Dagstuhl Seminar
on
Loop Parallelization

Organized by

Christian Lengauer (Universität Passau)
Lothar Thiele (ETH Zürich)
Michael Wolfe (Oregon Graduate Institute)
Hans Zima (Universität Wien)

Schloß Dagstuhl 15. – 19.4.1996

Contents

1 Preface

As parallelism emerges as a viable and important concept of computer technology, the automatic parallelization of loops is becoming increasingly important and receiving increased attention from researchers. The reasons are (1) that programming parallel computers by hand is impractical in all but the simplest applications and (2) that, by exploiting the parallelism in nested loops, potentially large numbers of processors can be utilized easily and a speed-up of orders of magnitude can be attained.

Methods of loop parallelization have been developed in two research communities: regular array design and parallelizing compilation.
Researchers in regular array design impose regularity constraints on loop nests in order to apply a geometric model in which the set of all parallelizations of the source loop nest can be characterized, the quality of each member of the set can be assessed and an optimal choice from the set can be made automatically. That is, regular array design methods identify optimal parallelizations of regular loop nests.
Researchers in parallelizing compilation are interested in faster methods than their colleagues in regular array design and, therefore, often apply heuristics to attain reasonable but not necessarily optimal parallelizations. Parallelizing compilation methods can often cope with less regular loop nests but do, in general, not produce provably optimal results.

The primary goal of this seminar was to intensify communication between the two communities. In recent years, the methods used in both communities have increasingly converged on the theory of linear algebra and linear programming.

Questions discussed at the seminar included:

- Algorithms that yield optimal parallelizations are usually computationally complex. For what applications is this (not) a serious restriction? For what applications do heuristic algorithms yield better performance, and what are the heuristics?
- Loop parallelization methods have yielded static parallelism in the past. How can they be made more dynamic, e.g., for the treatment of while loops or of irregular data structures?
- What parallel programmable computer architectures should the research in loop parallelization aim at?
- What do the users of parallelizing compilers expect from loop parallelization?

- What are the special requirements on design methods for multi-media applications?
- How can memory management in parallelized programs be made more efficient?

The 41 participants of the workshop came from 9 countries: 14 from the US (funded by the National Science Foundation), 10 from France, 9 from Germany and 8 from other European countries. The organizers would like to thank everyone who has helped to make this workshop a success.

Christian Lengauer Lothar Thiele Michael Wolfe Hans Zima

2 Abstracts

The Omega Library

William Pugh
University of Maryland at College Park, USA

The Omega library provides routines for analyzing constraints over integer variables built using linear constraints, the usual logical connectives (and, or, not) and universal and existential quantification. This is Presburger arithmetic; the tightest upper bound on the complexity of verifying the satisfiability of Presburger arithmetic is $2^{2^{2^{O(n)}}}$.
The Omega library is targeted at problems that arise in the context of analyzing and transforming scientific programs for execution on parallel computers and within that context is generally fast. It has also found some use in other domains (such as analyzing the requirements for the A7 attack fighter).
We describe the capabilities of and interface to the Omega library, discuss some of the design decisions we made in creating that interface, and give a high-level overview of how the Omega library operates.

Compiler Techniques for Multiprocessors and the Polaris Restructurer

David Padua
The University of Illinois at Urbana-Champaign, USA

Multiprocessor computers are rapidly becoming the norm. Parallel workstations are widely available today and it is likely that most PCs in the near future will also be parallel. Some classes of applications will have to be developed in explicitly parallel form. Yet, in order to avoid a substantial increase in software development costs, compilers to translate conventional programs into efficient parallel form will clearly be necessary. In the ideal case, multiprocessor parallelism should be as transparent to programmers as functional level parallelism is to programmers of today's superscalar machines. However, compiling for multiprocessors is substantially more complex than compiling for functional unit parallelism, in part because successful parallelization often requires a very accurate analysis of long sections of code.
We discuss our recent experience at Illinois on the automatic parallelization of scientific codes. New techniques for dependence analysis, idiom recognition, and

privatization have been developed in recent years based on an extensive analysis of the characteristics of real Fortran codes. These techniques, which are based on both static and dynamic parallelization strategies, have been incorporated in Polaris, a source-to-source Fortran restructurer developed at the University of Illinois. Polaris accepts Fortran 77 with vector extensions and generates programs in parallel Fortran dialects including those of SGI, Sun, and the Cray T3D, as well as *step, an implementation of the last version of the ANSI X3H5 standard. Polaris is implemented in C++ around a class hierarchy representing the source program internally. Preliminary results of the effectiveness of Polaris on parallel workstations are encouraging and we expect that, once the implementation of the new techniques is complete, Polaris will be able to obtain good speedups for most scientific codes on parallel workstations.

The Loop Parallelizer LooPo

Martin Griebl
University of Passau, Germany

We report on a prototype for testing different methods of space-time mapping loop nests. LooPo admits perfect or imperfect loop nests in a number of imperative languages, takes data dependences from the user or derives them itself from the source code, provides a choice of strategies for scheduling and allocating the loop nest's iterations, and produces synchronous or asynchronous parallel target code for shared-memory or distributed-memory machines.

Optimal Loop Parallelization under Register Constraints

Christine Eisenbeis and Antoine Sawaya
INRIA-Rocquencourt, Le Chesnay, France

We deal with the interaction between instruction scheduling and register allocation, in the case of straight line code and in the case of loops. This problem is at the heart of code optimization in microprocessors with instruction-level parallelism. Usual solutions use heuristics based on a decoupled approach. We propose here a formulation via linear integer programming that allows dependence, resource and register constraints to be integrated in the same framework. By varying the parameters, all kinds of optimization problems can be solved exactly (maximization of the throughput, minimization of the number of registers).

We report on examples of computation timings that turn out to be prohibitive in some specific cases, but tractable on average.

Automatic Performance Analysis for SVM-Fortran Programs

Michael Gerndt
KFA Jülich, Germany

Programming massively parallel machines is much simplified by using a high-level programming language providing a global view on data and computation. The most well known example is High Performance Fortran. HPF compilers generate SPMD code with explicit message passing. If the compiler is unable to identify access patterns, such as in irregular grid applications, the generated code has to rely on costly run-time analysis.
Another approach is to combine the strength of hardware and software technology to solve the problem of mapping high-level languages to distributed memory machines. One can implement a global address space on the system level, as is done with shared virtual memory where remote accesses are automatically resolved via data migration among the different memories.
SVM-Fortran is a high-level language based on task and data parallelism. The user can enforce data locality with respect to the local memories by specifying a work distribution enforcing reuse of local data. Although it is easy to parallelize programs with SVM-Fortran according to a high-level parallelization strategy, i.e., domain decomposition, it is necessary to optimize the program carefully. This optimization relies on performance information.
The SVM-Fortran programming environment provides a new approach for tracing performance data: selective tracing based on a combination of compile time and run time instrumentation. Selective tracing enables an incremental performance analysis cycle, starting from coarse information and gathering more and more precise information. The process of incremental parallelization can be automated based on rules describing the information required and appropriate predicates to proof the bottlenecks.
There are two classes of rules: refinement rules and proof rules. Refinement rules handle the problem of making proven hypotheses more precise, for example, refining the statement that a locality bottleneck exists into region-specific hypotheses that a locality problem exists in that region. Proof rules instead are used to prove a hypothesis.
The current set of hypotheses in one step of the performance analysis cycle requires information determined by the applicable set of proof rules. This leads to the second important aspect of automatic performance analysis, the synthesis of

new instrumentation requests. This synthesis has to take into account the execution time of the program, the amount of required information and the possible intrusion.
The presentation outlines the structure of an automatic performance analysis tool for SVM-Fortran programs consisting of selective tracing, hypothesis refinement and instrumentation synthesis.

Parallelizing Nested Loops with Approximation of Distance Vectors: A Survey

Alain Darte and Frédéric Vivien
LIP, ENS Lyon, France

We compare three nested loop parallelization algorithms (Allen and Kennedy's algorithm, Wolf and Lam's algorithm and Darte and Vivien's algorithm) that use different representations of distance vectors as input. We study the optimality of each with respect to the dependence analysis it uses. We propose well chosen examples that illustrate the power and limitations of the three algorithms. This study permits to identify which algorithm is the most suitable for a given representation of dependences.

New Algorithms for Address Generation for HPF-Style Mappings

Arun Venkatachar and Jagannathan Ramanujam
Louisiana State University, Baton Rouge, Louisiana, USA

Two new algorithms are discussed for address generation for one-level and two-level mappings for distributed address space machines.
The first algorithm is shown to have a complexity of $O(\log(\min(s, pk)))$, the same as that of the complexity of computing GCD. Here s is the stride, k is the block size and p is the number of processors. The address generation problem is viewed as an integer lattice problem and then basis vectors are computed in order to scan the lattice to enumerate elements in lexicographic order. The idea of this algorithm is to generate these basis vectors in $O(\log(\min(s, pk)))$ time. The time taken to compute these vectors is markedly faster than the time taken by other existing methods to solve the same problem. The results are compared to those of the algorithms developed at Rice University and our own previous work. Methods to compute the starting elements on each processor in $O(k)$ time are also discussed.

The second algorithm talks about generating addresses for the case of a two-level mapping. In this case, the alignment factor is greater than 1. A new method of solving this problem is to keep track of the offsets of the allocated elements and then reusing them to compute the new set of accessed elements. Results show that this method is about 6 to 10 times faster than the approach used by RIACS.

Modular Mappings of Loop Nests

Hyuk Jae Lee and Jose A. B. Fortes
Purdue University, West Lafayette, Indiana, USA

Many optimizations (of programs with loops) used in parallelizing compilers and systolic array design are based on linear transformations of loop iteration spaces. Additional important optimizations and designs are possible by using recently proposed modular mappings, which are described by linear transformations modulo a constant vector. We identify and characterize a class of (BLAS-like) algorithms that can be optimized for parallel execution by modular mappings. A formal technique to derive optimal modular schedules (i.e., the time component of a modular mapping) is provided. Subsequently, conditions that guarantee the injectivity of a modular mapping (i.e., a modular schedule plus a modular processor allocation) are discussed and techniques are provided to generate efficiently constrained injective modular mappings. Given a BLAS-like algorithm with a nested loop of depth n, the complexity of these generation techniques is $O(n^{2n!})$. We also propose a new class of data alignments, called expanded modular data alignments (EMDAs), for algorithms that are mapped by modular time-space transformations. An EMDA subsumes multiple modular data alignments (MDAs) which are described by affine functions modulo a constant vector. Conditions for perfect alignment between a modular time-space mapping and an EMDA are provided. However, these conditions together with other conditions discussed above introduce non-linear constraints in the problem of generating modular mappings. A method with $O(n^2)$ complexity is provided to choose some entries of a transformation matrix so that non-linear constraints are transformed into linear ones. Although the solution space of the problem is reduced by assigning fixed values to some entries, the proposed heuristic attempts to reduce the number of the fixed entries and exclude as few solutions as possible.
We also consider the issue of deriving the inverse transformation of a given modular mapping. We identify a class of modular functions whose inverses result directly from computing the inverse of the (coefficient) matrix used to specify a modular mapping. An efficient method with $O(n^2)$ complexity is provided to formulate the problem of generating such modular mappings as an integer linear programming problem.

Transforming Imperfectly Nested Loops

Induprakas Kodukula and Keshav Pingali
Cornell University, Ithaca, New York, USA

Loop transformations are critical for compiling high-performance code for modern computers. Existing work has focused on transformations for perfectly nested loops (that is, loops in which all assignment statements are contained within the innermost loop of a loop nest). In practice, most loop nests, such as those in matrix factorization codes, are imperfectly nested. In some programs, imperfectly nested loops can be transformed into perfectly nested loops by loop distribution, but this is not always legal. We present an approach to transforming imperfectly nested loops directly. Our approach is an extension of the linear loop transformation framework for perfectly nested loops, and it models permutation, reversal, skewing, scaling, alignment, distribution and jamming. We also give a completion procedure which generates a complete transformation from a partial transformation.

On the Removal of Anti and Output Dependences

Pierre-Yves Calland, Alain Darte,
Yves Robert and Frédéric Vivien
LIP, ENS Lyon, France

We build upon results of Padua and Wolfe (1986), who introduced two graph transformations to eliminate anti and output dependences. First, we give a unified framework for such transformations. Then, given a loop nest, we aim at determining which statements should be transformed so as to break artificial cycles involving anti or output dependences. The problem of finding the minimum number of statements to be transformed is shown to be NP-complete in the strong sense, and we propose two efficient heuristics.

Memory Reuse Analysis in the Polyhedral Model

Sanjay Rajopadhye and Doran Wilde
IRISA, Rennes, France, and Brigham Young University, Provo, Utah, USA

We address the problem of compiling programs expressed in the polyhedral model (systems of affine recurrences over polyhedral domains). Such languages, being functional, are inherently single assignment. Some parallelizing compilers which use the polyhedral model also use an intermediate single assignment form due to array expansion. We present a static analysis method that enables a compiler to generate multiple assignment (and hence memory-efficient) code. First, we give an algorithm to determine the "usage table", a key piece of information that specifies the set of index points at which a particular value is used. Based on the usage table, we develop an analysis method that gives necessary and sufficient conditions under which an index domain can be projected so that memory can be reused.

Static Analysis to Reduce Synchronization Costs in Data-Parallel Programs

Manish Gupta
IBM T. J. Watson Research Center, Yorktown Heights, New York, USA

For a program with sufficient parallelism, reducing synchronization costs is one of the most important objectives for achieving efficient execution on any parallel machine. We present a novel methodology for reducing synchronization costs of programs compiled for SPMD execution. This methodology combines data flow analysis with communication analysis to determine the ordering between production and consumption of data on different processors, which helps in identifying redundant synchronization. The resulting framework is more powerful than any that have been previously presented, as it provides the first algorithm that can eliminate synchronization messages even from computations that need communication. We show that several commonly occurring computation patterns, such as reductions and stencil computations with reciprocal producer-consumer relationship between processors, lend themselves well to this optimization, an observation that is confirmed by an examination of some HPF benchmark programs. Our

framework also recognizes situations where the synchronization needs for multiple data transfers can be satisfied by a single synchronization message. This analysis, while applicable to all shared memory machines as well, is especially useful for those with a flexible cache-coherence protocol, as it identifies efficient ways of moving data directly from producers to consumers, often without any extra synchronization.

Systolic Parallelization of Programs with Multiple Loops

Friedrich Wichmann
University of Paderborn, Germany

The presented approach tries to combine systolic parallelized loop nests by use of a set of communication patterns to generate a parallel program. This *loop combination model* is targeted at numerical programs containing sequences of loops or loop nests and assignment statements. The systolic parallelization technique is applied to regular loop nests in order to get different possible parallel implementations. Their input/output data builds the interface between subsequent loop nests.
The loop nests are accessing data elements regularly. If a systolic space-time transformation is applied to the *access pattern*, the positions of data elements can be described. Qualitative features of the possible *position patterns* for regular systolic loops have been described. Stationary data elements can have different displacements, directions (forward/backward for one-dimensional arrays), "holes" in processor usage, or skip of some data element indices. Pipelined data elements can differ additionally in timing. Thus, their starting or final position can have offsets from the processor boundary, their speed is determined by "delays" in the communication between processors or "slow", i.e., the occurrence of unused places in the data stream. Also, broadcast input or tree-structured output from reductions can be considered. The data element accesses can be analyzed and described qualitatively (cf. "utilization sets"), and after applying the space-time transformation this yields an exact description of position patterns.
Finally, the possible pairs of position pattern types are looked at to give optimized *communication patterns* for the transport of data elements across the interface between two subsequent loop nests. These can be shift operations for displacements and offsets, a shift "mirrored" at the end of the array for turning the direction, and gathering/scattering to deal with holes. Mirroring can also be used to turn stationary into pipelined data (or vice versa in the symmetric case). It is possible to deal with part of the *reorganization* of data elements in parallel to the computation phase of the loop nests. If there is no such opportunity, as

for pipelined data with different speed, the reorganization code has to transport the data in an extra phase ("default solution").
The model and strategy presented can be used for programming real parallel machines if one estimates the costs of different means of communication. The selection of parallel solutions for the loop nests and the necessary communication patterns is planned to be done iteratively, driven by parameters like the dimension of loop nests and data. The loop combination scheme, which is still going to be realized and evaluated, is related to work from both areas, data parallel loop parallelization and parallel task optimization, as well as the compilation and synthesis of regular systolic arrays.

Compilation Techniques for Sparse Matrix Computations

Harry Wijshoff
Leiden University, The Netherlands

The problem of compiler optimizations for sparse matrix codes is well known and no satisfactory solutions have been found yet. One of the major obstacles is caused by the fact that sparse matrix programs deal explicitly with the particular data structures selected for storing sparse matrices. This explicit data structure handling obscures the functionality of a code to such an extent that the optimization of the code is prohibited, i.e., by the introduction of indirect addressing and variables not known at compile time.
We present a method which postpones data structure selection until the compile phase, thereby allowing the compiler to combine code optimization with explicit data structure selection. This method not only enables the compiler to generate efficient code for sparse matrix computations, also the complexity of the programming effort is greatly reduced.

HPF+ – Language and Implementation Support for Irregular Computations

Hans Zima
University of Vienna, Austria

The High Performance Fortran Forum (HPFF), which first convened during 1992, set itself the task of defining language extensions for Fortran to facilitate data parallel programming on a wide range of parallel architectures without sacrificing

performance. Much of the work focussed on extending Fortran 90 by directives for specifying alignment and distribution of a program's data. These enable the programmer to influence the locality of computation by controlling the manner in which the data is mapped to processors. Other major extensions include data parallel constructs, such as the FORALL statement and construct, the INDEPENDENT directive, and a number of library routines.
However, the current version of the language, HPF-1, has not fully achieved the stated goal. While the basic distribution functions offered by the language – regular block, cyclic, and block cyclic distributions – can support regular numerical algorithms, advanced applications such as multiblock codes, particle-in-cell codes or sweeps over unstructured grids require added functionality that cannot be expressed adequately. This is a major weakness of HPF, which until now has significantly reduced its chances of becoming accepted in the numerical community.
We outline the major features of *HPF+*, a language which, on the one hand, eliminates some unnecessary or ill-defined features of HPF-1, and, on the other hand, extends its functionality, with an emphasis towards providing non-standard data and work distributions, much along the lines of the Vienna Fortran language. More specifically, the following set of extensions has been proposed:

- data distribution to processor subsets
- processor views
- general block distributions
- indirect distributions
- user-defined distribution functions, and
- on-clauses for the control of the work distribution in an INDEPENDENT loop.

At this time, the syntax and semantics of HPF+ have been informally defined. In the ESPRIT IV project "HPF+", which is coordinated by the University of Vienna and includes the University of Pavia, NAS Software, and three application developers (ECMWF, Engineering Systems International (ESI), and AVL), a full language specification is being developed and implemented in the framework of the *Vienna Fortran Compilation System*. The results of this work are also being input to the HPF Forum, which currently works on the definition of an HPF-2 language.

Compilation Issues for Irregular Problems

Guillermo P. Trabado and Emilio L. Zapata
University of Malaga, Spain

Many large-scale computational applications contain irregular data access patterns related to unstructured problem domains. Examples include finite element methods, computational fluid dynamics, and molecular dynamics codes. Such codes are difficult to parallelize efficiently with current HPF compilers. However, most of these problems exhibit spatial locality. We review how data is handled in such programs, which problems arise at the time of parallelization and the techniques that a compiler will use to parallelize these codes.
Unordered sets of particles or locations are stored as coordinate lists that can be distributed using Multiple Recursive Decomposition (MRD), a pseudo-regular distribution, which combines efficient implementation with good load balancing and communication behavior. Unstructured domains are accessed via indirection arrays. We introduce a new directive that serves to identify indirection arrays and the boundaries of the associated domains. Indirection arrays are aligned with the data arrays. Using the information provided in the directive, the compiler can produce a target program with significantly better performance than an approach based on indirect distributions and the inspector/executor paradigm.

Foundations of Parallel Speculative Execution

Paul Feautrier and Jean-François Collard
University of Versailles, France

Static scheduling consists of compile-time mapping of operations onto logical execution dates. However, scheduling so far only applies to static control programs, i.e., roughly to nests of DO (or FOR) loops. To extend scheduling to dynamic control programs, one needs a method that (1) is consistent with unpredictable control flows (and thus unpredictable iteration domains), (2) is consistent with unpredictable data flows, and (3) permits speculative execution. We discuss first the several types of dependences which are to be considered in the scheduling process: data dependences and control dependences. Then we show that speculative execution is obtained if one ignores some control dependences when selecting a schedule. To restore program correctness, one has to include compensating dependences. Provided that the schedules are selected in such a way that the total amount of work to be done before any given instant is finite, one can prove that the parallel program terminates and gives correct results. A simple criterion for

the finiteness condition is given. In the conclusion, we point to the many questions which are still unsolved and have to be answered before speculative parallel execution becomes a practical method.

Optimistic Parallel Computation

Damal K. Arvind
University of Edinburgh, Scotland, UK

The correct execution of a parallel program demands that the causal relationships, either due to data or control dependencies, be respected. For extracting greater concurrency in programs, it may be advantageous to be able to speculate on conditional control dependencies: for instance in the case of loops with run-time dependencies or programs which are rich in control structures. This requires additional mechanisms during the program execution for detecting and recovering from causal violations. We describe the fundamentals of optimistic parallel computation and explore the intimate interactions between the compiler and the hardware for supporting this efficiently.

On the Optimal Size and Shape of Supernode Transformation

Edin Hodzic and Weijia Shang
Santa Clara University, California, USA

Supernode transformation has been proposed to reduce the communication startup cost by grouping a number of iterations in a nested loop as a supernode which is assigned to a processor as a single unit. A supernode transformation is specified by n families of hyperplanes, which slice the iteration space into parallelepiped supernodes, the grain size of a supernode, and the relative side lengths of the parallelepiped supernode. The total running time is affected by the three factors. We discuss how to find an optimal grain size and an optimal relative side length vector. Our results show that the optimal grain size is the ratio of the communication startup cost to the computation speed of the processor, and that the optimal supernode shape is similar to the shape of the index space, in the case of hypercube index spaces and supernodes.

Reducing Storage Size for Static Control Programs Mapped onto Parallel Architectures

Eddy De Greef, Francky Catthoor and Hugo De Man
IMEC, Leuven, Belgium

We report new insights in the problem of reducing storage size for static control programs that are being mapped onto several classes of parallel architectures. These insights and the accompanying mathematical descriptions allow us to employ more aggressive data transformations than previously possible. These transformations can result in a considerable reduction of the storage size by reusing memory locations several times. This is especially important for data-intensive algorithms implemented in embedded systems, such as multimedia applications. In these applications, the memory cost is usually dominant, such that a large reduction in memory area is certainly desirable. The presented techniques (almost) do not interfere with traditional parallelization techniques and are in that sense complementary to them. They are embedded in a larger methodology for memory management which focuses on both optimizing transformations and improved memory organization when mapping multimedia applications onto processors.

Optimization of Inspector/Executor Code Generated by Parallelizing Compilers

Thilo Ernst
GMD-FIRST, Berlin, Germany

In compiling irregular codes for distributed-memory architectures, the lack of compile-time knowledge about data access patterns is hampering sophisticated organization of communication. Combined compile-time/run-time analysis techniques, often called *inspector/executor methods*, are known as one of the most promising approaches to attack this problem.
However, if such methods are applied by the compiler to data-parallel assignments (e.g., HPF FORALL) in a simple, local manner, prohibitive performance penalties can be caused by *redundant inspector computations*. Notably, invariant inspector computations occurring in (possibly nested) loops are identified as damaging the performance.
These and other cases can successfully be attacked by applying *partial redundancy elimination* (*PRE*) optimization techniques to the inspector computations under consideration. Such techniques find a more optimal *placement* of the computations in question while satisfying the usual optimization criteria.

However, in contrast to classical optimization, an inspector computation also entails a *memory allocation* for the (possibly huge) *run-time descriptor* in which the result is stored. This gives rise to the question at which point to *deallocate* the memory for each descriptor, now that the descriptor computations/allocations are placed differently.
A simple algorithm based on statement-level *descriptor liveness analysis* is presented to find a safe, sufficient and optimal placement of deallocation primitives based on the computation/allocation placement found by the PRE algorithm.
Finally, preliminary work is presented about integrating a PRE method based on the notion of *districts* with a solution of the deallocation placement problem discussed before.

On Two Examples of the Need for Non-Traditional Optimizations Involving Reductions

P. Sadayappan
Ohio State University, Columbus, Ohio, USA

We present two examples to point out the need for optimizations pertaining to reduction operations – an aspect that has traditionally not been addressed much by the work on parallelizing compilers.
The first example is parallel sparse Cholesky factorization. Several domain-specific ideas used in generating efficient "hand-crafted" parallel codes for this computation are first explained. It is then shown for a sample sparse matrix that a re-constructed directed acyclic graph corresponding to a "hand-crafted" parallel algorithm is in fact different from that corresponding to an efficient sequential sparse Cholesky algorithm. The difference is fundamentally due to changes in the order of application of commutative and associative operations, facilitated by domain-specific heuristics that use deep insight into the application domain. If a general-purpose DAG scheduling heuristic were to be used with the different DAG corresponding to the sequential program, it seems impossible to create the same (or a comparable) schedule as that possible by use of the domain heuristics. This is at least partly due to the fact that, by using the sequential program's DAG, the degree of freedom possible with reordering commutative/associative operations is not exploited.
The second example is motivated by an application in computational physics. There is the potential for a significant reduction in the number of arithmetic operations in a nested loop computation by judicious application of a distributive law to a multi-dimensional summation expression. This kind of optimization involving reduction operations is traditionally not performed by compilers.

Parallel Programming with PEI

Guy-René Perrin
Université Louis Pasteur, Strasbourg, France

A wide range of research work on the static analysis of programs forms the foundation of parallelization techniques which improve the efficiency of codes: loop nest rewriting, directives to the compiler to align or distribute the data or the operations, etc. These techniques are of particular interest in data parallel programming. They are based on geometric transformations either of the iteration space or of the index domains of arrays.
In some sense, this shows that, beside a classical functional point of view on programs, geometric issues in parallel programming or parallelizing compilation have to be considered of main importance for the mastery of efficient computations. This geometric approach entails an abstract manipulation of array indices, to define and transform the data dependences in the program, the way the data are, or are not, locally accessible, their expansion in a multidimensional space of virtual processors, etc. This requires to be able to express, compute and modify the placement of the data and operations in an abstract discrete reference domain. Then, the programming activity may refer to a very small set of primitive issues to construct, transform or compile programs. PEI is a program notation and includes such issues.
We focus on some applications in parallel programming, which are induced by algebraic laws in PEI: parallelization techniques, memory storage optimization, reduction and data alignment in data-parallel languages.

Parallel Loops in High Performance Fortran

Robert Schreiber
HP Labs, Palo Alto, California, USA

High Performance Fortran (HPF) is widely known as a data-parallel dialect of Fortran. In fact, HPF supports more than one means for expressing parallelism. Simple array-based data parallelism is there, but so is a quite general parallel loop construct, a DO loop whose iterations are asserted to be independent. This allows an arbitrary collection of tasks to run in parallel provided, in HPF version 1, that they do not communicate.

Of course, this prohibition is too limiting. In HPF version 2, an additional capability allows loops whose iterations perform reduction operations to also run in parallel. Another construct of HPF 2 will allow more general task structures, such as task pipelines, to be expressed. Finally, HPF 2 supports the ability to map data structures to processor subsets and to operate in parallel on different data structures using different submachines.

Dataflow Analysis in Parallel Programs

Jean-François Collard and Martin Griebl
Université de Versailles, France, and University of Passau, Germany

We describe a dataflow analysis of array data structures for data-parallel and/or control- (or task-) parallel imperative languages. This analysis departs from previous work because (1) it handles simultaneously both parallel programming paradigms, and (2) it does not rely on the usual iterative solving process of a set of dataflow equations but extends array dataflow analysis based on integer linear programming, thus improving the precision of results.

A Library for Operations on Z-Polyhedra

Tanguy Risset
IRISA, Rennes, France

Z-polyhedra are in the intersection of polyhedra and integral lattices. Z-polyhedra are used to model loop iteration domains; operations on Z-polyhedra are useful for loop transformations. We present a practical approach to the problem of computation upon Z-polyhedra. We introduce a canonic representation of Z-polyhedra, which allows to perform comparisons and transformations of Z-polyhedra with the help of a computational kernel for polyhedra. This contribution is a step towards the manipulation of images of polyhedra by affine functions. It has applications in the domain of automatic parallelization and parallel VLSI synthesis.

3 List of Participants