ISSN (Online) : 2348 - 2001

International Refereed Journal of Reviews and Research

Volume 4 Issue 1 January-February 2016

International Manuscript ID : 23482001V4I101022016-19

(Approved and Registered with Govt. of India)

MODELS FOR PARALLEL COMPUTING REVIEW AND PERSPECTIVES

Nahar Priyank Hasmukhbhai

Research Scholar

Shri Venkateshwara University

Uttar Pradesh, India

Dr. Parveen Kumar

Professor

Shri Venkateshwara University

Uttar Pradesh, India

Abstract: Parallelism has now become pervasive in today's league of computers, so it seems important to provide an analysis of the models that have evolved in past years for parallel computation. In this paper the concentration is specifically on models with certain pragmatic application in sync with a viewpoint on models with possible future applicability. Along with citing, the models paper also ascribes implementation in programming language and means.

Keywords: Parallel Computational Model, 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 both in the desktop and embedded.

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.

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. data flow and declarative

ISSN (Online) : 2348 - 2001

International Refereed Journal of Reviews and Research

Volume 4 Issue 1 January-February 2016

International Manuscript ID : 23482001V4I101022016-19

(Approved and Registered with Govt. of India)

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.

A brief survey of parallel programming models and comment on their merits and perspectives from both a theoretical and practical view is been presented. The focus is on brief classification of parallel models and languages. Basic references are articles and books in the literature such as by [7], Skillicorn[44], Giloi[24], Maggs[38], Skillicorn and Talia[45,46], Lengauer[35], and Leopold[36].

2. Model Survey

Before presenting parallel programming models, in next section (2.1) two fundamental issues in parallel program execution that occur in implementations of several models is discussed.

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.

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.

Nested parallelism can be achieved with SPMD style as well, name if a group of p processors is subdivided into s subgroups of p_i processors each, where $\sum p_i \leq p$. 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.

2.2 Parallel Random Access Machine

The Parallel Random Access Machine (PRAM) model was proposed by Fortune and Wyllie [20] 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

ISSN (Online) : 2348 - 2001

International Refereed Journal of Reviews and Research

Volume 4 Issue 1 January-February 2016

International Manuscript ID : 23482001V4I101022016-19

(Approved and Registered with Govt. of India)

CRCW even allows simultaneous write accesses by several processors to the same memory location in the same clock cycle.

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 JaJa[29]. It is the most basic model for parallel algorithms[32] and focuses on pure parallelism only, rather than data locality and communication efficiency.

A cost effective realization of PRAMs is possible using hardware techniques such as multithreading and smart combining networks, such as the NYU Ultracomputer [25], SBPRAM by Wolfgang Paul's group in Saarbrucken [1,30,40], XMT by Vishkin [49], and ECLIPSE by Forsell [19]. PRAM is a general purpose model that is completely insensitive to data locality.

PRAM model variants have been proposed within the parallel algorithms theory community such as asynchronous PRAM variants [13, 23], the hierarchical PRAM (H PRAM) ,the block PRAM [4], the queuing PRAM (Q PRAM), and the distributed PRAM(DRAM), to name only a few. The BSP model, discussed in Section 2.4, regarded a relaxed PRAM, introduced to bridge the gap between idealistic models and actual parallel machines.

Implementations: Several PRAM programming languages have been, such as Fork [30,33]. Also methods for translating PRAM algorithms automatically for other models such as BSP or message passing have been proposed.

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. 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.

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[14]. 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. The LogP model has been extended to the LogGP model [5], 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. With the definition of vendor independent message passing libraries, message passing became the dominant programming style on large parallel computers.

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++. Open source implementation exists such as OpenMPI.

2.4 Bulk Synchronous Parallelism

The bulk synchronous parallel (BSP) model, proposed by Valiant in 1990 [48] and modified by McColl [39], enforces a structuring of message

ISSN (Online) : 2348 - 2001

International Refereed Journal of Reviews and Research

Volume 4 Issue 1 January-February 2016

International Manuscript ID : 23482001V4I101022016-19

(Approved and Registered with Govt. of India)

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.

Practical Relevance: BSP model allows deriving realistic predictions of execution time and can thereby guide algorithmic design decisions and balance trade-offs.

Implementations: The BSP model is mainly realized in the form of libraries such as BSPlib [27] or PUB [9] for an SPMD execution style.

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, the threads of execution run asynchronously.

A recent development is transactional memory ([26],[2]), 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

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.

Implementations:Cilk [8], is a shared memory parallel language for algorithmic multithreading.

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.The Linda system [10] 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.

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.

Practical Relevance: Vector computing was the paradigm used by the early vector super computers in the 1990s and 1980s and is still an essential part of modern high performance computer architectures. It is a special case of the SIMD computing paradigm. Most modern high end processors have vector units extending their instruction set by SIMD/vector operations. VLIW is today also a popular concept in high performance processors for the digital signal processing (DSP) domain.

Implementations: APL [28] is an early SIMD programming language. Other SIMD languages include Vector-C [37] and C* [43]. Fortran90 supports vector computing and even a simple form of data parallelism. With the HPF [31] extensions, it became a full-fledged data parallel language. Other data parallel languages include ZPL [47] NESL Dataparallel C and Modula-2* [42].

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

ISSN (Online) : 2348 - 2001

International Refereed Journal of Reviews and Research

Volume 4 Issue 1 January-February 2016

International Manuscript ID : 23482001V4I101022016-19

(Approved and Registered with Govt. of India)

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.

Practical Relevance: 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 data flow computation was the MIT Alewife machine with the ID functional programming language [3].There are several grid middlewares, most prominently Globus [22] and Unicore [17], for which a multitude of schedulers exists.

3. General Parallel Programming Methodologies

In this section, the features, advantages and drawbacks of several widely used approaches to the design of parallel software has been reviewed.

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 [21] 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 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.

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.

3.4 Automatic Parallelization

Automatic parallelization of sequential legacy code is of high importance to industry but 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.

3.5 Skeleton Based and Library-Based Parallel Programming

Structured parallel programming, also known as skeleton programming [12,41] restricts the many ways of expressing parallelism to compositions of only a few, predefined patterns, called skeletons. Skeletons [12, 15] 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 [6, 41], SCL [15, 16], eSkel [11], MuesLi [34], or QUAFF [18], 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. While non-nestable skeletons can be implemented by generic

ISSN (Online) : 2348 - 2001

International Refereed Journal of Reviews and Research

Volume 4 Issue 1 January-February 2016

International Manuscript ID : 23482001V4I101022016-19

(Approved and Registered with Govt. of India)

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.

4. Conclusion

The review of parallel programming models presented in this paper surfaces the current trends and provide speculation about the future of parallel programming models.

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 require the programmer and the compiler to solicit parallelism, orchestrate computations and manage data locality at several levels in order to achieve reasonable performance for example the Cell BE processor. Because of their relative simplicity, purely sequential languages will remain for certain applications that are not performance critical, applications that are not performance critical such as word processors. New software engineering techniques such as aspect oriented and view based programming and model driven development may help in managing complexity.

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 [31], where HPC programmers instead switched to the lower level MPI as their main programming model.

5. References

1. Ferri Abolhassan, Reinhard Drefenstedt, Jorg Keller, Wolfgang J. Paul, and Dieter Scheerer. On the physical design of PRAMs. Computer J 36(8):756-762, 1993 December.
2. Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency: multicore programming with transactional memory. ACMQueue, (Dec. 2006/jan. 2007), 2006.
3. Anant Agarwal, Ricardo Bianchini, The MIT Alewife machine: Architecture and performance. In Proc. 22nd Int. Symp. Computer Architecture, pages 2-13, 1995.
4. A Aggarwal. A. K.Chandra, and M.Snir.Communication complexity of PRAMs. Theoretical Computer Science, 71:3-28, 1990.
5. 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.
6. 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):225-225, 1995.
7. Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. ACM Computing Surveys 21(3):261-322, September1989.
8. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul. Cilk: an efficient multi-threaded run-time system. In Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, pages 207-216, 1995.
9. Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library. Parallel Computing, 29:187-207, 2003.
10. Nicholas Carriero and David Gelernter. Linda in context. Commun. ACM, 32(4):444-458, 1989.
11. Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing, 30(3):389-406, 2004.
12. Murray I. Cole. Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman and MIT Press, 1989.
13. 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.
14. David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schause. LogP: Towards a realistic model of parallel computation. In Principles & Practice of Parallel Programming pages 1-12, 1993.
15. J. Darlington, A. J. Field, P. G. Harrison. Parallel Programming Using Skeleton Functions. In Proc. Conf. Parallel Architectures and Languages Europe, pages 146-160. Springer LNCS 694, 1993.
16. 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 38(3), pp. 19-28.
17. 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.
18. Joel Falcou and Jocelyn Serot. Formal semantics applied to the implementation of a skeleton based parallel

ISSN (Online) : 2348 - 2001

International Refereed Journal of Reviews and Research

Volume 4 Issue 1 January-February 2016

International Manuscript ID : 23482001V4I101022016-19

(Approved and Registered with Govt. of India)

programming library. In Proc. ParCo-2007. IOS press, 2008.
19. Martti Forsell. A scalable high performance computing solution for networks on chips. IEEE Micro, pages 46-55, September 2002.
20. S. Fortune and J. Wyllie Parallelism in random access machines. In Proc. 10th Annual ACM Symp. Theory of Computing, pages 114-118, 1978.
21. Ian Foster. Designing and Building Parallel Programs. Addison Wesley, 1995.
22. 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.
23. Phillip B. Gibbons, A. More Practical PRAM Model. In Proc. 1st Annual ACM Symp. Parallel Algorithms and Architectures, pages 158-168, 1989.
24. 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.
25. 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.
26. Maurice Herlihy and J. Eliot B. Moss. Transactional memory Architectural support for lock free data structures. In Proc. Int. Symp. Computer Architecture, 1993.
27. Jonathan M. D. Hill, Bill McColl. BSPlib: the BSP Programming Library. Parallel Computing, 24(14): 1947-1980, 1998.
28. Kenneth E. Iverson. A Programming Language. Wiley, New York, 1962.
29. Joseph JaJa. An Introduction to Parallel Algorithms. Addison Wesley, 1992.
30. Jorg Keller, Christoph Kessler, and Jesper Traff. Practical PRAM Programming. Wiley, New York, 2001.
31. 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.
32. Christoph W. Kessler. A practical access to the theory of parallel algorithms. In Proc. ACM SIGCSE'04 Symposium on Computer Science Education, March 2004.
33. Christoph W. KeBler and Helmut Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. Int. J. Parallel Programming, 24(1):17-50, February 1997.
34. Herbert Kuchen, A skeleton library. In. Proc. Euro Par'02, pages 620-629, 2002.
35. Christian Lengauer. A personal, historical perspective of parallel programming for high performance. In Gunter Hommel, editor, Communication Based Systems (CBS 2000), pages 111-118, Kluwer, 2000.
36. Claudia Leopold. Parallel and Distributed Computing. A survey of models, paradigms and approaches. Wiley, New York, 2000.
37. K. C. Li and H. Schwetman. Vector C: A Vector Processing Language. J. Parallel and Distrib. Comput., 2:132-169, 1985.
38. 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.
39. 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.
40. Wolfgang J. Paul, Peter Bach, Michael Bosch.Real PRAM programming. In Proc. Int. Euro Par Conf.'02, August 2002.
41. Susanna Pelagatti. Structured Development of Parallel Programs. Taylor Francis, 1998.
42. Michael Philippsen and Walter F. Tichy Modula and its Compilation. In Proc. 1st Int. Conf. of the Austrian Center for Parallel Computation, pages 169-183. Springer LNCS 591, 1991.
43. J. Rose and G. Steele. C*: an Extended C Language for Data Parallel Programming. Technical Report PL87-5, Thinking Machines Inc., Cambridge, MA, 1987.
44. D. B. Skillicorn. Models for Practical Parallel Computation. Int. J. Parallel Programming, 20(2):133-138, 1991.
45. David B. Skillicorn and Domenico Talia, editors. Programming Languages for Parallel Processing. IEEE Computer Society Press, 1995.
46. David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. ACM Computing Surveys, June 1998.
47. Lawrence Snyder. The design and development of ZP.L In Proc. ACM SIGPLAN Third symposium on history of programming languages (HOPL III). ACM Press, June 2007.
48. Leslie G. Valiant. A Bridging Model for Parallel Computation. Comm. ACM, 33(8):103-111, August 1990.
49. 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.