Dagstuhl Seminar
on
Theory and Practice of Higher-Order Parallel Programming

Organized by

Murray Cole (University of Edinburgh)
Sergei Gorlatch (Universität Passau)
Christian Lengauer (Universität Passau)
David Skillicorn (Queen's University at Kingston)

Schloß Dagstuhl 17. – 21.2.1997

Contents

1 Preface

The diversity and complexity of parallel computers (both proposed and implemented) drives the quest for portable, tractable and efficiently implementable parallel programming models and languages. One approach to the problem focuses on the design and exploitation of higher-order programming and algorithmic constructs which can become the fundamental building blocks in such a model. An analogy can be drawn with the historical development of sequential programming, in which simple, relatively unstructured mechanisms, closely tied to the underlying architecture, have given way to more powerful, structured and abstract concepts. Similar progress in the parallel case would raise the level of abstraction from models in which communications (or the like) are primitive to a world in which more complex patterns of computation and interaction are combined and presented as parameterized higher-order program-forming constructs.

There are three more or less interwoven threads of research in the area. Work on algorithmic skeletons focuses on the selection and implementation of appropriate primitives (the "skeletons"), typically with a bias towards coarse-grain primitives and control-parallel implementation. Researchers in program development apply and extend techniques from the sequential branch of the field to the parallel case, developing new design methodologies and models in which, as discussed above, the level of abstraction is raised from the base level of simple concurrent activities, and in which there are facilities for understanding the cost of the program under development during its design. The final thread seeks to design and understand the formal models which underpin the area, whether by viewing existing models in a parallel light (for example, work on the Bird-Meertens Formalism as a parallel programming model) or by developing new ones. Much work in this thread takes a more data-parallel view of the process.

The concerns spanned by these groups range from the very pragmatic questions of implementation techniques and real applications to the abstract world of formal program development. Many researchers from all of the areas had already been brought together electronically, via e-mail lists and WWW pages. The seminar was an opportunity to establish the community and intensify the contacts.

Questions discussed at the seminar included:

- Which primitives are suitable for a parallel programming model?
- Can the selection of primitives be justified pragmatically and/or theoretically?
- To what extent should the parallelism within the primitives be apparent to the programmer?

- To what extent can and should the implementation mechanisms attempt to optimize the parallel structure of composed primitives?
- To what extent can this optimization be reflected in a compositional cost model (or cost calculus)?
- To what extent do 'sequential' techniques in program development shed light on the parallel case?
- What are the new transformations for the new primitives?
- How well does the approach compare with conventional models in its ability to express 'classical' parallel algorithms and to derive new ones?

The 43 participants of the workshop came from 10 countries: 16 from Germany, 8 from the UK, 6 from the USA, 5 from France, 2 from Italy and 1 from each Australia, Canada, Japan, New Zealand and Sweden. The organizers would like to thank everyone who has helped to make this workshop a success.

Murray Cole Sergei Gorlatch Christian Lengauer David Skillicorn

2 Abstracts

Skeletal Programming – Present and Future

Murray Cole
University of Edinburgh, UK

This talk considers the common theme which unites the participants, namely that abstraction and control of structure should play an important role in parallel programming, and suggests four related areas in which further research is required. Firstly, in the area of "expressivity" it is as yet unclear what might constitute the right set of types and operators (or whether such a bounded set even exists) and what might be the most appropriate language framework within which to express the base level of a skeletal scheme. Secondly, the challenge of building a tractable, compositional cost calculus for such a model, and the choice of underlying cost/computational model must be resolved. Thirdly, the techniques and frameworks required for formal program derivation and transformation when the target context is parallel must be further developed. Finally, our implementations and concrete programming languages must be sensitive to the needs of "real" parallel programmers, who must be convinced that our ideas have something to offer in practice.

Combining Task and Data Parallelism within Skeleton-Based Models

Susanna Pelagatti
University of Pisa, Italy

This talk approaches the problem of providing both task and data parallel abstractions within a system which is easy to use, portable and reaches high levels of performance on different architectures. To achieve all this, an automatic and efficient solution of mapping, scheduling, data distribution and grain size determination must be found. These problems are proved intractable for models allowing parallel computations with arbitrary structure to be expressed. Moreover, parallel applications appear to use a restricted set of recurring structures, possibly nested, for both data and task parallel parts. The talk analyzes the most common paradigms of task and data parallelism (task queue, pipeline, independent and composed data parallelism) and shows how they can be provided within a skeleton based language (P3L). The P3L support is organized according to a template based methodology and embeds in a transparent way good implementation strategies for each skeleton.

The Higher-Order Parallel Programming (HOPP) Model

Roopa Rangaswami
University of Edinburgh, UK

Programming parallel computers remains a difficult task. An ideal parallel programming environment should enable the user to concentrate on the problem-solving activity at a convenient level of abstraction, while managing the intricate low-level details without sacrificing performance.
In an attempt to achieve these goals, a model of parallel programming based on the Bird-Meertens Formalism (BMF) is investigated. It comprises of a predefined set of higher-order functions, many of which are implicitly parallel. Programs are expressed as compositions of these higher-order functions. A parallel implementation is defined for each of these functions and associated costs are derived. An analyzer estimates the costs associated with different possible implementations of a given program and selects a cost-effective one. The parallel implementation strategy output by the analyzer is input to a code generator which produces parallel SPMD code in C++, using MPI to handle communication.
Initial experiments involving the generation and execution of parallel code for predicted cost-effective implementations of simple problems on the Cray-T3D are encouraging. Further work will concentrate on the expressivity of the model.

PAR Considered Harmful

Luc Bougé
ENS Lyon, France

The evolution of computer languages is dominated by the quest of abstraction. The programming models develop independently of the execution models, and compiler technology becomes a central concern. A typical example of this continuous trend is the advent of "structured programming" in the 70's and Dijkstra's claim "GOTO Considered Harmful".
We claim that parallel programming languages are now developing along the same lines. The advent of data-parallel programming in the 90's introduces structure

and abstraction. A parallel composition of sequential processes (PAR/SEQ) is recast into a sequential composition of actions on parallel objects (SEQ/PAR). The price to pay for this is a possible loss in expressivity and performances. We beleive this will be outweighted by major gains:

- performance predictability (BSP);
- compositional semantic methods for program validation and refinement (skeletons);
- new perspectives in software engineering (in particular, reusability).

The next step is probably to go from flat parallel objects (arrays) to structured ones (lists, trees, etc.).

Development of Parallel Programs: Towards a Systematic Approach

Sergei Gorlatch
Universitt Passau, Germany

This paper reports a case study in the development of parallel programs in the Bird-Meertens formalism (BMF), starting from divide-and-conquer algorithm specifications. Our long-term goal is to come up with a parallel programming methodology which would simplify the development of correct programs with predictable efficiency over different parallel architectures.

The contribution of the paper is two-fold: (1) we classify divide-and-conquer algorithms and formally derive a parameterized family of parallel implementations for an important subclass of divide-and-conquer, called DH (distributable homomorphisms); (2) we systematically adjust the mathematical specification of the Fast Fourier Transform (FFT) to the DH format and thereby obtain a generic SPMD program, well suited for implementation under MPI. The target program includes the efficient FFT solutions used in practice – the binary-exchange and the 2D- and 3D-transpose implementations – as its special cases.

The case study demonstrates a potential of systematizing the development of parallel programs by using higher-order skeletons and formal transformations. More information can be found at: http://www.uni-passau.de/~gorlatch

Systematic Mapping of Higher-Order Functional Specifications

Zully Grant-Duff
Imperial College, UK

The talk described a general method for mapping combinator (higher-order objectless functions) applications onto loosely-coupled multiprocessors. Programs in ACL, the source language where the combinators are a set of constant constructs, are represented as weighted task graphs. The weight associated with a node represents computation cost, while that associated with an arc represents communication cost. Analogously, the network is represented by a system graph whose nodes (processors) have an associated computation speed and whose arcs (communication links) have an associated communication bandwidth. The mapping algorithm clusters both graphs through identifying concurrency and connectivity respectively, and proceeds to match clusters according to demand and capacity. We showed experiments where the resulting mapping – and inherent scheduling – achieves a speed up which is approximately 66 % of the optimum.

SPMD Programming in Java

Susan Flynn Hummel
IBM T.J. Watson Research Center, USA

We consider the suitability of the Java concurrent constructs for writing high-performance SPMD code for parallel machines. More specifically, we investigate implementing a financial application in Java on a distributed-memory parallel machine. Despite the fact that Java was not expressly targeted to such applications and architectures per se, we conclude that efficient implementations are feasible. Finally, we propose a library of Java methods to facilitate SPMD programming.

Abstract Parallel Machines: Organizing Higher Order Functions for Parallel Program Derivation

John O'Donnell and Gudula Rnger
University of Glasgow, UK, and Universitt Saarbrcken, Germany

We need to take a flexible approach in designing a family of higher order functions to support parallel program derivation. For example, it isn't enough just to define scan and give it a log time cost model, because there are actually many different implementations with different costs, suitable in different circumstances. To help a programmer decide what to do next in a derivation, the parallel operations need to be defined at several levels of abstraction and to have a suitable operational semantics at each level.
We propose Abstract Parallel Machines to address these problems, and we apply them to two case study derivations: a parallel heat equation program and a parallel addition algorithm.
An abstract parallel machine defines a set of parallel operations, and it expresses these definitions using a set of computational sites ("abstract processors") and coordination functions ("abstract network"). This makes a suitable model of implementation available at each level. The framework can describe higher order parallel operations at high levels (SPMD), intermediate levels (scan) and low levels (digital circuits). We have found that this approach helps guide the derivation process by clarifying the relationships between alternative realizations of a function, and we plan to experiment with it on more complex case studies.

Costs, Transformations, and Parallel Programming

David Skillicorn
Queen's University at Kingston, Canada

A programming model cannot be useful for designing programs unless it possesses a transformation system. A transformation system is not useful unless it possesses a cost model. A cost model maps functions to program texts (that is, implicit implementations) and transformation rules to rewrite rules.
It is useful to consider what we might expect of a cost-based transformation system. There are two properties of interest. The first is the degree of confluence.

A rewriting system may be confluent, simply confluent (on components of programs), directed, or without costs altogether. The second property of interest is convexity: it is not possible to increase the cost of a piece of a program to decrease the cost of the whole.
Neither convexity or some degree of confluence are easy to get for parallel systems. Convexity is usually destroyed by congestion phenomena. It is possible to get a directed convex transformation system for homomorphic skeletons by the following process: start from a small fixed set of skeletons, form all short compositions, define a new skeleton for each composition with a cheaper than expected implementation, and add the defining equation as a new rewrite rule. It also seems possible to get simply confluent rewrite systems using variants of BSP such as miniBSP.

Deriving Programs with Mixed Method and Data Parallelism

Gudula Rünger
Universitt Saarbrcken, Germany

The talk presented an overview of a methodology for deriving parallel programs in the area of scientific computing with mixed method and/or data parallelism. The final parallel programs are group-SPMD programs in a message-passing style aimed at distributed memory machines.
The top-down derivation process contains three stages: (i) the hierarchical module specification of the algorithmic structure designed by the application programmer, (ii) a parallel frame program resulting from decision steps (multitask-scheduling, partitioning the sets of processors into groups, data distribution types for input/output of each module) , and (iii) the executable message passing program with a small set of collective communication operations. The emphasis lies in the methodology how to transform one stage into the next one by being more precise concerning the parallelism to be exploited in the final program.
The performance analysis is based on the structure of the frame program which enables the programmer to test different alternatives of parallel implementation decisions before realizing the best. There is a library of programs in the area of linear and nonlinear algebraic or differential equation solvers realized according to this methodology. The performance prediction mechanism was verified for the library programs on the Intel Hypercube and Paragon, the Cray T3D, and the IBM SP2.

Higher-Order Functions in Hardware Design

Mary Sheeran
Chalmers University of Technology, Gteborg, Sweden

I presented my experiences in using higher-order functions in hardware design. I briefly described μFP, an extension of Backus' FP to synchronous streams, in which the combining forms are given a geometric as well as a behavioural interpretation. The result is a hardware design language that is particularly well suited for designing and reasoning about regular array algorithms. In 1986-87, it was used by engineers at Plessey to design a video picture motion estimator circuit that became a product. Unfortunately, the design group was closed down shortly afterwards.

Next, I presented Ruby, a generalisation of μFP in which circuits are modelled as binary relations on synchronous streams. I showed the kinds of higher order functions that are used in circuit design and some examples of their algebraic properties. Groups of higher order functions for various design idioms have been studied, for example regular arrays (including grids and hex-connected arrays), state machines, and high wire area networks (such as butterflies). An extensive suite of tools has been built (by people other than me): a formalisation in Isabelle, a transformation assistant, symbolic evaluators, visualisers, VHDL generators etc. Ruby can perhaps be classified as an academic success. But, it is not used in practice. We have provided many of the tools for hardware design that the skeletons community plans to provide for high level parallel programming, yet we have not succeeded in finding users. Why is this? A possible reason is that our papers are cryptic and published in the wrong places. But maybe it is just that circuit design by formal derivation is just too hard to do! I asked this question in order to prompt the skeletons community into thinking about its plans for the future.

Finally, I briefly described my current work with Satnam Singh on a hardware design system that uses Haskell as the hardware description language. Further information can be found on the Lava web page:

http://www.dcs.gla.ac.uk/~satnam/lava/main.html

BSP Cost Analysis and the Implementation of Skeletons

Jonathan M.D. Hill
Oxford University Computing Laboratory, UK

The Bulk Synchronous Parallel (BSP) model provides a theoretical framework for the development of *predictable* architecture independent parallel algorithms. One of the strengths of BSP compared to alternative models of parallel computation is its simple yet *realistic* cost model that decomposes costs into communication and computational parts. By placing an emphasis on these two fundamental aspects of parallel algorithms, the aim of this talk is to show how BSP cost analysis can be used to guide algorithm design towards optimal solutions across a wide variety of machines.
In contrast to BSP, the skeletal approach to parallelism provides an expressive way of encapsulating general patterns of computation and communication, yet it lacks "flesh". The fundamental problem is the naivity of skeletal cost analysis, which has produced a plethora of techniques that are unusable on today's parallel machines.
The aim of this talk is to show how BSP cost analysis can be used to develop architecture independent skeletons at a high level of abstraction, whilst providing a basis for optimal performance on todays (and future) parallel machines.

A High-Level Programming Environment for Distributed Memory Architectures

Wolfgang K. Giloi
GMD-FIRST and TU Berlin, Germany

The talk presents the PROMOTER programming model for the abstract, problem-oriented programming of message-passing architectures. PROMOTER enables its user to formulate parallel applications in an abstract, algebraic form so that the PROMOTER compiler can take care of the correct and time-optimal execution. PROMOTER allows the user to deal with a large variety of array-like or hierarchical data structures that may be regular or irregular, static or dynamic. Programs are written at a level at which the data types look homogeneous – thus allowing the use of distributed types – although there may be local differences in thread execution. Domains of computation are created as finite, possibly irregular and/or dynamic substructures of a regular, static "structured universe", which is an index domain with group property. Communication is viewed by the programmer as the observation of state in some domain points by other domain points as specified by the definition of a "communication domain". Coordination of parallel threads is guided by a "coordination scheme" selected by the programmer. Existing coordination schemes are: lock step (bulk synchronization), wave fronts, asynchronous iteration, chaotic iteration. PROMOTER has been implemented on a variety of platforms and competes in efficiency with PVM or MPI.

Distributed Types: A Unifying Model of Spatial Structure in Parallel Programming

Andreas Schramm
GMD-FIRST, Germany

In programming massively-parallel machines, preserving locality is the major issue (because of the physical communication incurred).
Thus, the intent of Distributed Types is to formalize the spatial structure of data-parallel applications explicitly at a problem-specific level.
The advantages of expressiveness at a high level are that the low-level implementation aspects such as mapping and message generation can be delegated to the system. The benefits that can be expected are more compact and comprehensible programs and improved architecture independence.
The actual challenge of this approach is that the formalization should reflect the structuring principles that actually occur in the relevant class of applications.
The presented talk identified the most important problems that massively parallel applications, mainly numeric ones, pose with respect to spatial structures, and introduced a solution.

Deriving Parallel Algorithms using Data Distribution Algebras

Thomas Nitsche
TU Berlin, Germany

Parallel and distributed programming are much more difficult than the development of sequential algorithms due to data distribution issues and communication requirements.

This talk presents a methodology which allows the abstract description of the distribution of algebraic data structures using data distribution algebras. The key idea behind the concept is that a data structure is split into a *cover* of overlapping subobjects which may be allocated to different processors. The own parts of the subobjects form a partitioning of the original structure, while the overlapping foreign parts specify communication requirements and possible data dependencies.
Algorithms are formulated in a functional setting using skeletons as certain higher order functions based on (parallel or sequential) covers.
This allows the programmer to specify data distribution issues on an abstract level. Such specifications enable the derivation of explicit lower–level communication statements.
To illustrate the concept, Wang's partition algorithm for the solution of tridiagonal linear equations is derived.

Exploiting Maximum Parallelism in Hierarchical Numerical Applications

Alexander Pfaffinger
TU Mnchen, Germany

Using hierarchical basis functions for the d-dimensional multilinear function representation, the number of the corresponding grid points can be reduced drastically from n^d to $O(n \log^{(d-1)} n)$ without significant increase of the approximation error. This leads to so-called sparse grids. Instead of flat arrays, binary trees and d-dimensional product graphs of binary trees are the natural implementation.
This product graph also reflects the dependency structure of the algorithm. Because of its complexity, exploiting the maximum inherent parallelism is tedious. An intuitive domain decomposition formulation of a sparse grid algorithm leads to a parallel complexity of $O(\log^d n)$ whereas an optimal implementation would achieve $O(\log n)$ complexity. The intuitive algorithm also results in an inefficient communication and synchronization pattern.
On the other side, coding an optimal program within conventional imperative languages (e.g., C with PVM) is a hard issue for general dimensions d. In the new data flow language FASAN the programmer has only to specify the mathematical data dependencies between the parts of the algorithm. The semantics of "wrapper streams" automatically generates direct communication channels between the dependent nodes, whereas the data flow semantics sends the data immediately after they are produced. Thus, the optimal parallel complexity can be expressed even with an intuitive divide-and-conquer description.

A Data Flow Approach to Higher Order Functions for Recursive Numerical Applications

Ralf Ebner
TU Mnchen, Germany

Multilevel algorithms for the numerical solution of partial differential equations like multi-grid methods, recursive substructuring techniques or algorithms based on sparse grids are naturally implemented as recursive programs that operate on adaptive tree structures. They typically apply a function to each tree node during two different kinds of tree traversal: a downward mapping, where each function application to a node's value depends on the result of its parent node, and an upward mapping, where the result of all children nodes are needed. A sequence of these algorithmic patterns can easily be expressed as combination of higher order functions.

A higher order extension of the functional language FASAN, which generates data flow graphs as abstract evaluation machines, permits an efficient combination of downward and upward mappings. The unnecessary overhead of construction and decomposition of the tree structures induced by the higher order function combination is deviated by the data flow concept of *wrapper streams* replacing ordinary tree constructors. Wrapper streams yield direct communication channels in the data flow graph, so locality of the tree node values, often large matrices or equation systems, is achieved.

A special syntax for partial function applications allows the neatless integration of higher order functions within the Pascal-like syntax of FASAN.

The Elements, Structure, and Taxonomy of Divide-and-Conquer

Z. George Mou
The Johns Hopkins University USA

Despite its broad range of successful applications, divide-and-conquer (DC) seems to many nothing more than an informal strategy with neither clearly defined elements nor a mathematical structure.

We identify three types of functionally orthogonal operations: relational, communication, and strongly local. The relational operations take apart and put together the data structures, the communication operations allow the nodes in the data structure to exchange the values, the strongly local operations map each node independent to others to its new value.

The elementary operations are structured under the pseudomorphism model, which is a generalization of the homomorphism. The additional mapping – pre

and post adjust functions – are always compositions of communication and, in general, weakly local functions. A weakly local function makes the computation over divided components independent of each other, but not necessarily that over each data point. A weakly local function can be in turn a DC and results in higher-order DC's.
DC algorithms can be classified in terms of their elements and their structures. The concepts of premorphism, postmorphism, strict, sequential, and polymorphic DC algorithms are explained among others. It is pointed out that the model can be used to derive fast parallel DC algorithms for many problems in numerical analysis, computational geometry, and other areas of applications. The Divacon language has been implemented on a number of parallel computers, and used to build a linear algebra library. Optimal mappings of DC algorithms to a large class of communication networks have been proposed and used to implement many DC algorithms on real machines. It is also pointed out that recursive doubling, cyclic reduction, and many BMF-based algorithms are special cases of the pseudomorphism model. The model also subsumes most known programming skeletons such as broadcast, reduction, scan, and nested scan.

Translation of Divide-and-Conquer Algorithms to Nested Parallel Loop Programs

Christoph Herrmann and Christian Lengauer
Universitt Passau, Germany

We present a top-down classification of Divide-and-Conquer by skeletons expressed in the functional language Haskell.
Depending on the specializations made, the resulting skeleton can be transformed into a nested parallel loop schema using equational reasoning. Building this into a compiler allows the user to translate a recursive Divide- and-Conquer algorithm, expressed in terms of a skeleton, automatically to an imperative, data-parallel loop program, e.g., in HPF. The advantages of a loop program are that it can be optimized for a given objective function, using integer linear programming, and that it can be implemented on various architectures using existing compilers.
The ease of use of our skeletons is illustrated by the non-trivial example of Strassen's matrix multiplication.

Algorithm + Strategy = Parallelism

Phil Thrinder
University of Glasgow, UK

The process of writing large parallel programs is complicated by the need to specify both the parallel behaviour of the program and the algorithm that is to be used to compute its result. The talk introduced evaluation strategies, lazy higher-order functions that control the parallel evaluation of non-strict functional languages. Using evaluation strategies, it is possible to achieve a clean separation between algorithmic and behavioural code. The result is enhanced clarity and shorter parallel programs.

Evaluation strategies are a very general concept: the talk outlined how they can be used to model a wide range of commonly used programming paradigms, including divide-and-conquer, pipeline parallelism, producer/consumer parallelism, and data-oriented parallelism. Because they are based on unrestricted higher-order functions, they can also capture irregular parallel structures.

Evaluation strategies are not just of theoretical interest: they have evolved out of our experience in parallelising several large-scale parallel applications, where they have proved invaluable in helping to manage the complexities of parallel behaviour. The largest application we have studied to date, Lolita, is a 60,000 line natural language parser. Initial results show that for these programs we can achieve acceptable parallel performance, while incurring minimal overhead for using evaluation strategies.

Runtime Interprocedural Data Placement Optimisation for Lazy Parallel Libraries

Paul H J Kelly
Imperial College, London, UK

We are developing a lazy, self-optimising parallel library of vector-matrix routines. The aim is to allow users to parallelise certain computationally expensive parts of numerical programs by simply linking with a parallel rather than sequential library of subroutines. The library performs interprocedural data placement optimisation at runtime, which requires the optimiser itself to be very efficient. We achieve this firstly by working from aggregate loop nests which have been optimised in isolation, and secondly by using a carefully constructed mathematical formulation for data distributions and the distribution requirements of library operators, which together make the optimisation algorithm both simple and efficient.

Functions Compute, Relations Co-ordinate

Manuel M. T. Chakravarty
TU Berlin / GMD FIRST, Germany

Parallel implementations of declarative languages suffer from two problems: first, the lack of an explicit, but declarative notion of parallelism; and second, latency that is induced by remote data accesses. We get a declarative notion of parallelism by a combination of functional and relational programming elements, where purely functional expressions denote step-at-a-time computations, whereas relations express co-ordination of sequential computations. The essence of such languages is captured by an extension of the lambda calculus – called $\mathcal{D}$ – that adds co-ordinating relations. $\mathcal{D}$ has an abstract semantics, which is realized by an encoding into linear logic, and a parallel operational semantics, which is derived from the abstract one.
To tackle the second problem, namely latency induced by remote access, an integration of multi-threading into the Spineless Tagless G-machine is proposed. It uses the freedom in the evaluation order, which is guaranteed by the semantics, to overlap different communication operations and computation with communication.

Practical PRAM Programming with Fork95

Christoph W. Kessler
Universitt Trier, Germany

With the existence of the SB-PRAM architecture, the PRAM model of parallel computation gains practical importance. We present the PRAM programming language Fork95. A superset of ANSI C, Fork95 offers language constructs to control different levels of relaxed exact synchronicity and shared address subspaces according to a hierarchical group concept.
Fork95 supports many important parallel algorithmic paradigms, including data parallelism, parallel divide-and-conquer, and pipelining. It turns out that explicit new language constructs ("skeletons") are not required to implement these paradigms in Fork95.

A compiler for Fork95 is operational and available on the WWW, together with a simulator for the SB-PRAM, documentation, source code, and example programs. (http://www.informatik.uni-trier.de/~kessler/fork95)
We introduce a new language construct, the join statement, which allows to easily express synchronous parallel critical sections and provides a flexible means to switch from asynchronous to synchronous mode of program execution. We visualize its semantics by an excursion bus analogy, and exemplify how it can be applied in practice to speed up, e.g., parallel shared heap memory allocation.

Costs and Semantics in Parallel Languages

Gaétan Hains
Université d'Orleans, France

We observe that parallel algorithmics requires a special point of view on programming languages and that specific language support for it should exist.
We then ask whether there exists a general theory of parallel languages distinct from sequential and concurrent languages. Such a theory would constitute a platform for the design of general-purpose parallel languages. The introduction of higher- order user-defined functions makes it hard to characterise what a parallel language should be. This is because the standard semantics of functional languages is not fully abstract.
Our work (with Loulergue, Mullins and Charloton) on functional parallel programming with concrete data structures is introduced as a small step towards resolving the tension between denotational and operational views of parallel programs.
More general observations lead us to call for an integration of semi-ring theory with domain theory with the following problem in mind. Semi-rings are distributive and non-sequential domains are not. The semi-ring cost operations need not coincide with the information order operations; but for the sake of parallel languages they should somehow interact. The open question is how.

Vectors are Shaped Lists

C. Barry Jay
University of Technology at Sydney, Australia

Shape theory supports abstraction of data structures independently of the data that is stored within them. For example, one may talk of a shape that stores real numbers, without knowing whether the shape is a tree or an array. Shape

information is one of the keys to successful parallel programming, as it supports error detection, efficient data distribution and redistribution, and load balancing. This talk will introduce our higher-order functional language Vec for vectors and arrays, and show how shape analysis yields these desired benefits, including a new divide-and-conquer algorithm.
If there is time, I will discuss our plans for extending Vec to support imperative features, so that we can embed native code from other languages.

Alpha: A Functional Data Parallel Language Based on Polyhedra

Sanjay Rajopadhye
IRISA, France

Alpha was originally designed by Christophe Mauras (1989) to serve as a tool for manipulating and transforming systems of affine recurrence equations in the context of systolic array synthesis. In this talk, I present the basic language, discuss the motivations behind its design and describe how affine dependency functions, polyhedral domains and unimodular transformations interact in a coherent manner an empower two important properties of the language: normalization and change-of-basis.
Recent work on Alpha involves the addition of reductions to the language, the development of subsystems so that computations can be expressed in a modular and hierarchical manner, definition of a (proper) subset called AlpHard for defining regular VLSI (systolic) arrays, development of a transformation system based on the Mathematica system, tools for static analysis of Alpha programs, compilation of Alpha to sequential and parallel general purpose machines, extensions of the language to sparse domains (domains which are defined as the intersections of lattices and polyhedra), and some ongoing work on verification.

Structured Parallel Programming: Parallel Abstract Data Types

Hing Wing To
Imperial College, UK

The work we present is motivated by the observation that irregular computation underlies many important applications. In addressing this problem we propose solutions to the problem of introducing new skeletons into an existing skeleton language.

At Imperial College we have developed a language for structured parallel programming (SPP(X)) using functional skeletons to compose and co-ordinate concurrent activities written in a standard imperative language. The kernel language, which is currently under construction, is based around the distributed array data type. The language is thus highly suitable for expressing regular computations.
The kernel language is less suitable for expressing programs over irregular data structures. We propose the introduction of Parallel Abstract Data Types (PADT) to enable such programs to be expressed naturally. A PADT consists of a data type definition of the irregular data structure and a set of parallel operators over this type.
The introduction of PADTs raises the question of how they are to be implemented. A direct approach would be to hand code the operators in some imperative language with message passing. In this approach the PADT becomes part of the kernel language. This is undesirable as this could possibly lead to an ever increasing kernel language as new PADTs are found to be necessary. The disadvantages of increasing the kernel language with the introduction of each new PADT include the need to extend and reformulate any system for optimising combinations of skeletons and any system for compiling programs.
We propose an alternative approach to implementing PADTs. The philosophy of our approach is to build the PADTs on top of the existing language constructs rather than to extend the kernel language. The encoding of a PADT into the kernel language can be arrived at through several data type transformations thus ensuring correctness and providing the opportunity for reasoning about optimisations. We demonstrate the approach by outlining a derivation from an example program over an irregular triangular mesh to its final regular array encoding.
We conclude the talk by suggesting that the concept can be applied to High Performance Fortran (HPF). Currently, HPF-1 cannot directly express irregular applications. There are proposals to extend the language with constructs for expressing irregular applications. It may be possible to apply our techniques to HPF-1, thus enabling irregular applications to be expressed without the need for kernel language extensions.
There are open questions including whether HPF-1 or SPP(X) are sufficiently rich to build all "useful" PADTs.

Yet Another Bridging Model – The Parallel Memory Hierarchy

Larry Carter
University of California at San Diego, USA

The Parallel Memory Hierarchy (PMH) model of computation is a tree of "modules", where each module is a finite RAM, and communication of blocks of contiguous data occurs on tree edges. Each module is parameterized by its memory capacity and number of children. Each edge is parameterized by the block size and transfer time. Modules nearer the root have larger memories; those nearer the leaves are smaller but faster (and more numerous). High performance is achieved by moving small, compute-intensive subproblems towards the leaves, and finding independence among the subproblems that are assigned to separate children. It is asserted that:

- Appropriate parameter choices can be made to model a wide variety of real computers.
- Efficient PMH programs correspond to efficient real programs.
- Future machines will have more levels, more multi-child levels, (i.e., more parallelism), and more heterogeneity.
- Portable, efficient PMH programs can be written by having each module break its subproblems into still smaller sub-subproblems (whose number and size is influenced by the PMH's parameters) that are passed to the module's children in a pipelined fashion.
- Hierarchical tiling, or formal methods (e.g., skeletons) may reduce the difficulty of writing PMH programs.

Data-parallel Skeletons

Herbert Kuchen
RWTH Aachen, Germany

Algorithmic skeletons are abstracts of common parallel programming patterns. We are particularly interested in using them for distributed memory machines. In order to reach the required generality, they are typically defined as polymorphic higher order functions. It is possible to distinguish data parallel, task parallel, and application oriented skeletons. Typically, data parallelism offers a better scalability and it can be easier implemented efficiently than task parallelism. Thus, we focus (at least for the time being) onto data parallel skeletons, i.e. skeletons working on a distributed data structure, in our case arrays. These skeletons can mainly be divided into monolithic computation operations (like *map* and *fold*), working in parallel on the elements of an array, and monolithic communication operations, which change the mapping of partitions of an array onto the (local stores of the) available processors. These communication operations can perform a coordinated overall communication (rather than the transmission of individual

messages) and they can thus be implemented in a way which guarantees that typical communication problems like deadlocks known from low-level message passing cannot occur.
A couple of techniques have been developed which allow run times similar to C with message passing. Among them is a technique which instantiates higher order functions to a set of corresponding first order functions. Moreover a decentralized execution model is used where "sequential computations" are replicated on every processor and skeletons are split in such a way that every processor takes care of computation concerning its local share of data.

Skeleton-Based Implementation of Adaptive Multigrid Algorithms

George Horatiu Botorog
RWTH Aachen, Germany

We are considering two issues, which, we believe, are very important in the context of algorithmic skeletons: on the one hand, finding an adequate level of generality for the skeletons and on the other hand, using skeletons to solve "real-world" problems in parallel.
To the first issue, we argue that very general skeletons may be inadequate for expressing certain algorithms and may lead to implementations that are not very efficient, whereas too specialized skeletons may degenerate to plain parallel library functions. Our approach is therefore somewhere in between these two extremes, more precisely, we try to design skeletons that can be used in solving related problems, or problems from different areas, but having a similar (e.g., hierarchical) structure.
We choose as our class of applications adaptive multigrid algorithms. There are several reasons for this. Firstly, multigrid methods are non-trivial numerical applications. Secondly, there is an entire class of multigrid methods, which build however on the same blocks, thus fitting naturally in the skeletal concept. Thirdly, multigrid skeletons can be used in the implementation of other classes of hierarchically structured algorithms, like for instance N-body methods.
We have then analyzed the data dependencies and access patterns that occur in multigrid and have derived from them two kinds of operations: "horizontal" or intra-grid operations, like relaxation or correction computation and "vertical" or inter-grid operations like grid refinement/coarsening and data prolongation and restriction.
Based on these data dependencies, we have derived a series of skeletons for multigrid. Besides point wise operations like *map*, we also employed "environment operations", which are applied to single points, together with their environment,

i.e. their neighboring points in the horizontal case and their parents/children in the vertical case, respectively.

Classification of Parallel Implementations for Linearly Recursive Functions

Christoph Wedler and Christian Lengauer
Universitt Passau, Germany

Broadcast, Reduction and Scan are popular functional skeletons which are used in distributed algorithms to distribute and gather data. We derive new parallel implementations of combinations of Broadcast, Reduction and Scan via a tabular classification of linearly recursive functions. The trick in the derivation is to not simply combine the individual parallel implementations of Broadcast, Reduction and Scan, but to transform these combinations to skeletons with a better performance. These skeletons are also linearly recursive.

Types in Parallel Programming

Eric Violard
Université Louis Pasteur, France

A wide range of research works on the static analysis of programs and forms the foundations of parallelization techniques. They greatly benefit from methods for the automatic synthesis of VLSI systolic circuits from recurrence equations.
These methods are based on geometrical transformations either of the iteration space or of the index domain of arrays. In some sense, it shows that beside a classical functional point of view on programs, geometrical issues in parallel programming or parallelizing compilation have to be considered of main importance. The recent developments on the data-parallel programming model reinforces this idea.
PEI was born of this approach and was introduced to express and transform parallel programs. It enables to define and apply pointwise operations on the basic objects of the language, called data fields – a kind of parallel variables. Abstract reference domains are also defined. Conformity of data fields or hypotheses on their domains have to be checked in order to apply operations or to transform programs. This requires PEI to be a strongly typed language in which the types of data fields, definition domains of partial functions on Z^n, can be inferred. The type system any correct PEI expression must solve is defined by induction on the

syntactic constructs of the language through inference rules. It can be expressed in a normal form by applying confluent rules and then be solved.
The typechecking algorithm has been implemented by using the Omega library which evaluates Presburger formulae and it works for a wide range of data field expressions.

Formal Validation of Data-Parallel Programs: The Assertional Approach

Luc Bougé and David Cachera
ENS Lyon, France

In this talk, we specifically address the validation of Alpha programs, as specified by Quinton and Radopadhye, IRISA, Rennes, France. Alpha programs are made of a set of recurrence equations, one for each variable. A rich set of transformations has been developed for them, including changes of basis to separate time and space components. But little has been done to prove abstract properties of such programs, especially partial properties of their functional behavior. We do not consider here properties which may depend on a particular scheduling of the program.
We present here a framework to prove input/output properties $\{p\}\ S\ \{q\}$. It is based on the search for a program invariant, very much as in the proof of a while-loop in Hoare's logic. Our main result is the completeness of the method: for any valid property, there exists an invariant to prove it. In fact, this invariant is nothing but the logical encoding of all data-flow dependencies specified by the program. Its external from is actually exactly the same as the program text, up to a suitable interpretation of the "=" sign. Of course, a property may also be proved using weaker ad-hoc invariants.
Using this approach sets up a firm foundation for all kinds of "handwaving" manipulations found in the literature about Alpha programs. It is especially crucial when the properties under study cannot be expressed as another Alpha program: non-linear indexing of arrays, induction arguments, etc.

Data-Parallel Programming: Can we Have High-Level Languages *and* High-Performance?

Jan F. Prins
UNC Chapel Hill, USA

Nested arrays and nested data-parallelism combine to provide a uniquely expressive mechanism for the specification of irregular parallel computations. The flattening transformation first described by Blelloch in 1990 provides a technique to realize all nested parallelism to within a constant factor of that specified without increasing total work and with perfect load balance in execution. The price for all these advantages is that the absolute performance and performance predictability fall precipitously in the presence of non-uniform memory access costs, and this appears, in some sense, to be inherent.
Yet uniform memory access (UMA) on real machines can be guaranteed through a bulk-reference condition that is easily satisfied by flattened parallel programs. Current commercial machines (such as the Cray and NEC parallel vector processors) provide UMA for vector operations. Using these machines and the flattening technique, some important irregular applications, such as the solution of unstructured sparse linear systems, may achieve the highest absolute performance currently available.

Load Balancing Strategies to Implement Non-Uniform Data Parallel Computations

Salvatore Orlando
University of Venice, Italy

The efficient implementation of data parallel loops on distributed memory multicomputers is a hot topic of research. To this end, data parallel languages such as HPF generally exploit static data layout and static scheduling of iterations. Unfortunately, when iteration execution costs vary considerably and are unpredictable, some processors may be assigned more work than others. Workload imbalance can be mitigated by cyclically distributing data and associated computations. Though this strategy often solves load balance issues, it may worsen data locality exploitation.
In this talk we present SUPPLE (SUPort for Parallel Loop Execution), an innovative run-time support for parallel loops with regular stencil data references and

non-uniform iteration costs. SUPPLE relies upon a static block data distribution to exploit locality, and combines static and dynamic policies for scheduling non-uniform iterations. It adopts, as far as possible, a static scheduling policy derived from the owner computes rule, and moves data and iterations among processors only if a load imbalance actually occurs. SUPPLE always tries to overlap communications with useful computations by reordering loop iterations and prefetching remote ones in the case of workload imbalance.
The SUPPLE approach has been validated by many experiments conducted by running a multi-dimensional flame simulation kernel on a 64-node Cray T3D. We have fed the benchmark code with several synthetic input data sets built on the basis of a load imbalance model. We have obtained very interesting results: for example, for some data sets, the total overheads due to communications and the residual load imbalance only range between 0.88% and 1.69% of the optimal time. These results are better than those those obtained with a CRAFT Fortran (an HPF-like language) implementation of the benchmark. This work is described in the paper:

S.Orlando and R.Perego. "SUPPLE: an Efficient Run-Time Support for Non-Uniform Parallel Loops", Tech. Report TR-CS-96-17, Dip. di Matematica Appl. ed Informatica, Universita' Ca' Foscari di Venezia, 1996 - Submitted for publication to IEEE TPDS. http://www.dsi.unive.it/~orlando/TR96-17-paper.ps.gz

3 List of Participants