An Integer Programming Approach to the Call Tree Reversal Problem

Johannes Lotz, Uwe Naumann
Software and Tools for Computational Engineering
RWTH Aachen University, Germany
lotz@stce.rwth-aachen.de
www.stce.rwth-aachen.de

Sumit Mitra
Department of Chemical Engineering
Carnegie Mellon University
Pittsburgh, PA, USA
Context of this work

"lib-version"  "source-version"

non-academics  academia

NAG

dco (derivative code by overloading)
dco/c++  dco/fortran

external functions
mind the gap

checkpointing

hand-written  continuous approach
source code transformation (dcc)

user-defined intrinsics
(interface to tape)

adjoint LAPACK
adjoint BLAS
adjoint PETSc

instrumentation
Adjoint / Adjoinable MPI: Michel Schanen

dcc, adjoint OpenMP: M. Förster

dco/c++: J. Lotz, V. Mosenkis (NAG), K. Leppkes, L. Razik

dco/fortran: J. Riehme, V. Mosenkis (NAG), K. Leppkes

AD in global optimization: M. Beckers

Tool Application to ...: M. Towara, A. Sen, A. Dastouri

... plus various projects.

... a “bit” (or MB) of involvement everywhere: U. Naumann
Context of this work (II)

"lib-version"  "source-version"
non-academics  academia

NAG

dco (derivative code by overloading)
dco/c++  dco/fortran

external functions
mind the gap

checkpointsing

hand-written  continuous approach
source code transformation (dcc)

user-defined intrinsics
(interface to tape)

adjoint LAPACK
adjoint BLAS
adjoint PETSc

instrumentation
a) memory consumption of tape as ratio of caller routine

visualization tool by courtesy of Max Sagebaum (CCES, RWTH)
a) memory consumption of tape as ratio of caller routine

visualization tool by courtesy of Max Sagebaum (CCES, RWTH)

b) call tree information

- call tree structure
- number of inputs (size of possible checkpoint)
- tape size information (per routine)
original

call tree

\[ f_i \quad \rightarrow \quad f_j \]
Short Introduction to Call Tree Reversal (I)

original call tree

- $f_i$ (advance (run routine))
- $f_j$ (store all (tape))
- $f_i$ (checkpoint arguments)
- $f_j$ (restore arguments)
- $f_i$ (reverse)

Checkpointing usually saves memory.
Short Introduction to Call Tree Reversal (I)

original call tree

A
T
C
R
I

advance (run routine)
store all (tape)
checkpoint arguments
restore arguments
reverse

no checkpointing \((y = 0)\)

\(f_i\)
\(f_j\)
\(f_i\)
\(f_j\)

\(⇒\) checkpointing usually saves memory.
Short Introduction to Call Tree Reversal (I)

Original call tree:

- $f_i$ 
- $f_j$

Advance (run routine):
- $A$

Store all (tape):
- $T$

Checkpoint arguments:
- $C$

Restore arguments:
- $R$

Reverse:
- $I$

No checkpointing ($y = 0$):

- $f_i$ 
- $f_j$

With checkpointing ($y = 1$):

- $f_i$ 
- $f_j$

Checkpointing usually saves memory.
original call tree

\[ A \] advance (run routine)

\[ T \] store all (tape)

\[ C \] checkpoint arguments

\[ R \] restore arguments

\[ I \] reverse

\[ f_i \]

\[ f_j \]

⇒ checkpointing usually saves memory.

no checkpointing \((y = 0)\)

\[ f_i \rightarrow f_i \]

\[ f_j \rightarrow f_j \]

with checkpointing \((y = 1)\)

\[ f_i \rightarrow f_i \]

\[ f_j \rightarrow f_j \rightarrow f_j \]
Short Introduction to Call Tree Reversal (II)

- **A**: advance (run routine)
- **T**: store all (tape)
- **C**: checkpoint arguments
- **R**: restore arguments
- **I**: reverse

Diagram:
- **checkpointing on middle layer**

- $f_i$ (forward)
- $f_j$ (middle layer)
- $f_k$ (backward)
Example: Where to Set Checkpoints?

![Diagram showing the set of possible checkpoints]

- Set of possible checkpoints: $s_1$, $s_2$, $s_3$, $s_4$
- Connections and delays:
  - $s_1$ to $s_2$: 15
  - $s_2$ to $s_3$: 10
  - $s_2$ to $s_4$: 10
  - $s_3$ to $s_4$: 10
- Delays:
  - $s_1$: 5
  - $s_2$: 5
  - $s_3$: 10
  - $s_4$: 5
  - Total time for $s_3$: 200
  - Total time for $s_4$: 50

⇒ i.e., an exponential number of possible “reversal schemes.”
Example: Where to Set Checkpoints?

\[ \Rightarrow \]

- i.e., an exponential number of possible "reversal schemes"
Example: Where to Set Checkpoints?

⇒ i.e., an exponential number of possible “reversal schemes”
Example: Search Space

\[ y \in \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\} \]
Starting with zero memory \((M = 0)\) and zero operations \((O = 0)\).
Starting with zero memory \((M = 0)\) and zero operations \((O = 0)\).

1. tape first part of \(s_1\): \(M = 15, O = 15\)
Starting with zero memory ($M = 0$) and zero operations ($O = 0$).

1. tape first part of $s_1$: $M = 15, O = 15$
2. checkpoint $s_s$, run $s_2, s_3, s_4$: $M = 20, O = 295$
Cost: Memory Requirement and Operations Count

Starting with zero memory \((M = 0)\) and zero operations \((O = 0)\).

1. tape first part of \(s_1\): \(M = 15, O = 15\)
2. checkpoint \(s_s\), run \(s_2, s_3, s_4\): \(M = 20, O = 295\)
3. tape second part of \(s_1\): \(M = 25, O = 300\)
Starting with zero memory ($M = 0$) and zero operations ($O = 0$).

1. tape first part of $s_1$: $M = 15, O = 15$
2. checkpoint $s_s$, run $s_2, s_3, s_4$: $M = 20, O = 295$
3. tape second part of $s_1$: $M = 25, O = 300$
4. interpret last part of $s_1$: $M = 20, O = 300$
Starting with zero memory \((M = 0)\) and zero operations \((O = 0)\).

1. tape first part of \(s_1\): \(M = 15, O = 15\)
2. checkpoint \(s_s\), run \(s_2, s_3, s_4\): \(M = 20, O = 295\)
3. tape second part of \(s_1\): \(M = 25, O = 300\)
4. interpret last part of \(s_1\): \(M = 20, O = 300\)
5. restore checkpoint of \(s_2\): \(M = 15, O = 300\)
Starting with zero memory ($M = 0$) and zero operations ($O = 0$).

1. tape first part of $s_1$: \( M = 15, O = 15 \)
2. checkpoint $s_s$, run $s_2, s_3, s_4$: \( M = 20, O = 295 \)
3. tape second part of $s_1$: \( M = 25, O = 300 \)
4. interpret last part of $s_1$: \( M = 20, O = 300 \)
5. restore checkpoint of $s_2$: \( M = 15, O = 300 \)
6. tape $s_2, s_3, s_4$: \( M = 295, O = 580 \)
Starting with zero memory \((M = 0)\) and zero operations \((O = 0)\).

1. tape first part of \(s_1\): \(M = 15, O = 15\)
2. checkpoint \(s_s\), run \(s_2, s_3, s_4\): \(M = 20, O = 295\)
3. tape second part of \(s_1\): \(M = 25, O = 300\)
4. interpret last part of \(s_1\): \(M = 20, O = 300\)
5. restore checkpoint of \(s_2\): \(M = 15, O = 300\)
6. tape \(s_2, s_3, s_4\): \(M = 295, O = 580\)
7. interpret \(s_2, s_3, s_4\): \(M = 0, O = 580\)
The Optimization Problem

For a

- search space $S$ (all possible adjoint call trees) and an
- upper bound on the available persistent memory $M$
- minimize the operations count $O$. 
Why Call Tree Reversal?

- If you know the structure and the math and AD principles and ...

\[ f \]

\[ g_1 \quad g_2 \quad g_3 \quad g_4 \quad g_5 \quad g_6 \]

\[ \Rightarrow \text{revolve, equidistant checkpointing} \]
Why Call Tree Reversal?

- ... if you know the structure and the math and AD principles and ...

\[ \Rightarrow \text{revolve, equidistant checkpointing} \]

- ... else ...

\[ \begin{align*}
  & g_1 & g_2 & g_3 & g_4 & g_5 & g_6 \\
  & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \\
  & f & & & & & \\
\end{align*} \]
Transform CTR to an IP (Integer Programming) formulation

- **Goals**
  - as little self-written buggy programs as possible
  - use existing efficient algorithms

\[^1\]... that means, real problems will arise soon.
Transform CTR to an IP (Integer Programming) formulation

Goals

- as little self-written buggy programs as possible
- use existing efficient algorithms

... mission accomplished\(^1\).

Network flow graph:

\(^1\)... that means, real problems will arise soon.
Transform CTR to an IP (Integer Programming) formulation

- Goals
  
  - as little self-written buggy programs as possible
  - use existing efficient algorithms

- ... mission accomplished\(^1\).

Network flow graph:

- flow variable: memory
- sources (tape / checkpoint)
- sinks (restore checkpoint / interpret)
- upper bound \(M\) for the flow on edges

\(^1\) ... that means, real problems will arise soon.
Network Flow Problem (I)

memory sources

memory sinks
Network Flow Problem (II) – no checkpoint

memory sources

memory sinks
Network Flow Problem (III) – w/ checkpoint

memory sources

memory sinks
Network Flow Problem (IV)

memory sources

memory sinks
1. Definition of the connectivity of all nodes
2. Sources and sinks for nodes
3. Objective function
   \[\min c(y) = \sum_{s \in \Sigma} c_s + \sum_{s \in \Sigma} y_s \cdot \bar{c}_s\]
4. Bounds on edges
   \[
   \begin{align*}
   0 & \leq F_e \leq M & \forall e \in G_s \\
   0 & \leq F_e \leq M \cdot (1 - y_s) & \forall e \in L_s \\
   0 & \leq F_e \leq M \cdot y_s & \forall e \in R_s
   \end{align*}
   \]
The current “tool-chain”

From a given computer program $P$, we need to extract the call tree information $T$ (I). With a given upper bound on the available memory, we generate the network flow graph $N$ (II), whose solution $y$ (III) defines the adjoint call tree $\overline{T}$ (IV). This configuration needs to be implemented in an adjoint program $\overline{P}$ (V).
The current “tool-chain”

From a given computer program $\mathbb{P}$, we need to extract the call tree information $\mathcal{T}$ (I). With a given upper bound on the available memory, we generate the network flow graph $\mathcal{N}$ (II), whose solution $y$ (III) defines the adjoint call tree $\overline{\mathcal{T}}$ (IV). This configuration needs to be implemented in an adjoint program $\overline{\mathbb{P}}$ (V).

(I) dco/c++ instrument mode
The current “tool-chain”

From a given computer program \( P \), we need to extract the call tree information \( \mathcal{T} \) (I). With a given upper bound on the available memory, we generate the network flow graph \( N \) (II), whose solution \( y \) (III) defines the adjoint call tree \( \overline{\mathcal{T}} \) (IV). This configuration needs to be implemented in an adjoint program \( \overline{P} \) (V).

(I) dco/c++ instrument mode

(II) helper program (C++)
From a given computer program \( P \), we need to extract the call tree information \( \mathcal{T} \) (I). With a given upper bound on the available memory, we generate the network flow graph \( \mathcal{N} \) (II), whose solution \( y \) (III) defines the adjoint call tree \( \overline{\mathcal{T}} \) (IV). This configuration needs to be implemented in an adjoint program \( \overline{P} \) (V).

(I) dco/c++ instrument mode

(II) helper program (C++)

(III) GAMS (General Algebraic Modeling System)
The current “tool-chain”

From a given computer program $\mathbb{P}$, we need to extract the call tree information $\mathcal{T}$ (I). With a given upper bound on the available memory, we generate the network flow graph $\mathcal{N}$ (II), whose solution $y$ (III) defines the adjoint call tree $\overline{\mathcal{T}}$ (IV). This configuration needs to be implemented in an adjoint program $\overline{\mathbb{P}}$ (V).

(I) dco/c++ instrument mode
(II) helper program (C++)
(III) GAMS (General Algebraic Modeling System)
(IV) implicitly
The current “tool-chain”

\[ \mathbb{P} \xrightarrow{(I)} \mathcal{T} \xrightarrow{(II)} \mathcal{N} \xrightarrow{(III)} y \xrightarrow{(IV)} \overline{\mathcal{T}} \xrightarrow{(V)} \overline{\mathbb{P}} \]

*From a given computer program \( \mathbb{P} \), we need to extract the call tree information \( \mathcal{T} \) (I). With a given upper bound on the available memory, we generate the network flow graph \( \mathcal{N} \) (II), whose solution \( y \) (III) defines the adjoint call tree \( \overline{\mathcal{T}} \) (IV). This configuration needs to be implemented in an adjoint program \( \overline{\mathbb{P}} \) (V).*

(I) dco/c++ instrument mode  
(II) helper program (C++)  
(III) GAMS (General Algebraic Modeling System)  
(IV) implicitly  
(V) manually with dco/c++ external functions
Problems / Decisions to be made

- How to extract the call tree information?
  - pure C++? (includes more code change)
  - “easy-to-use” gcc plugin (e.g. -finstrument-functions)
  - “real” gcc plugin (i.e. work on AST)
- How to reduce the problem size (inlining)?
- How to implement the resulting checkpointing scheme (manually)?