Models for Parallel Computing: Review and Perspectives

Christoph Kessler and Jörg Keller

Journal Article

Original Publication:

Christoph Kessler and Jörg Keller, Models for Parallel Computing: Review and Perspectives, *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen*, 2007. 24, pp. 13-29.

Postprint available at: Linköping University Electronic Press

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-40734

PARS-Mitteilungen **24: 13–29, Dec. 2007, ISSN 0177-0454, GI/ITG PARS.**

Models for Parallel Computing: Review and Perspectives

Christoph Kessler[1] and Jörg Keller[2]

[1] PELAB, Dept. of Computer Science (IDA)
Linköping university, Sweden
chrke@ida.liu.se
[2] Dept. of Mathematics and Computer Science
Fernuniversität Hagen, Germany
joerg.keller@FernUni-Hagen.de

Abstract. As parallelism on different levels becomes ubiquitous in today's computers, it seems worthwhile to provide a review of the wealth of models for parallel computation that have evolved over the last decades. We refrain from a comprehensive survey and concentrate on models with some practical relevance, together with a perspective on models with potential future relevance. Besides presenting the models, we also refer to languages, implementations, and tools.

Key words: Parallel Computational Model, Survey, Parallel Programming Language, Parallel Cost Model

1 Introduction

Recent desktop and high-performance processors provide multiple hardware threads (technically realized by hardware multithreading and multiple processor cores on a single chip). Very soon, programmers will be faced with hundreds of hardware threads per processor chip. As exploitable instruction-level parallelism in applications is limited and the processor clock frequency cannot be increased any further due to power consumption and heat problems, exploiting (thread-level) parallelism becomes unavoidable if further improvement in processor performance is required—and there is no doubt that our requirements and expectations of machine performance will increase further. This means that parallel programming will actually concern a majority of application and system programmers in the foreseeable future, even in the desktop and embedded domain, and no longer only a few working in the comparably small HPC market niche.

A *model of parallel computation* consists of a parallel programming model and a corresponding cost model. A *parallel programming model* describes an abstract parallel machine by its basic operations (such as arithmetic operations, spawning of tasks, reading from and writing to shared memory, or sending and receiving messages), their effects on the state of the computation, the constraints of when and where these can be applied, and how they can be composed. In particular, a parallel programming model also contains, at least for shared memory programming models, a *memory model* that describes how and when memory accesses can become visible to the different parts of a parallel computer. The memory model sometimes is given implicitly. A *parallel cost model* associates a cost (which usually describes parallel execution time and resource occupation) with each basic operation, and describes how to predict the accumulated cost of composed operations up to entire parallel programs. A parallel programming model is often associated with one or several *parallel programming languages* or *libraries* that realize the model.

Parallel algorithms are usually formulated in terms of a particular parallel programming model. In contrast to sequential programming, where the von Neumann model is the predominant programming model (notable alternatives are e.g. dataflow and declarative programming), there are several competing parallel programming models. This heterogeneity is partly caused by the fact that some models are closer to certain existing hardware architectures than others, and partly because some parallel algorithms are easier to express in one model than in another one.

Programming models abstract to some degree from details of the underlying hardware, which increases portability of parallel algorithms across a wider range of parallel programming languages and systems.

In the following, we give a brief survey of parallel programming models and comment on their merits and perspectives from both a theoretical and practical view. While we focus on the, in our opinion, most relevant models, this paper is not aimed at providing a comprehensive presentation nor a thorough classification of parallel models and languages. Instead, we refer to survey articles and books in the literature, such as by Bal *et al.* [10], Skillicorn [74], Giloi [39], Maggs *et al.* [65], Skillicorn and Talia [76, 77], Hambrusch [42], Lengauer [61], and Leopold [62].

2 Model Survey

Before going through the various parallel programming models, we first recall in Section 2.1 two fundamental issues in parallel program execution that occur in implementations of several models.

2.1 Parallel Execution Styles

There exist several different parallel *execution styles*, which describe different ways how the parallel activities (e.g. processes, threads) executing a parallel program are created and terminated from the programmer's point of view. The two most prominent ones are fork-join-style and SPMD-style parallel execution.

Execution in *Fork-join style* spawns parallel activities dynamically at certain points (fork) in the program that mark the beginning of parallel computation, and collects and terminates them at another point (join). At the beginning and the end of program execution, only one activity is executing, but the number of parallel activities can vary considerably during execution and thereby adapt to the currently available parallelism. The mapping of activities to physical processors needs to be done at run-time by the operating system, by a thread package or by the language's run-time system. While it is possible to use individual library calls for fork and join functionality, as in the pthreads package, most parallel programming languages use scoping of structured programming constructs to implicitly mark fork and join points; for instance, in OpenMP they are given by the entry and exit from an `omp parallel` region; in Cilk, they are given by the `spawn` construct that frames a function call.

Nested parallelism can be achieved with fork-join style execution by statically or dynamically nesting such fork-join sections within each other, such that e.g. an inner fork spawns several parallel activities from each activity spawned by an outer fork, leading to a higher degree of parallelism.

Execution in *SPMD style* (single program, multiple data) creates a fixed number p (usually known only from program start) of parallel activities (physical or virtual processors) at the beginning of program execution (i.e., at entry to `main`), and this number will be kept constant throughout program execution, i.e. no new parallel activities can be spawned.

In contrast to fork-join style execution, the programmer is responsible for mapping the parallel tasks in the program to the p available processors. Accordingly, the programmer has the responsibility for load balancing, while it is provided automatically by the dynamic scheduling in the fork-join style. While this is more cumbersome at program design time and limits flexibility, it leads to reduced run-time overhead, as dynamic scheduling is no longer needed. SPMD is used e.g. in the MPI message passing standard and in some parallel programming languages such as UPC.

Nested parallelism can be achieved with SPMD style as well, namely if a group of p processors is subdivided into s subgroups of p_i processors each, where $\sum_i p_i \leq p$. Each subgroup takes care of one subtask in parallel. After all subgroups are finished with their subtask, they are discarded and the parent group resumes execution. As group splitting can be nested, the group hierarchy forms a tree at any time during program execution, with the leaf groups being the currently active ones.

This schema is analogous to exploiting nested parallelism in fork-join style if one regards the original group G of p processors as one p-threaded process, which may spawn s new p_i-threaded processes G_i, $1 \leq i \leq s$, such that the total number of active threads is not increased. The parent process waits until all child processes (subgroups) have terminated, and reclaims their threads.

2.2 Parallel Random Access Machine

The *Parallel Random Access Machine (PRAM)* model was proposed by Fortune and Wyllie [34] as a simple extension of the Random Access Machine (RAM) model used in the design and analysis of sequential algorithms. The PRAM assumes a set of processors connected to a shared memory. There is a global clock that feeds both processors and memory, and execution of any instruction (including memory accesses) takes exactly one unit of time, independent of the executing processor and the possibly accessed memory address. Also, there is no limit on the number of processors accessing shared memory simultaneously.

The memory model of the PRAM is strict consistency, the strongest consistency model known [3], which says that a write in clock cycle t becomes globally visible to all processors in the beginning of clock cycle $t + 1$, not earlier and not later.

The PRAM model also determines the effect of multiple processors writing or reading the same memory location in the same clock cycle. An EREW PRAM allows a memory location to be exclusively read or written by at most one processor at a time, the CREW PRAM allows concurrent reading but exclusive writing, and CRCW even allows simultaneous write accesses by several processors to the same memory location in the same clock cycle. The CRCW model specifies in several submodels how such multiple accesses are to be resolved, e.g. by requiring that the same value be written by all (COMMON CRCW PRAM), by the priority of the writing processors (PRIORITY CRCW PRAM), or by combining all written values according to some global reduction or prefix computation (COMBINING CRCW PRAM). A somewhat restricted form is the CROW PRAM, where each memory cell may only be written by one processor at all, which is called the cell's *owner*. In practice, many CREW algorithms really are CROW.

Practical Relevance. The PRAM model is unique in that it supports deterministic parallel computation, and it can be regarded as one of the most programmer-friendly models available. Numerous algorithms have been developed for the PRAM model, see e.g. the book by JaJa [49]. Also, it can be used as a first model for teaching parallel algorithms [57] as it allows students to focus on pure parallelism only, rather than having to worry about data locality and communication efficiency already from the beginning.

The PRAM model, especially its cost model for shared memory access, have however been strongly criticized for being unrealistic. In the shadow of this criticism, several architectural approaches demonstrated that a cost-effective realization of PRAMs is nevertheless possible using hardware techniques such as multithreading and smart combining networks, such as the NYU Ultracomputer [41], SBPRAM by Wolfgang Paul's group in Saarbrücken [1, 51, 67], XMT by Vishkin [82], and ECLIPSE by Forsell [33]. A fair comparison of such approaches with current clusters and cache-based multiprocessors should take into consideration that the latter are good for special-purpose, regular problems with high data locality while they perform poorly on irregular computations. In contrast, the PRAM is a general-purpose model that is completely insensitive to data locality.

Partly as a reaction to the criticism about practicality, variants of the PRAM model have been proposed within the parallel algorithms theory community. Such models relax one or several of the PRAM's properties. These include asynchronous PRAM variants [23, 38], the hierarchical PRAM (H-PRAM) [45], the block PRAM [5], the queuing PRAM (Q-PRAM), and the distributed PRAM (DRAM), to name only a few. Even the BSP model, which we will discuss in Section 2.4, can be regarded a relaxed PRAM, and actually was introduced to bridge the gap between idealistic models and actual parallel machines. On the other hand, also an even more powerful extension of the PRAM was proposed in the literature, the BSR (Broadcast with selective reduction) [64].

Implementations. Several PRAM programming languages have been proposed in the literature, such as Fork [51, 58]. For most of them there is (beyond a compiler) a PRAM simulator available, sometimes additional tools such as trace file visualizers or libraries for central PRAM data structures and algorithms [51]. Also, methods for translating PRAM algorithms automatically for other models such as BSP or message passing have been proposed in the literature.

2.3 Unrestricted Message Passing

A *distributed memory machine*, sometimes called *message-passing multicomputer*, consists of a number of RAMs that run asynchronously and communicate via messages sent over a communication network. Normally it is assumed that the network performs message routing, so that a processor can send a message to any other processor without consideration of the particular network structure. In the simplest form, a message is assumed to be sent by one processor executing an explicit send command and received by another processor with an explicit receive command (*point-to-point communication*). Send and receive commands can be either blocking, i.e. the processors get synchronized, or non-blocking, i.e. the sending processor puts the message in a buffer and proceeds with its program, while the message-passing subsystem forwards the message to the receiving processor and buffers it there until the receiving processor executes the receive command. There are also more complex forms of communication that involve a group of processors, so-called *collective communication operations* such as broadcast, multicast, or reduction operations. Also, *one-sided communications* allow a processor to perform a communication operation (send or receive) without the processor addressed being actively involved.

The cost model of a message-passing multicomputer consists of two parts. The operations performed locally are treated as in a RAM. Point-to-point non-blocking communications are modelled by the *LogP* model [24], named after its four parameters. The latency L specifies the time that a message of one word needs to be transmitted from sender to receiver. The overhead o specifies the time that the sending processor is occupied in executing the send command. The gap g gives the time that must pass between two successive send operations of

a processor, and thus models the processor's bandwidth to the communication network. The processor count P gives the number of processors in the machine. Note that by distinguishing between L and o it is possible to model programs where communication and computation overlap. The LogP model has been extended to the LogGP model [6] by introducing another parameter G that models the bandwidth for longer messages.

Practical Relevance. Message passing models such as CSP (communicating sequential processes) have been used in the theory of concurrent and distributed systems for many years. As a model for parallel computing it became popular with the arrival of the first distributed memory parallel computers in the late 1980s. With the definition of vendor-independent message-passing libraries, message passing became the dominant programming style on large parallel computers.

Message passing is the least common denominator of portable parallel programming. Message passing programs can quite easily be executed even on shared memory computers (while the opposite is much harder to perform efficiently). As a low-level programming model, unrestricted message passing gives the programmer the largest degree of control over the machine resources, including scheduling, buffering of message data, overlapping communication with computation, etc. This comes at the cost of code being more error-prone and harder to understand and maintain. Nevertheless, message passing is as of today the most successful parallel programming model in practice. As a consequence, numerous tools e.g. for performance analysis and visualization have been developed for this model.

Implementations. Early vendor-specific libraries were replaced in the early 1990s by portable message passing libraries such as PVM and MPI. MPI was later extended in the MPI 2.0 standard (1997) by one-sided communication and fork-join style. MPI interfaces have been defined for Fortran, C and C++. Today, there exist several widely used implementations of MPI, including open-source implementations such as OpenMPI.

A certain degree of structuring in MPI programs is provided by the hierarchical group concept for nested parallelism and the communicator concept that allows to create separate communication contexts for parallel software components. A library for managing nested parallel multiprocessor tasks on top of this functionality has been provided by Rauber and Rünger [70]. Furthermore, MPI libraries for specific distributed data structures such as vectors and matrices have been proposed in the literature.

2.4 Bulk Synchronous Parallelism

The *bulk-synchronous parallel (BSP)* model, proposed by Valiant in 1990 [80] and slightly modified by McColl [66], enforces a structuring of message passing computations as a (dynamic) sequence of barrier-separated *supersteps*, where each superstep consists of a computation phase operating on local variables only, followed by a global interprocessor communication phase. The cost model involves only three parameters (number of processors p, point-to-point network bandwidth g, and message latency resp. barrier overhead L), such that the worst-case (asymptotic) cost for a single superstep can be estimated as $w + hg + L$ if the maximum local work w per processor and the maximum communication volume h per processor are known. The cost for a program is then simply determined by summing up the costs of all executed supersteps.

An extension of the BSP model for nested parallelism by nestable supersteps was proposed by Skillicorn [75].

A variant of BSP is the CGM (coarse grained multicomputer) model proposed by Dehne [28], which has the same programming model as BSP but an extended cost model that also accounts for aggregated messages in coarse-grained parallel environments.

Practical Relevance. Many algorithms for the BSP model have been developed in the 1990s in the parallel algorithms community. Implementations of BSP on actual parallel machines have been studied extensively. In short, the BSP model, while still simplifying considerably, allows to derive realistic predictions of execution time and can thereby guide algorithmic design decisions and balance trade-offs. Up to now, the existing BSP implementations have been used mainly in academic contexts, see e.g. Bisseling [11].

Implementations. The BSP model is mainly realized in the form of libraries such as BSPlib [46] or PUB [15] for an SPMD execution style.

NestStep [53, 56] is a parallel programming language providing a partitioned global address space and SPMD execution with hierarchical processor groups. It provides deterministic parallel program execution with a PRAM-similar programming interface, where BSP-compliant memory consistency and synchronicity are controlled by the `step` statement that represents supersteps. NestStep also provides nested supersteps.

2.5 Asynchronous Shared Memory and Partitioned Global Address Space

In the *shared memory* model, several threads of execution have access to a common memory. So far, it resembles the PRAM model. However, the threads of execution run asynchronously, i.e. all potentially conflicting accesses to the shared memory must be resolved by the programmer, possibly with the help of the system underneath. Another notable difference to the PRAM model is the visibility of writes. In the PRAM model, each write to the shared memory was visible to all threads in the next step (strict consistency). In order to simplify efficient implementation, a large number of weaker consistency models have been developed, which however shift the responsibility to guarantee correct execution to the programmer. We will not go into details but refer to [3].

The cost model of shared memory implementations depends on the realization. If several processors share a physical memory via a bus, the cost model resembles that of a RAM, with the obvious practical modifications due to caching, and consideration of synchronization cost. This is called *symmetric multiprocessing (SMP)*, because both the access mechanism and the access time are the same independently of address and accessing processor. Yet, if there is only a shared address space that is realized by a distributed memory architecture, then the cost of the shared memory access strongly depends on how far the accessed memory location is positioned from the requesting processor. This is called *non-uniform memory access (NUMA)*. The differences in access time between placements in local cache, local memory, or remote memory, may well be an order of magnitude for each level. In order to avoid remote memory accesses, caching of remote pages in the local memory has been employed, and been called *cache coherent NUMA (CC-NUMA)*. A variant where pages are dynamically placed according to access has been called *cache only memory access (COMA)*. While NUMA architectures create the illusion of a shared memory, their performance tends to be sensitive to access patterns and artefacts like thrashing, much in the same manner as uniprocessor caches require consideration in performance tuning.

In order to give the programmer more control over the placement of data structures and the locality of accesses, *partitioned global address space (PGAS)* languages provide a programming model that exposes the underlying distributed memory organization to the programmer. The languages provide constructs for laying out the program's data structures in memory, and to access them.

A recent development is *transactional memory* ([44], see e.g. [2] for a recent survey), which adopts the transaction concept known from database programming as a primitive for parallel programming of shared memory systems. A transaction is a sequential code section enclosed in a statement such as `atomic { ... }` that should either fail completely or

commit completely to shared memory as an atomic operation, i.e. intermediate state of an atomic computation is not visible to other processors in the system. Hence, instead of having to hard-code explicit synchronization code into the program, e.g. by using mutex locks or semaphores, transactions provide a declarative solution that leaves the implementation of atomicity to the language run-time system or to the hardware. For instance, an implementation could use speculative parallel execution. It is also possible to nest transactions.

If implemented by a speculative parallel execution approach, atomic transactions can avoid the sequentialization effect of mutual exclusion with resulting idle times, as long as conflicts occur only seldom.

A related approach is the design of *lock-free parallel data structures* such as shared hash tables, FIFO queues and skip lists. The realization on shared memory platforms typically combines speculative parallel execution of critical sections in update operations on such data structures with hardware-supported atomic instructions such as *fetch&add*, *compare&swap*, or *load-linked/store-conditional*.

Practical Relevance. Shared memory programming has become the dominant form of programming for small scale parallel computers, notably SMP systems. As large-scale parallel computers have started to consist of clusters of SMP nodes, shared memory programming on the SMPs also has been combined with message passing concepts. CC-NUMA systems seem like a double-edged sword because on the one hand, they allow to port shared memory applications to larger installations. However, to get sufficient performance, the data layout in memory has to be considered in detail, so that programming still resembles distributed memory programming, only that control over placement is not explicit. PGAS languages that try to combine both worlds currently gain some attraction.

Implementations. POSIX threads (pthreads) were, up to now, the most widely used parallel programming environment for shared memory systems. However, pthreads are usually realized as libraries extending sequential programming languages such as C and Fortran (whose compiler only knows about a single target processor) and thus lack a well-defined memory model [14]. Instead, pthreads programs inherit memory consistency from the underlying hardware and thus must account for worst-case scenarios e.g. by inserting flush (memory fence) operations at certain places in the code.

Cilk [13] is a shared-memory parallel language for algorithmic multithreading. By the `spawn` construct, the programmer specifies independence of dynamically created tasks which can be executed either locally on the same processor or on another one. Newly created tasks are put on a local task queue. A processor will execute tasks from its own local task queue as long as it is nonempty. Processors becoming idle steal a task from a randomly selected victim processor's local task queue. As tasks are typically light-weight, load balancing is completely taken care of by the underlying run-time system. The overhead of queue access is very low unless task stealing is necessary.

OpenMP is gaining popularity with the arrival of multicore processors and may eventually replace Pthreads completely. OpenMP provides structured parallelism in a combination of SPMD and fork-join styles. Its strengths are in the work sharing constructs that allow to distribute loop iterations according to one of several predefined scheduling policies. OpenMP has a weak memory consistency model, that is, flush operations may be required to guarantee that stale copies of a shared variable will no longer be used after a certain point in the program. The forthcoming OpenMP 3.0 standard supports a task-based programming model, which makes it more appropriate for fine-grained nested parallelism than its earlier versions. A drawback of OpenMP is that it has no good programming support yet for controlling data layout in memory.

In PGAS languages, accesses to remote shared data are either handled by separate access primitives, as in NestStep [53], while in other cases the syntax is the same and the language run-time system automatically traps to remote memory access if nonlocal data have been accessed, as in UPC (Universal Parallel C) [17]. Loops generally take data affinity into account, such that iterations should preferably be scheduled to processors that have most of the required data stored locally, to reduce communication.

The Linda system [19] provides a shared memory via the concept of tuple spaces, which is much more abstract than linear addressing, and partly resembles access to a relational database.

A programming language for transactional programming, called ATOMOS, has been proposed by Carlstrom *et al.* [18]. As an example of a library of lock-free parallel data structures, we refer to NOBLE [79].

In parallel language design, there is some consensus that extending sequential programming languages by a few new constructs is not the cleanest way to build a sound parallel programming language, even if this promises the straightforward reuse of major parts of available sequential legacy code. Currently, three major new parallel languages are under development for the high-performance computing domain: Chapel [20] promoted by Cray, X10 [31] promoted by IBM, and Fortress [7] promoted by Sun. It remains to be seen how these will be adopted by programmers and whether any of these will be strong enough to become the dominant parallel programming language in a long range perspective. At the moment, it rather appears that OpenMP and perhaps UPC are ready to enter mainstream computing platforms.

2.6 Data Parallel Models

Data parallel models include SIMD (Single Instruction, Multiple Data) and vector computing, data parallel computing, systolic computing, cellular automata, VLIW computing, and stream data processing.

Data parallel computing involves the elementwise application of the same scalar computation to several elements of one or several operand vectors (which usually are arrays or parts thereof), creating a result vector. All element computations must be independent of each other and may therefore be executed in any order, in parallel, or in a pipelined way. The equivalent of a data parallel operation in a sequential programming language is a loop over the elementwise operation, scanning the operand and result vector(s). Most data parallel languages accordingly provide a data parallel loop construct such as `forall`. Nested `forall` loops scan a multidimensional iteration space, which maps array accesses in its body to possibly multidimensional slices of involved operand and result arrays. The strength of data parallel computing is the single state of the program's control, making it easier to analyze and to debug than task-parallel programs where each thread may follow its own control flow path through the program. On the other hand, data parallel computing alone is quite restrictive; it fits well for most numerical computations, but in many other cases it is too unflexible.

A special case of data parallel computing is *SIMD computing* or *vector computing*. Here, the data parallel operations are limited to predefined SIMD/vector operations such as element-wise addition. While any SIMD/vector operation could be rewritten by a data parallel loop construct, the converse is not true: the elementwise operation to be applied in a data parallel computation may be much more complex (e.g., contain several scalar operations and even nontrivial control flow) than what can be expressed by available vector/SIMD instructions. In SIMD/vector programming languages, SIMD/vector operations are usually represented using array notation such as `a[1:n-1] = b[2:n] + c[1:2*n:2]`.

Systolic computing is a pipelining-based parallel computing model involving a synchronously operating processor array (a so-called *systolic processor array*) where processors have local memory and are connected by a fixed, channel-based interconnection network. Systolic algorithms have been developed mainly in the 1980s, mostly for specific network topologies such as meshes, trees, pyramids, or hypercubes, see e.g. Kung and Leiserson [60]. The main motivation of systolic computation is that the movement of data between processors is typically on a nearest-neighbor basis (so-called shift communications), which has shorter latency and higher bandwidth than arbitrary point-to-point communication on some distributed memory architectures, and that no buffering of intermediate results is necessary as all processing elements operate synchronously.

A *cellular automaton* (CA) consists of a collection of finite state automata stepping synchronously, each automaton having as input the current state of one or several other automata. This neighbour relationship is often reflected by physical proximity, such as arranging the CA as a mesh. The CA model is a model for massively parallel computations with structured communication. Also, systolic computations can be viewed as a kind of CA computation. The restriction of nearest-neighbor access is relaxed in the *global cellular automaton* (GCA) [47], where the state of each automaton includes an index of the automaton to read from in this step. Thus, the neighbor relationship can be time dependent and data dependent. The GCA is closely related to the CROW PRAM, and it can be efficiently implemented in reconfigurable hardware, i.e. field programmable gate arrays (FPGA).

In a *very large instruction word* (VLIW) processor, an instruction word contains several assembler instructions. Thus, there is the possibility that the compiler can exploit instruction level parallelism (ILP) better than a superscalar processor by having knowledge of the program to be compiled. Also, the hardware to control execution in a VLIW processor is simpler than in a superscalar processor, thus possibly leading to a more efficient execution. The concept of *explicitly parallel instruction computing (EPIC)* combines VLIW with other architectural concepts such as predication to avoid conditional branches, known from SIMD computing.

In stream processing, a continuous stream of data is processed, each element undergoing the same operations. In order to save memory bandwidth, several operations are interleaved in a pipelined fashion. As such, stream processing inherits concepts from vector and systolic computing.

Practical Relevance. Vector computing was the paradigm used by the early vector supercomputers in the 1970s and 1980s and is still an essential part of modern high-performance computer architectures. It is a special case of the *SIMD computing* paradigm, which involves SIMD instructions as basic operations in addition to scalar computation. Most modern high-end processors have vector units extending their instruction set by SIMD/vector operations. Even many other processors nowadays offer SIMD instructions that can apply the same operation to 2, 4 or 8 subwords simultaneously if the subword-sized elements of each operand and result are stored contiguously in the same word-sized register. Systolic computing has been popular in the 1980s in the form of multi-unit co-processors for high-performance computations. CA and GCA have found their niche for implementations in hardware. Also, with the relation to CROW PRAMs, the GCA could become a bridging model between high-level parallel models and efficient configurable hardware implementation. VLIW processors became popular in the 1980s, were pushed aside by the superscalar processors during the 1990s, but have seen a re-birth with Intel's Itanium processor. VLIW is today also a popular concept in high-performance processors for the digital signal processing (DSP) domain. Stream processing has current popularity because of its suitability for real-time processing of digital media.

Implementations. APL [48] is an early SIMD programming language. Other SIMD languages include Vector-C [63] and *C** [73]. Fortran90 supports vector computing and even a simple form of data parallelism. With the HPF [52] extensions, it became a full-fledged data parallel language. Other data parallel languages include ZPL [78], NESL [12], Dataparallel C [43] and Modula-2* [69].

CA and GCA applications are mostly programmed in hardware description languages. Besides proposals for own languages, the mapping of existing languages like APL onto those automata [50] has been discussed. As the GCA is an active new area of research, there are no complete programming systems yet.

Clustered VLIW DSP processors such as the TI 'C62xx family allow parallel execution of instructions, yet apply additional restrictions on the operands, which is a challenge for optimizing compilers [55].

Early programming support for stream processing was available in the Brook language [16] and the Sh library (Univ. Waterloo). Based on the latter, RapidMind provides a commercial development kit with a stream extension for C++.

2.7 Task-Parallel Models and Task Graphs

Many applications can be considered as a set of *tasks*, each task solving part of the problem at hand. Tasks may communicate with each other during their existence, or may only accept inputs as a prerequisite to their start, and send results to other tasks only when they terminate. Tasks may spawn other tasks in a fork-join style, and this may be done even in a dynamic and data dependent manner. Such collections of tasks may be represented by a *task graph*, where nodes represent tasks and arcs represent communication, i.e. data dependencies. The *scheduling* of a task graph involves ordering of tasks and mapping of tasks to processing units. Goals of the scheduling can be minimization of the application's runtime or maximization of the application's efficiency, i.e. of the machine resources. Task graphs can occur at several levels of granularity.

While a superscalar processor must detect data and control flow dependencies from a linear stream of instructions, *dataflow computing* provides the execution machine with the application's dataflow graph, where nodes represent single instructions or basic instruction blocks, so that the underlying machine can schedule and dispatch instructions as soon as all necessary operands are available, thus enabling better exploitation of parallelism. Dataflow computing is also used in *hardware-software co-design*, where computationally intensive parts of an application are mapped onto reconfigurable custom hardware, while other parts are executed in software on a standard microprocessor. The mapping is done such that the workloads of both parts are about equal, and that the complexity of the communication overhead between the two parts is minimized.

Task graphs also occur in *grid computing* [37], where each node may already represent an executable with a runtime of hours or days. The execution units are geographically distributed collections of computers. In order to run a grid application on the grid resources, the task graph is scheduled onto the execution units. This may occur prior to or during execution (*static* vs. *dynamic* scheduling). Because of the wide distances between nodes with corresponding restricted communication bandwidths, scheduling typically involves clustering, i.e. mapping tasks to nodes such that the communication bandwidth between nodes is minimized. As a grid node more and more often is a parallel machine itself, also tasks can carry parallelism, so-called *malleable* tasks. Scheduling a malleable task graph involves the additional difficulty of determining the amount of execution units allocated to parallel tasks.

Practical Relevance. While dataflow computing in itself has not become a mainstream in programming, it has seriously influenced parallel computing, and its techniques have found their way into many products. Hardware software co-design has gained some interest by the integration of reconfigurable hardware with microprocessors on single chips. Grid computing has gained considerable attraction in the last years, mainly driven by the enormous computing power needed to solve grand challenge problems in natural and life sciences.

Implementations. A prominent example for parallel dataflow computation was the MIT Alewife machine with the ID functional programming language [4].

Hardware-software codesign is frequently applied in digital signal processing, where there exist a number of multiprocessor systems-on-chip (DSP-MPSoC), see e.g. [83]. The Mitrion-C programming language from Mitrionics serves to program the SGI RASC appliance that contains FPGAs and is mostly used in the field of bio informatics.

There are several grid middlewares, most prominently Globus [36] and Unicore [30], for which a multitude of schedulers exists.

3 General Parallel Programming Methodologies

In this section, we briefly review the features, advantages and drawbacks of several widely used approaches to the design of parallel software.

Many of these actually start from an existing sequential program for the same problem, which is more restricted but of very high significance for software industry that has to port a host of legacy code to parallel platforms in these days, while other approaches encourage a radically different parallel program design from scratch.

3.1 Foster's PCAM Method

Foster [35] suggests that the design of a parallel program should start from an existing (possibly sequential) algorithmic solution to a computational problem by partitioning it into many small tasks and identifying dependences between these that may result in communication and synchronization, for which suitable structures should be selected. These first two design phases, partitioning and communication, are for a model that puts no restriction on the number of processors. The task granularity should be as fine as possible in order to not constrain the later design phases artificially. The result is a (more or less) scalable parallel algorithm in an abstract programming model that is largely independent from a particular parallel target computer. Next, the tasks are agglomerated to macrotasks (processes) to reduce internal communication and synchronization relations within a macrotask to local memory accesses. Finally, the macrotasks are scheduled to physical processors to balance load and further reduce communication. These latter two steps, agglomeration and mapping, are more machine dependent because they use information about the number of processors available, the network topology, the cost of communication etc. to optimize performance.

3.2 Incremental Parallelization

For many scientific programs, almost all of their execution time is spent in a fairly small part of the code. Directive-based parallel programming languages such as HPF and OpenMP, which are designed as a semantically consistent extension to a sequential base language such as Fortran and C, allow to start from sequential source code that can be parallelized incrementally. Usually, the most computationally intensive inner loops are identified (e.g., by profiling) and parallelized first by inserting some directives, e.g. for loop parallelization. If performance is not yet sufficient, more directives need to be inserted, and even rewriting of some of the original code may be necessary to achieve reasonable performance.

3.3 Automatic Parallelization

Automatic parallelization of sequential legacy code is of high importance to industry but notoriously difficult. It occurs in two forms: *static parallelization* by a smart compiler, and *run-time parallelization* with support by the language's run-time system or the hardware.

Static parallelization of sequential code is, in principle, an undecidable problem, because the dynamic behavior of the program (and thereby the exact data dependences between statement executions) is usually not known at compile time, but solutions for restricted programs and specific domains exist. Parallelizing compilers usually focus on loop parallelization because loops account for most of the computation time in programs and their dynamic control structure is relatively easy to analyze. Methods for data dependence testing of sequential loops and loop nests often require index expressions that are linear in the loop variables. In cases of doubt, the compiler will conservatively assume dependence, i.e. non-parallelizability, of (all) loop iterations. Domain-specific methods may look for special code patterns that represent typical computation kernels in a certain domain, such as reductions, prefix computations, dataparallel operations etc., and replace these by equivalent parallel code, see e.g. [29]. These methods are, in general, less limited by the given control structure of the sequential source program than loop parallelization, but still rely on careful program analysis.

Run-time parallelization techniques defer the analysis of data dependences and determining the (parallel) schedule of computation to run-time, when more information about the program (e.g., values of input-dependent variables) is known. The necessary computations, prepared by the compiler, will cause run-time overhead, which is often prohibitively high if it cannot be amortized over several iterations of an outer loop where the dependence structure does not change between its iterations. Methods for run-time parallelization of irregular loops include doacross parallelization, the inspector-executor method for shared and for distributed memory systems, the privatizing DOALL test [71] and the LRPD test [72]. The latter is a software implementation of speculative loop parallelization.

In *speculative parallelization of loops*, several iterations are started in parallel, and the memory accesses are monitored such that potential misspeculation can be discovered. If the speculation was wrong, the iterations that were erroneously executed in parallel must be rolled back and re-executed sequentially. Otherwise their results will be committed to memory.

Thread-level parallel architectures can implement general thread-level speculation, e.g. as an extension to the cache coherence protocol. Potentially parallel tasks can be identified by the compiler or on-the-fly during program execution. Promising candidates for speculation on data or control independence are the parallel execution of loop iterations or the parallel execution of a function call together with its continuation (code following the call). Simulations have shown that thread-level speculation works best for a small number of processors [81].

3.4 Skeleton-Based and Library-Based Parallel Programming

Structured parallel programming, also known as *skeleton programming* [22, 68] restricts the many ways of expressing parallelism to compositions of only a few, predefined patterns, so-called skeletons. *Skeletons* [22, 25] are generic, portable, and reusable basic program building blocks for which parallel implementations may be available. They are typically derived from higher-order functions as known from functional programming languages. A skeleton-based parallel programming system, like P3L [8, 68], SCL [25, 26], eSkel [21], MuesLi [59], or QUAFF [32], usually provides a relatively small, fixed set of skeletons. Each skeleton

represents a unique way of exploiting parallelism in a specifically organized type of computation, such as data parallelism, task farming, parallel divide-and-conquer, or pipelining. By composing these, the programmer can build a structured high-level specification of parallel programs. The system can exploit this knowledge about the structure of the parallel computation for automatic program transformation [40], resource scheduling [9], and mapping. Performance prediction is also enhanced by composing the known performance prediction functions of the skeletons accordingly. The appropriate set of skeletons, their degree of composability, generality, and architecture independence, and the best ways to support them in programming languages have been intensively researched in the 1990s and are still issues of current research.

Composition of skeletons may be either non-hierarchical, by sequencing using temporary variables to transport intermediate results, or hierarchical by (conceptually) nesting skeleton functions, that is, by building a new, hierarchically composed function by (virtually) inserting the code of one skeleton as a parameter into that of another one. This enables the elegant compositional specification of multiple levels of parallelism. In a declarative programming environment, such as in functional languages or separate skeleton languages, hierarchical composition gives the code generator more freedom of choice for automatic transformations and for efficient resource utilization, such as the decision of how many parallel processors to spend at which level of the compositional hierarchy. Ideally, the cost estimations of the composed function could be composed correspondingly from the cost estimation functions of the basic skeletons. While non-nestable skeletons can be implemented by generic library routines, nestable skeletons require, in principle, a static preprocessing that unfolds the skeleton hierarchy, e.g. by using C++ templates or C preprocessor macros.

The exploitation of *nested parallelism* specified by such a hierarchical composition is quite straightforward if a fork-join mechanism for recursive spawning of parallel activities is applicable. In that case, each thread executing the outer skeleton spawns a set of new threads that execute also the inner skeleton in parallel. This may result in very fine-grained parallel execution and shifts the burden of load balancing and scheduling to the run-time system, which may incur tremendous space and time overhead. In a SPMD environment like MPI, UPC or Fork, nested parallelism can be exploited by suitable group splitting.

Parallel programming with skeletons may be seen in contrast to parallel programming using parallel library routines. Domain-specific parallel subroutine libraries e.g. for numerical computations on large vectors and matrices are available for almost any parallel computer platform. Both for skeletons and for library routines, reuse is an important purpose. Nevertheless, the usage of library routines is more restrictive because they exploit parallelism only at the bottom level of the program's hierarchical structure, that is, they are not compositional, and their computational structure is not transparent for the programmer.

4 Conclusion

At the end of this review of parallel programming models, we may observe some current trends and speculate a bit about the future of parallel programming models.

As far as we can foresee today, the future of computing is parallel computing, dictated by physical and technical necessity. Parallel computer architectures will be more and more hybrid, combining hardware multithreading, many cores, SIMD units, accelerators and on-chip communication systems, which requires the programmer and the compiler to solicit parallelism, orchestrate computations and manage data locality at several levels in order to achieve reasonable performance. A (perhaps extreme) example for this is the Cell BE processor.

Because of their relative simplicity, purely sequential languages will remain for certain applications that are not performance critical, such as word processors. Some will have standardized parallel extensions and be slightly revised to provide a well-defined memory model for use in parallel systems, or disappear in favor of new, true parallel languages. As parallel programming leaves the HPC market niche and goes mainstream, simplicity will be pivotal especially for novice programmers. We foresee a multi-layer model with a simple deterministic high-level model that focuses on parallelism and portability, while it includes transparent access to an underlying lower-level model layer with more performance tuning possibilities for experts. New software engineering techniques such as aspect-oriented and view-based programming and model-driven development may help in managing complexity.

Given that programmers were mostly trained in a sequential way of algorithmic thinking for the last 50+ years, migration paths from sequential programming to parallel programming need to be opened. To prepare coming generations of students better, undergraduate teaching should encourage a massively parallel access to computing (e.g. by taking up parallel time, work and cost metrics in the design and analysis of algorithms) early in the curriculum [54].

>From an industry perspective, tools that allow to more or less automatically port sequential legacy software are of very high significance. Deterministic and time-predictable parallel models are useful e.g. in the real-time domain. Compilers and tools technology must keep pace with the introduction of new parallel language features. Even the most advanced parallel programming language is doomed to failure if its compilers are premature at its market introduction and produce poor code, as we could observe in the 1990s for HPF in the high-performance computing domain [52], where HPC programmers instead switched to the lower-level MPI as their main programming model.

Acknowledgments. Christoph Kessler acknowledges funding by Ceniit 01.06 at Linköpings universitet; by Vetenskapsrådet (VR); by SSF RISE; by Vinnova SafeModSim; and by the CUGS graduate school.

References

1. Ferri Abolhassan, Reinhard Drefenstedt, Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. On the physical design of PRAMs. *Computer J.*, 36(8):756–762, December 1993.
2. Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency: multicore programming with transactional memory. *ACM Queue*, (Dec. 2006/Jan. 2007), 2006.
3. Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: a Tutorial. *IEEE Comput.*, 29(12):66–76, 1996.
4. Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, David Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. 22nd Int. Symp. Computer Architecture*, pages 2–13, 1995.
5. A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.
6. Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.
7. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification version 1.0 β, March 2007. http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.
8. Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P3L: A structured high level programming language and its structured support. *Concurrency – Pract. Exp.*, 7(3):225–255, 1995.
9. Bruno Bacci, Marco Danelutto, and Susanna Pelagatti. Resource Optimisation via Structured Parallel Programming. In [27], pages 1–12, April 1994.
10. Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.
11. R. Bisseling. *Parallel Scientific Computation – A Structured Approach using BSP and MPI.* Oxford University Press, 2004.

12. Guy Blelloch. Programming Parallel Algorithms. *Comm. ACM*, 39(3):85–97, March 1996.
13. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multi-threaded run-time system. In *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pages 207–216, 1995.
14. Hans-J. Boehm. Threads cannot be implemented as a library. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 261–268, 2005.
15. Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29:187–207, 2003.
16. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.
17. William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, second printing, IDA Center for Computing Sciences, May 1999.
18. Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christoforos E. Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *Proc. Conf. Prog. Lang. Design and Impl. (PLDI)*, pages 1–13. ACM, June 2006.
19. Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.
20. Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. submitted, 2007.
21. Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
22. Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman and MIT Press, 1989.
23. Richard Cole and Ofer Zajicek. The APRAM: Incorporating Asynchrony into the PRAM model. In *Proc. 1st Annual ACM Symp. Parallel Algorithms and Architectures*, pages 169–178, 1989.
24. David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles & Practice of Parallel Programming*, pages 1–12, 1993.
25. J. Darlington, A. J. Field, P. G. Harrison, P. H. B. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *Proc. Conf. Parallel Architectures and Languages Europe*, pages 146–160. Springer LNCS 694, 1993.
26. J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming.* ACM Press, July 1995. *SIGPLAN Notices* 30(8), pp. 19–28.
27. K. M. Decker and R. M. Rehmann, editors. *Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser, Basel (Switzerland), 1994. Proc. IFIP WG 10.3 Working Conf. at Monte Verita, Ascona (Switzerland), April 1994.
28. F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM Symp. on Comput. Geometry*, pages 298–307, 1993.
29. Beniamino di Martino and Christoph W. Keßler. Two program comprehension tools for automatic parallelization. *IEEE Concurr.*, 8(1), Spring 2000.
30. Dietmar W. Erwin and David F. Snelling. Unicore: A grid computing environment. In *Proc. 7th Int.l Conference on Parallel Processing (Euro-Par)*, pages 825–834, London, UK, 2001. Springer-Verlag.
31. Vijaj Saraswat et al. Report on the experimental language X10, draft v. 0.41. White paper, http://www.ibm.research.com/ comm/research_projects.nsf/pages/x10.index.html, February 2006.
32. Joel Falcou and Jocelyn Serot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In *Proc. ParCo-2007.* IOS press, 2008.
33. Martti Forsell. A scalable high-performance computing solution for networks on chips. *IEEE Micro*, pages 46–55, September 2002.
34. S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Annual ACM Symp. Theory of Computing*, pages 114–118, 1978.
35. Ian Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.
36. Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *Proc. IFIP Int.l Conf. Network and Parallel Computing*, LNCS 3779, pages 2–13. Springer, 2006.
37. Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int.l J. Supercomputer Applications*, 15(3):200–222, 2001.
38. Phillip B. Gibbons. A More Practical PRAM Model. In *Proc. 1st Annual ACM Symp. Parallel Algorithms and Architectures*, pages 158–168, 1989.
39. W. K. Giloi. Parallel Programming Models and Their Interdependence with Parallel Architectures. In *Proc. 1st Int. Conf. Massively Parallel Programming Models.* IEEE Computer Society Press, 1993.
40. Sergei Gorlatch and Susanna Pelagatti. A transformational framework for skeletal programs: Overview and case study. In Jose Rohlim et al., editor, *IPPS/SPDP'99 Workshops Proceedings, IEEE Int. Parallel Processing Symp. and Symp. Parallel and Distributed Processing*, pages 123–137. Springer LNCS 1586, 1999.
41. Allan Gottlieb. An overview of the NYU ultracomputer project. In J.J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 25–95. Elsevier Science Publishers, 1987.

42. Susanne E. Hambrusch. Models for Parallel Computation. In *Proc. Int. Conf. Parallel Processing, Workshop on Challenges for Parallel Processing*, 1996.
43. Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.
44. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. Int. Symp. Computer Architecture*, 1993.
45. Heywood and Leopold. Dynamic randomized simulation of hierarchical PRAMs on meshes. In *AIZU: Aizu International Symposium on Parallel Algorithms/Architecture Synthesis.* IEEE Computer Society Press, 1995.
46. Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: the BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, 1998.
47. Rolf Hoffmann, Klaus-Peter Völkmann, Stefan Waldschmidt, and Wolfgang Heenes. GCA: Global Cellular Automata. A Flexible Parallel Model. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 66–73, London, UK, 2001. Springer-Verlag.
48. Kenneth E. Iverson. *A Programming Language.* Wiley, New York, 1962.
49. Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.
50. Johannes Jendrsczok, Rolf Hoffmann, Patrick Ediger, and Jörg Keller. Implementing APL-like data parallel functions on a GCA machine. In *Proc. 21st Workshop Parallel Algorithms and Computing Systems (PARS)*, 2007.
51. Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM Programming.* Wiley, New York, 2001.
52. Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proc. Int. Symposium on the History of Programming Languages (HOPL III)*, June 2007.
53. Christoph Kessler. Managing distributed shared arrays in a bulk-synchronous parallel environment. *Concurrency – Pract. Exp.*, 16:133–153, 2004.
54. Christoph Kessler. Teaching parallel programming early. In *Proc. Workshop on Developing Computer Science Education – How Can It Be Done?, Linköpings universitet, Sweden*, March 2006.
55. Christoph Kessler and Andrzej Bednarski. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience*, 18:1353–1390, 2006.
56. Christoph W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The J. of Supercomputing*, 17:245–262, 2000.
57. Christoph W. Kessler. A practical access to the theory of parallel algorithms. In *Proc. ACM SIGCSE'04 Symposium on Computer Science Education*, March 2004.
58. Christoph W. Keßler and Helmut Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. J. Parallel Programming*, 25(1):17–50, February 1997.
59. Herbert Kuchen. A skeleton library. In *Proc. Euro-Par'02*, pages 620–629, 2002.
60. H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. In C. Mead and L. Conway, editors, *Introduction to VLSI systems*, pages 271–294. Addison-Wesley, 1980.
61. Christian Lengauer. A personal, historical perspective of parallel programming for high performance. In Günter Hommel, editor, *Communication-Based Systems (CBS 2000)*, pages 111–118. Kluwer, 2000.
62. Claudia Leopold. *Parallel and Distributed Computing. A survey of models, paradigms and approaches.* Wiley, New York, 2000.
63. K.-C Li and H. Schwetman. Vector C: A Vector Processing Language. *J. Parallel and Distrib. Comput.*, 2:132–169, 1985.
64. L. Fava Lindon and S. G. Akl. An optimal implementation of broadcasting with selective reduction. *IEEE Trans. Parallel Distrib. Syst.*, 4(3):256–269, 1993.
65. B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of Parallel Computation: a Survey and Synthesis. In *Proc. 28th Annual Hawaii Int. Conf. System Sciences*, volume 2, pages 61–70, January 1995.
66. W. F. McColl. General Purpose Parallel Computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 1993.
67. Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. Real PRAM programming. In *Proc. Int. Euro-Par Conf.'02*, August 2002.
68. Susanna Pelagatti. *Structured Development of Parallel Programs.* Taylor&Francis, 1998.
69. Michael Philippsen and Walter F. Tichy. Modula-2* and its Compilation. In *Proc. 1st Int. Conf. of the Austrian Center for Parallel Computation*, pages 169–183. Springer LNCS 591, 1991.
70. Thomas Rauber and Gudula Rünger. Tlib: a library to support programming with hierarchical multi-processor tasks. *J. Parallel and Distrib. Comput.*, 65:347–360, March 2005.
71. Lawrence Rauchwerger and David Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. 8th ACM Int. Conf. Supercomputing*, pages 33–43. ACM Press, July 1994.
72. Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 218–232. ACM Press, June 1995.
73. J. Rose and G. Steele. C*: an Extended C Language for Data Parallel Programming. Technical Report PL87-5, Thinking Machines Inc., Cambridge, MA, 1987.
74. D. B. Skillicorn. Models for Practical Parallel Computation. *Int. J. Parallel Programming*, 20(2):133–158, 1991.

75. D. B. Skillicorn. miniBSP: a BSP Language and Transformation System. Technical report, Dept. of Computing and Information Sciences, Queens's University, Kingston, Canada, Oct. 22 1996. http://www.qucis.queensu.ca/home/skill/mini.ps.
76. David B. Skillicorn and Domenico Talia, editors. *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1995.
77. David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, June 1998.
78. Lawrence Snyder. The design and development of ZPL. In *Proc. ACM SIGPLAN Third symposium on history of programming languages (HOPL-III).* ACM Press, June 2007.
79. Håkan Sundell and Philippas Tsigas. NOBLE: A non-blocking inter-process communication library. Technical Report 2002-02, Dept. of Computer Science, Chalmers University of Technology and Göteborg University, SE-41296 Göteborg, Sweden, 2002.
80. Leslie G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8):103–111, August 1990.
81. Fredrik Warg. *Techniques to reduce thread-level speculation overhead.* PhD thesis, Dept. of Computer Science and Engineering, Chalmers university of technology, Gothenburg (Sweden), 2006.
82. Xingzhi Wen and Uzi Vishkin. Pram-on-chip: first commitment to silicon. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 301–302, New York, NY, USA, 2007. ACM.
83. Wayne Wolf. Guest editor's introduction: The embedded systems landscape. *Computer*, 40(10):29–31, 2007.