Algorithmic Differentiation of Pragma-Defined Parallel Regions
Algorithmic Differentiation of Pragma-Defined Parallel Regions

Differentiating Computer Programs Containing OpenMP
Abstract

The goal of this dissertation is to develop a source code transformation that exploits the knowledge that a given input code is parallelizable in a way that it generates derivative code efficiently executable on a supercomputer environment.

There is barely a domain where optimization does not play a role. Not only in science and engineering, also in economics and industry it is important to find optimal solutions for a given problem. The size of these optimization problems often requires large-scale numerical techniques that are capable of running on a supercomputer architecture.

For continuous optimization problems the calculation of derivative values of a given function is crucial. If these functions are given as a computer code implementation $Q$ then techniques known as algorithmic differentiation (AD) alias automatic differentiation can be used to obtain an implementation $Q'$ that is capable of computing the derivative of a given output of $Q$ with respect to a certain input. This thesis focuses on algorithmic differentiation by source transformation. The implementation $Q$ is transformed into $Q'$ such that $Q'$ contains assignments for computing the derivative values.

On the one side, the size of optimization problems is rising. On the other side, the number of cores per central processing unit (CPU) in modern computers is growing. A typical supercomputer node has up to 32 cores or even more in case that multiple physical nodes form a compound. In order to allow $Q$ to compute its output values efficiently, the implementation of $Q$ should exploit the underlying multicore computer architecture. An easy approach of using parallel programming is to declare a certain code region inside of $Q$ as parallelizable. This declaration is done by setting a certain kind of pragma in front of the corresponding code region. The pragma is a compiler directive and in our case this special directive informs the compiler that the corresponding code region should be executed concurrently. This code region is denoted as a parallel region $P$ and the parallel instances which execute $P$ are called threads.

There are two fundamental modes in AD, the forward and the reverse mode. We present source transformation rules for a simplified programming language, called $SPL$. In addition, we show that these rules provide derivative code either in forward or in reverse mode. One crucial goal of this work is that the knowledge that the original code contains a parallel region $P$ leads to a parallel region $P'$
in the derivative code. This allows a concurrent computation of the derivative values. We exhibit a proof to ensure that the parallel execution of $P'$ is correct. In case that the user of AD wants to achieve higher derivative code the possibility of reapplying the source transformation is important. Therefore, we exhibit that the source transformation is closed in the sense that the output code language is the same as the input language.

The reverse mode of AD builds the so-called adjoint code. The term ‘reverse’ indicates that the adjoint code requires a dataflow reversal of the execution of $P$. Suppose that $P$ consists of code where a memory location is read by multiple threads. The dataflow reversal of $P$ leads to the situation that the corresponding derivative component of this memory location is a target of multiple store operations from different threads during the execution of the adjoint code. These store operations must be synchronized. Otherwise, the adjoint code would have a race condition at runtime. Conservatively, one could assume that all memory locations in $P$ are read by multiple threads, which leads to the result that the adjoint source transformation generates a lot of synchronization constructs to ensure a correct parallel execution. In the worst case the synchronization overhead leads to a concurrent runtime of the derivative code $P'$ that is bigger than the sequential runtime. In order to avoid as much synchronization as possible, we develop a static program analysis that collects information about $P$ at compile time whether or not a memory location is exclusively read by a thread. If a memory location read exclusively, the adjoint source transformation does not need to emit a synchronization method for the corresponding derivative computation. This can make a major difference.

We demonstrate how the context-free grammar for the language $SPL$ can be extended in order to recognize pragmas defined in the OpenMP standard. Beside the extension of the grammar we present source transformation rules for these OpenMP constructs. With the source transformation rules for constructs such as the barrier, the critical, or the worksharing loop construct, this work provides rules for generating derivative code for most of the occurring OpenMP parallel regions.

The approach of this work has been implemented in a tool, called simple parallel language compiler ($SPLc$). We give evidence that our approach is applicable through the implementation of two optimization problems. On the one hand, we use a first derivative code provided by $SPLc$ to solve a nonlinear least-squares problem. On the other hand, a nonlinear constrained optimization problem has been solved with the second derivative code provided by $SPLc$ as well.
Acknowledgments

First and foremost, I would like to thank my adviser Prof. Dr. Uwe Naumann for his thoughtful guidance and constant encouragement. He has been inspiring since my days as a computer science student and I am grateful that I did get the chance to make my PhD at his institute.

Special thanks goes to my co-supervisor Prof. Dr. Christian Bischof for his support and constructive suggestions. I also would like to thank Prof. Dr. Thomas Noll for the fruitful discussions about static program analysis.

Further thanks goes to all current and former employees of the Lehr- und Forschungsgebiet Computer Science 12 at the RWTH Aachen University. We had a lot of fun and many discussions mainly about algorithmic differentiation but besides about all sorts of things. I would especially like to thank Johannes Lotz, Lukas Razik, Michel Schanen, Arindam Sen, and Markus Towara for reading early drafts of this manuscript and for finding most of the contained errors and unintelligible explanations.

Many thanks to the IT center of the RWTH Aachen University. The HPC group provided a lot of useful material about parallel programming and in particular about OpenMP. At this point, a special thank goes to Sascha Bücken for providing us access to the SUN cluster that had served its time, but has never lost its capabilities for being a good environment for parallel programming.

I really appreciate all my family and I am grateful for all their support during all these years. I wish my father could read this but he passed away much too early. Anyway, thanks for all the fun during our successful motocross years. The ups and downs during my sports career have been a major influence on my further life.

Last, but not least, I would like to thank my wife Mirjam. Without her understanding and support during the past few years, I may not have finished this thesis. Many thanks for all the pleasure that we had together with our two little daughters Emmy and Matilda.
## Contents

Abstract \hspace{1.5cm} V

Acknowledgments \hspace{1.5cm} VII

1 Motivation and Introduction \hspace{1.5cm} 1
  1.1 Numerical Optimization in the Multicore Era \hspace{1.5cm} 1
    1.1.1 A Nonlinear Least-Squares Problem \hspace{1.5cm} 6
  1.2 Algorithmic Differentiation \hspace{1.5cm} 11
    1.2.1 Second Derivative Code \hspace{1.5cm} 18
    1.2.2 \texttt{dcc} - A Derivative Code Compiler \hspace{1.5cm} 21
    1.2.3 A Nonlinear Constrained Optimization Problem \hspace{1.5cm} 26
  1.3 OpenMP Standard 3.1 \hspace{1.5cm} 29
  1.4 Related Work \hspace{1.5cm} 52
  1.5 Contributions \hspace{1.5cm} 55
  1.6 Outline of the Thesis \hspace{1.5cm} 58

2 Transformation of Pure Parallel Regions \hspace{1.5cm} 61
  2.1 Formalism and Notation \hspace{1.5cm} 61
  2.2 \textit{SPL} - A Simple Language for Parallel Regions \hspace{1.5cm} 79
  2.3 AD Source Transformation of \textit{SPL} Code \hspace{1.5cm} 85
    2.3.1 Tangent-Linear Model of \textit{SPL} - Transformation $\tau(P)$ \hspace{1.5cm} 86
    2.3.2 Adjoint Model of \textit{SPL} - Transformation $\sigma(P)$ \hspace{1.5cm} 94
    2.3.3 \textit{SPL} Code Inside of C/C++ Code \hspace{1.5cm} 113
  2.4 Closure of the Source Transformation \hspace{1.5cm} 116
    2.4.1 Closure Property of $\tau(P)$ \hspace{1.5cm} 116
    2.4.2 Closure Property of $\sigma(P)$ and the Exclusive Read Property \hspace{1.5cm} 118
  2.5 Summary \hspace{1.5cm} 134

3 Exclusive Read Analysis \hspace{1.5cm} 137
  3.1 Control Flow in \textit{SPL} code \hspace{1.5cm} 142
  3.2 Integer Interval Analysis \hspace{1.5cm} 146
  3.3 Directed Acyclic Graphs and Partial Orders \hspace{1.5cm} 148