Lecture Notes in Computer Science 1808
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Santosh Pande Dharma P. Agrawal (Eds.)

Compiler Optimizations for Scalable Parallel Systems

Languages, Compilation Techniques, and Run Time Systems

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Santosh Pande
Georgia Institute of Technology, College of Computing
801 Atlantic Drive, Atlanta, GA 30332, USA
E-mail: santosh@cc.gatech.edu

Dharma P. Agrawal
University of Cincinnati, Department of ECECS
P.O. Box 210030, Cincinnati, OH 45221-0030, USA
E-mail: dpa@ececs.uc.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Compiler optimizations for scalable parallel systems : languages, compilation techniques, and run time systems / Santosh Pande ; Dharma P. Agrawal (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; 1808)
ISBN 3-540-41945-4

CR Subject Classification (1998): D.3, D.4, D.1.3, C.2, F.1.2, F.3

ISSN 0302-9743
ISBN 3-540-41945-4 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10720238 06/3142 5 4 3 2 1 0

Preface

Santosh Pande[1] and Dharma P. Agrawal[2]

[1] College of Computing
801 Atlantic Drive,
Georgia Institute of Technology,
Atlanta, GA 30332
[2] Department of ECECS, ML 0030,
PO Box 210030,
University of Cincinnati,
Cincinnati, OH 45221-0030

We are very pleased to publish this monograph on Compiler Optimizations for Scalable Distributed Memory Systems. Distributed memory systems oer a challenging model of computing and pose fascinating problems regarding compiler optimizations ranging from language design to run time systems. Thus, the research done in this area serves as foundational to many challenges from memory hierarchy optimizations to communication optimizations encountered in both stand-alone and distributed systems. It is with this motivation that we present a compendium of research done in this area in the form of this monograph.

This monograph is divided into ve sections : section one deals with languages, section two deals with analysis, section three with communication optimizations, section four with code generation, and section ve with run time systems. In the editorial we present a detailed summary of each of the chapters in these sections.

We would like to express our sincere thanks to many who contributed to this monograph. First we would like to thank all the authors for their excellent contributions which really make this monograph one of a kind; as readers will see, these contributions make the monograph thorough and insightful (for an advanced reader) as well as highly readable and pedagogic (for students and beginners). Next, we would like to thank our graduate student Haixiang He for all his help in organizing this monograph and for solving latex problems. Finally we express our sincere thanks to the LNCS Editorial at Springer-Verlag for putting up with our schedule and for all their help and understanding. Without their invaluable help we would not have been able to put this monograph into its beautiful nal shape!!! We sincerely hope the readers nd the monograph truly useful in their work be it further research or practice.

Introduction

Santosh Pande[1] and Dharma P. Agrawal[2]

[1] College of Computing
801 Atlantic Drive,
Georgia Institute of Technology,
Atlanta, GA 30332
[2] Department of ECECS, ML 0030,
PO Box 210030,
University of Cincinnati,
Cincinnati, OH 45221-0030

1. Compiling for Distributed Memory Multiprocessors

1.1 Motivation

The distributed memory parallel systems offer elegant architectural solutions for highly parallel data intensive applications primarily because:

- They are highly scalable. These systems currently come in a variety of architectures like 3D torus, mesh and hypercube that allow addition of extra processors should the computing demands increase. Scalability is an important issue especially for high performance servers such as parallel video servers, data mining and imaging applications.
- With increase in parallelism, there is insignificant degradation in memory performance since memories are isolated and decoupled from direct accesses from processors. This is especially good for data intensive applications such as parallel databases and data mining that demand considerable memory bandwidths. In contrast, the memory bandwidths may not match the increase in number of processors in shared memory systems. In fact, the overall system performance may degrade due to increased memory contention. This in turn jeopardizes scalability of application beyond a point.
- Spatial parallelism in large applications such as Fluid Flow, Weather Modeling and Image Processing, in which the problem domains are perfectly decomposable, is easy to map on these systems. The achievable speedups are almost linear and this is primarily due to fast accesses to the data maintained in local memory.
- The interprocessor communication speeds and bandwidths have dramatically improved due to very fast routing. The performance ratings offered by newer distributed memory systems have improved although they are not comparable to shared memory systems in terms of Mflops.
- Medium grained parallelism can be effectively mapped onto the newer systems like the Meiko CS-2, Cray T3D, IBM SP1/SP2 and EM4 due to a

low ratio of communication/computation speeds. Communication bottleneck has decreased compared with earlier systems and this has opened up parallelization of newer applications.

1.2 Complexity

However, programming distributed memory systems remains very complex. Most of the current solutions mandate that the users of such machines must manage the processor allocation, data distribution and inter-processor communication in their parallel programs. Programming these systems for achieving the desired high performance is very complex. In spite of frantic demands by programmers, current solutions provided by (semi-automatic) parallelizing compilers are rather constrained. As a matter of fact, for many applications the only practical success has been through hand parallelization of codes with communication managed through MPI. In spite of a tremendous amount of research in this area, applicability of many of the compiler techniques remains rather limited and the achievable performance enhancement remains less than satisfactory. The main reasons for the restrictive solutions offered by parallelizing compilers is the enormous complexity of the problem. Orchestrating computation and communication by suitable analysis and optimizing their performance through judicious use of underlying architectural features demands a true sophistication on the part of the compiler. It is not even clear whether these complex problems are solvable within the realm of compiler analysis and sophisticated restructuring transformations. Perhaps they are much deeper in nature and go right into the heart of design of parallel algorithms for such an underlying model of computation.

The primary purpose of this monograph is to provide an insight into current approaches and point to potentially open problems that could have an impact. The monograph is organized in terms of issues ranging from programming paradigms (languages) to effective run time systems.

1.3 Outline of the Monograph

Language design is largely a matter of legacy and language design for distributed memory systems is no exception to the rule. In section I of the monograph we examine three important approaches (one imperative, one object-oriented and one functional) in this domain that have made a significant impact. The first chapter on HPF 2.0 provides an in-depth view of data parallel language which evolved from Fortran 90. They present HPF 1.0 features such as `BLOCK` distribution and `FORALL` loop as well as new features in HPF 2.0 such as `INDIRECT` distribution and `ON` directive. They also point to the complementary nature of MPI and HPF and discuss features such as `EXTRINSIC` interface mechanism. HPF 2.0 has been a major commercial success with many vendors such as Portland Group and Applied Parallel Research providing highly optimizing compiler support which generates

message passing code. Many research issues especially related to supporting irregular computation could prove valuable to domains such as sparse matrix computation etc. The next chapter on Sisal 90 provides a functional view of implicit paralleism specification and mapping. Shared memory implementation of Sisal is discussed, which involves optimizations such as *update in place copy elimination* etc. Sisal 90 and a distributed memory implemenatation which uses message passing are also discussed. Finally multi-threaded implementations of Sisal are discussed, with a focus on multi-threaded optimizations. The newer optimizations which perform memory management in hard-ware through dynamically scheduled multi-threaded code should really prove beneficial for the performance of functional languages (including Sisal) which have an elegant programming model. The next chapter on HPC++ provides an object oriented view as well as details on a library and compiler strategy to support HPC++ level 1 release. The authors discuss interesting features related to multi-threading, barrier synchronization and remote procedure invocation. They also discuss library features that are especially useful for scientific programming. Extensions of this work relating to newer portable languages such as Java is currently an active area of research. We also have a chapter on concurrency models of OO paradigms. The authors specifically address a problem called *inheritance anomaly* which arises when synchronization constraints are implemented within methods of a class and an attempt is made to specialize methods through inheritance mechanisms. They propose a solution to this problem by separating the specification of synchronization from the method specification. The synchronization construct is not a part of the method body and is handled separately. It will be interesting to study the compiler optimizations on this model related to strength reduction of barriers, and issues such as data partitioning vs. barrier synchronizations.

In section II of the monograph, we focus on various analysis techniques. Parallelism detection is very important and the first chapter presents a very interesting comparative study of different loop parallelization algorithms by Allen and Kennedy, Wolf and Lam, Darte and Vivien and by Feautrier. They provide comparisons in terms of their performance (ability to parallelize as well as quality of schedules generated for code generation) as well as complexity. The comparison also focusses on the type of dependence information available. Further extensions could involve run-time parallelization given more precise dependence information. Array data-flow is of utmost importance in optimizations : both sequential as well as parallel. The first chapter on array data-flow analysis examines this problem in detail and presents techniques for exact data flow as well as for approximate data flow. The exact solution is shown for static control programs. Authors also show applications to interprocedural cases and some important parallelization techniques such as privatization. Some interesting extensions could involve run-time data flow analysis. The next chapter discusses interprocedural analysis based on guarded (predicated) array regions. This is a framework based on path-sensitive predi-

cated data-flow which provides summary information. The authors also show application of their work to improve array privatization based on symbolic propagation. Extensions of these to newer object oriented languages such as Java (which have clean class hierarchy and inheritance model) could be interesting since these programs really need such summary MOD information for performing any optimization. We finally present a very important analysis/optimization technique for array privatization. Array privatization involves removing memory-related dependences which have a significant impact on communication optimizations, loop scheduling etc. The authors present a demand-driven data-flow formulation of the problem; an algorithm which performs single pass propagation of symbolic array expressions is also presented. This comprehensive framework implemented in a Polaris compiler is making a significant impact in improving many other related optimizations such as load balancing, communication etc.

The next section is focussed on communication optimization. The communication optimization can be achieved through data (and iteration space) distribution, statically or dynamically. These approaches further classify into data and code alignment or simply interation space transformations such as in tiling. The communication can also be optimized in data-parallel programs through array region analysis. Finally one could tolerate some communication latency through novel techniques such as multi-threading. We have chapters which cover these broad range of topics about communication in depth.

The first chapter in this section focusses on tiling for cache-coherent multicomputers. This work derives optimal tile parameters for minimal communication in loops with affine index expressions. The authors introduce a notion of data footprints and tile the iteration spaces so that the volume of communication is minimized. They develop an important lattice theoretic framework to precisely determine the sizes of data footprints which are very valuable not only in tiling but in many array distribution transformations. The next two chapters deal with the important problem of communication free loop partitioning.

The second chapter in this section focusses on comparing different methods of achieving communication-free partitioning for DOALL loops. This chapter discusses several variants of the communication-free partitioning problem involving duplication or non-duplication of data, load balancing of iteration space and aspects such as statement level vs. loop level partitioning. Several aspects such as trading parallelism to avoid inter-loop data distribution are also touched upon. Extending these techniques to broader classes of DOALL loops could enhance their applicability.

The next chapter by Pingali et al. proposes a very interesting framework which first determines a set of constraints on data and loop iteration placement. They then determine which constraints should be left unsatisfied to relax an overconstrained system to find a solution involving a large amount of parallelism. Finally, the remaining constraints are solved for data and code

distribution. The systematic linear algebraic framework improves over many ad-hoc loop partitioning approaches.

These approaches trade parallelism for codes that allow *decoupling* the issues of parallelism and communication by relaxing an appropriate constraint of the problem. However, for many important problems such as image processing applications such a relaxation is not possible. That is, one must resort to a different partitioning solution based on relative costs of communication and computation. In the next chapter, for solving such a problem, a new approach has been proposed to partition iteration space by determining directions which maximally cover the communication by minimally trading parallelism. This approach allows mapping of general medium grained DOALL loops. However, the communication resulting from this iteration space partitioning can not be easily aggregated without sophisticated 'pack'/'unpack' mechanisms present at send/receive ends. Such extensions are desirable since aggregating communication has as significant impact as reducing the volume.

The static data distribution and alignment typically solve the problems of communication on a loop nest by loop nest basis but rarely in an intraprocedural scope. Most of the inter-loop nest level and interprocedural boundaries require dynamic data redistribution. Banerjee et al. develop techniques that can be used to automatically determine which data partitions are most beneficial over specific sections of the program by accounting for redistribution overhead. They determine split points and phases of communication and redistribution are performed at split points.

When communication must take place, it should be optimized. Also, any redundancies must be captured and eliminated. Manish Gupta in the next chapter proposes a comprehensive approach for performing global (interprocedural) communication optimizations such as vectorization, PRE, coalescing, hoisting etc. Such an interprocedural approach to communication optimization is highly profitable in substantially improving the performance. Extending this work to irregular communication could be interesting.

Finally, we present a multi-threaded approach which could hide the communication latency. Two representative applications involving bitonic sort and FFT are chosen and using fine grained multi-threading on EM-X it is shown that multi-threading can substantially help in overlapping computation with communication to hide latencies up to 35 %. These methods could be especially useful for irregular computation.

The final phase of compiling for distributed memory systems involves solving many code generation problems. Code generation problems involve, determining communication generation and doing address calculation to map global references to local ones. The next section deals with these issues. The first chapter presents structures and techniques for communication generation. They focus on issues such as flexible computation partitioning (going beyond owner computes rule), communication adaptation based upon manipulating integer sets through abstract inequalities and control flow simpli-

cation based on these. One good property of this work is that it can work with many di erent front ends (not just data parallel languages) and the code generator has more opportunities to perform low level optimizations due to simpli ed control ow.

The second chapter discusses basis vector based address calculation mechanisms for e cient traversals of partitioned data. While one important issue of code generation is communication generation, a very important issue is to map global address space to local address space e ciently. The problem is complicated due to data distributions and access strides. Ramanujam et al. present closed form expressions for basis vectors for several cases. Using the closed form expressions for the basis vectors, they derive a non-unimodular linear transformation.

The nal section is on supporting task parallelism and dynamic data structures. We also present a run-time system to manage irregular computation. The rst chapter by Darbha et al. presents a task scheduling approach that is optimal for many practical cases. The authors evaluate its performance for many practical applications such as the Bellman-Ford algorithm, Cholesky decomposition, the Systolic algorithm etc. They show that schedules generated by their algorithm are optimal for some cases and near optimal for most others. With HPF 2.0 supporting task parallelism, this could open up many new application domains.

The next two chapters describe language supports for dynamic data structures such as pointers in distributed address space. Gupta describes several extensions to C with declarations such as *TREE, ARRAY, MESH* to declare dynamic data structures. He then describes name generation and distribution strategies for name generation and distribution strategies. Finally he describes support for both regular as well as irregular dynamic structures. The second chapter by Rogers et al. presents an approach followed in their *Olden* project which uses a distributed heap. The remote access is handled by software caching or computation migration. The selection of these mechanisms is done automatically through a compile time heuristic. They provide a data layout annotation to the programmer called *local path lengths* which allows programmers to give hints regarding expected data layout thereby x-ing these mechanisms. Both of these chapters provide highly useful insights into supporting dynamic data strutures which are very important for scalable domains of computation supported by these machines. Thus, these works should have a signi cant impact on future scalable applications supported by these systems.

Finally, we present a run-time system called *CHAOS* which provides e - cient support for irregular computations. Due to indirection in many sparse matrix computations, the communication patterns are unknown at compile time in these applications. Indirection patterns have to be preprocessed, and the sets of elements to be sent and received by each processor precomputed,

in order to optimize communication. In this work, the authors provide details of e cient run time support for an *inspector executor* model.

1.4 Future Directions

The two important bottlenecks for the use of distributed memory systems are the limited application domains and the fact that the performance is less than satisfactory. The main bottleneck seems to be handling communication. Thus, e cient solutions must be developed. Application domains beyond regular communication can be handled by supporting a general run-time communication model. This run-time communication model must be latency hiding and should give su cient exibility to the compiler to defer the hard decisions to run time yet allow static optimizations involving communication motion etc. One of the big problems compilers face is that estimating cost of communication is almost impossible. They can however gauge criticality (or relative importance) of communication. Developing such a model will allow compilers to more e ectively deal with issues of *relative* importance betwen computation and communication and communication and communication.

Probably *the* best reason to use distributed memory systems is to bene t from *scalability* even though application domains and performance might be somewhat weaker. Thus, new research must be done in *scalable code generation*. In other words, as size of the problem and number of processors increase, should there be a change in data/code partition or should it remain the same? What code generation issues are related to this? How could one potentially handle the hot spots that inevitably (although at much lower levels than shared memory systems) arise? Can one bene t from the above communication model and dynamic data ownerships discussed earlier?

Table of Contents

Section II : Analysis

Section III : Communication Optimizations