Dagstuhl Seminar
on

High Level Parallel Programming: Applicability, Analysis and Performance

Organized by

Murray Cole (University of Edinburgh)
Sergei Gorlatch (Universität Passau)
Jan Prins (University of North Carolina)
David Skillicorn (Queen's University at Kingston)

Schloß Dagstuhl 26. – 30.4.1999

Contents

1 Preface

It is generally acknowledged that programming parallel computers effectively and correctly is a conceptually challenging task for all but the simplest of applications. Consequently, there is widespread research interest in models and methodologies which can assist the process. In order to provide some degree of durability, such approaches must abstract from the detailed characteristics of specific systems, while remaining efficiently implementable by those systems.

The previous Dagstuhl Seminar 9708 brought together a spectrum of researchers with interests related to the "higher order" aspects of this area, ranging from those with theoretical interests in program development to practical systems builders. The present seminar has aimed to focus more closely on the developments in two of the areas which emerged at the original workshop as being on the critical path to progress to include approaches which are more generally "high level" rather than specifically "higher order". By "high level" we mean going beyond the simple extension of sequential languages with communications or shared data primitives, to models and languages in which the expression of conceptual structure is encouraged and supported.

Most interest in parallel programming is motivated by the quest for dramatically improved performance in processing large applications. To gain credibility with that community, we must be able to show that our methods are competitive. The quantification of performance is simplified by the computational structure inherent in the high level approach. This applies equally to attempts to predict performance on the basis of static program analysis and a small number of architecture specific parameters (more commonly know as "cost-modeling") and to the benchmarking and post-hoc analysis of the behaviour of implemented systems.

Similarly, we must be able to demonstrate convincingly that high level parallelism enhances the *applicability* of the underlying technology by simplifying the expression of real programs for real problems (rather than the sanitized and simplified examples appropriate to the early stages of research). Paradoxically, our target audience must also be convinced that there is no loss of expressiveness when dealing with those parallel programming sub-tasks in which there is no well behaved structure to capture.

In summary the following questions and the many subordinate issues they raise were addressed during the seminar:

- How well can high-level parallel programming methods match the performance of more machine specific approaches?
- What do (and should) we mean by performance in this context?
- Can such systems be effectively cost-modelled, and if so, would this be an attractive feature to practitioners?
- Can we support such models with other conceptual tools in ways which enhance their attractiveness?
- Can the tension between the use of abstraction and the requirement for detailed ad-hoc control in certain problems be satisfactorily resolved?

In order to ensure that contributions remain focused on cost and applicability, and to provide some common ground on which debate can be conducted, we circulated the participants well in advance with two realistic problems to act as case studies: (1) the Frequent Sets Problem in data mining, and (2) the Barnes-Hut algorithm for the N-Body problem, Both problems are important in practice but have not been treated well to date because of methodological limitations. Lively and deep discussions on both problems, in particular about such evaluation criteria as succinctness, correctness and clarity, as well as performance, contributed a lot to the success of the seminar:

- The Frequent Sets Problem is one of the basic building blocks of many data mining algorithms. Suppose that an organization has recorded the set of objects purchased by each customer on each visit. The goal of the frequent set problem is to find those (smaller) subsets of objects that appear in more than a given fraction of the sets. This information can be used to, for example, place objects that are often purchased together near each other on the shelf. The algorithm is also applicable in scientific domains, for example to find the 'interesting '' parts of complex simulations.

  Given a set M (all of the possible objects that can be sold) and a bag N of subsets of M (each element of the bag records the subset of objects purchased by one customer in one visit), find all subsets of M that appear in more than a fraction x of the elements of N.

  The problem is trivial in the sense that there is an obvious algorithm. However, the size of the data concerned is so large that it becomes critical to do as little work as possible. Clever algorithms are necessary.

  There have been two main approaches. The first depends on the observation that a set can only be frequent if all of its subsets are frequent. This reduces the number of sets whose frequencies need to be checked. A summary of this approach can be found at `http://www.cs.helsinki.fi/ htoivone/pubs/toivonen.ps.gz` in Toivonen's thesis. The other tries to use the vast mathematical literature on lattices to improve the search. An example is the work of Zaki at Rochester: `http://www.cs.rochester.edu/trs/system-trs.html`

  The frequent set problem fits well with calculational approaches in the sense that it is straightforward to write down a solution, but harder to transform it to an efficient solution.

  This problem was presented to Dagstuhl participants in advance of the workshop. During the workshop, two attacks on the problem were made. Zhenjiang Hu was able to derive a much-improved version of a functional implementation. Its performance was compared to a direct implementation by Christoph Herrmann. On some small synthetic datasets, performance improvements of an order of magnitude were demonstrated. Second, Charles Leiserson pointed out that an approximate algorithm due to P. Gibbons and J. Matias for computing distributions in datasets might be applicable to the problem. Discussion along this line took place during the meeting, although no substantial progress was achieved (the approach has since been extended and shown to work well, however).

- The N-Body Problem: given a self-gravitating system consisting of n distinct particles characterized by their mass, initial position, and velocity, the problem is to compute the force on each particle that is induced by the other particles.

A direct force calculation would require the computation of O(n2) interactions, a large amount of work for particle systems encountered in practice. There exist a variety of methods that compute approximations to the exact solution with reasonable accuracy and with an improved asymptotic complexity.

The Barnes-Hut hierarchical force-calculation algorithm exploits the fact that, at a distance, the combined potential of a group of particles can be approximated by the potential of the center of mass of that group. The algorithm makes use of a hierarchical quad-tree (or oct-tree in 3D) decomposition of the space containing the particles, and associates with each region its center of mass. After the tree is built, the force calculation traverses the tree top down for each particle to accumulate the total force on the particle. Subregions are explored only if the region's center of mass is not sufficiently far away from the particle to be used as an approximation.

The treeForce computation for each particle is independent and can be computed in parallel. Moreover, the recursive force-computations in the tree traversal are independent and can be computed in parallel. The parallelism specified in treeForce is dynamic since the available parallelism increases with the depth of the recursion. It is irregular because the degree of parallelism specified and the locality of the interactions depends on the distribution of the particles.

This problem has been suggested for consideration in Dagstuhl because the algorithm can be succinctly expressed in a high-level notation, yet an efficient implementation is challenging. Furthermore, some comparative performance data is available for low-level implementations.

In order to encourage critical comment on our ideas, we invited a small number of open-minded participants who are prominent in the use of currently dominant parallel programming technologies (such as MPI, HPF and multi-threading). Their pragmatic perspective on our proposals was illuminating.

The 33 participants of the workshop came from 8 countries: 12 from Germany, 6 from the USA, 5 from the UK, 3 from each France, Italy and Japan, and 1 from each Australia and Canada. The organizers would like to thank everyone who has helped to make this workshop a success.

Murray Cole Sergei Gorlatch Jan Prins David Skillicorn

2 Abstracts

Meeting the Real World Half Way

Murray Cole
University of Edinburgh, UK

Higher order skeletal parallel programming has been studied for at least ten years but a breakthrough into the mainstream remains elusive. We speculate that our community must be prepared to "meet the real world half way", by embedding our concepts in frameworks which are familiar to everyday parallel programmers. Two possible avenues are suggested.

"Frame" represents an attempt to define an imperative, block-structured skeletal language in which the skeletons have the status of conventional control constructs and in which ad-hoc parallelism may be introduced at the leaves of the computation.

The "patterns" concept has attracted increasing popularity in the Software Engineering world. Its central concern (of abstraction and sharing of "good solutions" appears related to our own interests. We consider whether "patterns" may offer a bridge between our communities.

Combining Template-based Task Parallelism and HPF Data Parallelism

Salvatore Orlando
University of Venice, Italy

We present $COLT_{\texttt{HPF}}$, a run–time support specifically designed for the coordination of concurrent and communicating HPF tasks. The version of $COLT_{\texttt{HPF}}$ iimplemented on top of MPI requires only small changes to the run–time support of the HPF compiler used. There exists a more portable version of $COLT_{\texttt{HPF}}$, built on top of PVM, which also permits HPF tasks to be dynamically created. Although the $COLT_{\texttt{HPF}}$ API can be used directly by programmers to write applications as a flat collection of interacting data-parallel tasks, we believe that it can be used more productively through a compiler of a simple *high-level coordination language* which facilitates programmers in structuring a set of data-parallel HPF tasks according to common forms of *task-parallelism*. We outline design and implementation issues, and discuss the main differences from other approaches to exploiting task parallelism in the HPF framework. We show how $COLT_{\texttt{HPF}}$ can be used to implement common forms of parallelism, e.g. pipeline and processor farms, and we present experimental results regarding a sample application. The experiments were conducted on an SGI/Cray T3E using Adaptor, a public domain HPF compiler.

P3L, FAN and Transformations

Susanna Pelagatti
University of Pisa, Italy

Structured parallel programming allows to construct applications by composing skeletons, i.e. recurring patterns of task and data parallelism. First academic and commercial experiences with skeleton based systems has demonstrated both the benefits of the approach and also the lack of a special methodology for systematic algorithm design in this framework.

In this talk, I discuss the features of P3L, a skeleton-based system developed at the university of Pisa and its industrial version SKIECL. SKIECL comes with a skeleton integrated environment (SKIE) currently under development by QSW Ltd. Then, I discuss a first step towards a methodology for systematic and sound software development in skeleton system systems by describing a general transformation framework named FAN (Functional Abstract Notation) which is to be integrated within the systems discussed before. The framework includes a new notation for expressing parallel algorithms, a set of semantic preserving transformation rules and analytical estimates of rules' impact on the program performance (joint work with Sergei Gorlatch from the University of Passau).

High-Level Programming for Parallel Cellular Computation

Domenico Talia
ISI-CNR, Italy

Cellular automata provide an abstract parallel computation model that can be efficiently used for modeling and simulation of complex phenomena and systems. This talk discusses the use of cellular automata programming languages and tools for the parallel implementation of real-life high-performance applications. Parallel cellular languages based on the cellular automata model are a significant example of restricted-computation programming that can be used to model parallel computation in a large number of application areas such as biology, physics, geophysics, chemistry, economics, artificial life, and engineering. The design and implementation of parallel languages based on cellular automata provide efficient high-level tools for the development of scalable algorithms and applications.

As a practical example, we discuss the design of parallel cellular programs by a language called CARPET, introducing the main features of this language specifically designed for the development of high-performance cellular algorithms. Furthermore, CARPET programs performance on parallel MIMD architectures are presented.

SEND-RECV Considered Harmful: a New Way To Structure and Optimize Parallelism?

Sergei Gorlatch
Universität Passau, Germany

The current difficulties in the programming for parallel and distributed systems are mainly due to the low level of the language constructs employed. In particular, the communication libraries like MPI and PVM are often compared with Assembler-like languages of the 60's, and their use is mainly explained by the high performance.

We argue that the way out of this crises of parallelism could be similar to the "structured programming" approach taken previously in the sequential setting. Similar to famous Dijkstra's motto "GOTO considered harmful", we propose to consider individual SEND and RECEIVE primitives harmful for parallel programs. Structured parallel programming should try and avoid using pairwise communications, and rely mostly on the collective communication primitives like broadcast, reduce, scan, etc. We demonstrate that many important applications can be efficiently implemented in such kind of parallel language.

The main advantage of the new way of programming, which we call *collective parallelism*, is its formally-based, easy-to-use design methodology. We provide a set of composition rules for programming with collective operations, and report some experimental results with collective parallelism on modern parallel machines.

Using Explicit Metadata to Separate the Performance Issues in Adaptive Applications

Paul H. J. Kelly
Imperial College, UK and University of California, San Diego, USA

Adaptive applications explicitly modify their structure in response to the evolution of the computation, and/or of the computational environment. The key issue in building tools to support adaptivity is the design of metadata. This talk concerns the role of metadata both for data structures and to represent the iteration space of the computation itself. Three examples are considered: a run-time self-optimising parallel matrix library, a library for block-irregular adaptive mesh applications with explicit metadata manipulation, and, as an example of an application with semi-structured irregularity both in data and in its iteration space, the Barnes-Hut n-body algorithm. This analysis leads to a focus for future research on tools which allow adaptive software to be built from pre-existing components, with very rich opportunities for adaptive, run-time restructuring.

Virtual BSP: Parallel Processing on Transient Processors

Boleslaw Szymanski
Rensselaer Polytechnic Institute, USA

Low cost, high speed and wide availability of workstations combined with improving bandwidth and latency of the interconnecting networks make Networks of Workstations (NOWs) an increasingly popular environment for parallel processing. A large number of workstations in a NOW are idle at any given time and their unused cycles can be used to perform additional parallel computations. A particular workstation is available for the additional computation only when it is not being used by its owner. However, a parallel program whose component processes synchronize during execution, stops making any progress if even a single participating workstation becomes unavailable. Parallel computations in such an environment must therefore adapt to the changing computing environment to deliver acceptable performance.

In this paper we describe an extension of the Oxford Bulk Synchronous Parallel Library to enable the BSP user to harvest idle cycles in a network of non-dedicated workstations for frequently synchronizing parallel computations. The extended library, called the Adaptive BSP (A-BSP for short) Library employs, transparently to the user, eager replication of data and lazy replication of computations and process migration to continuously map a set of BSP processes to currently available workstations. We describe results of applying the A-BSP Library to two large computational problems.

SCL: Flexible library support for data-parallel languages

Gabriele Keller
University of Tsukuba, Japan

Many high-level, data-parallel languages are collection oriented. That is, parallelism can be expressed by applying a fixed set of possibly higher-order operations to a designated aggregate data-structure. The SCL library is designed to facilitate the implementation of such languages by providing run-time support for parallel irregular data-structures, load-balancing, and communication operations. It differs from existing libraries as it offers no monolithic parallel computations on the data-structures, as such an approach often hinders important optimisations. Instead, it provides the communication operations as building blocks, which, combined with optimised sequential code, make up the parallel computations. The number of functions whose implementation is machine-dependent is kept intentionally small to obtain portability.

Costing Parallel Programs - What Next?

C. Barry Jay
University of Technology at Sydney, Australia

A realistic cost model for parallel programs will allow programs to be customized to minimize costs under changing circumstances. The most pressing circumstance has been the wide variety of hardware and architectures available, so that portability to new platforms has been the key goal. Models such as BSP have been successful at exploiting a small set of hardware parameters, such as latency. However, the costings could be refined by other performance factors which are required for better accuracy.

The most important of these is knowledge of the size and shape of the data structures which are in play. Even in the sequential case, one can find specialist libraries of algorithms intended to match the data structures to machine properties, e.g. number of registers. The same issues arise in the parallel setting, but magnified. Such shape information may be obvious for the inputs, but composition of programs, or divide-and-conquer techniques, will quickly obscure the structure. The potential of this approach is shown by the FISh language. It supports static analysis of shapes which are then available for use in costing. Each program $f : X - - > Y$ has a shape $\#f : X - > Y$ which maps the shape of the input to that of the output.

The impact of registers, and other parameters such as cache size, etc. on performance shows that the hardware model should be customized for each individual platform, to take account of those parameters deemed important by the manufacturers and/or implementers of the language primitives. This can be done by using a computational monad to represent costs as a function of shapes and hardware parameters. Each machine would support its own monad, using its own hardware parameters, but the costing of composite programs would be performed in a standard way. More precisely, each program f as above will have a cost given by $costf : X - > (H - > C) * Y$, which will take the shape of the input and produce a pair, consisting of the cost function (from hardware parameters to costs, say, time) and the shape of the result. The latter can be used in costing the next step of the computation, and the costs added pointwise.

The Algebraic Path Problem Revisited

Sanjay Rajopadhye
IRISA, France

We derive an efficient linear SIMD architecture for the algebraic path problem. For a graph with n nodes, our array has n processors, each with $3n$ memory cells, and computes the result in $3n^2 - 2n$ steps. Our array is ideally suited for VLSI, since the controls is simple and the memory can be implemented as FIFOs. I/O is straightforward, since the array is linear. It can be trivially adapted to run in multiple passes, and moreover, this version *improves* the work efficiency. For any constant α, the running time on $\frac{n}{\alpha}$ processors is no more than $(\alpha + 2)n^2$. The work is no more than $(1 + \frac{2}{\alpha})n^3$ and can be made as close to n^3 as desired by increasing α.

The Network of Tasks Model

David B. Skillicorn
Queen's University at Kingston, Canada

A parallel programming model requires good properties with respect to both software development (clean semantics, transformation system, cost model) and execution (reasonable efficiency). Several such models are known, for example skeleton-based languages such as P3L, and less-structured alternatives such as BSP. These models have a common property: their operations are machine-filling and their composition is simple sequential composition.

The resulting structure seems too limiting for the whole range of possible high-performance applications. It is interesting to see how much composition can be generalized while still preserving, among other things, a compositional cost model. This is the goal of the Network of Tasks Model.

Programs in the NOT model are acyclic networks of nodes (except for a single loop construct) joined by directed edges representing communication. Nodes may be written in any combination of parallel programming languages provided only that (a) a cost expression parameterized in the number of processors is provided, and (b) the node may be adapted to use fewer processors. Under these conditions, a technique called work-based allocation schedules the graph so that each layer completes at roughly the same time and the total work of the nodes is preserved by the implementation (which makes costs transparent).

The simple semantics of NOT graphs makes it possible to define a refinement calculus derivation methodology for them. In particular, it is straightforward to handle residuals on graphs, which is of particular importance when software reuse is a concern.

The primary difference between the NOT model and other task graph approaches of the same general kind (e.g. COLT, Skie, 2L) is their granularity. NOT programs are intended to have quite large nodes and to preserve no state between them; whereas these other models have nodes that are like those of conventional skeleton languages.

Abstract Parallel Machines for the Polytope Model

Nils Ellmenreich
Universität Passau, Germany

Since scientific algorithms are often easily expressible in a functional language and they are often used for huge problem sizes, we propose an approach to statically parallelize this class of algorithms. The method, polyhedral parallelization, was adapted to the functional setting. For the code generation, we chose a tree of Abstract Parallel Machines (APMs), each being specified in the non-strict functional language Haskell. Each node in the tree represents a design decision, probably leading to different APMs. Program transformation ports code from one APM to the next. In future work, APM code will be translated down to native parallel code.

The Parallel Functional Language Eden

Ulrike Klusik, Rita Loogen, Steffen Priebe
University of Marburg, Germany

Eden is a Haskell-based functional language which provides process abstractions and instantiations for the explicit definition of dynamically evolving process networks. Treating parallelism explicitly with declarative features promises runtime advantages in comparison to more conventional approaches to parallel functional programming, where annotations are used to indicate implicit parallelism and a process notion is only implicit. Eden process systems are distributed, there exists no shared memory or global address space. Processes communicate via head-strict communication channels.

Eden has been implemented by modifying the Glasgow Haskell compiler. The runtime system is based on the GUM system, the runtime system of Glasgow parallel Haskell, but incorporates substantial changes and extensions. A simple case study with the bitonic merge sort algorithm shows that the pulsating process system produced by the divide and conquer version of this algorithm can easily be replaced by a stable process system in which the data pulsates. Measurements with the Eden system show runtime benefits for the stable process tree.

BSP Cost Analysis of Skeletal Programs

Yasushi Hayashi and Murray Cole
University of Edinburgh, UK

The skeletal approach to parallel programming advocates the use of program constructors, or "skeletons", which abstract useful patterns of parallel computation and interaction. This is held to ease the burden on the programmer, who is freed to think in terms of these higher-level strategies while being absolved of responsibility for their detailed implementation. When embedded in a purely functional language opportunities naturally arise for program development in the transformational style in which an initial obviously correct version is refined by a sequence of meaning-preserving rewrites into a more efficient, but semantically equivalent form.

The program transformation process has long been recognised as a suitable target for automated support. The bulk of work has focused on the fundamental question of the semantic soundness of each step, with responsibility for choosing steps and for judging their effect on performance left to the programmer's intuition. Support for cost-prediction is hampered by its intractability in the general case. In response, C.B. Jay et al. defined a cost calculus for a simple shapely functional language Vec, targeting the PRAM and its cost model as the underlying parallel machine. we are developing a similar framework but choose the more realistic BSP model as the target architecture. This requires us to account for communication costs which are ignored by the PRAM. These costs impact upon the implementation choices made by the (hypothetical) Vec to BSP compiler and consequently upon the structure of the cost calculus which reflects them. The talk outlines the issues which must be addressed, and describes the implementation mechanism and the new cost calculus itself. An example of its application is presented.

High level BSP programming

Gaetan Hains
Universite d'Orleans, France

We argue against the use of concurrent semantics in MPI+C programs for implementing parallel algorithms. A language design called BSML (Bulk Synchronous ML) is outlined and formalised by the BS-lambda theory. A library implementation of BSML shows the agreement of timings with the BSP model: in most cases to within 50often to within 10defined with barrier fusion semantics. It is shown that BS-lambda extended with this operation is non-confluent. Network-recursive definitions and nested parallelism therefore appear difficult to include in the pure-functional part of this framework.

Parallelizing Compilation of Higher-Order Functional Programs using Skeletons - A Compiler for HDC

Christoph A. Herrmann and Christian Lengauer
Universität Passau, Germany

We present a compiler for the functional language HDC, which aims at the generation of efficient code from high-level programs. HDC, which is syntactically a subset of the widely used language Haskell, facilitates the clean integration of skeletons with a predefined efficient parallel implementation into a functional program. Skeletons are higher-order functions which represent program schemata that can be specialized by providing customizing functions as parameters. The only restriction on customizing functions is their type; they can be composed of skeletons again. With HDC, we focus on the divide-and-conquer paradigm, which has a high potential for efficient parallelization.

The most important skeletons we use are explained. We describe the compiler phases, especially the higher-order elimination as the most important phase, on a simple example. Then, the generic implementation of skeletons is presented on the example of map, for three different parallel models. We discuss the example of polynomial product using Karatsuba's schema. Our experimental results are the basis of a comparison of the sequential code for this example with the code generated by the Glasgow Haskell compiler ghc-4.01. We concluded with an example program for the Frequent Set problem, which illustrates the large variety of possible skeleton usages.

Evaluation Strategies for Parallel Haskell

Kevin Hammond
University of St. Andrews, UK

This talk introduces evaluation strategies, higher-order functions that can be used to separate algorithmic from behavioural code.

An evaluation strategy has type: `type Strategy a = a -> ()`, and is defined to have some effect on the argument of type "a", through shared graph. Basic strategies are used to control sequential evaluation degree, so that for example the "rnf" strategy will cause reduction of an expression to full normal form. Since evaluation strategies are completely normal functions, they can be naturally composed, nested, returned as program results, passed to other functions etc.

An evaluation strategy can be applied with the "using" function, which separates the two components of the program.

```
using ::  a -> Strategy a -> a
x `using` s = s x `seq` x
```

For example,

```
parMap ::  Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

The algorithm is "map f xs", the behaviour is "parList strat", where "strat" parameterises "parMap". So `parMap rnf` reduces all the results of the map in parallel to full normal form. `parMap (parList rnf)` would create two levels of nested parallelism etc.

Evaluation capture many common programming paradigms, including divide-and-conquer, pipelining, bounded buffers, and SPMD. They act as a dual to the skeleton approach: skeletons are parameterised on control, whereas strategies parameterise the algorithm. Efficient skeletons could be used to implement particular strategies, or strategies could be used to specify the semantics of particular skeletons.

Evaluation strategies have been used successfully in a number of large parallel Haskell programs, including the 47000 line Lolita natural language processor, and reasonable speedups have been obtained. This talk shows how they can be applied to the Dagstuhl n-body challenge application.

We are now looking at ways to extend strategies to new settings such as Java, and to apply new ideas such as shape, cost models or type-based reasoning.

Parallel Functional Programs Translated into Java

Ralf Ebner
TU München, Germany

The functional coordinating language FASAN was designed for distributed numerical applications that are based on recursive algorithms with hierarchical data structures. The execution model of a FASAN program is a dynamic data flow graph. It avoids unnecessary synchronization and ensures data locality as well as direct communication channels between function calls using the concept of "data wrappers".

A first C- and PVM-based implementation of FASAN was successful w.r.t. execution time and speedup. But the resulting programs were difficult to debug since the programmer was faced with the internals of the data flow runtime system.

The actual implementation does not construct the data flow graph explicitly but rather uses features of Java to simulate its behavior. For this purpose, it maps parallel function calls to threads and distributed function calls to remote method invocations (RMI). Elements of data structures and result parameters of functions get stored in wrappers subclassing Java's "UnicastRemoteObjects". The wrappers serve as global references, as keys in object caches improving data locality, and as monitors that delay synchronization until actual calls to selector functions occur.

Implementations of the Barnes-Hut algorithm and of the adaptive recursive substructuring method for finite element simulations have demonstrated the usability of the new FASAN system, but still suffer from poor performance of RMI, which hopefully will be improved according to recommendations of the Java Grande Forum Report.

The Stampede Cluster Programming System for Interactive Multimedia Applications

Rishiyur S. Nikhil
Compaq Cambridge Research Laboratory, USA

Stampede is a C-based parallel programming system with: (1) high-level support for writing applications involving streams and dynamic parallelism, and (2) ability to run transparently on an SMP or a cluster of SMPs. We describe an example intelligent computer vision application, Stampede's implementation, and performance.

Programming Shared-Memory Multiprocessors Using the Cilk Multithreaded Language

Charles E. Leiserson
MIT, USA

Cilk is a language being developed in the MIT Laboratory for Computer Science with the goal of making parallel programming easy. Cilk minimally extends the C programming language to allow interactions among computational threads to be specified in a simple and high-level fashion. Cilk's provably efficient runtime system dynamically maps a user's program onto available physical resources using a "work-stealing" scheduler, freeing the programmer from concerns of communication protocols and load balancing. In addition, Cilk provides an abstract performance model that a programmer can use to predict the multiprocessor performance of his application from its execution on a single processor. Not only do Cilk programs scale up to run efficiently on multiple processors, but unlike existing parallel-programming environments, such as MPI and HPF, Cilk programs "scale down": the efficiency of a Cilk program on one processor rivals that of a comparable C program.

In this talk, I provide a brief tutorial on the Cilk language. I explain how to program multithreaded applications in Cilk and how to analyze their performance. I illustrate some of the ideas behind Cilk using the example of MIT's championship computer-chess programs, *Socrates and Cilkchess. I also briefly sketch how the software technology underlying Cilk works.

See `http://supertech.lcs.mit.edu/cilk` for more background on Cilk and to download the Cilk-5.2 manual and software release.

A Practical Solution to Programming Parallel Computers

Lawrence Snyder
University of Washington, Seattle, USA

To program parallel computers well, i.e. so applications realize high performance across all parallel platforms, it is essential to be able to compare alternate solutions to subcomputations. This implies there must be a faithfully implemented performance model available to programmers with which to make the comparisons.

ZPL, an implemented and freely available data parallel array language, is a witness to the feasibility of this idea. The performance model presented by ZPL is known as the what-you-see-is-what-you-get model. The approach is described with running examples of ZPL code.

The parallel programming language ForkLight

Christoph Kessler
Universität Trier, Germany

ForkLight is an imperative, task-parallel programming language for massively parallel shared memory machines. It is based on ANSI C, follows the SPMD model of parallel program execution, provides a sequentially consistent shared memory, and supports dynamically nested parallelism. While no assumptions are made on uniformity of memory access time or instruction-level synchronicity of the underlying hardware, ForkLight offers a simple but powerful mechanism for coordination of parallel processes in the tradition and notation of PRAM algorithms: Beyond its asynchronous default execution mode, ForkLight offers a mode for control–synchronous execution that relates the program's block structure to parallel control flow. This directly maps to parallel divide-and-conquer implementations. We give an overview of the implementation of ForkLight by a source-to- source compiler. The run-time system uses a very small target machine interface consisting of only seven core routines for shared memory access and process management, that are currently implemented using P4. We also report implementation results for the N-body problem. We observe very good scalability on the SB-PRAM even for very small problem sizes. Finally we briefly introduce some key features of the BSP language NestStep, namely nesting of supersteps and a BSP-compliant virtual shared memory implementation.

A Coordination Language for Mixed Task and Data Parallel Programs

Thomas Rauber and Gudula Rünger
Universität Halle and Universität Leipzig, Germany

We present a coordination model to derive efficient implementations using mixed task and data parallelism. The model provides a specification language in which the programmer defines the available degree of parallelism and a coordination language in which the programmer determines how the potential parallelism is exploited for a specific implementation. Specification programs depend only on the algorithm whereas coordination programs may be different for different target machines in order to obtain the best performance. The transformation of a specification program into a coordination program is performed in well-defined steps where each step selects a specific implementation detail. Therefore, the transformation can be automated, thus guaranteeing a correct target program. A cost model is used to ensure that the transformations result in an efficient program. We demonstrate the usefulness of the model by applying it to solution methods for differential equations. We show that for different parallel machines, different execution schemes lead to the most efficient parallel program.

Irregular Computation in Fortran -or- Does Fortran Count as a High-Level Programming Model?

Jan Prins
University of North Carolina, Chapel Hill, USA

Computations over non-uniform and/or recursive data structures such as nested sequences and variable-depth trees can be found at the heart of many efficient algorithms. However, irregular computations such as these pose complex performance challenges as the source of parallelism and reference patterns are not known a priori.

We are exploring the efficient implementation of such algorithms on various shared-memory multiprocessors. Our approach is to express irregular computations in fully parallel form using the data abstraction and parallel loop facilities found in modern Fortran dialects (Fortran 90 or Fortran 95), and to transform the parallel loops to optimize the performance for particular target processors. The transformations are (1) loop serialization, (2) loop scheduling via OpenMP directives, (3) loop range splitting and (4) flattening of nested parallel loops. We use Fortran because (1) it is widely used as a performance-oriented programming language and consequently tends to have high quality compilers and implementations, and (2) the language is closed under the transformations used, in particular Fortran supports the primitives introduced in the flattening transformation.

In this talk the implementation of the Barnes-Hut n-body algorithm on an SGI Origin 2000 is described. Using all four transformations above we were able to improve single-processor performance of the tree-force calculation by a factor of 6 and obtain a superlinear speedup (due to cache effects) on parallel runs up to 16 processors. The performance appears to match or exceed that reported in the literature, although this is a difficult claim to make since the B-H algorithm has many parameters affecting performance, most of which are unreported in the literature.

Is Nested Data Parallelism Portable?

Manuel M. T. Chakravarty
University of Tsukuba, Japan

Research on the high-performance implementation of nested data parallelism has, over time, covered a wide range of architectures. Both scalar and vector processors as well as shared-memory and distributed memory machines were targeted. The Nepal Compilation System integrates this technology, in an attempt to provide a portable parallel programming system. Essential are two program transformations, flattening and calculational fusion, which even out irregular parallelism and increase locality of reference, respectively. The resulting program is mapped to C plus calls to a portable, light-weight collective-communication library. First experiments on scalar, vector, and distributed-memory machines support the feasibility of the approach.

Diffusion: Calculating Efficient Parallel Programs

Zhenjiang Hu, Masato Takeichi and Hideya Iwasaki
University of Tokyo, Japan

Parallel primitives (skeletons) intend to encourage programmers to build a parallel program from ready-made components for which efficient implementations are known to exist, making the parallelization process easier. However, programmers often suffer from the difficulty to choose a combination of proper parallel primitives so as to construct efficient parallel programs. To overcome this difficulty, we shall propose a new transformation, called *diffusion*, which can efficiently decompose a recursive definition into several functions such that each function can be described by some parallel primitive. This allows programmers to describe algorithms in a more natural recursive form. We demonstrate our idea with several interesting examples. Our diffusion transformation should be significant not only in development of new parallel algorithms, but also in construction of parallelizing compilers.

Compilation of a Specialized Language for Massively Parallel Computers

Julien Mallet
Irisa/Inria, Rennes, France

We propose a specialized language based on program skeletons encapsulating data and control flow for which an accurate cost analysis of the parallel implementation exists. The compilation process deals with the automatic choice of the data distributions on the processors through the accurate cost guaranteed by the source language. This allows to obtain an automatic compilation with an efficient parallel code (the distributions representing a global choice of parallel implementation).

The compilation process is described as a series of program transformations, each transformation mapping one intermediate skeleton-based language into another. The target language is an SPMD-like skeleton-based language straightforwardly translating into a sequential language with calls to communication library routines. The main compilation steps are : the size analysis, the in-place updating transformation, the explicitation of the communications and the data distributions choice.

The approach can be seen as a cross-fertilization between techniques developed within the FORTRAN parallelization and skeleton communities.

EPOS: Paving the Path for Parallel Applications

Antonio Augusto Froehlich and Wolfgang Schroeder-Preikschat
GMD-FIRST and Universität Magdeburg, Germany

Every time more applications demand performance levels that can only be achieved by parallelization. In order to properly support them, new operating systems and tools are to be conceived. Our experiences developing runtime support systems for parallel applications convinced us that adjectives such as "generic" and "all purpose" do not fit together with "high performance", whereas different parallel applications have quite different requirements regarding the operating system. Even apparently flexible designs, like micro-kernel based operating systems, may imply in waste of resources that, otherwise, could be used by applications.

The promotion of configurability has been properly addressed by the PURE operating system. PURE is designed as a collection of configurable classes that can be seen as building blocks to assemble application-oriented operating systems. Approaches like this, although doing much for performance, reusability and maintainability, usually are not enough to support application programmers, since the number and the complexity of available building blocks grows quickly with the system evolution. In such a context, selecting and configuring the proper building blocks becomes a nightmare and yields a gap between that what the operating system offers and that what the applications expect.

EPOS aims to deliver, whenever possible automatically, a customized runtime support system for each parallel application. In order to achieve this, EPOS introduces three main concepts: 1 - adaptable, scenario independent system abstractions that result from composing PURE building blocks into application-ready abstractions. These abstractions are designed to be as much independent from the execution scenario as possible. 2 - Scenario adapters that adapt existing system abstractions to a given execution scenario, for instance, by making an existing thread abstraction ready to run in a SMP configuration. 3 - Inflated interfaces that export the system abstraction repository by gathering several different implementations of each system abstraction in a single, well-known interface.

An application designed and implemented following the guidelines behind these concepts can be submitted to a tool that will proceed syntactical and data flow analysis to extract an operating system blueprint. This blueprint is then refined by dependency analysis against information about the execution scenario acquired from the user via visual tools; and then submitted to another tool that will generate the application-oriented operating system. With this approach, EPOS shall diminish the gap that usually separates operating systems from parallel applications.

Scheduling of Data Parallel Modules

Thomas Rauber and Gudula Rünger
Universität Halle and Universität Leipzig, Germany

Algorithms from scientific computing often exhibit a two-level parallelism based on potential task and data parallelism. We consider the parallel implementation of those algorithms on distributed memory machines. The efficient parallel implementation depends on several design decisions including the execution order of the tasks and the building of subgroups of processors for a group parallel execution. This gives rise to a multiprocessor task scheduling problem. The task structure is described in a directed acyclic graph where the nodes represent data parallel modules and the edges represent data dependencies mainly caused by multidimensional data structures. After a normalization step, phases of independent modules are scheduled one after another. We propose two algorithms: a greedy algorithms which first decides on the execution order of tasks and then on the group sizes and a dynamic programming approach which determines a module execution plan in one step.

Piecewise Execution of Nested Parallel Programs

Wolf Pfannenstiel
Technische Universität Berlin, Germany

Flattening nested parallel programs can lead to high memory requirements of the transformed programs because all the parallelism is exposed at once. Piecewise execution is a special kind of serialisation that computes only vector pieces of constant size at a time. This reduces memory requirements but comes at the price of a runtime overhead for managing the computation across the pieces and more complicated implementation. However, piecewise execution can be combined with other optimisation techniques like fusion and tupling transformations. It can be applied to memory-critical parts of a program (e.g. matching pairs of generators and accumulators) such that other parts are not affected. This property holds for a "formal specification" of program transformations as well as for the multi-threaded execution model that is proposed here. First experiments on a small example program show that piecewise execution and fusion techniques mix well and lead to the best measured running times on a Cray T3E.

3 List of Participants