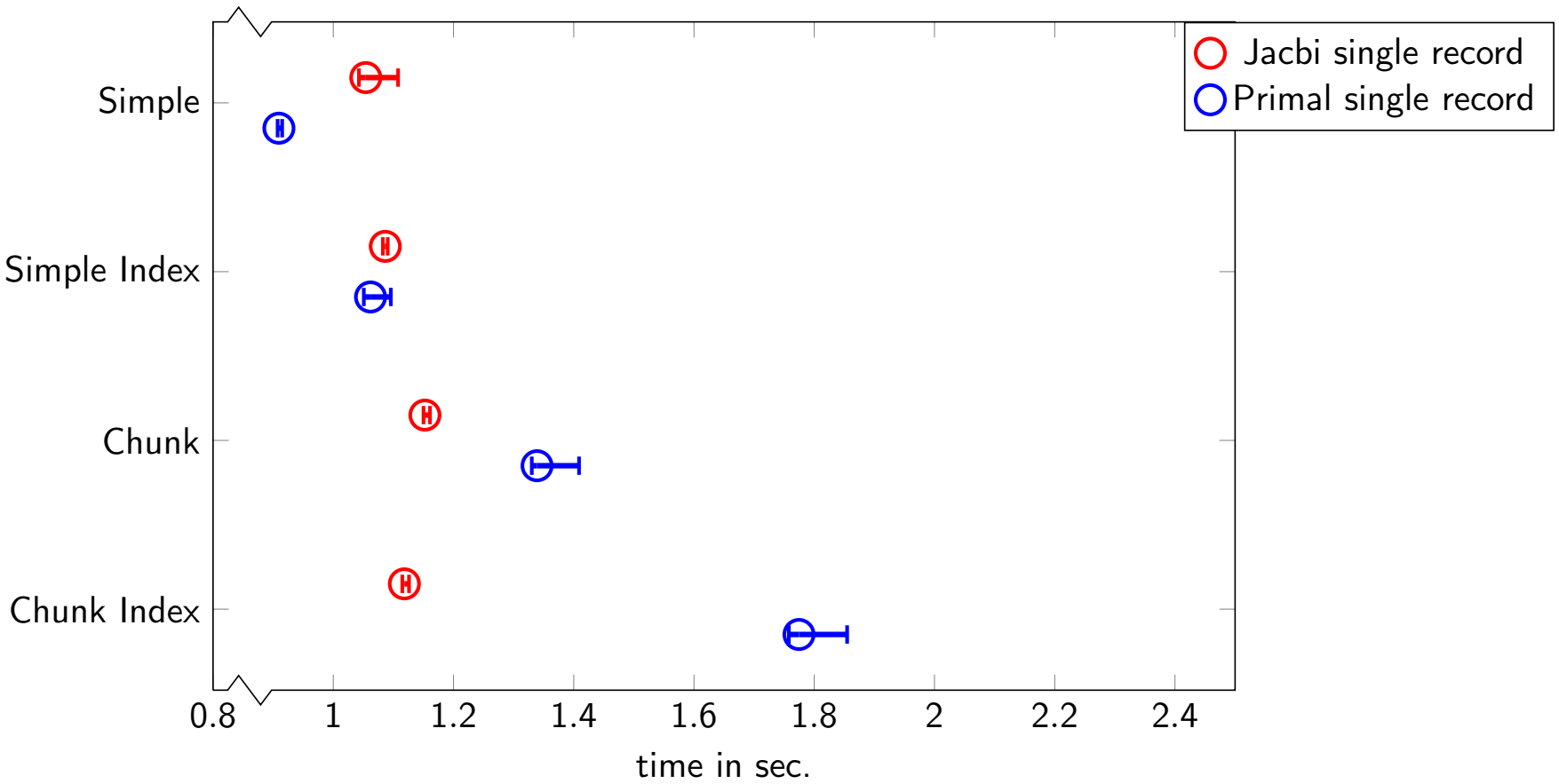
$$u_t + uu_x + vu_y = \frac{1}{R}(u_{xx} + u_{yy}) \tag{1}$$

$$v_t + uv_x + vv_y = \frac{1}{R}(v_{xx} + v_{yy}) \tag{2}$$

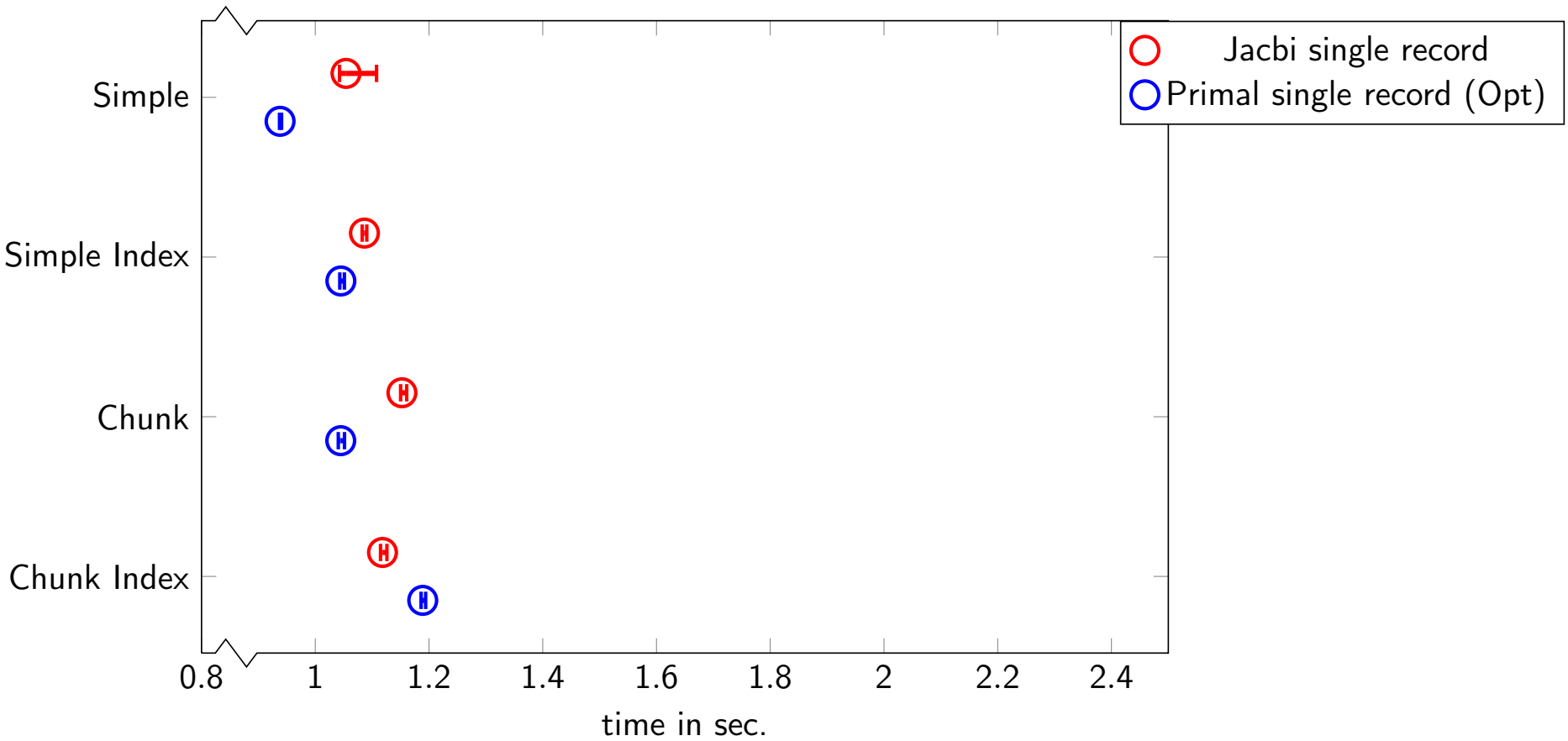
- Evaluated on one node of the Elwetritsch cluster with two Intel E5-2670 cpu's (16 cores)
- *Single* test case: Run only one process
  - Full memory bandwidth available
- *Multi* test case: Run 16 times the same problem
  - Memory bandwidth is limited

	Jacobi	Jacobi index	Primal	Primal index
Memory in MB	4830	4496	4030	3853
Reduction	0 %	7 %	17 %	20 %

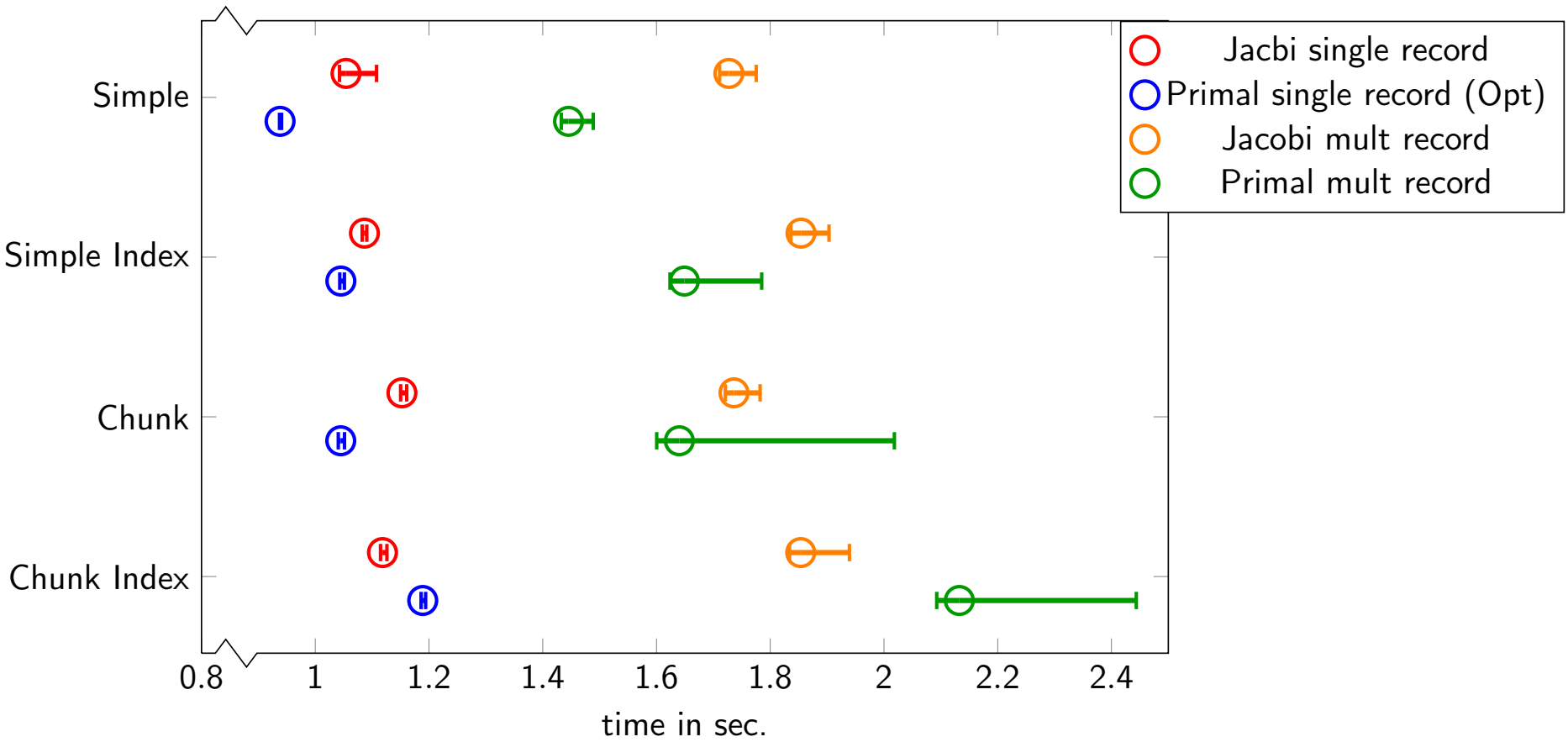
Time measurements - Recording time



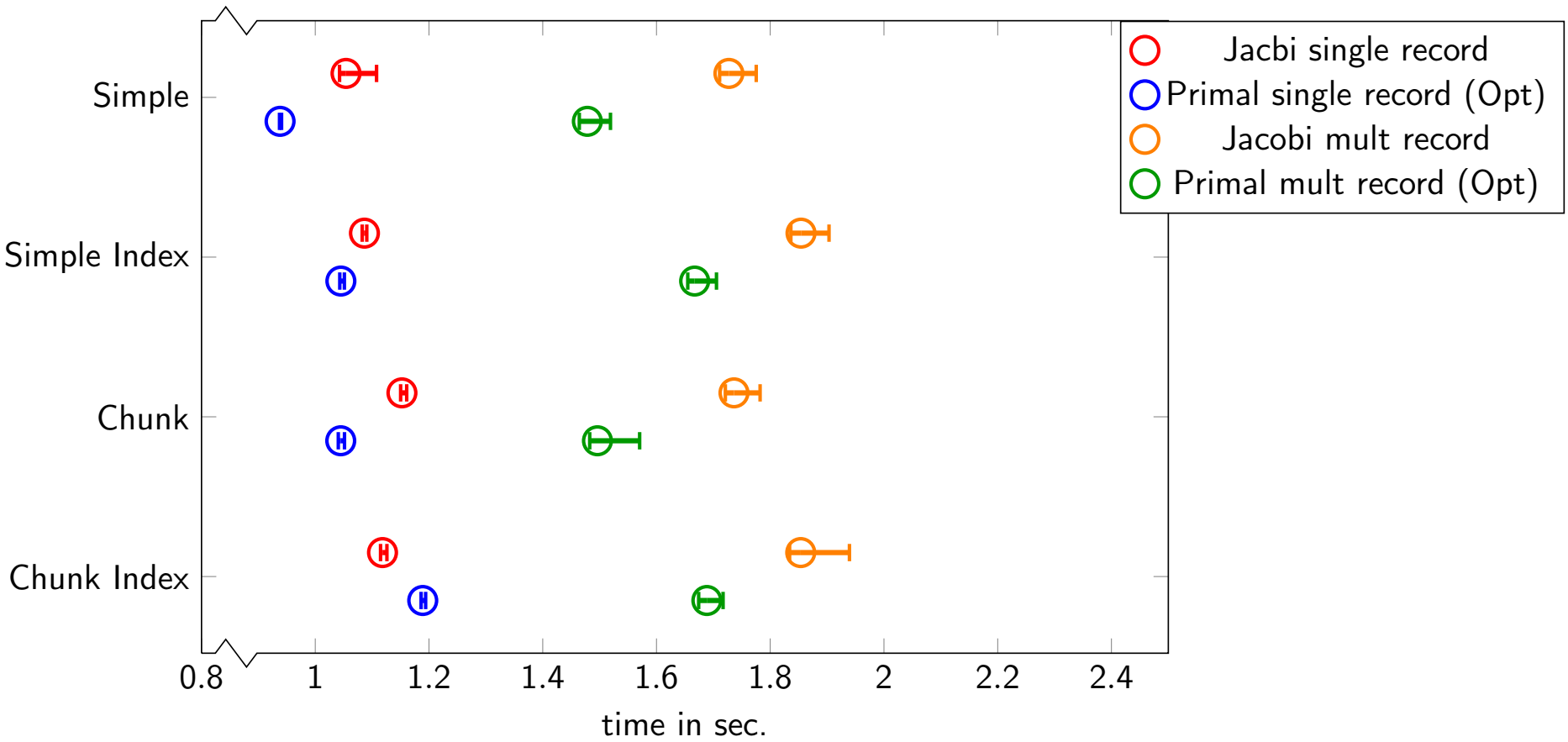
Time measurements - Recording time



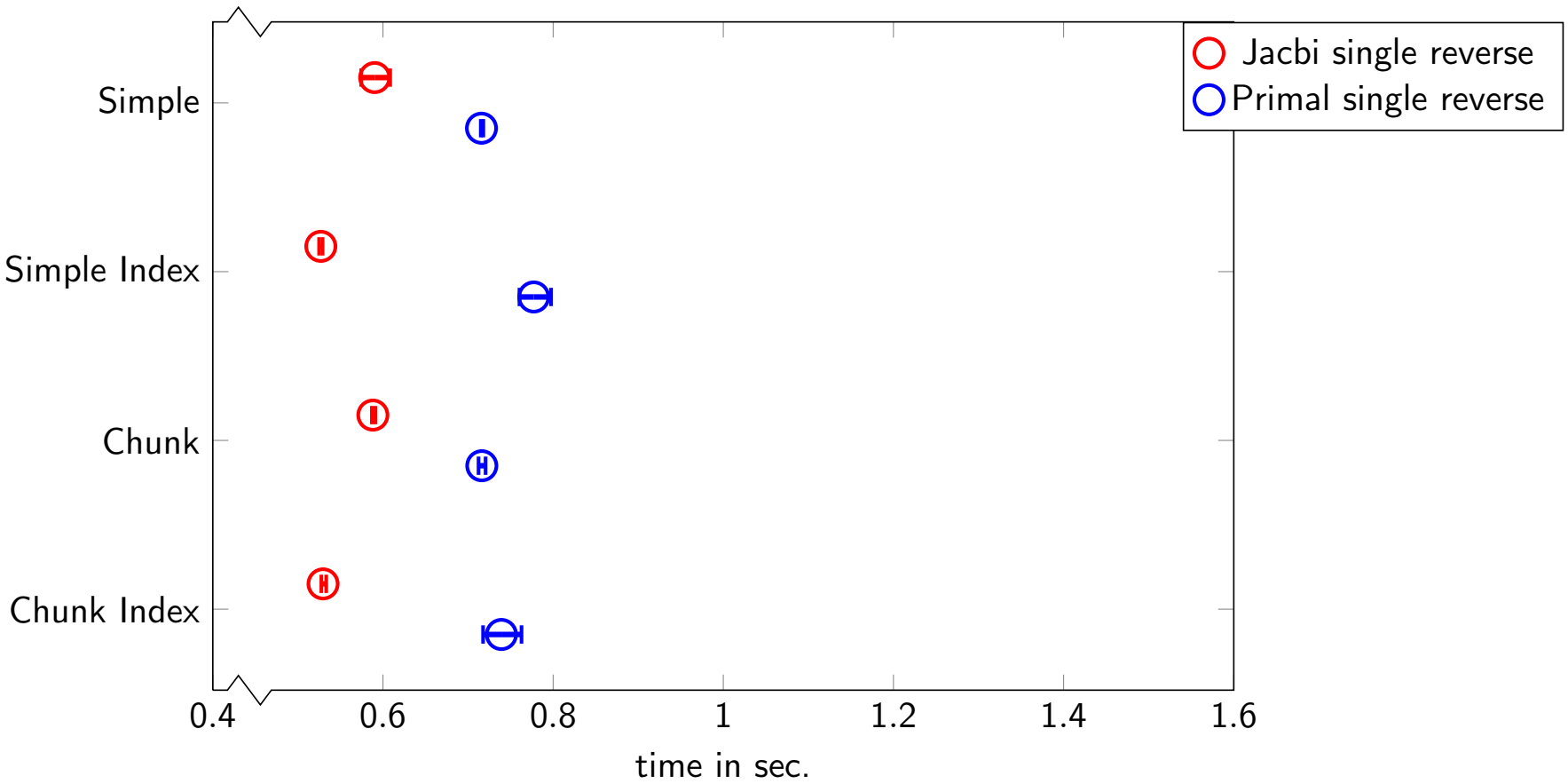
Time measurements - Recording time



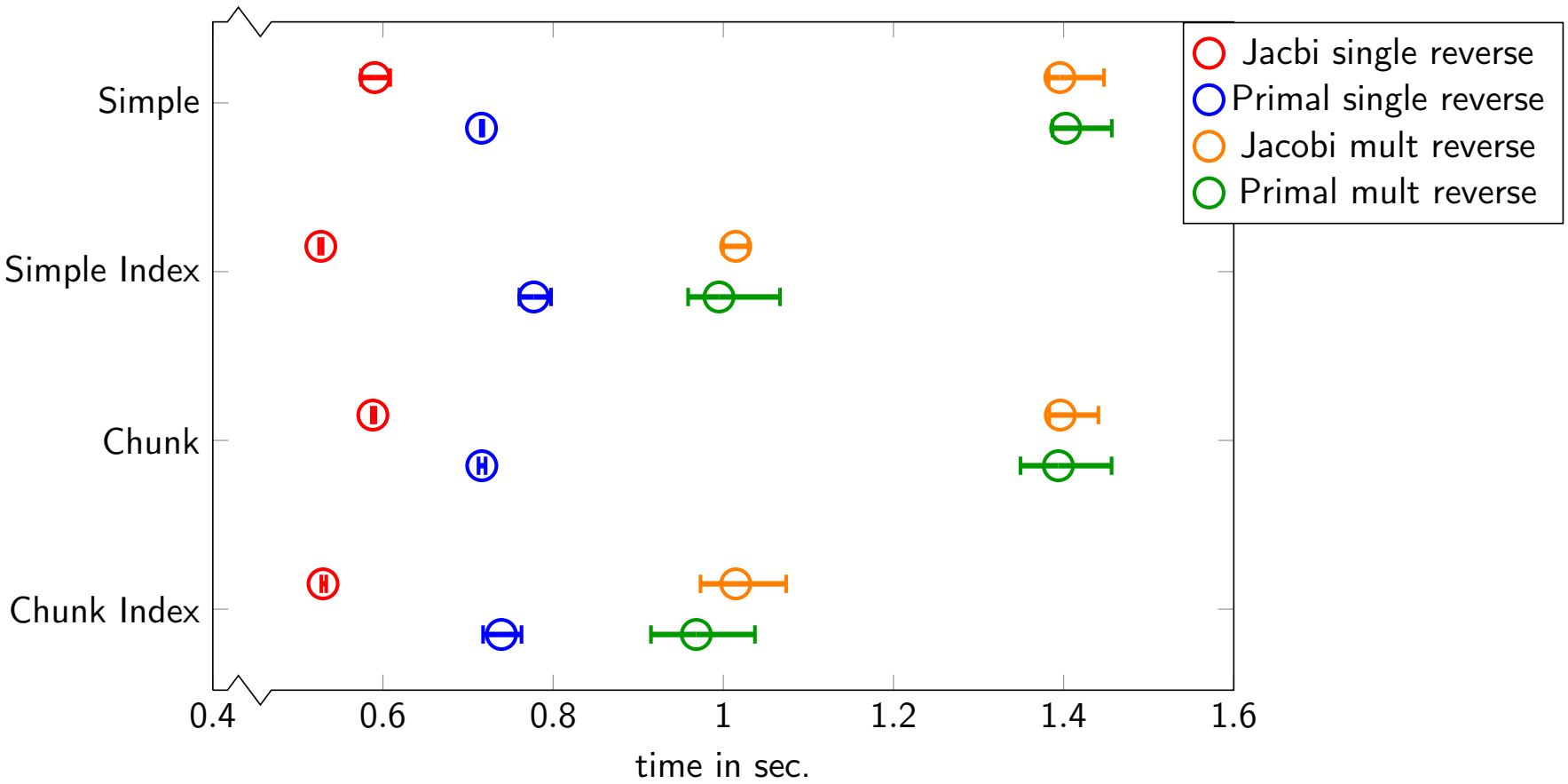
Time measurements - Recording time



Time measurements - Reverse evaluation time



Time measurements - Reverse evaluation time



Conclusion & Guidelines & Outlook

Conclusion:

- Fast implementation of primal value taping with expression templates
- Saves memory w.r.t. Jacobi taping
- Passive values need to be looked at

Conclusion & Guidelines & Outlook

Conclusion:

- Fast implementation of primal value taping with expression templates
- Saves memory w.r.t. Jacobi taping
- Passive values need to be looked at

Guidelines:

- Don't use constants e.g. 4.0
- Use active constants e.g.

```
RealReverse FOUR = 4.0;
tape.registerInput(FOUR);
```

- Legacy codes with a lot of passive values may prefer Jacobi taping

Conclusion & Guidelines & Outlook

Conclusion:

- Fast implementation of primal value taping with expression templates
- Saves memory w.r.t. Jacobi taping
- Passive values need to be looked at

Guidelines:

- Don't use constants e.g. 4.0
- Use active constants e.g.

```
RealReverse FOUR = 4.0;
tape.registerInput(FOUR);
```

- Legacy codes with a lot of passive values may prefer Jacobi taping

Outlook:

- Investigate inline problems
- Implement different passive value handling for index tapes
- Improve memory per statement (from 16 byte to 10 byte)

CoDiPack Release 1.3

CoDiPack release 1.3

- Primal value tape implementation
  - RealReversePrimal, RealReversePrimalIndex
  - RealReversePrimalVec, RealReversePrimalIndexVec
  - RealReversePrimalUnchecked, RealReversePrimalIndexUnchecked
- *numeric_limits* specialization
- Added *erf* and *erfc*
- Newsletter: codi-info@uni-kl.de
- Contact: codi@scicomp.uni-kl.de

Thank you for your attention!