Automatic Parallelization An Overview Of Fundamental Compiler Techniques Samuel P Midkiff

With the era of increasing clock speeds coming to an end, parallel computing architectures have now become main-stream. Due to the wide range of architectures available today that can be used to exploit parallelism, ranging from multicore CPUs, to GPGPUs, to distributed memory machines; adapting applications for efficient execution on all these architectures poses a significant challenge.

I Unidimensional Problems.- 1 Scheduling DAGs without Communications.- 2 Scheduling DAGs with Communications.- 3 Cyclic Scheduling.- II Multidimensional Problems.- 4 Systems of Uniform Recurrence Equations.- 5 Parallelism Detection in Nested Loops.

Developed in the context of science and engineering applications, with each abstraction motivated by and further honed by specific application needs, Charm++ is a production-quality system that runs on almost all parallel computers available. Parallel Science and Engineering Applications: The Charm++ Approach surveys a diverse and scalable collecti

Automatic Parallelization of Non-uniform Loops

for Multicore and Cluster Systems

Scheduling and Automatic Parallelization: Multidimensional Problems. 4. Systems of Uniform Recurrence Equations. 5. Parallelism Detection in Nested Loops

Using OpenMP

Empirically Examining the Roadblocks to the Automatic Parallelization and Analysis of Open Source Software Systems

Automatic Performance Prediction of Parallel Programs presents a unified approach to the problem of automatically estimating the performance of parallel computer programs. The author focuses primarily on distributed memory multiprocessor systems, although large portions of the analysis can be applied to shared memory architectures as well. The author introduces a novel and very practical approach for predicting some of the most important performance parameters of parallel programs, including work distribution, number of transfers, amount of data transferred, network contention, transfer time, computation time and number of cache misses. This approach is based on advanced compiler analysis that carefully examines loop iteration spaces, procedure calls, array subscript expressions, communication patterns, data distributions and optimizing code transformations at the program level; and the most important machine specific parameters including cache characteristics, communication network indices, and benchmark data for computational operations at the machine level. The material has been fully implemented as part of P3T, which is an integrated automatic performance estimator of the Vienna Fortran Compilation System (VFCS), a state-of-the-art parallelizing compiler for Fortran77, Vienna Fortran and a subset of High Performance Fortran (HPF) programs. A large number of experiments using realistic HPF and Vienna Fortran code examples demonstrate highly accurate performance estimates, and the ability of the described performance prediction approach to successfully guide both programmer and compiler in parallelizing and optimizing parallel programs. A graphical user interface is described and displayed that visualizes each program source line together with the corresponding parameter values. P3T uses color-coded performance visualization to immediately identify hot spots in the parallel program. Performance data can be filtered and displayed at various levels of detail. Colors displayed by the graphical user interface are visualized in greyscale. Automatic Performance Prediction of Parallel Programs also includes coverage of fundamental problems of automatic parallelization for distributed memory multicomputers, a description of the basic parallelization strategy and a large variety of optimizing code transformations as included under VFCS.

Parallel software is now required to exploit the abundance of threads and processors in modern multicore computers. Unfortunately, manual parallelization is too time-consuming and error-prone for all but the most advanced programmers. While automatic parallelization promises threaded software with little programmer effort, current auto-parallelizers are easily thwarted by pointers and other forms of ambiguity in the code. In this dissertation we profile the loops in SPEC CPU2006, categorize the loops in terms of available parallelism, and focus on promising loops that are not parallelized by IBM's XL C/C++ V10 auto-parallelizer. For those loops we propose methods of improved interaction between the programmer and compiler that can facilitate their parallelization. In particular, we (i) suggest methods for the compiler to better identify to the programmer the parallelization-blockers; (ii) suggest methods for the programmer to provide guarantees to the compiler that overcome these parallelization-blockers; and (iii) evaluate the resulting impact on performance.

To effectively translate real programs written in standard, sequential languages into parallel computer programs, parallelizing compilers need advanced techniques such as powerful dependence tests, array privatization, generalized induction variable substitution, and reduction parallelization. All of these techniques need or can benefit from symbolic analysis. To determine what kinds of symbolic analysis techniques can significantly improve the effectiveness of parallelizing Fortran compilers, we compared the automatically and manually parallelized versions of the Perfect Benchmarks. The techniques identified include: data dependence tests for nonlinear expressions, constraint propagation, interprocedural constant

propagation, array summary information, and run time tests. We have developed algorithms for two of these identified symbolic analysis techniques: nonlinear data dependence analysis and constraint propagation. For data dependence analysis nonlinear expressions, (e.g., A(n * i + j), where $1 le j le n), $ we developed a data dependence test called the Range Test. The Range Test proves independence by determining whether certain symbolic inequalities hold for a logical permutation of the loop nest. We use a technique called Range Propagation to prove these symbolic inequalities. For constraint propagation, we developed a technique called Range Propagation. Range Propagation computes the range of values that each variable can take at each point of a program. A range is a symbolic lower and upper bound on the values taken by a variable. Range propagation also includes a facility to compare arbitrary expressions under the constraints imposed by a set of ranges. We have developed both a simple but slow algorithm and a fast and demand-driven but complex algorithm to compute these ranges. The Range Test and Range Propagation have been fully implemented in Polaris, a parallelizing compiler being developed at the University of Illinois. We have found that these techniques significantly improve the effectiveness of automatic parallelization. We have also found that these techniques are reasonably efficient.

A Unifying Language Study of Automatic Parallelization

Programmer-Assisted Automatic Parallelization

Advanced Parallel Processing Technologies

Automatic Parallelization Via Loop Separation

A Case Study

This book constitutes the proceedings of the 14th IFIP WG 10.3 International Conference on Network and Parallel Computing, NPC 2017, held in Hefei, China, in October 2017.The 9 full papers and 10 short papers presented in this book were carefully reviewed and selected from 88 submissions. The papers cover traditional areas of network and parallel computing including parallel applications, distributed algorithms, software environments, and distributed tools.?

This book constitutes the refereed proceedings of the 9th International Symposium on Advanced Parallel Processing Technologies, APPT 2011, held in Shanghai, China, in September 2011. The 13 revised full papers presented were carefully reviewed and selected from 40 submissions. The papers are organized in topical sections on parallel distributed system architectures, architecture, parallel application and software, distributed and cloud computing.

This handbook offers a comprehensive treatise on Grammatical Evolution (GE), a grammar-based Evolutionary Algorithm that employs a function to map binary strings into higher-level structures such as programs. GE's simplicity and modular nature make it a very flexible tool. Since its introduction almost twenty years ago, researchers have applied it to a vast range of problem domains, including financial modelling, parallel programming and genetics. Similarly, much work has been conducted to exploit and understand the nature of its mapping scheme, triggering additional research on everything from different grammars to alternative mappers to initialization. The book first introduces GE to the novice, providing a thorough description of GE along with historical key advances. Two sections follow, each composed of chapters from international leading researchers in the field. The first section concentrates on analysis of GE and its operation, giving valuable insight into set up and deployment. The second section consists of seven chapters describing radically different applications of GE. The contributions in this volume are beneficial to both novices and experts alike, as they detail the results and researcher experiences of applying GE to large scale and difficult problems. Topics include: • Grammar design • Bias in GE • Mapping in GE • Theory of disruption in GE · Structured GE · Geometric semantic GE · GE and semantics · Multi- and Many-core heterogeneous parallel GE · Comparing methods to creating constants in GE · Financial modelling with GE · Synthesis of parallel programs on multi-cores · Design, architecture and engineering with GE · Computational creativity and GE · GE in the prediction of glucose for diabetes · GE approaches to bioinformatics and system genomics · GE with coevolutionary algorithms in cybersecurity · Evolving behaviour trees with GE for platform games · Business analytics and GE for the prediction of patient recruitment in multicentre clinical trials

Automatic Parallelization of Loop Programs for Distributed Memory Architectures

Semi-automatic Parallelization of FORTRAN Programs

Autotuning for Automatic Parallelization on Heterogeneous Systems

An Overview of Fundamental Compiler Techniques

Readership This book is devoted to the study of compiler transformations that are needed to expose the parallelism hiddenin a program. This book is notan introductory book to parallel processing, nor is it an introductory book to parallelizing compilers. Weassume thatreaders are familiar withthebooks High Performance Compilers for Parallel Computingby Wolfe [121] and Super compilers for Parallel and Vector Computers by Zima and Chapman [125], and that they want to know more about scheduling transformations. In this book we describe both task graph scheduling and loop nest scheduling. Taskgraphschedulingaims atexecuting tasks linked by prece dence constraints; it is a run-time activity. Loop nest scheduling aims at ex ecutingstatementinstances linked bydata dependences;it is a compile-time activity. We are mostly interested in loop nestscheduling,butwe also deal with task graph scheduling for two main reasons: (i) Beautiful algorithms and heuristics have been reported in the literature recently; and (ii) Several graphscheduling, like list scheduling, are the basis techniques used in task ofthe loop transformations implemented in loop nest scheduling. As for loop nest scheduling our goal is to capture in a single place the fantastic developments of the last decade or so. Dozens of loop trans formations have been introduced (loop interchange, skewing, fusion, dis tribution, etc.) before a unifying theory emerged. The theory builds upon the pioneering papers of Karp, Miller, and Winograd [65] and of Lam port [75], and it relies on sophisticated mathematical tools (unimodular transformations, parametric integer linear programming, Hermite decom position, Smithdecomposition, etc.).

Modern multicore architectures have become ubiquitous and are present in almost all of today's computers and mobile devices. That is, the hardware resources are available to parallelize all types of general-purpose software applications. This fact has pushed the need for software engineers to rethink how the code they write can better

utilize the underlying hardware. Because most of existing software systems were developed with sequential processors in mind, they typically make inefficient use of the multicore technology (by using only one core in many cases). Parallelizing existing software systems can be a very time consuming and a risky task because of the expected errors and bugs that can be introduced if a manual approach is considered. The dissertation explores issues related to the analysis of large-scale general-purpose software systems developed in C/C++ and if it is practical and warranted to parallelize such systems. A series of empirical studies is conducted to examine of a variety of general-purpose open source software systems to better understand the roadblocks for applying automated and/or semi-automated parallelization tools. The primary contributions presented in this dissertation are, broadly, the study and description of inhibitors to automated parallelization and demonstrate which inhibitors are most prevalent. That is, the main interest is determining the most prevalent inhibitors that occur in a wide variety of software applications and if there are general trends. Additionally, the development of new source code analysis techniques and tools for analyzing large-scale software repositories are presented. Empirical studies of the historical change, over the lifetime of the software systems, in the number of inhibitors are also conducted. Empirical analysis of the prevalence and usage of function calls that involve function pointers or virtual methods is also conducted as this can greatly increase the computational cost of performing accurate analysis. The results of the empirical study demonstrate that the most prevalent inhibitor by far, is functions called within for-loops that have side effects. This single inhibitor poses the greatest challenge in adapting and re-engineering systems to better utilize modern multi-core architectures. This fact is somewhat contradictory to the literature, which is primarily focused on the removal of data dependencies within loops. The results also show that in a majority of the time function pointers are used in situations that make analysis very difficult (i.e., NP-hard). Thus, conducting accurate program analysis (e.g., program slicing, call graph generation) becomes very costly or impractical to conduct. Analysis of the historical data over a ten-year period of these systems shows that there is an increase in the usage of both calls using function pointers and virtual method over the lifetime the systems, thus posing further problems for inter-procedural analysis.

Innovations in hardware architecture, like hyper-threading or multicore processors, mean that parallel computing resources are available for inexpensive desktop computers. In only a few years, many standard software products will be based on concepts of parallel programming implemented on such hardware, and the range of applications will be much broader than that of scientific computing, up to now the main application area for parallel computing. Rauber and Rünger take up these recent developments in processor architecture by giving detailed descriptions of parallel programming techniques that are necessary for developing efficient programs for multicore processors as well as for parallel cluster systems and supercomputers. Their book is structured in three main parts, covering all areas of parallel computing: the architecture of parallel systems, parallel programming models and environments, and the implementation of efficient application algorithms. The emphasis lies on parallel programming techniques needed for different architectures. For this second edition, all chapters have been carefully revised. The chapter on architecture of parallel systems has been updated considerably, with a greater emphasis on the architecture of multicore systems and adding new material on the latest developments in computer architecture. Lastly, a completely new chapter on general-purpose GPUs and the corresponding programming techniques has been added. The main goal of the book is to present parallel programming techniques that can be used in many situations for a broad range of application areas and which enable the reader to develop correct and efficient parallel programs. Many examples and exercises are provided to show how to apply the techniques. The book can be used as both a textbook for students and a reference book for professionals. The material presented has been used for courses in parallel programming at different universities for many years.

The Data Parallel Programming Model

9th International Symposium, APPT 2011, Shanghai, China, September 26-27, 2011, Proceedings Language Support, Automatic Parallelization, Advanced Optimization, and Runtime Systems

Semi-automatic parallelization of FORTRAN programs

Automatic Parallelization Techniques for Massively Parallel Machines

Massively parallel processing is currently the most promising answer to the quest for increased computer performance. This has resulted in the development of new programming languages and programming environments and has stimulated the design and production of massively parallel supercomputers. The efficiency of concurrent computation and input/output essentially depends on the proper utilization of specific architectural features of the underlying hardware. This book focuses on development of runtime systems supporting execution of parallel code and on supercompilers automatically parallelizing code written in a sequential language. Fortran has been chosen for the presentation of the material because of its dominant role in high-performance programming for scientific and engineering applications.

A comprehensive overview of OpenMP, the standard application programming interface for shared memory parallel computing—a reference for students and professionals. "I hope that readers will learn to use the full expressibility and power of OpenMP. This book should provide an excellent introduction to beginners, and the performance section should help those with some experience who want to push OpenMP to its limits." —from the foreword by David J. Kuck, Intel Fellow, Software and Solutions Group, and Director, Parallel and Distributed Solutions, Intel Corporation OpenMP, a portable programming interface for shared memory parallel computers, was adopted as an informal standard in 1997 by computer scientists who wanted a unified model on which to base programs for shared memory systems. OpenMP is now used by many software developers; it offers significant advantages over both hand-threading and MPI. Using OpenMP offers a comprehensive introduction to parallel programming concepts and a detailed overview of OpenMP. Using OpenMP discusses hardware developments, describes where OpenMP is applicable, and compares OpenMP to other programming interfaces for shared and distributed memory parallel architectures. It introduces the individual features of OpenMP, provides many source code examples that demonstrate the use and functionality of the language constructs, and offers tips on writing an efficient OpenMP program. It describes how to use OpenMP in full-scale applications to achieve high performance on large-scale architectures, discussing several case studies in detail, and offers in-depth troubleshooting advice. It

explains how OpenMP is translated into explicitly multithreaded code, providing a valuable behind-the-scenes account of OpenMP program performance. Finally, Using OpenMP considers trends likely to influence OpenMP development, offering a glimpse of the possibilities of a future OpenMP 3.0 from the vantage point of the current OpenMP 2.5. With multicore computer use increasing, the need for a comprehensive introduction and overview of the standard interface is clear. Using OpenMP provides an essential reference not only for students at both undergraduate and graduate levels but also for professionals who intend to parallelize existing codes or develop new parallel programs for shared memory computer architectures.

Compiling for parallelism is a longstanding topic of compiler research. This book describes the fundamental principles of compiling "regular" numerical programs for parallelism. We begin with an explanation of analyses that allow a compiler to understand the interaction of data reads and writes in different statements and loop iterations during program execution. These analyses include dependence analysis, use-def analysis and pointer analysis. Next, we describe how the results of these analyses are used to enable transformations that make loops more amenable to parallelization, and discuss transformations that expose parallelism to target shared memory multicore and vector processors. We then discuss some problems that arise when parallelizing programs for execution on distributed memory machines. Finally, we conclude with an overview of solving Diophantine equations and suggestions for further readings in the topics of this book to enable the interested reader to delve deeper into the field. Table of Contents: Introduction and overview / Dependence analysis, dependence graphs and alias analysis / Program parallelization / Transformations to modify and eliminate dependences / Transformation of iterative and recursive constructs / Compiling for distributed memory machines / Solving Diophantine equations / A guide to further reading

Symbolic Analysis Techniques for Effective Automatic Parallelization

Automatic Parallelization of Nested Loop Programs for Non-manifest Real-time Stream Processing Applications

The Charm++ Approach

Foundations, HPF Realization, and Scientific Applications

Automatic Parallelization for a Class of Regular Computations

Automatic introduction of OpenMP for sequential applications has attracted significant attention recently because of the proliferation of multicore processors and the simplicity of using OpenMP to express parallelism for shared-memory systems. However, most previous research has only focused on C and Fortran applications operating on primitive data types. Modern applications using high-level abstractions, such as C++ STL containers and complex user-defined class types, are largely ignored due to the lack of research compilers that are readily able to recognize high-level object-oriented abstractions and leverage their associated semantics. In this paper, we use a source-to-source compiler infrastructure, ROSE, to explore compiler techniques to recognize high-level abstractions and to exploit their semantics for automatic parallelization. Several representative parallelization candidate kernels are used to study semantic-aware parallelization strategies for high-level abstractions, combined with extended compiler analyses. Preliminary results have shown that semantics of abstractions can help extend the applicability of automatic parallelization to modern applications and expose more opportunities to take advantage of multicore processors.

Abstract: "This paper discusses the techniques used to hand- parallelize, for the Alliant FX/80, four Fortran programs from the Perfect- Benchmark suite. The paper also includes the execution times of the programs before and after the transformations. The four programs considered here were not effectively parallelized by the automatic translators available to the authors. However, most of the techniques used for hand parallelization, and perhaps all of them, have wide applicability and can be incorporated into existing translators."

Automatic ParallelizationAn Overview of Fundamental Compiler TechniquesMorgan & Claypool Publishers

Parallel Programming

New Approaches to Code Generation, Data Distribution, and Performance Prediction

Automatic Performance Prediction of Parallel Programs

Parallel Science and Engineering Applications

Handbook of Grammatical Evolution

This book constitutes the refereed proceedings of the IFIP International Conference on Network and Parallel Computing, NPC 2005, held in Beijing, China in November/December 2005. The 48 revised full papers and 20 revised short papers presented together with 3 invited papers were carefully selected from a total of 320 submissions. The papers are organized in topical sections on grid and system software, grid computing, peer-to-peer computing, web techniques, cluster computing, parallel programming and environment, network architecture, network security, network storage, multimedia service, and ubiquitous computing.

A complete source of information on almost all aspects of parallel computing from introduction, to architectures, to programming paradigms, to algorithms, to programming standards. It covers traditional Computer Science algorithms, scientific computing algorithms and data intensive algorithms.

This monograph-like book assembles the thorougly revised and cross-reviewed lectures given at the School on Data Parallelism, held in Les Menuires, France, in May 1996. The book is a unique survey on the current status and future perspectives of the currently very promising and popular data parallel programming model. Much attention is paid to the style of writing and complementary coverage of the relevant issues throughout the 12 chapters. Thus these lecture notes are ideally suited for advanced courses or self-instruction on data parallel programming. Furthermore, the book is indispensable reading for anybody doing research in data parallel programming and related areas.

Network and Parallel Computing

Automatic Parallelization of C Programs Using Shared Data-objects [microform]

Automatic Parallelization

Input/Output Intensive Massively Parallel Computing

Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions

The automatic generation of parallel code from high level sequential description is of key importance to the wide spread use of high performance machine architectures. This text considers (in detail) the theory and practical realization of automatic mapping of algorithms generated from systems of uniform recurrence equations (do-lccps) onto fixed size architectures with defined communication primitives. Experimental results of the mapping scheme and its implementation are given.

Distributed-memory multiprocessing systems (DMS), such as Intel's hypercubes, the Paragon, Thinking Machine's CM-5, and the Meiko Computing Surface, have rapidly gained user acceptance and promise to deliver the computing power required to solve the grand challenge problems of Science and Engineering. These machines are relatively inexpensive to build, and are potentially scalable to large numbers of processors. However, they are difficult to program: the non-uniformity of the memory which makes local accesses much faster than the transfer of non-local data via message-passing operations implies that the locality of algorithms must be exploited in order to achieve acceptable performance. The management of data, with the twin goals of both spreading the computational workload and minimizing the delays caused when a processor has to wait for non-local data, becomes of paramount importance. When a code is parallelized by hand, the programmer must distribute the program's work and data to the processors which will execute it. One of the common approaches to do so makes use of the regularity of most numerical computations. This is the so-called Single Program Multiple Data (SPMD) or data parallel model of computation. With this method, the data arrays in the original program are each distributed to the processors, establishing an ownership relation, and computations defining a data item are performed by the processors owning the data.

Introduction to Parallel Computing

Manual Versus Automatic Parallelization Using Parallel Virtual Machine and High Performance FORTRAN

Automatic Parallelization of a Crystal Growth Simulation Program for Distributed-memory Systems

Portable Shared Memory Parallel Programming

Automatic Parallelization of Programs