

Our Pattern Language (OPL)

Author: Tim Mattson

Shepherd: Kurt Keutzer

Introduction

The case has been well established in the literature [Asanovic09]: parallel processors will dominate most if not every niche of computing. In response, the software industry must quickly evolve into a “parallel software” industry.

This does not mean that every programmer needs to master parallel programming. Modern software practices are changing. In the past, software was developed by teams of programmers that created applications as “top to bottom” programs. Current trends are breaking with this “top to bottom” model and are more modular. Applications programmers working within domain-specific software frameworks compose modules to create a high level application. Smaller groups of programmers with deeper knowledge of computer science and how software maps onto hardware create the frameworks and tune modules to the needs of a particular platform. This smaller group of programmer, sometimes called “technology programmers” or efficiency layer programmers, need to master parallel programming and understand how to manage concurrency efficiently. The vastly larger population of applications programmers, however, only need to be weakly aware of parallel programming.

To foster the emergence of a new parallel-software industry, therefore, we need to match our approach to modern software engineering trends. While earlier research in parallel programming emphasized parallel algorithms for specific computations, we need a more nuanced approach. We need to understand the software architecture that guides the design of high level application frameworks. We need to understand the key classes of computations and how they map onto parallel algorithms. Finally, we need to map these onto foundational models for how parallel modules execute, so we can intelligently compose modules and support their efficient execution.

Solving this parallel-software engineering problem is an ambitious undertaking. To accomplish this we need a “roadmap” to guide us and a conceptual framework to organize our understanding of the problem. This is the only way we can create the right elements of our solution and ensure that they correctly fit together. For our conceptual framework, we will build a pattern language of parallel programming ([Alexander72], [Gamma94], [Mattson04]). Design patterns give a name to recurring solutions experts “take for granted” in a problem domain. These patterns can be organized into a web of interlocking of patterns that guide a designer from the beginning of a design problem to the end point; the realization of a design on real hardware. Our working title for this pattern language is OPL or “Our Pattern Language”. OPL will be published on our project wiki:

<http://parlab.eecs.berkeley.edu/wiki/patterns/patterns>

Our Pattern language of parallel-software engineering will serve three distinct purposes:

1. Education: Patterns have pedagogical value and help new parallel programmers more quickly master the field.
2. Communication: Patterns give experts in parallel software design a common terminology to use as they advance the field.
3. Design: Patterns, and the way they are organized into a pattern language, guide the design of frameworks.

In this paper, we will present the overall structure of Our Pattern Language (OPL) and briefly list the patterns present at each layer of the language. Since a pattern language is mined from existing practice, experienced parallel programmers will most likely understand the intent of the patterns just from these brief descriptions. More detailed descriptions are available on our wiki.

The Scope of OPL

Patterns are a recurring solution to a well known problem within a well defined context. The “well defined” context is essential to keep the patterns focused. When the context is vague, the solutions degrade into general platitudes as opposed to concrete actions that solve real problems. This need for carefully defining context so the scope of a solution is clear is just as important for a pattern language.

OPL is restricted to describing the design of parallel software. It is concerned with software architecture and ways to design and implement parallel algorithms. The emphasis is on the needs of the application programmer, not compiler writers or programmers that write operating systems or parallel runtime libraries.

OPL is not specific to any particular application domain. Programmers working within an application domain will find their own patterns. These may be of vital importance to programmers within that domain, but they are not applicable across application domains and hence they are out of scope for OPL.

In addition, OPL can't cover every aspect of computer science. There are patterns that play a foundational role for serial, parallel, system and applications software. These patterns are important, but if we tried to include them in OPL, the project's scope would become unmanageable. Hence, you will not find patterns in OPL that discuss, for example, techniques for inlining functions or rearranging instructions by a compiler to improve throughput. OPL is a pattern language for parallel application programming, not a synthesis of all knowledge in computer science.

OPL and programmer roles

To describe the patterns and explore ways they might be used, we find it useful to think of four types of programmers:

1. Application programmer: an expert in the problems that matter to a community of users. These programmers may have little background in computer science and usually lack detailed knowledge of hardware architecture. A vision processing expert concerned with

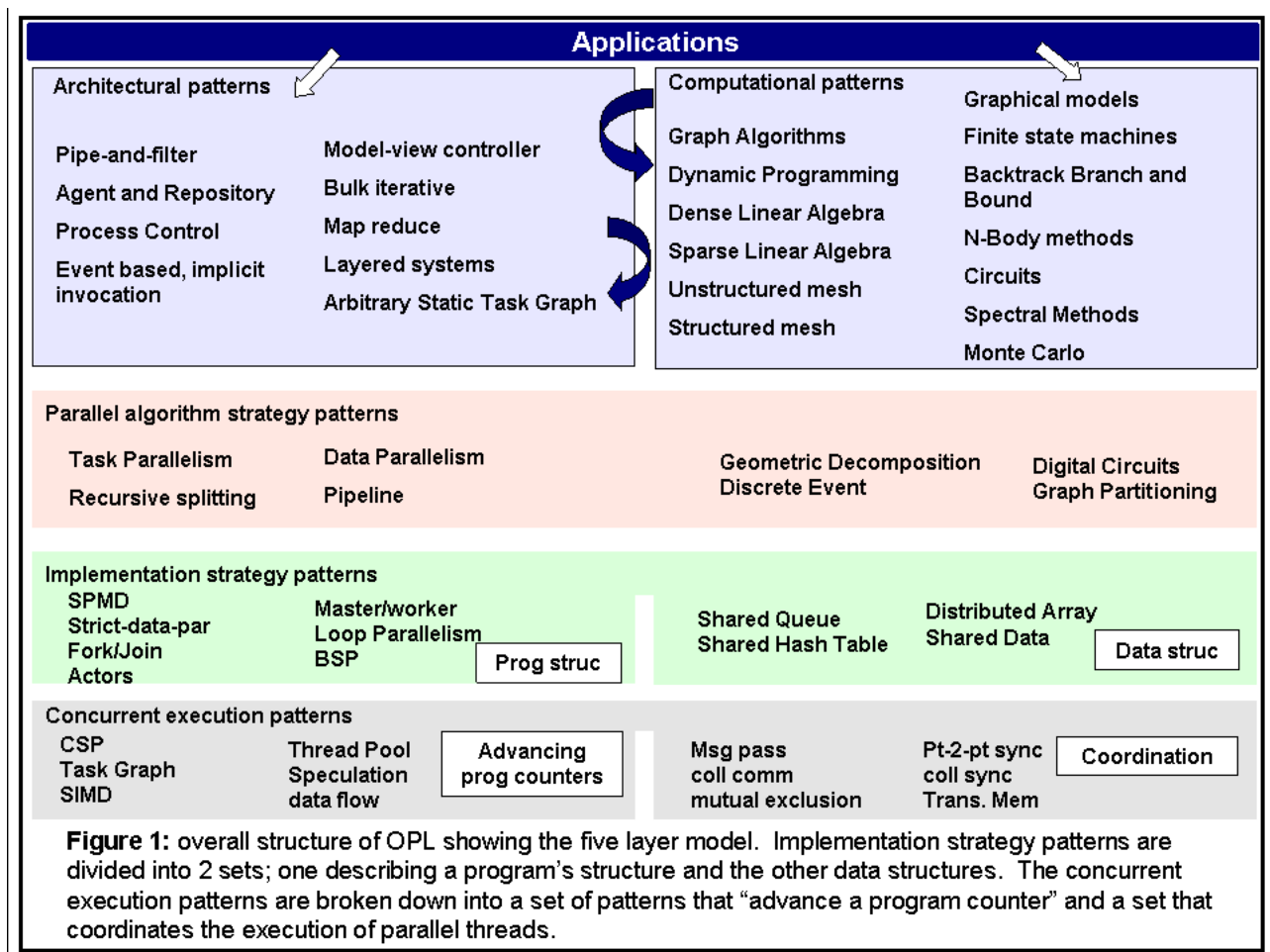
image features, their extraction, and techniques for image classification and recognition is an example of such an application programmer.

2. Application framework developer: These programmers build frameworks and key software modules to support the application programmers. These developers will typically have a broad range of skills: They must have enough understanding of the target application domain so they can have meaningful interaction with application programmers, but they are not domain experts. They must understand the computational implications of various algorithmic choices but they probably would not be developers of original algorithms. They need to understand the state-of-the-art in software technology but probably not develop general software tools or languages themselves. An image retrieval framework developer who builds libraries of various feature extraction and classification procedures is an example of such a programmer.
3. Parallel programming framework developers: These are the parallel programming experts who create the libraries and generic parallel programming frameworks used to create the applications frameworks. They must understand a great deal about parallel algorithms, concurrency control mechanisms and at a high level, the way software interacts with the hardware platform.
4. Platform programmers: These programmers are computer scientists that understand details of a particular platform. They create the runtime libraries and low level tools to enable programmers working at higher levels of abstractions. For example, these are the programmers who build runtime libraries, implement frameworks such as CUDA or OpenCL, or optimize the low level math libraries used in applications.

The numbers of programmers in each role drop as you move from application programmers to the platform programmers. For example, in the computer gaming industry today, on the order of 1% of programmers are “platform programmers”. While we believe that in time programming will become stratified in this way, we acknowledge that today the required application frameworks are not broadly available. Only “tall skinny” programmers that span, to some degree all of these categories, will be able to build highly-parallel applications and application frameworks.

Overall Structure of OPL

Engineering software is an extremely complex undertaking. To manage complexity, we commonly break the problem down into layers and think of the software as a stack. Each stack addresses a portion of the problem. Programmers need to understand the overall layout of the stack but in practice; they can restrict their work to the layers that directly impact their part of the software project.



Hence, a pattern language for software design should also be organized into a stacked, layered system. As shown in Figure 1, we organize OPL into five major boxes

1. Architectural patterns: the architectural styles or overall organization of the application and the way the elements that make up the computation interact. These constitute the "boxes and arrows" a software architect would draw on a whiteboard to describe for the application.
2. Computational patterns: These are the core classes of computations that make up the application. These patterns describe what is computed "in the boxes on the whiteboard".
3. Parallel Algorithm strategy patterns: These are the high level strategies that describe how to exploit concurrency in a parallel computation.
4. Implementation strategy patterns: These are the structures that are realized in source code to support (a) how the program itself is organizes and (b) the common data structures specific to parallel programming.
5. Concurrent execution patterns: These are the approaches used to support the execution of a parallel algorithm. This includes (a) strategies that advance a program counter and (b) basic building blocks to support the coordination of concurrent tasks.

In OPL, the top two layers, software architecture and computations, are placed side by side with connecting arrows. This shows the tight coupling between these high level patterns and the iterative nature of how a designer works with them. The lower three layers of OPL build off earlier and more traditional work in parallel algorithms [Mattson04]. It is important to note, however, that even though we present OPL as a top to bottom layered model, in practice a programmer may move around in more complex ways; moving back and forth between layers as decisions made at lower layers cause the software designer to rethink design decisions made at higher layers.

In the following sections, we will briefly define the patterns that make up each layer of OPL. For each layer, we will define the goal of the patterns at that layer, the artifacts from the design process produced at that layer, and the primary target audience for these patterns.

Architectural patterns

Goal: These patterns define the overall architecture of a program.

Output: The overall organization of a program; the “boxes and arcs” an architect would write on a whiteboard in describing an application.

Primary target audience: Application programmers and application-framework developers

- Pipe-and-filter: view the program as filters (pipeline stages) connected by pipes (channels). Data flows through the filters to take input and transform into output.
- Agent and Repository: a collection of autonomous agents update state managed on their behalf in a central repository.
- Process control: the program is structured analogously to a process control pipeline with monitors and actuators moderating feedback loops and a pipeline of processing stages.
- Event based implicit invocation: The program is a collection of agents that execute asynchronously: listening for events they are subscribed to, responding to events, and issuing events for other agents into a shared medium. The architecture enforces a high level abstraction so invocation of an agent is implicit; i.e. not hardwired to a specific controlling agent.
- Model-view-controller: An architecture with a central model for the state of the program, a controller that manages updates of the state, and one or more agents that export views of the model.
- Bulk Iterative: A program that proceeds iteratively ... update state, check against a termination condition, complete coordination, and proceed to the next iteration.
- Map reduce: This makes more sense to me: A two phase program: In phase one, a single function is *mapped* onto independent sets of data. In phase two the results of mapping that function on the sets of data are reduced. The reduction may be a summary computation, or merely a data reduction.
- Layered systems: an architecture composed of multiple layers that enforces a separation of concerns wherein (1) only adjacent layers interact and (2) interacting layers are only concerned with the interfaces presented by other layers.
- Arbitrary static task graph: the program is represented as a graph that is statically determined. In other words the structure of the task graph does not change once the

computation is established. This is a broad class of programs in that any arbitrary graph can be used.

Computational patterns

Goal: These patterns define the computations carried out by the components that make up a program.

Output: Definitions of the types of computations that will be carried out. In some cases, specific library routines will be defined.

Primary target audience: Application programmers and application-framework developers
These patterns describe computations that define the components in a program's architecture.

- Backtrack, branch and bound: Used in search problems ... where instead of exploring all possible points in the search space, we continuously divide the original problem into smaller sub-problems, evaluate characteristics of the sub-problems, set up constraints according to the information at hand, and eliminate sub-problems that do not satisfy the constraints.
- Circuits: used for Boolean or bit-level computations, representing them as Boolean logic or combinational circuits wired together with state elements such as flip-flops.
- Dynamic programming: recursively split a larger problem into sub-problems but with memorization to reuse past sub-solutions. The optimal result of the problem is assembled from the optimal results of the sub-problems.
- Dense linear algebra: represent a problem in terms of dense matrices using standard operations defined in terms of Basic linear algebra (BLAS).
- Finite state machine: Used in problems for which the system can be described by a language of strings. The problem is to define a piece of software that distinguishes between valid input strings (associated with proper behavior) and invalid input strings (improper behavior).
- Graph algorithms: a diverse collection of algorithms that operate on graphs. Solutions involve representation of the problem as a graph, and developing a graph traversal or partitioning that results in the desired computation.
- Graphical models: probabilistic reasoning problems where the problem is defined in terms of probability distributions represented as a graphical model.
- Monte Carlo: A large class of problems where the computation is replicated over a large space of parameters. In many cases, random sampling is used to avoid exhaustive search strategies. (AKA replicated evaluation, parameter sweep)
- N-body: Problems in which each member of a system depends on the state of every other particle in the system. The problems typically involve some scheme to approximate the naïve $O(N^2)$ exhaustive sum.
- Sparse Linear Algebra: Problems represented in terms of sparse matrices. Solutions may be iterative or direct.
- Spectral methods: Computations where a data set is transformed between domains the idea being that once transformed, the problem will be easier to solve. Examples include Z-transform, FFT, DCT, etc.

- Structured mesh: Problem domains are mapped onto a regular mesh and solutions computed as averages over neighborhoods of points (explicit methods) or as solutions to linear systems of equations (implicit methods)
- Unstructured mesh: The same as the structured mesh problems, but the address of grid elements in the data-structures cannot be directly inferred from the physical location of the elements; hence, the computations involve scatter and gather operations.

Parallel Algorithm Strategy patterns

Goal: These patterns describe the high level strategies used when creating the parallel algorithms used to implement the computational patterns.

Output: Definition of the parallel algorithms to support the application frameworks.

Primary target audience: Application Framework Developers and Parallel programming Framework Developers

- Task parallelism: Parallelism is expressed as a collection of explicitly defined tasks. This pattern includes the embarrassingly parallel pattern (no dependencies) and separable dependency pattern (replicated data/reduction).
- Data parallelism: Parallelism is expressed as a single stream of tasks applied to each independent element of a data structure. This is generalized as an index space with the stream of tasks applied to each point in the index space.
- Recursive splitting: A problem is recursively split into smaller problems until the problem is small enough to solve directly. This includes the divide and conquer pattern as a subset wherein the final result is produced by first recursing from root to leaves and then by reversing the splitting process to assemble solutions starting with the leaf-node solutions.
- Pipeline: Fixed coarse-grained tasks with data flowing between them in an assembly-line like manner. Multiple tasks are typically working in parallel on partial workproducts of the same computation.
- Geometric decomposition: A problem is expressed in terms of a domain that is decomposed spatially into smaller chunks. Solution is composed of updates across chunk boundaries, updates of local chunks, and then updates to the boundaries of the chunks.
- Discrete event: a collection of tasks that coordinate among themselves through discrete events. This pattern is often used for GUI design and discrete event simulations.
- Graph partitioning: Tasks generated by decomposing recursive data structures (graphs)
- Digital Circuits: Tasks which are restricted to operate on single bit or bit-vector operands with a small set of logical (AND, OR, XOR, NAND, NOR, MUX) and arithmetic (ADD, SUBTRACT, ALU) operations.

Implementation Strategy patterns

Goal: These patterns focus on how a software design is implemented in software. They describe how threads or processes execute code within a program; i.e. they are intimately connected with how an algorithm design is implemented in source code. These patterns fall into two sets: program structure patterns and data structure patterns.

Output: pseudo-code defining how a parallel algorithm will be realized in software.

Primary target audience: Parallel Programming Framework Developers

- Program structure
 - Single-Program Multiple Data (SPMD): One program used by all the threads or processes, but based on ID different paths or different segments of data are executed.
 - Strict data parallel: A single instruction stream is applied to multiple data elements. This includes vector processing as a subset.
 - Loop-level parallelism: Parallelism is expressed in terms of loop iterations that are mapped onto multiple threads or processes.
 - Fork/join: Threads are logically created (forked), used to carry out a computation, and then terminated after possibly combining results from the computations (joined).
 - Master-worker/Task-queue: A master sets up a collection of work-items (tasks), a collection of workers pull work-items from the master (a task-queue), carry out the computation, and then go back to the master for more work.
 - Actors: a collection of active software agents (the actors) interact over distinct channels.
 - BSP: A computation is organized as a sequence of super-steps [Valiant90]. Within a super-step, computation occurs on a local view of the data. Communication events are posted within a super-step but the results are not available until the subsequent super-step. Communication events from a super-step are guaranteed to complete before the subsequent super-step starts.
- Data Structure Patterns
 - Shared queue: this pattern describes ways to any of the common queue data structures and manage them in parallel
 - Distributed array: An array data type that is distributed about a threads or processes involved with a parallel computation.
 - Shared hash table: A hash table shared/distributed among a set of threads or processes with any concurrency issues hidden behind an API.
 - Shared data: a “catch all” pattern for cases where data is shared within a shared memory region but the data can not be represented in terms of a well defined and common high level data structure.
 - Data Locality: describes the techniques used to keep data close to the processing elements that will work with it.

Concurrent execution Patterns

Goal: These patterns describe how a parallel algorithm is organized into software elements that execute on real hardware and interact tightly with a specific programming model. We organize these into two sets: (1) process/thread control patterns and (2) coordination patterns.

Output: task-graphs, data flow diagrams and other structure to define how a parallel algorithms will be supported at runtime.

Primary target audience: Platform programmers

- Process/thread control patterns
 - CSP or Communicating Sequential Processes: Sequential processes execute independently and coordinate their execution through discrete communication events.
 - Data flow: sequential processes organized into a static network with data flowing between them.
 - Task-graph: A directed acyclic graph of threads or processes is defined in software and mapped onto the elements of a parallel computer.
 - Single-Instruction Multiple Data (SIMD): A single stream of program instructions execute in parallel for different lanes in a data structure. There is only one program counter for a SIMD program. This pattern includes vector computations.
 - Thread pool: The system maintains a pool of threads that are utilized dynamically to satisfy the computational needs of a program. The pool of threads work on queues of tasks. Work stealing is often used to enforce a more balanced load.
 - Speculation: a thread or process is launched to pursue a computation, but any update to the global state is held in reserve to be entered once the computation is verified as valid.
- Coordination Patterns
 - Message passing: Processes or threads cooperatively move data about the system as distinct messages.
 - Collective communication: reductions, broadcasts, prefix sums, scatter/gather etc.
 - Mutual exclusion: Blocks of code or updates of memory that can only be executed by one process or thread at a time.
 - Point to point synchronization: Synchronization operations such as the mutex which operate between a distinct pair of processes or threads.
 - Collective synchronization: synchronization operations such as barriers that impact collections of threads or processes.
 - Transactional memory: transactions with memory subsystems to support atomicity.

Other patterns

It is important to appreciate that OPL is not complete. OPL is restricted to those parts of the design process associated with architecting parallel application software. There are countless additional patterns that software development teams utilize. Some examples include:

- Finding concurrency patterns [Mattson04]: These patterns capture the process used by experienced parallel programmers when looking at a problem and understanding how to exploit the concurrency available in the problem.

- Object oriented design [Gamma94]: The object oriented design process introduces its own unique set of patterns. We made no attempt to include them in OPL. An interesting framework that supports common patterns in parallel object oriented design is TBB [Reinders07].
- Application patterns: Different classes of applications have their own patterns. These patterns are used across families of applications, but are restricted in their scope to a class of applications. For example, we have found design patterns applicable to machine learning software as a class regardless of whether the actual application is image recognition or speech analysis. These domain specific patterns are beyond the scope of OPL, but they need to be documented elsewhere in a catalog of applications patterns.

Conclusion and Next Steps

The patterns in OPL define the full range of design patterns programmers are likely to encounter when designing software for parallel platforms. A programmer working with a particular class of computers, however, may only sample a portion of these patterns. For example, a programmer mapping general purpose programs onto a GPU platform, however, will tend to focus on the data parallel, SIMD and SPMD design patterns. This GPGPU programmer will spend much more time thinking about the data involved in a computation and how to define computations in terms of updates to that data (which is the crux of the data parallel pattern). We believe that the granularity of OPL and its organization into relatively simple five layer model makes it easy for this GPGPU programmer to ignore the patterns that are not appropriate for a GPU and focus on the ones he or she needs.

In other words, we do not believe it make sense to create a different pattern language for clusters, SMP computers, GPUs and each class of parallel platform. One pattern language can apply to design problems that span a range of platforms. Furthermore, there is a benefit to exposing a GPGPU programmer, for example, to the wider range of patterns since the rapid pace of platform evolution means there is a significant probability that at some point, this programmer may need to port his or her software to other platforms; and hence might need to understand how a particular design would map onto other platforms.

Defining the layers in OPL and listing the patterns is useful, but it is only a first step. We need to write all the patterns and have them carefully reviewed. In addition, we need to:

- Understand how our patterns map onto more formal models of computation, for example how should we align the patterns in OPL with the models of computation from the Ptolemy project [Eker03]?
- Document case studies to show that the patterns in OPL are sufficiently complete. We believe the patterns will map onto higher level frameworks, but outside of a few simple cases, we have not collected case studies to validate this connection to frameworks.
- Currently, we define the output, goals and target audience for each layer in the language informally. It would be useful to formalize these for each layer.
- Many of the computational patterns are better understood as entry points into collections of patterns. For example. Sparse linear algebra is not a single pattern but a family of patterns

as documented in [Berry94]. We need to understand how to best capture this complexity inside OPL.

Acknowledgements

OPL is a team effort. It would not exist without the hard work by members of Kurt Keutzer's group at UC Berkeley and the students taking CS294 in the spring semesters of 2008 and 2009.

References

[Alexander77] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977.

[Asanovic09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiawicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A View of the Parallel Computing Landscape", Submitted to Communications of the ACM, May 2008, to appear in 2009.

[Berry94] M. berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, CH.Romine, H. van der Vorst, Templates for the solution of Linear Systems: Building blocks for Iterative methods, SIAM, 1994.

[Eker03] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, "Taming Heterogeneity---the Ptolemy Approach," Proceedings of the IEEE, v.91, No. 2, January 2003.

[Gamma94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of reusable Object Oriented Software, Addison-Wesley, 1994.

[Mattson04] T. G. Mattson, B. A. Sanders, B. L. Massingill, patterns for Parallel Programming, Addison Wesley, 2004.

[Reinders07] J. Reinders, Intel Threaded Building Blocks, O'Reilly Press, 2007.

[Valiant90] L. G. Valiant, "A Bridging Model for parallel Computation", Communication of the ACM, vol, 33, pp. 103-111, 1990.