Massively Parallel Programming Languages – A Classification of Design Approaches

Wolfgang Gellerich

Department of Computer Science
University of Stuttgart
D-70565 Stuttgart, Germany,
gellerich@informatik.uni-stuttgart.de

Michael M. Gutzmann

Department of Computer Science
University of Jena
D-07743 Jena, Germany
gutzmann@informatik.uni-jena.de

Abstract

This paper presents the results of a study in which we examined about 50 parallel programming languages in order to detect typical approaches towards supporting massive parallelism. Based on a classification into nine classes, semantic properties affecting the development of parallel programs are compared. From a consideration of the general function of programming languages in software engineering, we derive basic requirements on parallel languages.

1 Introduction

Phrases like "parallel software crisis" [52] or "software dilemma" [7] are commonly used when scientists discuss today's situation of parallel programming. To some degree, this is up to programming languages which provide only poor support for parallel software development as some of their properties "hinder advanced analysis and optimization" [68].

Our goal was to find out what kind of languages have emerged and what their properties and possible strengths and weaknesses are. We examined about 50 parallel programming languages to detect typical design approaches [29, 30]. In order to derive basic requirements for a "good" parallel language, we discuss properties of each language class and relate them to steps performed in program development and to major tasks performed by parallelizing compilers.

We were interested in languages that utilize massive parallelism to achieve high performance and therefore excluded approaches that only provide pseudo-parallel concepts primarily for simulation purposes (e.g., coroutines in Modula-2) as well as languages which provide operating-system-like large grain processes (e.g., tasks in Ada [2, 6, 12]. Furthermore, we concentrated on approaches that concern languages design and we will mention approaches that basically provide a set of communication routines (e.g. Linda, PVM) only for completeness. Finally, we considered only languages that had already been implemented for at least one parallel architecture.

1.1 Terminology

The term *parallel language* subsumes all languages used to program parallel computers. *Problem-oriented programming* is commonly used to express that the programmer can concentrate on a specification of the problem to be solved without worrying over (too many) technical details. The opposite, *system-oriented programming*, requires the program to be organized around the parallelization method of a compiler, properties of a certain machine or characteristics of an architecture class.

1.2 Programming Languages Overview

Programming languages are traditionally classified as logical (Prolog-stream), functional (Lisp-stream), imperative (Fortran-stream, Algol-stream, CPL-stream, Simula67-stream) or applicative (VAL-stream). In contrast to this distinction, we present a classification according to the kind of support for parallel processing. For a discussion of general properties of programming languages, the reader is referred to textbooks like [31, 61].

Programming languages serve as interface between programmers and computers. In order to derive basic requirements on parallel languages, we take a short look at software engineering and at parallel compiler construction.

1.3 Software Engineering and Compilation Aspects

Program development usually consists of four general steps, which belong to the classic life-cycle models and are also found in other models developed in software engineering [57]:

1. The *software requirements* describe structure and details of a given problem.
2. During software design, this description is used to construct or to choose an *abstract algorithm* as well as details like interface structures and data representation. The algorithm is abstract in the sense that it does not include any details of a possible implementation.
3. Next, the programmer codes the algorithm in a programming language yielding a *program text*. Depending on the degree of abstraction provided by the language, this representation may already include details of the final implementation.
4. Finally, the program text is compiled into *machine code* which is then executed under control of a runtime system.

Aspects of parallelism must already be considered in the second step as different algorithms may vary in their inherent suitability for parallelization. In order to get maximum speedup when executing the machine code, the information on independence of certain operations in the abstract algorithm must be preserved during coding and compilation. This leads to additional requirements on programming languages and on compilation techniques.

A refinement of steps 3 and 4 with respect to parallel programming yields the following tasks which must be performed on *any* platform. Intentionally, we kept this description very general to cover most systems. Different systems may, however, differ in whether these tasks are performed by the programmer, a compiler or at runtime – and this is one of the distinctive properties we used for our evaluation.

Analysis. Analyze an *implicitly parallel* representation of the algorithm in order to detect operations that are executable in parallel. The result is an *explicitly parallel* representation. Depending on the system, these representations may be program text or a less formal description. Further information needed in the following steps may be collected, too.

Time/Space Mapping. Map parallel structures found in the explicitly parallel representation onto parallel structures available in hardware, yielding a *mapped* representation.

This consists of two steps: the determination of an execution order (time mapping, scheduling) and the determination of an execution location (processor) for parallel operations (space mapping). This may include partitioning, distribution and alignment of data.

Code generation. This may be conventional machine code or, in the case of data-flow architectures, a dependence graph. As we do not consider assembly languages, code generation is done by the compiler.

A compiler performing the analysis step will usually do traditional *data-flow analysis* and *dependence analysis for arrays*. This is preceded by *control flow analysis* if the source language is based on a control flow model [3, 67, 69, 68]. In the case of arrays with affine dependence structures, powerful strategies for automatized time/space mapping exist [23, 24, 42].

Our classification and evaluation of parallel languages is based on their support for requirements from software engineering and compiler construction. A study comparing 8 languages by programming a set of typical parallel applications, the so-called *Salishan Problems* was published by J. Feo [25].

1.4 Run-Time Behaviour

As languages are usually claimed to be "fast", it would be interesting to compare the run-time behaviour of one problem programmed in different languages but running on the same machine – speedup is the only reason for doing massively parallel computation. However, such experiments are found very rarely. We only know of one such study [14] which is cited in the context of data-flow languages.

2 Classes of Parallel Languages

Table 1 shows the structure of our classification and gives some typical examples for each class. The distinction between different classes is made according to the support of parallelism. We concentrate on semantic properties while ignoring purely syntactic details as well as distinctions arising from other requirements on programming languages. The following subsections describe major features of each class, while any conclusions are deferred to the next section.

Class 1: Automatically Parallelized Sequential Languages. When the first parallel architectures emerged, the most obvious approach was to implement well known sequential languages like Fortran by using

System-oriented languages		
	1. sequential languages	: CFT [56], PTRAN [60]
	2. hardware specific languages	: CFD [63, 54, 55], MPL [47] : DAP-Fortran [55]
	3. architecture class oriented languages	
	MIMD–based	: Occam [35, 13]
	SIMD–based	: C* [59, 58], Parallaxis [9, 10]
Problem-oriented languages		
	explicitly parallel	
	4. task parallel languages	: PVM [27], Linda [28], : Ada [2, 6, 12]
	5. data parallel languages	: Modula-2* [34], Vector C [46] : Actus [53, 55], Fortran D [26] : HPF [43]
	implicitly parallel	
	6. functional languages	: Haskell [37, 38], : extended ML [5]
	7. data-flow languages	: VAL [49], Id [21, 51], : Sisal [48, 15]
	8. equational languages	: ASL [33, 22], EPL [64], : Crystal [39]
	9. logical languages	: Concurrent Prolog [62] : Parlog [32, 18, 16], GHC [66]

Table 1: Classification of parallel languages

parallelizing compilers, in order to use existing sequential code on parallel machines.

Unfortunately, there are several problems with this approach. To a varying degree, sequential languages are based on the von-Neumann model of computation that assumes a single flow of control and a contiguous main memory. Obviously, this is appropriate only for programming SIMD machines, but is likely to cause problems on other parallel architectures.

Many parallel architectures are distributed memory machines and require the distributed storage of large data structures in order to utilize parallelism. In general, the kind of distribution will depend on the problem size and on the expected communication costs. Two arrays could be distributed differently when they are used in different contexts. Therefore, the components of a matrix need neither be stored contiguously nor can we assume a common distribution scheme. Access methods based on uniformly calculated addresses (e.g., as in C) are inapplicable here.

An additional problem is the possibility of variable names to be aliases. Apart from well-known problems regarding program reliability and optimizations [31], aliasing makes data dependence analysis considerably more difficult and may create artificial dependences. In the presence of pointers, data dependence analysis becomes NP-hard and can only be approximated [45]. But, if the compiler can not prove the independence of some statements, it must enforce sequential execution.

To achieve substantial efficiency, the programmer must design (or restructure existing) code according to the compiler's parallelization capabilities. Characteristic examples are given in [55] and [4]. However, there has been significant progress in the field of automatic parallelization [67, 69, 68], so that today's systems can handle many situations that were not parallelized by early compilers. But it is still suggested that parallelization systems should be interactive in that they emit messages reporting which parts of code could not be parallelized and the programmer is then asked to restructure the code [67, 69].

Although this programming style is not *hardware-oriented*, it is at least *compiler-oriented* instead of being *problem-oriented*. An additional problem with taking sequential programs to parallel architectures is that an algorithm executing efficiently on a sequential machine may be not well suited for parallel execution.

Class 2: Hardware Specific Languages. A sequential language can be extended with some constructs which directly reflect the architecture of a certain parallel computer. Such languages typically provide a special syntax for arrays that are to be processed in parallel but limit the degree of parallelism to the capabilities of the hardware (e.g., CFD, DAP Fortran). Some languages (e.g., MPL) also provide statements to control communication between processors.

These languages are easy to implement as the compiler only performs code generation, while *analysis* and *mapping* are left to the programmer. This is a major disadvantage: The programmer must find out which parts of the program can be executed in parallel and then map them onto the hardware. Thus, if the problem parallelism does not match the parallelism of the machine, the programmer must change the data structures. This can significantly increase the complexity of the problem solution [54] and "turns these languages into higher level assembly languages" [55]. Such programs are not portable. Scalability can be achieved to some degree only if the number of processors is made available in the language and is used to make the program adaptive.

Class 3: Architecture Class Oriented Lan-

guages. A more abstract approach is to define the operational semantics of a language in terms of an architecture class. This leads to languages which support explicit parallelism but are not biased towards a certain machine. We distinguish between MIMD- and SIMD-based approaches.

For example, Parallaxis (version 2[1]) is an extension of Modula-2 and provides means to specify an abstract array processor. The programmer may declare any number of processing elements (PE) and choose an arbitrary interconnection network of named channels. Any declaration of variables is augmented with the information whether it is to be located in the control unit (CU) or in every PE's local memory. The same distinction is made at statement level: the programmer must specify whether a statement sequence is to be executed by the CU or in parallel on (the set of active) PEs. Occam provides a similar abstraction from MIMD architectures.

Programs in such languages are reasonably portable within the architecture class they are based upon. The programmer must still do *analysis*, but the compiler does all *mapping* as it assigns virtual processors to real processors and thereby maps the virtual topology onto the real topology. The quality of this mapping is crucial for efficiency. Although these languages provide a high level of abstraction, programming is still not purely problem-oriented.

Class 4: Task Parallel Languages. This approach consists of a sequential language (often Fortran or C) which is combined with a library of communication primitives. In the case of Linda, data exchange is done via a single mailbox (here called *tuple space*), while PVM is based on message passing in a heterogeneous computer network.

Class 5: Data Parallel Languages. Problem-oriented programming is achieved when the language supports the expression of parallel structures in the *algorithm* to be programmed instead of relying on the nature of parallelism in the hardware.

A data parallel language provides constructs for expressing that a statement sequence is to be executed in parallel on different data. The syntax is either a parallel loop (e.g., Modula-2*, many parallel Fortran dialects) or a kind of vector notation with similar semantics (e.g., Actus, Vector C). Some languages support both synchronous and asynchronous parallelism. This approach is always limited to data parallelism.

The programmer must analyze the algorithm to find those parts which can be executed in parallel. The compiler then maps the data parallel parts onto the parallel hardware structures. Since most of the data parallel algorithms concern arrays, many languages provide means to specify an alignment between arrays for minimizing communication overhead.

[1]Parallaxis-III [11] has considerably moved towards being a data parallel language. In particular, statements to be executed in parallel are no longer explicitly marked.

Languages with Implicit Parallelism. In general, neither users nor application programmers are interested in parallelism itself, but in performance [52]. Therefore it is very reasonable to build a language which allows using parallel resources but does not require to explicitly specify which parts of a program can be executed in parallel.

Instead, the language should not enforce any unnecessary specification of an execution order for independent steps. This can be achieved by replacing the idea of control-driven execution with data-driven execution: every instruction becomes executable as soon as all of its arguments have been evaluated. The compiler maps executable entities onto real processors, probably after collecting individual statements into larger sequences. Demand-driven execution models have also been suggested.

Class 6: Functional Languages. The term *purely functional languages* subsumes all languages that are based on Church's λ-calculus. This model is free from side effects and has no explicitly specified execution order. Due to these properties, it is claimed that purely functional languages should be "highly suited for parallel programming" [37]. In particular, function arguments can be evaluated in parallel.

However, modern functional languages (e.g., Haskell [37, 38]) provide features like high order functions (allowing the combination existing functions into new ones and then apply these to arguments) and non-strict semantics (i.e., lazy evaluation allowing the specification of infinite data structures, which is also used to model input/output). It seems very difficult to implement such concepts on parallel machines in a way that yields the same speedup and efficiency as imperative languages do.

A currently popular approach for the design of more efficient parallel functional languages are *algorithmic skeletons* [41, 17, 19]. The basic idea is to provide a set of high order functions which capture characteristic "algorithmic structures" found in parallel algorithms, such as divide-and-conquer. These templates for parallel algorithms are then instantiated by the programmer. Recent implementations, e.g., of ML extended by skeletons, yielded good performance [5]. However, the set of skeletons required to implement all possible

algorithms is large (if not infinite). Probably, no language providing a fixed set of skeletons can expected to be general.

It has also been suggested to extend functional languages by explicitly parallel constructs to support data-parallel or message-passing computation.

Class 7: Data-flow Languages. Data-flow languages (DFL) [20, 1] were originally developed to program data-flow architectures which provide parallelism at operator level and require appropriate languages.

The central idea in DFL is a *value-oriented* or *applicative* view of computation: any operations are considered to operate on values rather than on (abstractions of) memory cells. This leads to the *single assignment rule*: an assignment establishes an immutable binding between a name and a value. Problems like aliasing as well as far-reaching data dependences are thereby eliminated and the translation of programs into data-flow graphs is simplified. A further consequence is that DFLs are *function-* or *expression-based*: since there are no mutable variables and thus no global state, every construct must return a value. Data dependences are the only sequencing constraints in data-flow programs.

The difference to purely functional languages is that DFL are not based on the λ-calculus and do not allow using of high order function to create new functions at runtime because these functions could not be translated into a flow-graph at compile time. Some DFL do not support static function parameters and others (VAL) even forbid recursive function calls.

However, the lack of mutable variables introduces new optimizations problems: a Sisal compiler with straight-forward implementation of applicative semantics generated code for a vector algorithm which took about an hour to execute while the Fortran equivalent executed in less than half a second on the same machine [15]. The problem with strict adherence to applicative semantics is that the array *value* is constructed step by step by appending one element in every iteration. Although the problem can be solved by *copy elimination* in most cases [14], the fact remains that, while the single assignment rule removes some of the problems related to imperative languages, it makes optimizations necessary.

Another problem is that the single assignment rule would disallow iteration. However, many algorithms (e.g., array operations or converging approximations) can be expressed more naturally and more efficiently using loops and mutable variables. Therefore, data-flow languages like Sisal or Id provide "iterative constructs for efficiency" [21], and allow multiple assignments yet inside loops.

The properties of DFL turned out advantageous for implementation on any kind of parallel machine: e.g., Sisal is available on a large number of parallel computers and offers a high degree of portability. Significant Fortran programs have been rewritten in Sisal and yielded comparable performance [14].

Class 8: Equational Languages. A very problem-oriented design approach for a language intended for scientific and engineering applications is to adopt the notation commonly used in this area. Typically, algorithms used in scientific and engineering computations are based on (some variations of) multi-dimensional arrays [64]. *Equational languages* such as ASL [30, 22] and EPL [64] organize programs around equations over arrays. These languages are primarily designed for scientific application and do not claim to be general. However, it has also been suggested to extend the equation-based approach by more general data structures like lists and sets [36].

The model behind ASL are recurrence equations [40, 44]. ASL was developed for static problems that have the same algorithmic power as primitive recursive functions [33]. Further properties of ASL are single assignment variables and the renunciation of pointers which results in the absence of side effects. Although these restrictions cause a loss of generality, most algorithms of the intended application area can still be expressed. On the other hand, these restrictions strongly support dependence analysis as well as automatic time/space mapping and high-level algorithm optimization performed by algebraic transformations with respect to specific problem and hardware characteristics [22].

Class 9: Parallel Logic Languages. The mathematical model behind logic programming are predicate logic and the principle of resolution that allows inferring new propositions from given propositions. This requires repeated unification of the goals to be proven with facts and rules already known. In *pure logic programming*, the order of attempted matches is nondeterministic.

This offers three sources for parallel execution. The reduction of several (sub)goals at the same time is called *AND-parallelism*. *OR-parallelism* means trying to prove one goal in several ways at once by concurrent search for clauses that are unifyable with this goal. If either one *or* another way succeeds, the goal is proven. Algorithms for *parallel unification* have been considered, too.

However, efficient implementation of parallel logic

languages seems to be be difficult [16] and requires some mapping and load-balancing at run-time. There are major semantic difference between sequential Prolog and its parallel versions as sequential prolog allows control of resolution order for efficiency [62]. In particular, the "cut" allows explicit control of backtracking.

3 Conclusions

Based on the discussion of software engineering and compiler aspects in section 1.3, we will now closer examine typical properties of these classes and derive some general requirements.

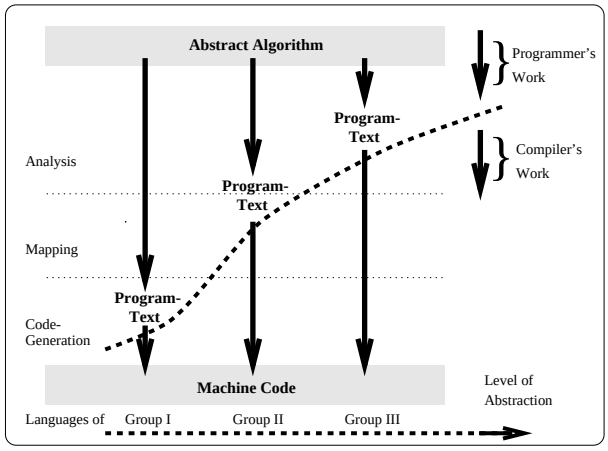

Figure 1: Languages with different levels of abstraction

3.1 How much is done by the Compiler?

We argued that users would prefer implicitly parallel languages as explicitly parallel programming can be rather difficult and error-prone. The implicit approach also supports general requirements like portability, maintainability, scalability, and machine independence. Therefore,

> *As much work as possible should be done by the compiler.*

A second argument for this request is the fact that work invested into a compiler once can be used by many application programs while work left to the programmer must be re-done for every program. Additionally, compilers used by a large user community can be expected to work correctly.

Figure 1 shows the program development process for three language classes providing different degrees of abstraction. This distinction is based on the observation that the program text is the result of the programmer's work and the input to the compiler. We now compare the language classes shown in table 1 with respect to their level of abstraction.

Hardware specific languages belong to group I as they leave analysis and mapping to the programmer. *Automatically parallelized sequential languages* belong to this group, too. Although this approach allows implicit usage of parallelism, the programmer is usually required to explicitly map the algorithm into program structures the compiler can parallelize.

Group II consists of *architecture class oriented languages* which ask the programmer to find data parallel parts in the algorithm (analysis) but often perform mapping automatically. *Data parallel languages* belong to this group too. E.g., languages like HPF ask the programmer to give detailed information about distribution and alignment of arrays as well as a specification of independent parts of the program, but some mapping and analysis is done automatically.

Group III shows the case of *implicitly parallel languages* where analysis and mapping are done by the compiler. However, compilers may still require or accept some user support for mapping in order to optimize runtime efficiency, e.g., by explicitly specifying inter-array alignment.

At the hardware level, only those steps that are independent in the abstract algorithm can be executed in parallel. The language must preserve this independence. In order to utilize this parallelism automatically, the language must enable the compiler to derive some specific information from any given program text. This is further discussed in the next sections.

3.2 Representation of Data

The compiler must be able to analyze existing data dependences efficiently, and the program text should not introduce artificial dependences. In particular, this concerns potential aliasing.

> *A language should not provide constructs that may introduce artificial data dependences or hinder the analysis of data dependence.*

Functional, *data-flow* and *equational languages* fulfill this condition while most approaches based on imperative languages have problems due to aliasing. But, since *architecture class oriented*, *task parallel* and *data parallel languages* explicitly express parallelism, aliasing is unlikely to cause problems here.

> *Any assumption about a given storage structure is disadvantageous.*

Constructs requiring a contiguous main memory or supporting an address-oriented view of variables (e.g., using pointers to access vector components as in C; Fortran programs depending on column-major storage of arrays) is likely to cause problems due to possible aliasing and because this view is unsuitable for distributed memory architectures. *Hardware specific languages* explicitly reflect the *parallel* storage structure of a certain machine. This hinders portability.

3.3 Execution Models

In general, program consists of a large number of *actions*[2]. Every language has rules specifying an execution order among the set of actions and therefore determining the moment at runtime when a certain action is to be executed. This mechanism is called the *execution model*.

Languages developed with the von-Neumann model in mind assume a single flow of control for this purpose, thereby creating a totally ordered instruction sequence. Statements like `GOTO` change this control flow but do not operate on data. Of course, these can not be directly translated onto parallel architectures but require control flow analysis in order to detect potentially parallelizable higher control structures. This requires that the overall control structure of a program is statically determined. Computed `GOTO`s – a feature still available even in Fortran 90 [50] – violate this requirement.

`GOTO` statements with statically determined targets *can* be handled but complicate the compiler as both, data dependences and the control flow representation must be updated when program transformations are applied. If only well-structured control structures can occur, control flow graphs can be avoided [8].

Task parallel and *data parallel languages* employ multi-control flow models that offer constructs like `cobegin`, explicitly parallel `FORALL` loops or processes. These allow splitting single control flow into multiple flows of control.

However, any control flow seems unnecessary as data dependences already contain complete information about possible execution orders, thereby creating a partial order on the set of actions. Control flow further restricts this order (except for cases where control flow does not respect data dependences – but these are errors such as reading an uninitialized variable). Data-driven execution seems advantageous as it allows for implicitly parallel execution and further supports compiler analysis. As a general requirement,

> *A language should not require to specify any execution order for operations that are not data dependent.*

This also concerns loop statements. For example, an execution order must be specified for independent steps of a vector addition if the language fails to provide an "unordered" (i.e., non-sequential) loop statement. It is not always possible to automatically parallelize unnecessarily sequential loops.

This raises the question how loop parallelism could be supported by a language. In general, it seems to be natural *not* to specify any order, unless necessary. Therefore, a language should provide a loop construct with unspecified execution order which implicitly supports parallelism. Sequential execution is then considered to be a special case and could be enforced by an additional keyword.

4 Summary and Future Work

We examined 50 parallel languages and found some typical design approaches. These were evaluated further based on criteria from software engineering and parallelizing compiler construction. Our final classification distinguishes the level of compiler support by the language.

The semantical properties of the programming language are an important factor to make efficient usage of parallel resources. Beyond general goals such as maintainability, correctness, completeness and a human-readable notation, additional goals should be fulfilled in parallel program development. Efficient usage of parallel resources, scalability and portability – both between different parallel architectures and between machines that differ only in the number of processors – are the most important requirements in this context.

A promising approaches are equational and dataflow languages which provide a high level of abstraction and thereby support the programmer as well as compilation techniques. However, these languages have been criticized for their single-assignment rule which fails to reflect that some objects have a state mutable over time. Also, it is considered "unnatural" by many of today's programmers. Currently, we are working on an execution model that is data-driven but allows multiple assignment to variables.

[2] A term intended to subsume terms like statement, operation or function which would be used in the context of a certain language.

References

[1] W.B. Ackerman. Data Flow Languages. *IEEE Computer*, 15(2):14–25, February 1982.

[2] *Ada 95 Reference Manual*. Intermetrics, Inc., 1995. ANSI/ISO/IEC-8652:1995.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers*. Addison-Wesley, 1986.

[4] R.G. Babb II and A.H. Karp. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, 5:52–67, September 1988.

[5] P. Bailey, M. Newwy, D. Sitsky, and R. Stanton. Supporting Coarse and Fine Grain Parallelism in an Extension of ML. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, pages 593–604. Springer, 1994.

[6] J. Barnes. *Programming in Ada 95*. Addison Wesley, 1995.

[7] J.E. Boillat, H. Burkhart, K.M. Decker, and P.G. Kropf. Parallel Computing in the 1990's: Attacking the Software Problem. *Physics Reports*, 207(3-5):141–165, 1991.

[8] M. M. Brandis. Building an Optimizing Compiler for Oberon: Implications on Programming Language Design. In P. Schulthess, editor, *Advances in Modular Languages*, pages 123–135. Universitaetsverlag Ulm, 1994.

[9] Th. Bräunl. *Massiv parallele Programmierung mit dem Parallaxis-Modell*. Springer, 1990.

[10] Th. Bräunl. Transparent massively parallel programming with parallaxis. *International Journal of Mini and Microcomputers*, 14(2):82–87, 1992.

[11] Th. Bräunl. Parallaxis-III: A structured data-parallel programming language. In *ICA3PP*, 1995.

[12] A. Burns and A.J. Wellings. Ada 95: An Effective Concurrent Programming Language. In *Reliable Software Technologies – Ada-Europe 1996*, volume 1088 of *LNCS*, pages 58–77. Springer, 1996.

[13] Alan Burns. *Programming in Occam 2*. Addison-Wesley, Wokingham, 1988.

[14] D.C. Cann. Retire Fortran? – A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[15] D.C. Cann, J. Feo, and R. Oldehoeft. A Report on the Sisal Language Project. Technical Report UCRL-102440, Lawrence Livermore National Laboratory, 1990.

[16] A. Cheese. *Parallel execution of Parlog*. Springer, Berlin, 1992.

[17] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[18] T. Conlon. *Programming in PARLOG*. Addison-Wesley, Wokingham, 1989.

[19] J. Darlington and A.J. Field. Parallel Programming Using Skeleton Functions. In *PARLE93, Parallel Architectures and Languages Europe*, 1993.

[20] J.B. Dennis. First Version of a Data Flow Procedure Language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *LNCS*. Springer, 1974.

[21] K. Ekanadham. A Perspective on Id. (in [65]).

[22] W. Erhard and M.M. Gutzmann. *ASL – Portable Programmierung massiv paralleler Rechner*. Teubner, 1995.

[23] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part I, One Dimensional Time. *International Journal of Parallel Programming*, 21(5), October 1992.

[24] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part II, Multidimensional Time. *International Journal of Parallel Programming*, 21(6), December 1992.

[25] J.T. Feo. *A Comparative Study of Parallel Programming Languages: The Salishan Problems* . North-Holland, 1992.

[26] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical report, Rice University, 1992.

[27] A. Geist, A. Beguin, J. Dingarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.

[28] D. Gelernter, N. Carriero, Chandran S, and S. Chang. Parallel Programming in Linda. In Douglas Degroot, editor, *Proceedings of the 1985 International Conference on Parallel Processing*, pages 255–263. Computer Society Press, 1985.

[29] W. Gellerich. Charakteristische Eigenschaften und Konzepte von Programmiersprachen für Parallelrechner. Master's thesis, Universität Erlangen-Nürnberg, 1993.

[30] W. Gellerich and M.M. Gutzmann. Konzepte von Parallelrechnersprachen im Vergleich zu den Entwurfsprinzipien von ASL-2. In *PARS Mitteilungen*, pages 127–140, April 1993.

[31] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley and Sons, 1987.

[32] S. Gregory. *Parallel logic programming in PARLOG : the language and its implementation*. Addison-Wesley, Wokingham, 1987.

[33] M.M. Gutzmann. *Leistungsbewertung von massiv parallelen Rechnermodellen*. PhD thesis, Universität Erlangen–Nürnberg, Sept. 1993. Arbeitsberichte des IMMD, Bd. 26, Nr. 13.

[34] E.A. Heinz, W. F. Tichy, M. Philippsen, and P. Lukowicz. From Modula-2* to Efficient Parallel Code. Technical Report 21/92, Universität Karlsruhe, 1992.

[35] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1986.

[36] C.M. Hoffmann and M.J. O'Donnell. Programming with Equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112, 1982.

[37] P. Hudak. Para-functional programming in Haskell. (in [65]).

[38] P. Hudak, S.L. Peyton-Jones, and P.L. Wadler. Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language. *ACM SIGPLAN Notices*, 16(5), May 1992.

[39] M. Jacquemin and J.A. Yang. Crystal Reference Manual Version 3.0. *Yale University - Internal Report*, pages 1–13, October 1991.

[40] R.M. Karp, R.E. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, 1967.

[41] P. Kelly. *Functional Pogramming for Loosely-coupled Multiprocessors*. Pitman, 1989.

[42] Wayne Kelly and William Pugh. Selecting Affine Mappings based on Performance Estimation. Technical Report CS-TR-3108, Dept. of Computer Science, University of Maryland, December 1993.

[43] C.H. Koelbel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[44] P. M. Kogge and H. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[45] W. Landi and B.G. Ryder. Pointer-induced Aliasing: A Problem Classification. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 142–152. ACM, ACM Press, 1991.

[46] K.C. Li. Vector C: A Vector Processing Language. *Journal of Parallel and Distributed Computing*, pages 132–169, 1985.

[47] *MasPar Programming Language (ANSI C compatible MPL) Reference Manual*.

[48] J. McGraw, S. Skedzielewski, R. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *Sisal Language Reference Manual Version 1.2*, 1985.

[49] J.R. McGraw. The VAL Language: Desription and Analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, 1982.

[50] M. Metcalf and J. Reid. *Fortran 90 explained*. Oxford University Press, Oxford, 1992.

[51] R.S. Nikil. ID Language Reference Manual. Technical Report Computation Structure Group Memo 284-2, MIT, July 1991.

[52] C.M. Pancake. Software Support for Parallel Computing: Where Are We Headed? *Communications of the ACM*, 34(11):53–64, November 1991.

[53] R.H. Perrott. A Language for Array and Vector Processors. *ACM Transactions on Programming Languages and Systems*, 1(2):177–195, Oktober 1979.

[54] R.H. Perrott. Language Design Approaches for Parallel Processors. In *CONPAR 81*. Springer, Berlin, 1981.

[55] R.H. Perrott. *Parallel Programming*. Addison-Wesley, 1988.

[56] R.H. Perrott. Parallel Languages and Parallel Programming. In *Parallel Computing 89*. North-Holland, 1990.

[57] R.S. Pressman. *Software Engineering*. McGraw-Hill, 3. edition, 1994.

[58] J. Rose. C*: A C++-like Language for Data Parallel Computation. Technical Report PL87-8, Thinking Machines Corporation, 1987.

[59] J. Rose and G. Steele. C*: An Extended C Language for Data Parallel Programming. Technical Report PL87-5, Thinking Machines Corporation, 1987.

[60] V. Sarkar. PTRAN – The IBM Parallel Translation System. (in [65]).

[61] R.W. Sebesta. *Concepts of Programming Languages*. The Benjamin-Cummings Publishing Company, 2. edition, 1993.

[62] E. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. In Ehud Shapiro, editor, *Concurrent Prolog Collected Papers*. The MIT Press, 1988.

[63] K.G. Stevens. CFD - A Fortran-Like Language for the Illiac IV. *ACM SIGPLAN Notices*, pages 72–76, März 1975.

[64] B.K. Szymanski. EPL – Parallel Programming with Recurrent Equations. (in [65]).

[65] B.K. Szymanski. *Parallel Functional Languages and Compilers*. Addison-Wesley, 1991.

[66] K. Ueda. Guarded Horn Clauses. In Ehud Shapiro, editor, *Concurrent Prolog Collected Papers*. The MIT Press, 1988.

[67] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989.

[68] M.J. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[69] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, 1991.

Bibliographic information:

```
@INPROCEEDINGS{gellerich:96b,
    AUTHOR    = "W. Gellerich and M.M Gutzmann",
    TITLE     = "Massively Parallel Programming Languages -- A Classification
                 of Design Approaches",
    BOOKTITLE = "Proceedings of the ISCA  International Conference on
                  Parallel and Distributed Computing Systems 1996",
    PUBLISHER = "ISCA",
    VOLUME    = "I",
    PAGES     = "110-118",
    YEAR      = "1996",
    NOTE      ={({\small \tt \allownewline
http://www.informatik.uni-stuttgart.de /ifi/ps/Gellerich/parlang-s.ps
              })}}
```