Framework Applications for Parallel Computers

by
Schrettner, Lajos

Abstract of the dissertation
submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
József Attila University
August, 1999

1 Introduction

Parallel computers and parallel programming have received considerable attention in recent years. This can be attributed to two factors. First, the ever growing need for processing power makes it necessary to look for new architectures because serial computers are approaching physical limits. It seems that the only way to increase the performance of computers is by using multiple processing elements which operate concurrently, in parallel. Second, the history of computing taught us that hardware and software are equally important, the complexity of problems can only be tackled by applying appropriate methods to design our systems. Parallelism may help allowing us to model real world and artificial phenomena in a natural way, thereby leading to cleaner, more reliable applications.

Besides its advantages, parallelism has drawbacks. The coordination of concurrent activities poses new problems and requires new methods, but these are inherently more difficult than in the sequential case. Further, despite its potential, parallel programming has not become widespread because there are no universally accepted general purpose parallel programming models. A large number of architectures and programming languages have been proposed, but at present, applications developed for a given architecture can only be ported to other architectures with substantial effort.

In the dissertation, I study several aspects of the parallel computing field, present applications that are the results of my work in this area and propose a parallel programming model (NOP) which can serve as a general programming model. The applications are not designed to solve a single problem, but provide a framework encapsulating functions that can be used to implement a range of systems similar in their characteristics.

1.1 Parallel hardware

Parallel computer architectures can be categorised in several ways depending on what characteristics are considered [14]. A parallel architecture consists of processors, memory module(s) and optionally some kind of interconnection mechanism to connect processors and/or processors and memory modules [10]. The simplest and most popular classification [11] distinguishes four classes based on the number of independent instruction and data streams: SISD (Single Instruction/Single Data stream), SIMD (Single Instruction/Multiple Data stream), MISD (Multiple Instruction/Single Data stream), and MIMD (Multiple Instruction/Multiple Data stream). SISD machines are the traditional sequential von Neumann machines. The SIMD category [13] contains vector computers, where a single instruction is exe-

cuted on multiple data items in a synchronised manner. There is no consensus on what the MISD class covers, most authors do not put any existing machines into it. MIMD machines contain autonomous processing elements (PEs, processors with private memory) working asynchronously.

Multiple data stream machines can be further classified based on where the data streams originate from. Shared memory (SM) machines have a memory module that all processors use for data storage. Multiple accesses to the same memory location have to be coordinated in order to avoid data corruption. In distributed memory (DM) machines each processor has only private memory. Nevertheless, PEs have to cooperate to achieve a common goal, so these machines contain an interconnection facility which enables PEs to exchange data. PEs can be connected by point-to-point communication links or they all can be connected to a broadcast network. The most widely used parallel machines are SM-MIMD (eg. multiuser UNIX machines, even multiprocessor PCs) and DM-MIMD, although we can find examples for others as well [13].

Ideally, parallel machines should be extensible, ie. users should be able to add PEs and/or memory to increase processing power. The SM-SIMD architecture has two disadvantages, both limiting its extendibility [10][12]. First there is a need for global synchronisation among processors, this becomes more and more problematic as the number of processors grows. Second, the shared interface to the global memory forms a bottleneck. To some extent, the bottleneck can be handled by using multistage switches, multiple memory banks and caching. Sadly, cache coherence is hard to maintain due to random accesses to memory. Cache performance is better if successive accesses are localised both in terms of processors and memory locations. It seems that distributed memory is more favourable in this respect, but programming with shared memory is easier because of its similarity to conventional sequential computer programming.

The parallel applications to be described have been developed on a transputer network [15][16][17][18][20]. A transputer network is a perfect example of a DM-MIMD machine, each processor has private memory, PEs are connected via point-to-point links. In contrast to SM-MIMD machines, DM-MIMD machines are easy to extend using only a constant number of communication links per processor.

1.2 Parallel software, programming models

The main obstacle in the way of widespread use of parallel machines is that software development is fundamentally more difficult. Owing to the more

complex structure of multicomputers, new language constructs, new software development methods have to be devised. The situation is similar to the one a few decades ago, when the software crisis stimulated the birth of software engineering. An active area of research in parallel software engineering [22][23][26] tries to explore the methods and tools that can be used to engineer reliable parallel systems. By now there is a great deal of knowledge and experience related to sequential algorithms, but parallel machines and programming are relatively new. There are many competing architectures, programming concepts and languages, but there are no universally accepted, general parallel programming models [27][28].

It is important that we make a distinction between architectures and programming models. A programming model is an abstract machine with its data structures and operations that connects the physical machine and the higher level language(s) used by humans. It has to meet conflicting requirements from the two sides, it has to be convenient to use for problem solving and has to allow efficient implementation(s). In sequential programming, the imperative programming model (the von Neumann model) is successful because high-level imperative languages can efficiently be compiled into machine code. In parallel programming, those models that allow us to write efficient programs use low-level primitives, while high-level paradigms are not universally applicable.

Existing parallel models can be categorised in various ways, but two categories seem to be the most important to discuss. The first category contains extensions to the von Neumann model. They inherit a number of features from it: they are imperative and use random access to fetch and store data from/to memory. The imperative style allows very efficient programs to be written, sometimes with considerable effort. Programmers are used to this style, although another important category of models, the declarative programming models [29][30][31][32] are gaining on. They have a number of advantages over the von Neumann model even if we are interested only in sequential implementations. The functional model for example provides explicit representation for data structures, allows functions to be treated as values and exhibits referential transparency. Problems (not directly related to parallel programming) that can cause inefficiencies are the lack of selective update of data structures and the missing concept of computational state.

1.2.1 Extensions of the von Neumann model

PRAM: Parallel Random Access Machine [34]. The data structure is global random access memory, it has a number of processors working in a

synchronised manner. It is easy to program, but harder to implement. The most suitable architecture for it is the SM-SIMD, in fact there is an obvious one-to-one correspondence between the components.

Process models. Processes are independent, asynchronous threads of control. Process models usually augment the von Neumann model with operations handling process creation/destruction and synchronisation/communication. A serious drawback common to all process models is that the execution of programs is very sensitive to timing conditions, making the design and testing of programs very hard. Asynchronous processes are very useful in certain application areas (eg. real-time systems), but they seem to cause more problems than they solve in others.

Shared memory process models: Various models use various methods for synchronisation, eg. semaphores, monitors [35] and derivatives of these. Shared memory models are very close to the SM-MIMD architecture, with all the problems of shared memory (see above). Attempts to simulate shared memory over physical distributed memory have not provided an efficient general solution either.

Distributed memory process models [36]: Here processes communicate dominantly by sending messages to each other. The models are very general and powerful: extensions of existing languages provide models in this category: well known are the PVM [37] and MPI libraries; more formal ones are Distributed Processes [38] and CSP [39]. These models are well suited to SM-MIMD and DM-MIMD architectures. OCCAM [45][46][47] is a parallel programming language based on CSP. The transputer and OCCAM were designed together to provide a uniform platform for parallel applications. INMOS Parallel C [48] is an other language for the transputer, it is a standard ANSI C implementation with added libraries for process control and communication facilities.

Other high-level models. While the models above were probably influenced by their underlying architectures, the ones in this group are farther away from the hardware.

BSP: The Bulk-Synchronous Parallel [27] model consists of PEs, memory units, a router, and a synchronising facility. Computation consists of supersteps with synchronisation at regular intervals to

determine the end of each superstep. With careful adjustment of machine parameters, good implementations are possible on SM-MIMD and DM-MIMD architectures. BSP is a very promising approach, the construct to express parallelism in the NOP model resembles BSP supersteps.

Linda: Linda has a global Tuple Space (TS) as its data structure, processes insert, read and remove tuples to/from TS. It is conceptually simple from the programmers point of view and allows several implementation strategies [41].

1.2.2 Declarative models

In these models [29][30][31][32] the data structures are certain aggregates of elementary types, the operations can access and operate on data that are passed to them, thus memory references are localised and exclusive (cf. caching in SM machines). Further, as their name suggests, they do not completely specify the sequence of events during program execution (evaluation). These two factors (localised references and incomplete sequencing) allow independent subproblems to be detected and evaluated in parallel [42][43][44]. These models are best suited for SM-MIMD machines, but MD-MIMD implementations are possible using dynamic load balancing techniques. There are attempts to design special purpose architectures as well.

1.3 Decomposition and load balancing

The central idea of parallel computation is that multiple processors work simultaneously to achieve a final result. For this to be possible, the problem to be solved has to be decomposed into independent parts: functional and/or domain decomposition can be used to achieve this.

In the case of *functional decomposition* the algorithm is partitioned and possibly assigned to different processors. Data are routed from one processor to the other as necessary during the computation. One popular form of functional decomposition is the *pipeline*, where processors are arranged in an assembly line fashion, each performing one particular transformation on data flowing trough it. The database query language executor described in the dissertation uses a pipeline arrangement. *Domain decomposition* means that data is partitioned into subsets which then can be processed more or less independently. When using data decomposition, often all processors execute the same program, this is called the SPMD (Single Program/ Multiple Data) paradigm. This approach was used in the load balancing environment, the other application presented in the dissertation.

Load balancing is the activity of ensuring that all processors perform useful work during the execution time of a given program. Some problems have a regular structure that allows us to define a decomposition in advance, before the program runs. In these cases load balancing is part of the design phase and is called *static load balancing*. Image processing problems, for example, tend to belong to this class as very often a large image can be decomposed into rectangular areas and processed in parallel. On the other hand, there are several important problems that cannot be decomposed in advance, therefore load balancing becomes an integral part of the algorithm. This is called *dynamic load balancing*. The performance of such an algorithm mainly depends on the effectiveness of load balancing. A widely used dynamic load balancing method is the so-called processor farm [49]. The particular load balancing method described in the dissertation can be regarded as a generalisation of the farm principle.

1.4 Performance of parallel programs

The primary aim of parallelisation is faster program execution, so we are most interested in the *running time* of parallel programs. Running time is influenced not only by the input data, but by the number of processors used. Other measures also yield useful information about the behaviour of programs [50]. *Speedup* is defined as the ratio of the running time of the fastest sequential program and the running time of the parallel program. This ratio indicates the improvement in solution time using parallelism. *Efficiency* is defined as the ratio of the speedup and the number of processors. It is an important measure of performance because it shows how effectively the processors are used. Higher efficiency means better utilisation, in the best case its value is 1.

1.5 Outline of the dissertation

Chapter 2 summarizes work carried out under the Copernicus project *Large Parallel Databases*. The most important results were

- specification and formal definition of the Object Functional Language (OFL)
- proposal for an extended OFL-based database machine
- design of an abstract machine to serve as a basis for the definition and implementation of the language

- multiprocessor implementation of an OFL executor
- tests showed that the executor was able to reduce execution time

Chapter 3 deals with dynamic load balancing, in particular it

- defines the class of (special) decomposible problems
- describes a test environment implemented to enable the investigation of special decomposible problems
- introduces the *processor commune* model as a generalisation of the processor farm model
- reports test results which indicate that the processor commune model is effective for the execution of declarative languages

Chapter 4

- defines a building block that can be used to construct deadlock-free processes
- shows by successive refinement and case analysis that the construction is correct
- shows how the processor commune can be built from the building block

Chapter 5 explains the structure of the processor commune implementation on a transputer network highlighting some of the difficulties and design decisions.

Chapter 6 introduces the NOP (NOde Processing) model

- defines the data structures and transformations
- shows how it relates to existing models
- describes a high level language based on the model
- gives examples of how problems can be solved with the model
- shows how the model can be implemented on shared memory multiprocessors

Programs are listed in the Appendices.

2 Database query language executor

In 1994–95 a team of department members took part in the Copernicus project named Large Parallel Databases (LPD). The aim of the collaborating parties was to explore diverse aspects of parallel database technology and provide both hardware and software solutions to identified problems. Our team ventured on creating a suite of tools for a database query language drafted by another member of the project.

Object oriented database management systems are based on an underlying object oriented data model and play an important role in the efficient management of structured objects [51]. The object oriented data model supports the construction of complex objects through user defined types. Similar objects are grouped together in classes encapsulating methods and attributes. Users of such databases express their queries in OQL (Object Query Language). The LPD project members decided that it was advantageous to translate OQL queries into an intermediate form, into expressions given in Object Functional Language (OFL).

The first task was to define the OFL language precisely based on earlier drafts [52][53]. As a result, a specification and formal definition for the syntax and semantics of the OFL language was given. It identified the data objects, object references and functions comprising the language. It clearly separated the components that belong to the language and those of the environment in which programs are run. The grammar was given in a form acceptable to the PROF-LP compiler generator system [54].

A low-level abstract machine was designed to serve as a basis for the definition of semantics and to ease the implementation of OFL. It was a convenient target for compilation and allowed efficient implementation on transputers [15][16][17][18][20]which was the target architecture of the system.

To take advantage of the parallel query executor, an extension to the conventional database management system structure was proposed. The extended structure contains the conventional one, so conventional query processing is possible without the users noticing any change. The extended structure clearly separates the responsibilities of the parallel OFL executor and the rest of the system (data access module), this allowed us to concentrate on the key issues throughout the development phase. The extended database machine is a MIMD architecture, the processors are arranged in a pipeline and grouped together into functional units. Each functional unit is assigned a portion of the program and data are passed from one unit to the next during execution, that is the problem was solved by functional decomposition.

The execution of an OFL program consists of two phases. In the first, preprocessing phase, the structure of the program is analysed and the functional decomposition takes place. The separated program fragments are assigned to the functional units of the extended database management system and translated into abstract machine code. In the second phase, the translated program fragments are downloaded to the functional units and executed. The executor itself is a multiprocessor implementation of the OFL abstract machine written in parallel C.

3 Dynamic load balancing system

For distributed memory multiprocessor systems the employment of appropriate load balancing strategies is crucial to the performance of most applications. In general, the strategy which leads to an even spread of work throughout the processing nodes is likely to produce the best overall performance ratings. In some instances this is straightforward to achieve: if it is possible to assess the computational and communication loads before program execution, the program sub-tasks can be appropriately allocated to the processing nodes. However for many applications these loads are not determinable in advance and hence load balancing becomes a dynamic activity that runs concurrently with the application.

One standard approach to the management of dynamic load balancing is the employment of *processor farms* [49]: a *master* processing node is responsible for the farming out of the tasks to the *worker* nodes, each worker computes and returns the task results and is allocated new work. This allows the work load to be well partitioned, even in the case where the computational complexity, and hence the execution time, varies from task to task. The drawback with the processor farm approach is the reliance on a central node to act as load balancer. In large multiprocessor machines this provides a communication bottleneck: this is particularly acute when subtasks are created dynamically within the worker nodes. In this case, work allocation involves conveyance of the newly formed task description to the central node, followed by transfer to its appropriately designated worker node. For many applications it is therefore likely to be of benefit to employ some form of locally based inter-processor work scheduling mechanism.

Work at Sheffield Hallam University on parallel declarative languages for the implementation of knowledge based systems led to the development of a parallel logic interpreter based on OR parallelism [24]. The significance of this application for the purpose of the load balancing investigation lies in the fact that the computational patterns that the system gives rise to are

typical of a large class of applications in the artificial intelligence and knowledge based systems fields. In these applications the program structure can be described by means of an execution tree; the evaluator (eg. the parallel logic interpreter) accepts a task, processes it and produces descendants of the input record. Program execution continues with evaluation of the descendant tasks, the parent having ceased to exist on completion of the task spawning operation. In these applications parallel execution gives rise to a highly dynamic system. During a program run, not only are tasks dynamically created in a manner which is unpredictable at the start of run time, but the tasks themselves vary considerably in their individual execution times thus producing the most difficult scenario as far as work scheduling and load balancing are concerned.

In Chapter 3, work on a load balancing system for *decomposible problems* is described. The load balancing system has been developed to provide an environment in which the means of mapping decomposible applications to distributed memory multi-processor systems can be explored. The processor farm approach of designating one master node to perform the task of controlling work distribution has been replaced with a *processor commune* model: under this approach, work allocation is a community activity with each processing node combining master-worker functionality. Each processing element continues to execute work (ie. evaluate tasks) and store any newly created tasks for local evaluation unless a state of work load imbalance is recognised. Processors maintain information about their own work load and compare it with that of neighbouring processors at appropriate times during program execution. When predefined thresholds are reached, work is transferred from a heavily loaded node to a lighter one.

In order to support investigation of different load balancing strategies, a test environment has been developed and comprises the following components:

- application model generator
- application executor
- load balancing analyser

The application model generator allows the computational characteristics of a particular program to be captured in the form of an application "mimic". This skeletal version, which provides a real time emulation of the application under consideration, is used as the basis for investigation of load balancing strategies with the application executor. This operates in the form of a local balancer process, installed on every processing element and works

concurrently with the local application tasks. Because of the separation of load balancing and application functionality, the system can be used to explore the performance of different applications under a range of predefined work scheduling algorithms. The runtime data obtained are then passed to the analyser tools.

Results obtained using this toolset are presented: these show that good performance can be obtained under nearest neighbour scheduling approaches. The overheads associated with the employment of work allocation mechanism are low and the manner of task distribution ensures that the system operates with maximum available parallelism throughout the application execution.

4 Correctness of the load balancing protocol

Except for the most trivial cases, the correct behaviour of programs is far from obvious and rapidly becomes less and less so as complexity increases. Parallel systems are typically amongst those of great complexity, so our confidence in their proper behaviour should be strengthened by some form of validation. Validation can be carried out after the system has been designed (and possibly implemented) by enumerating and checking the system states. However, as the number of system states grows exponentially when the number of system components or number of component states grows, this form of validation is limited [65].

Another possibility is to ensure correctness in the design phase. One advantage of this method is that we can use building blocks (components whose correctness have been proved) to construct larger systems. Once the correct behaviour of the building blocks is ensured, the system composed of these blocks is also correct as far as the interface requirements of the blocks are observed. A good example in this category is the client-server type systems [66]. In this approach processes are constrained to act in active (client) or passive (server) modes with respect to communication interaction. By following client-server interaction rules deadlock/livelock freedom can be guaranteed.

Synthesized from work with the processor commune system, a building block has been defined which enables processes to run independently and communicate with each other without a predefined communication pattern. This means that at any given time it is not known in advance which of two processes will send a message to the other. The processes work independently as long as they can, connection is established only when needed. The basic building block is successively extended in order to be able to deal

with more complex situations.

The setup for the problem consists of two processes connected by a pair of channels, they want to communicate with each other at unpredictable moments without risking getting deadlocked. All protocols were defined in OCCAM.

The first (*basic dynamic interaction*) version of the building block enables the two parties to exchange simple messages without deadlock. The second (*bounded resource dynamic interaction*) version introduces a *store* process which stores items to be exchanged between the two parties. There are a limited number of items, this restriction complicates the protocol, but deadlock freedom can be guaranteed.

The third version (*general dynamic interaction*) merges the former two. It stipulates that components use two types of data during their activity: *control data* and *mass data*. Control data items are small and carry control information, they do not accumulate but are processed upon receipt. Mass data items are large and carry the workload to be processed by the system, they accumulate if arrive faster than processing takes place. In fact the basic and bounded resource versions deal with these two types of data respectively. It is possible to combine them in a way that guarantees correct functioning. In general, correctness means that all specification requirements are met by the system. While most requirements can be problem specific, one of them is universal: absence of deadlock and livelock, and this is what is meant by 'correct' in this general setting. Beside this, the issue of fair resource handling is adressed where appropriate.

A building block suitable for using in parallel systems contains a coordinator *boss* process, a *store* process, and any number of general dynamic interaction *worker* processes. Such a system of processes is deadlock free. In the load balancing test system, each processing element has a copy of the general building block described above and as such, the load balancing system is deadlock free. In this particular case, a circulating token is used to detect termination of the computation [67], the token is implemented as a control data item. The tasks to be processed are of course mass data items.

5 Implementation of the load balancing system

By the end of the development of the parallel C version of the test system, based on the experience gained through the tests, it became apparent that OCCAM provides much better facilities for writing concurrent applications [47][19]. Among its chief advantages are the seamless integration of parallel and communication primitives into the language, the possibility of declar-

ing channel protocols, the strict type checks on data communicated on channels. Besides these, OCCAM's constructs are well defined, making it easier to discuss program correctness. All these factors influenced the decision that the system should be rewritten in OCCAM, the result of which is presented in Chapter 5.

6 Node processing model

Parallel programming languages have two advantages over sequential ones. One is that a parallel program can run faster if suitable hardware is available, the other is that as programs often model real world phenomena, the solution may be expressed more naturally using explicit or implicit parallelism. Despite this potential, parallel programming has not become widespread because there are no general purpose parallel programming models [27][28]. It is important that we make a distinction between architectures and programming models. A programming model is an abstract machine with its data structures and operations that connects the physical machine and the higher level language(s) used by humans. It has to meet conflicting requirements from the two sides, it has to be convenient to use for problem solving and has to allow efficient implementation(s). Humans are given in this scenario, the task is to find a model and design a machine such that the three together satisfy the conditions above. A number of parallel programming models and parallel architectures have been proposed, some are based on the successful sequential von Neumann model and architecture while others are based on new ideas. The von Neumann model has global random access memory as its data structure, the operations fetch, process and store data sequentially.

A general purpose parallel architecture has to be extendible, that is users have to be able to add PEs and memory to increase processing power. An ideal general purpose parallel programming model on the other hand has to hide the architectural details from programmers. They must be protected from having to deal with such details as the number of PEs, interconnection topologies, or placement of processes to processing elements. If these requirements are satisfied, portable parallel programs can be written, opening the way to widespread use of parallel computing. Note that portability does not mean that programs could be run on every machine, it only means that they can be run on the single architecture (or few architectures) which accompany the general programming model. As such machines are extendible, their speed and capacity can be adjusted by the users to their needs.

The NOP model proposed here incorporates a number of novel features. It has a single data structure that makes random memory accesses unnecessary. This radical approach has both advantages and disadvantages, but it is argued that it is worth considering as a viable alternative method of memory management in the context of parallel processing. The model has both imperative and declarative features, trying to combine the best of both worlds. Its simple imperative transformations can be easily and efficiently implemented, while larger programs can be constructed in a declarative style, benefiting from the substantial body of accumulated knowledge in the area of functional programming languages. By studying the existing programming models (see Introduction), we can observe that random access causes implementation difficulties in distributed and even in shared memory environments. A more regulated, more predictable flow of data would be more satisfactory. Further, the imperative style of programming is preferable if we want performance, declarative style is better at ensuring sound design and program correctness.

The NOP model uses implicit references exclusively, it is not possible to use variables. Operations cannot name their arguments, thus making random access impossible. The operations are commands in the imperative sense, they cause some action(s) to be performed. There are sequential, conditional and parallel composition operations. On the other hand, a NOP program is a mutually recursive equation set, in this sense it is declarative. Put into other terms, a functional program (equation set) determines the control structure of the imperative program, which is then executed to map the input into the final result. Using equations, programmers extend the transformation set of the model. These design choices are believed to produce a unique, radical, but satisfying model.

The ultimate goal of the model is of course to allow efficient multiprocessor implementations. A fully functional NOP simulator has been developed and implemented to carry out experiments and gain insight for further research. This sequential NOP simulator immediately translates into an SM-MIMD implementation, the only problem to solve is to regulate access to global data, but this can easily be done using standard mutual exclusion techniques. Research progresses toward investigating a combination of program transformations and dynamic load balancing to arrive at a distributed memory multiprocessor implementation.

References

[1] L. Schrettner, I. E. Jelly, Z. Alexin, T. Gyimóthy: **Parallel Execution of OFL Programs**, *Large Parallel Databases Technical Report*, Copernicus Project CP 93:6638, 1995

[2] L. Schrettner, T. Gyimóthy, Z. Alexin, J. Toczki: **Parallel Execution of Object Functional Queries**, *Annales Univ. Sci. Budapestinensis, Sectio Computatorica*, Vol. 17, 1998, pp. 339–354

[3] L. Schrettner, J. Toczki: **Dynamic Load Balancing for Decomposible Problems**, *Parallel Processing in Education*, Impact TEMPUS JEP's and Hungarian Transputer Users Group's Workshop, March 1993

[4] L. Schrettner, I. E. Jelly: **Investigation of Dynamic Load Balancing in Distributed Memory Multiprocessor Machines**, *Proc. of 15th Intl. Conf. on Information Technology Interfaces*, Pula, Croatia, June 1993

[5] L. Schrettner, I. E. Jelly: **A Test Environment for Investigation of Dynamic Load Balancing in Transputer Networks**, *Transputer Applications and Systems '93*, Transputer and Occam Engineering Series, Vol. 36, IOS Press, 1993, pp. 284–295

[6] J. Toczki, L. Schrettner: **Attribute Grammar Applications**, *Annales Univ. Sci. Budapestinensis, Sectio Computatorica*, Vol. 17, 1998, pp. 367–380

[7] L. Schrettner, M. Bohus: **A Parallel Design Method for Producing Valid Protocols**, *Proc. of 8th Symposium on Microcomputer and Microprocessor Applications*, October 1994, pp. 689–697

[8] L. Schrettner, I. E. Jelly: **Dynamic Process Interaction**, *Parallel Programming and Java*, Concurrent Systems Engineering Series, Vol. 50, IOS Press, 1997, pp. 261–273

[9] L. Schrettner: **The NOP Parallel Programming Model**, submitted for publication

[10] A. S. Tannenbaum: **Parallel Architectures**, in *Structured Computer Organization*, 3rd Edition, Ch. 8.2, Prentice-Hall, 1990

[11] M. J. Flynn: **Some Computer Organizations and their Effectiveness**, *IEEE Trans. Computers, Vol. C-21*, September 1972, pp. 948–960

[12] B. P. Lester: **The Art of Parallel Programming**, Ch. 3, Prentice-Hall, 1993

[13] R. Cypher, J. Sanz: **The SIMD Model of Parallel Computation**, Springer-Verlag, 1994

[14] D. Sima, T. Fountain, P. Kacsuk: , **Advanced Computer Architectures. A Design Space Approach**, Ch. 3, Addison Wesley, 1997

[15] INMOS Ltd.: **Transputer Databook**, 2nd Edition, 1989

[16] INMOS Ltd.: **The Transputer Development and *i*q Systems Databook**, 2nd Edition, 1991

[17] INMOS Ltd.: **Transputer Instruction Set**, Prentice Hall, 1988

[18] M. D. May, P. W. Thompson, P. H. Welch (Eds): **Networks, Routers and Transputers: Function, Performance and Application**, Transputer and OCCAM Engineering Series, IOS Press, 1993

[19] P. Kacsuk, Sz. Ferenczi: **Parallel and Concurrent Programming in Multitransputer Systems** BME Mérnöktovábbképző Intézet, Budapest, 1993 (in Hungarian)

[20] Telmat Informatique: **T.Node Hardware Manual**

[21] Telmat Informatique: **T.Node Software Manual**

[22] I. E. Jelly, I. Gorton: **Software Engineering for Parallel Systems**, *Information and Software Technology Journal*, Vol. 36, No. 7, 1994, pp. 381–396

[23] I. E. Jelly, I. Gorton, P. Croll (Eds): **Software Engineering for Parallel and Distributed Systems**, *Proc. 1st IFIP Intl. Workshop on Parallel and Distributed Software Engineering*, March 1996

[24] I. E. Jelly: **A Parallel Process Model and Architecture for a Pure Logic Language**, PhD Thesis, Sheffield City Polytechnic, 1990

[25] J. P. Gray, I. E. Jelly: **A Hybrid Transputer Based Architecture for Parallel Logic Language Execution**, *Proc. of First World Transputer Conference*, Transputing-91, IOS Press, April 1991

[26] S. C. Winter, P. Kacsuk: **Software Engineering for Parallel Processing**, *Proc. of 8th Symposium on Microcomputer and Microprocessor Applications*, October 1994

[27] L. G. Valiant: **A Bridging Model for Parallel Computation**, *Comm. ACM, Vol. 33, No. 8*, August 1990, pp. 103–111

[28] B. M. Maggs, L. R. Matheson, R. E. Tarjan: **Models of Parallel Computation: A Survey and Synthesis**, *Proc. of 28th Hawaii Intl. Conf. On System Sci. (HICSS)*, Vol. 2. 1995, pp. 61–70

[29] G. Michaelson: **An Introduction to Functional Programming Through Lambda Calculus**, International Computer Science Series, Addison-Wesley, 1988

[30] Å. Wikström: **Functional Programming Using Standard ML**, Prentice Hall International Series in Computer Science, Prentice Hall, 1987

[31] P. Hudak: **Conception, Evolution, and Application of Functional Programming Languages**, *ACM Computing Surveys*, Vol. 21, No. 3, ACM Press, September 1989, pp. 359–411

[32] W. Clocksin, C. Mellish: **Programming in PROLOG**, Springer Verlag, 1981

[33] S. A. Cook, R. A. Reckhow: **Time Bounded Random Access Machines**, *J. Comput. Systems Sci.*, Vol. 7, 1973, pp. 354–375

[34] R. M. Karp, V. Ramachandran: **Parallel Algorithms for Shared-Memory Machines**, in *Handbook of Theoretical Computer Science*, Vol. A, Ch. 17, Elsevier, 1990, pp. 869–942

[35] C. A. R. Hoare: **Monitors: An Operating System Structuring Concept**, *Comm. ACM*, Vol. 17, October 1974, pp. 549–557

[36] E. Bal, J. G. Steiner, A. S. Tannenbaum: **Programming Languages for Distributed Computing Systems**, *ACM Computing Surveys*, Vol. 21, No. 3, ACM Press, September 1989, 261–322

[37] V. S. Sunderan: : **PVM: A Framework for Distributed Computing**, *Concurrency, Practice and Experience*, Vol. 2, No. 4, December 1990, pp. 315–339

[38] P. B. Hansen: **Distributed Processes: A Concurrent Programming Concept**, *Comm. ACM*, Vol. 21, No. 11, November 1978, pp. 934–941

[39] C. A. R. Hoare: **Communicating Sequential Processes**, Prentice Hall, 1985

[40] A. W. Roscoe: **The Theory and Practice of Concurrency**, Prentice Hall, 1998

[41] D. Gelernter, N. Carriero: **Applications Experience with Linda**, in *Proc. PPEALS 1998*, SIGPLAN Not. (ACM), Vol. 23, No. 9, September 1988, pp. 173–187

[42] B. K. Szymanski (Ed): **Parallel Functional Languages and Compilers**, ACM Press Frontier Series, ACM Press, 1991

[43] P. Kelly: **Functional Programming for Loosely-coupled Multiprocessors**, Research Monographs in Parallel and Distributed Computing, Pitman, 1989

[44] E. Shapiro: **The Family of Concurrent Logic Programming Languages**, *ACM Computing Surveys*, Vol. 21, No. 3, ACM Press, September 1989, pp. 412–510

[45] INMOS Ltd.: **OCCAM 2 Reference Manual**, Prentice-Hall, 1988

[46] INMOS Ltd.: **OCCAM 2 Toolset User Manual**, *IMS D00205-DOCA*, 1991

[47] G. Jones, M. Goldsmith: **Programming in OCCAM 2**, Prentice-Hall, 1988

[48] INMOS Ltd.: **ANSI C Toolset User Manual**, *IMS D0214-DOCA*, 1990

[49] W. Day: **Farming: Towards a Rigorous Definition and Efficient Transputer Implementation**, *Transputer Systems—Ongoing Research*, IOS Press, 1992

[50] L. A. Crowl: **How to Measure, Present and Compare Parallel Performance**, *IEEE Parallel and Distributed Technology*, Spring 1994, pp. 9–25

[51] R. G. G. Cattell (Ed): **Object Databases: The ODMG Standard**, Morgan Kaufman, 1993

[52] G. Gardarin, F. Machuca, P. Pucheral: **A Functional Execution Model to Evaluate Object-Oriented Queries**, *PRiSM Technical Report*, University of Versailles, France, 1994

[53] G. Gardarin et al: **OFL: An Object Functional Language to Map Extended SQL**, *Large Parallel Databases Technical Report*, Copernicus Project CP 93:6638, 1994

[54] **PROF-LP User's Guide**, Research Group on the Theory of Automata, Szeged, 1987

[55] H. Alblas: **Attribute Evaluation Methods**, *Memoranda Informatica 89-20*, Faculteit der Informatica, University Twente, the Netherlands, 1989

[56] J. Toczki: **Attribute Grammars and their Applications**, *Dr. Univ. thesis*, Departments of Informatics, József Attila University, Szeged, Hungary, 1991, (in Hungarian)

[57] L. Schrettner: **Dynamic Attribute Evaluation**, *MSc. Thesis*, Dept. of Computer Science, JAU, 1992, (in Hungarian)

[58] M. Tóth: **Parallel execution of logic programs**, *BSc. Thesis*, Dept. of Computer Science, JAU, 1999, (in Hungarian)

[59] F. W. Burton, M. M. Huntbach: **Virtual Tree Machines**, *IEEE Trans. on Computers*, Vol. C-33, No. 3, March 1984

[60] D. L. McBurney, M. R. Sleep: **Transputer-Based Experiments with the ZAPP Architecture**, *University of East Anglia report no. SYS-C86-10*, October 1986

[61] F. C. H. Lin, R. M. Keller: **The Gradient Model Load Balancing Method**, *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 1, January 1987

[62] G. Cybenko: **Dynamic Load Balancing for Distributed Memory Multiprocessors**, *J. Parallel and Distributed Computing*, Vol. 7, October 1989

[63] M. H. Willebeck-Lemair, A. P. Reeves: **Strategies for Dynamic Load Balancing on Highly Parallel Computers**, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 9, September 1993

[64] P. K. K. Loh, W. J. Hsu, C. Wentorg, N. Sriskanthan: **How Network Topology Affects Dynamic Load Balancing**, *IEEE Parallel and Distributed Technology*, Fall 1996, pp. 25–35

[65] G. J. Holzmann: **Design and Validation of Computer Protocols**, Prentice-Hall International, 1991

[66] P. Welch, G. Justo and C. Willcock: **High-Level Paradigms for Deadlock-Free High-Performance Systems**, *Transputer Applications and Systems '93*, IOS Press, 1993 pp. 981–1004

[67] J. Misra: **Detecting Termination of Distributed Computation Using Markers**, *Proc. of the 2nd annual ACM Symposium on Principles of DC*, Montreal, August 1983, pp. 290–294

[68] M. Raynal: **Distributed Algorithms and Protocols**, Wiley, 1988