Parallel Implementations of Functional Languages

Wellington Santos Martins [*]
Universidade Federal de Goiás, Goiânia, BRAZIL
and
University of East Anglia, Norwich, UK

Abstract

With interest in parallel implementations of functional languages as a whole, we first discuss some issues related to parallel execution of functional programs, and then review some key proposals considering three differents approaches, namely Parallel lisp Systems, Dataflow Systems and Reduction Systems.

1. Introduction

Functional programming languages have received a special attention since the famous Backus' Turing Award lecture [Bac78], when Backus stated the advantages of functional programming languages over traditional imperative ones. During the last decade, a number of researchers have been attempted to put these ideas into practice by proposing different implementations of functional languages. Substantial progress has been obtained for the sequential implementations of functional languages [Aug84, Joh84, Kranz86], however the parallel implementations still do not present big advantages as it was expected theoretically.

2. Some issues related to parallelism

Every pure functional language can be considered as a version of the lambda-calculus with some syntactic sugar. A consequence of the Church-Rosser theorem is that if a given lambda-expression is reduced by two different reduction techniques, and if both reduction sequences yield normal forms, then the normal forms are identical. This way, if the sequence does not matter, the reduction can be done even in parallel. This suitability of functional programming languages for parallel processing and the present facilities to build parallel machines (VLSI technology) have led various researchers to propose different parallel implementations of functional languages.

[*]Supported by the Departamento de Estatística e Informática, UFG, Goiânia, Brazil and CNPq, Brazil, Research Grant Number 200721/91. Author's current address: School of Information Systems, UEA, Norwich, NR4 7TJ, UK. E-mail: wsm@uk.ac.uea.sys

2.1. Granularity of parallelism

In a parallel architecture, granularity is a measure of how much computation occurs between periods of communication. A fine-grained computation (small grain size) performs a small amount of computation between periods of communication. On the other hand, a coarse-grained computation (large grain size) performs a large amount of computation during communication intervals. In other words, communication occurs less frequently in a coarse-grained computation than in a fine-grained computation.

At first sight, very fine-grained computation appears to be more advantageous since more parallel computation can be obtained. However, any parallel machine has some overhead associated with the creation, execution and termination of a task. This means that the hardware sets lower bounds on how small the grain size can be; any value below this bound implies in more work spent in management than in computation. By contrast, too coarse-grained computation results in a small amount of parallelism. This suggests the choice of intermediate values for the grain, neither too small nor too large.

The division of a functional program in a number of suitable-grained tasks is certainly not easy. In general we have various redexes with different grain sizes so only a subset of these redexes may be proved useful for parallel evaluation. In order to decide if a redex is worth offloading we must calculate its work size, i.e., the amount of time needed for its execution considering a single processor, and compare this value to the machine's overhead costs. A task should only be offloading if these costs are compensated.

However, it is not always possible to determine the work size of a redex as it may depend on information only available at run time (input data). Therefore some approaches have been proposed to deal with this problem. Some systems rely only on the programmer's ability for indicating expressions, which are worth evaluating in parallel, through the use of constructs/annotations provided by the language.. Others make use of some heuristics to estimate the execution costs of some expressions and then automatically annotate the program. Others yet use execution-profiling information for providing empirical measurement of the time spent in each function and then decide which expressions must be annotated.

2.2. Distribution of Tasks

Once candidates for parallel evaluation have been determined they may be offloaded to other processors. This raises another issue, which is a trade-off between distributing the work effectively and exploiting locality of data structure. In general the methods employed vary according to the machine architecture employed in the implementation.

Some proposals are designed with special purpose architectures in mind. Dataflow machines have employed a ring structure where tasks and processors are separated. Each task contains all the information (operation code, input values and destination addresses) required for its processing and for the distribution of its results. The tasks are moved to the processing elements where they are evaluated with subsequent sending of the results. The results are captured by a special task which combine them in order to create new tasks. Some other proposals adopt a physical tree architecture where the processing is done at the leaves while the task distribution is performed by the elements which constitute the tree structure.

The majority of the proposals, however, employ conventional MIMD machines that can be built with hookups of conventional von Neumann chips. The subdivision of MIMD machines into shared memory machines and distributed memory machines gives rise to two different models of distribution of tasks. In shared memory machines the tasks selected are put in a global task pool where they wait for some idle processor in order to be reduced. An obvious problem with this model is the contention caused by the shared access; this problem can be alleviated by using a multi-level queue structure with each group of processors having access to a limited number or tasks. In distributed memory machines the tasks selected are usually spread over the nodes (processor/memory) according to some diffusion mechanism. This way, each node has its own task pool and the decision about offloading a task can depend on either local information, or local information and information about nearest neighbours, or yet information about all the nodes of the machine.

3. Proposals

The number of proposals for parallel implementations of functional languages is considerable. They differ from each other in so many aspects that is not easy to make a general classification. We decided to group the projects into three broad categories namely, Parallel Lisp Systems, Dataflow Systems and Reduction Systems. Although all these groups have a different emphasis in their implementations they all make use of some type of functional language.

3.1. Parallel Lisp Systems

This category has as its objective the parallel execution of Lisp dialects in existing parallel architectures. Most of the projects in this category make use of impure functional languages[1] (including constructs that cause side effects), support constructs to explicitly indicate parallelism and have a non-lazy semantics. The implementation technique usually used in these projects is the traditional one based on environments (SECD machine). The use of environments results in some waste of store since an environment may contain many objects that are not needed any more, however this problem is reduced when working with strict evaluation since suspended expressions don't occur. Another problem, particular to parallel implementations, is that the shared access to the environment may become a bottleneck. Nevertheless, an important aspect of environment-based implementations is that they can take advantage of an already well-understood technology developed to deal with environments.

Qlisp

Qlisp [GoGa88] is a dialect of Common Lisp proposed as a multiprocessing programming language which provides constructs that can be used by the programmer to indicate parallelism in the program. The approach used for Qlisp is queue-based multiprocessing, thus the new tasks, generated dynamically, are added to a queue for subsequent evaluation. The construct QLET is the basic construct used to introduce parallelism in Qlisp programs, its form is (QLET *pred* ((x_1 arg_1) . . . (x_n arg_n)) . *body*). The predicate *pred* is evaluated first and if its value is false the QLET will behave like an ordinary LET in Common Lisp, otherwise QLET will spawn a new process for each argument. The process evaluating the QLET is then blocked until all arguments have been evaluated, after which it is awakened and the statements in body are evaluated. An Alliant FX/8 computer system was the machine used to implement Qlisp. In order to avoid spawning too small tasks, the programs used an expression associate with QLET's predicate (e.g. > depth 0) to limit the tree of processes created. Real speed-ups were verified, but they required a great ability from the programmer.

Multilisp

In [Hal85], Halstead describes the language Multilisp, a version of the Lisp dialect Scheme extended with constructs for parallel execution. Like Scheme, Multilisp includes constructs for causing side effect, however they both have a side-effect-free subset which allows a pure functional style of programming. Multilisp's main construct for specifying parallelism is the future. The form (future X) immediately returns an object (future), which has initially an indeterminate status, and begins the evaluation of X concurrently

[1]Although allowing programs to be written in a completely side-effect free way since they possess a side-effect free subset. Modern lisps include first-class treatment of functions (higher-order functions), and lexical scoping (static binding).

with the computation containing (future X). When the evaluation of X finishes, the value produced replaces the future and the status becomes determined. Any task that attempts to access an indeterminate future will be suspended until it is resolved. Thus, futures can be used to compute nonstrict functions (functions that terminate independently of the termination of its arguments), but differently from lazy evaluation, future always begins the evaluation of its argument which can lead to the evaluation of unnecessary expressions. Although some kind of speculative work can be introduced by using future, Multilisp does not provide any means for halting the evaluation of a future which may be proved not needed. In addition, Multilisp provides the construct delay that implements lazy evaluation precisely. Multilisp was implemented on a Concert machine, a 32-processor shared-memory multiprocessor. The programs executed were first compiled into a machine-level byte-code language called MCODE, and subsequently interpreted by a native-code MC68000 program. The results obtained [Hal86] show that a substantial amount of concurrency can be exploited given a suitable parallel Multilisp program, however the programmer is responsible for the correct and optimum use of the constructs.

Mul-T

Another parallel Lisp system that makes use of futures for generating parallelism is Mul-T [KHM89], an extended version of the Yale T system, that was implemented on an Encore Multimax multiprocessor. Unlike Multilisp System which interprets its code, Mul-T uses a compiler (T3 ORBIT compiler) that generates native code for the Multimax's NS32332 processors. This resulted in a dramatic increase in speed, about 100 times faster than Multilisp.

Portable Standard Lisp on the BBN Butterfly

An implementation of Portable Standard Lisp (PSL) on the BBN Butterfly is described in [SKL88]. It differs from other PSL implementations because it includes the Multilisp's future construct for introducing parallelism. Another construct included is the touch construct that is used for forcing, explicitly, synchronization between the process evaluating a future and a process trying to access its value. The form (touch X) returns X if X is not a future, otherwise it either returns the result of the future's evaluation, or blocks the computation containing (touch X) until the evaluation is complete, when the result is returned. Thus, the programmer must insert, explicitly, the touch construct whenever a future might appear as an argument to a strict function. This use of touch touch facilitates the implementation but overload the programmer. However, unlike Multilisp which employs an interpreter for the byte-code language MCODE, Butterfly PSL compiles directly to the 68020 machine code. The results demonstrated good speed-ups which were constrained by memory/switch contention and non-local memory accesses.

3.2 Dataflow Systems

Conversely to the Parallel Lisp Systems and Reduction Systems, which employ the language first approach, Dataflow Systems came from concerns about a non conventional hardware (dataflow architectures). Thus the first languages employed were typically strict, reflecting the data-driven mode of operation, and had a first order semantics, as a consequence of the difficulty of constructing closures in the dataflow model. Dataflow languages are usually translated to an intermediate dataflow graph language where each instruction correspond to a node in the dataflow graph. We can think of an instruction as a task that is executed as soon as all its arguments are available. Parallel execution of this code is performed by distributing the instructions over different processors. More recently, some researchers have abandoned the traditional view of dataflow, with tokens flowing along the arcs of a dataflow graph, in favour of a multi-threaded style where a sequence of dataflow instructions is treated as a sequential thread. Thus the compiler produces code not as a dataflow graph but as a collection of sequential threads.

SISAL

SISAL (Streams and Iteration in a Single-Assignment language) [McGr84]] is a functional programming language designed to express algorithms, mainly in the area of numeric computation, for execution on highly parallel machines. Parallelism is detected automatically by the compiler which also decides on the amount of parallelism to exploit. However, SISAL was not designed with any particular target machine in mind, so we have implementations of SISAL on existing multiprocessors (e.g. HEP) as well as in a dataflow machine (e.g. Manchester Dataflow Machine). The implementations of SISAL make use of a powerful intermediate language called IF1 [SkGl83], a dataflow graph language. In [Sar89] Sarkar describes a quite successful approach for partitioning and scheduling SISAL programs. It is based on parameters describing the target multiprocessor and an execution profile of the program.

Id

The Id (Irvine dataflow) language has evolved considerably since its initial design [AGP78] and today it is considered a modern functional language with nonstrict semantics, higher order functions and a polymorphic type system. In [ArNi90] Arvind describes the execution of Id programs on the MIT tagged-token dataflow machine, a novel multiprocessor machine with purely data-driven instruction scheduling. In order to do that, Id programs are compiled into dynamic dataflow graphs, a parallel machine language, and, subsequently, are executed directly on the machine. More recently, much work has been done in implementing Id on the Monsoon machine, the

Tagged-token dataflow machine's successor. Besides the traditional dataflow style code, the multi-thread style is also being investigate [Tra91]. While in the traditional dataflow style we have a collection of threads each one containing just one instruction, in the multi-thread style we can have a collection of threads, each thread with several instructions. Within a thread, data is passed imperatively just as in conventional architectures, whereas, between threads, the data is passed via an stack frame, with tokens serving only to initiate and synchronize threads. These threads may be statically scheduled at compile time, allowing the compiler to exploit the implicit control flow of a sequential code stream.

3.3. Reduction Systems

The majority of the projects in this category use pure functional languages along with combinator-based reduction and MIMD architectures. Unlike the traditional environment-based reduction scheme, combinator-based reduction scheme requires no hierarchical environment structure since the compiler transforms the program into variable free machine code. The proposals are quite diverse covering different approaches: direct reduction of the original source program, use of combinators as machine code and compiled graph reduction.

ALICE

The first project here considered is ALICE (Applicative language Idealised Computing Engine) [DaRe81], a parallel graph reduction machine tailored to executing functional programs. The language employed is the functional language HOPE and the extraction of parallelism is intended to be automatic by the compiler. The program to be executed is compiled into a collection of packets that representing the graph to be reduced. The graph reduction is implemented by the creation and updating of packets. The so-called processing agents continuously select rewritable packets from the pool of packets, rewrite them and return the updated and newly created packets to the pool. The ALICE machine is composed of up to 40 processing agents and packet pool segments connected with a multistage switching network. The main problem verified was the grain size too small which prevented overcoming the latency of the transactions. The suggestion for future work included the development of a more sophisticated load sharing, direct compilation from ALICE CTL into OCCAM and the inclusion of a strictness analyser into the compiler.

ZAPP

Burton and Sleep [WaSl81] proposed a system called ZAPP (Zero Assignment Parallel Processor) which executes functional program by creating a virtual tree of processors. The system is specially designed to support divide and conquer algorithms. The ZAPP machine consists of a number of ZAPP elements (Processor and local memory) connected like an indirect binary n-cube. There is no shared memory and the ZAPP elements communicate to each other by message passing. Before executing a program the system must broadcast it to all ZAPP elements. After that, a single ZAPP element is chosen for receiving the initial problem and data. As the execution proceeds the divide and conquer processes are being distributed along the other elements. Load balancing is performed by all the ZAPP elements, but they are only permitted to offload tasks to their immediate neighbours. ZAPP is now [GMS91] being used to implement the language Concurrent Clean [BELP87], a graph rewriting language augmented with annotations for expressing parallelism. The system is called Concurrent Clean on ZAPP (CCOZ) and uses transputers to implement the ZAPP elements. Although the implementation is not complete, the results presented for popular benchmark programs are encouraging.

Rediflow

In [KeLi84], Keller describes the Rediflow system, a loosely-coupled multiprocessor system whose name is a combination of the words "reduction" and "dataflow". The system employs the graph reduction model but also allows the introduction of dataflow behaviour by using the pointers present in the structures of the reduction model to provide logical channels on which tokens flow. The language used is a dialect of Lisp called RediLisp and the intention is that the programmer should not bother about indicating parallelism. The Rediflow system is aimed at medium-grain tasks with function applications defining the unit of task granularity. The system is composed of several components called Xputers each one consisting of processor, memory and packet switch. A wide variety of interconnection networks can be used by the system which is said to be no topology-sensitive. The minimal topological assumptions are addressability (unique addressing in the entire system) and routability (given a request the switch must be able to determine the direction to route it) since the system creates a uniform logical address space. An interesting feature of the Rediflow system is its distributed load balancing technique which uses a "pressure" model to determine where to route excess tasks.

FFP

The FFP machine [MaMi84] is an unique proposal developed for the direct execution of the FFP (Formal Functional Programming) language of Backus. The machine consists of a large number of simple processing elements (cells) interconnected to form a binary tree. There are two kinds of cells, the leaf cells which are arranged in a linear array of processors and hold the string representing an FFP program, and the Tree cells which form a communication tree connecting the leaf cells through its leaves. The machine performs string reduction in a cyclical way, with each cycle composed of three phases: partitioning, execution and storage management. In the first phase, partitioning, the Tree cells are configured according to the string being reduced and the innermost applications are determined. In the execution phase the addresses of the symbols are generated, the microcode corresponding to the functional symbols is loaded and subsequently executed. Finally, in the last phase, storage management, the machine is rearranged to create space for inserting symbols. An important feature of the FFP machine is that it is not subjected to program decomposition, allocation and scheduling problems; the machine is dynamically reconfigured to fit the program and data. Some limitations of the FFP machine are the bottleneck near the root of the tree and the necessity of copying complete data structures for performing data movement.

GRIP

The GRIP (Graph Reduction in Parallel) machine [PCSH87] is another proposal for the parallel execution of functional programs. It is composed of up to eighty M68020 processing elements (PEs), 1 Mbyte of local memory per processor, up to 20 Intelligent Memory Units (each with 20 Mbytes of shared memory) and a high bandwidth bus to connect the components. One of the PEs is said to be the system manager and is responsible for global resource allocation and control within GRIP, the other PEs behave like supercombinators reduction machines. The program to be executed is held as a graph and is evaluated using supercombinator reduction. The program is expected to contain spark annotations which inform the run time system that certain expressions may be evaluated in parallel provided that there is a processor available. The unit of concurrency is a task which is associated with a sub-graph to be reduced. Every task is allowed to spark sub-tasks whose results are required, the new tasks are put in a task pool which is polled by PEs looking for work. The sparking mechanism is made dependent on the machine load in order to control the number of tasks.

Graphinators

Although most of the proposals use MIMD machines to implement graph reduction, Hudak and Mohr [HuMo88] proposed a really new graph reduction implementation which makes use of a SIMD machine, the Connection Machine. The idea is to treat the program graph as data and repeatedly execute a small set of primitive operations (instructions) called graphinators. A set of 7 graphinators was proved to be enough to implement all the Turner's standard set of combinator, {**S**, **K**, **I**, **B**, **C**, **S'**, **B'**, **C'**, **Y**}. The program is then transformed into an expression containing these standard combinators and subsequently evaluated in a 7-step reduction cycle. The parallelism exploited is the inherent parallelism of functional programs with no programmer annotations or compiler analysis. The results presented show that additional improvements are necessary for the system to be practical. The main problem verified was related to the communication time which was unacceptably long since no attempt to preserve locality was made.

Alfalfa and Buckwheat

In [Gol88] Goldberg investigates the viability of implementing functional languages on commercial multiprocessors. Two parallel graph reduction systems are presented: Alfalfa, implemented on the Intel iPSC hypercube multiprocessor (distributed memory), and Buckwheat, implemented on the Encore Multimax multiprocessor (shared-memory). Both systems are programmed using the ALFL language, a lazy functional language which has no constructs for specifying parallelism. The programs are intended to be automatically decomposed by the compiler. A key aspect of these implementations is the use of a special kind of combinator, called serial combinator [HuGo85], which is said to have optimal granularity. A serial combinator is basically a refinement of a supercombinator in which there is no concurrent substructure. Interesting techniques were presented to perform load balancing. Alfalfa used a diffusion scheduling (similar to that used in ZAPP) in which a processor only sends work to its neighbours. Buckwheat, in turn, used a two-level queue where each group of processors has direct access to its primary queue, and the secondary queue, which is shared among all the processors, is only accessed when the primary queue is empty or full. Both systems achieved good speed-ups for the benchmark programs considered. The performance of Buckwheat was verified to be superior to the Alfalfa indicating that more work remains to be done on partitioning data for loosely coupled machines.

MaRS

MaRS is a combinator graph reduction multiprocessor described in [Cont89]. The aim of this project is to obtain better performance by using specialized hardware (based on VLSI technology) for parallel graph reduction. The machine uses specific types of processor for reduction, memory and communication. A particular attention is paid to the communication processor since the machine is intended to provide a powerful communication medium. The communication processor is the basic element of an Omega switching interconnection network which, besides routing, is able to balance the activities in the machine. The language designed for MaRS programming is MaRS_Lisp, a pure functional language which provides annotations for expressing parallelism. The programs are subsequently translated into a machine language code based on a set of indexed combinators. The results presented, considering a simulated prototype, show good execution times for simple benchmark programs.

<ν, G>-machine

Augustsson and Johnsson proposed the <ν, G>-machine [AuJo89], a parallel graph reduction machine which performs compiled graph reduction. It is an extension of the G-machine which allows multiple executing threads of control, and incorporates the corresponding stacks in the graph structure (heap space). The introduction of parallelism is carried out by using spark annotations, as in the GRIP machine. The <ν, G>-machine was implemented on a Sequent Symmetry, a commercially available shared memory multiprocessor. The results presented are limited to a few small benchmark programs and demonstrated that good speed-ups can be achieved compared to the sequential G-machine.

4. Conclusions

Although functional programs may present a fair amount of implicit parallelism, the experiments so far conducted have demonstrated that the price to be paid, for parallel implementations, is not low. It is important to note, however, that some of the problems involved in the projects considered are not peculiar to functional programming languages, but are generic to all parallel programming languages, e.g. granularity, scheduling, locality. Some possible lines of research include techniques for determining program's execution costs, enhancing locality of data structures, and controlling granularity of parallelism.

5. Acknowledgements

I would like to thank the comments and suggestions provided by Professor M. R. Sleep which I found very useful while preparing this paper.

6. Bibliography

[AcDe79] Ackernan, W. B., and Dennis, J. B. "VAL - A value oriented algorithmic language , preliminary reference manual", *Tech. Rep. TR-218, Lab. for Computer Science, Massachusetts Institute of Technology*, June 1979.

[AGP78] Arvind, Gostelow, K. P., and Plouffe, W. "An asynchronous programming language and computing machine". *Tech. Rep. 114a, Dep. Information and Computer Science.* Univ. of California, Irvine, Dec. 1978.

[ArNi90] Arvind, and Nikhil, R. S. "Executing a Program on the MIT Tagged-Token Dataflow Architecture". *IEEE Transactions on Computers*, Vol. 39, No.3, March 1990.

[AuJo89] Augustsson, L. and Johnsson, T. "Parallel Graph Reduction with the <v,G>-machine, *Proceedings of the 1989 Conference on Functinal Programming Languages and Computer Architecture.*

[Aug84] Augustson, L. "A compiler for lazy ML". *In Symposium on Lisp and Functional Programming*, ACM, August 1984, pp. 218-227.

[Bac78] Backus, J. "Can programming be liberated from the von Neumann style ? A functional style and its algebra of programs", *Commun. ACM* 21, 8, 613-641, (1978).

[BELP87] Brus, T., Eekelen, M. C. J. D. van, Leer, M. van, Plasmeijer, M. J. "Clean - a language of functional graph rewriting". *Lecture Notes in Computer Science* 274, 1987, pp. 364-384.

[Cont89] Contessa, A., Cousin, W., Coustet, C., Cubero-Castan, M., Durrieu, G., Lecussan, B., Lemaitre, M., Ng, P. "MaRS, a combinator graph reduction multiprocessor". *Lecture Notes in Computer Science 365*, Springer-Verlag, 1989.

[DaRe81] Darlington, J. and Reeve, M. "ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages". *Proc. ACM Conf. on Functional Programming Languages and Computer Architectures*. New Hampshire, October 1981.

[DeMi75] Dennis, J. B., and Misunas, D. P. "A preliminary architecture for a basic data flow processor". *Proc. 2nd Int. Symp. Computer Architecture* , Houston, Tex., january 1975, pp. 126-132.

[FrWi78] Friedman, D. P. and Wise, D. "Aspects of Applicative Programming for Parallel Processing". *IEEE Transactions on Computers*, Vol.C-27, No.4, April 1978.

[GoGa88] Goldman, R. and Gabriel, R. P. "Preliminary Results with the Initial Implementation of Qlisp". *1988 ACM Symp. on Lisp and Functional Programming*. Snowbird, Utah, July 1988, pp. 143-152.

[Gol88] Goldberg, B. "Multiprocessor Exectution of Functional Programs". *International Journal of Parallel Programming*, Vol.17, No.5, 1988.

[GMS91] Goldsmith, R., McBurney, D. L., and Sleep, M. R. "Concurrent Clean on ZAPP". Proc. Symposium on the Semantics and Pragmatics of Generalized Graph Rewriting. Nijmegen, The Netherlands, December 1991.

[Hal85] Halstead, R. "Multilisp: A Language for Concurrent Symbolic Computation". *ACM Trans. on Prog. Languages and Systems 7:4*, October 1985, pp. 501-538.

[Hal86] Halstead, R. "An Assessment of Multilisp: Lessons from Experience". *International Journal of Parallel Programming*, Vol. 15, No. 6, 1986, pp. 459-501.

[HuGo85] Hudak, P. and Goldberg, B. "Serial Combinators: 'Optimal' Grains of Parallelism. *Lecture Notes in Computer Science 201*, Springer-Verlag, 1985.

[HuMo88] Hudak, P. and Mohr, E. "Graphinators and the duality of SIMD and MIMD". *Proceedings 1988 ACM Conference on LIsp and Functional Programming*, Salt Lake City, Utah, August 1988.

[Joh84] Johnsson, T. "Efficient compilation for lazy evaluation". *In symposium on Compiler Construction*, ACM, June 1984, pp. 58-69.

[KeLi84] Keller, R., and Lin, F. "Simulated performance of a reduction-based multiprocessor". *IEEE Comput. 17*, 7, pp. 70-82, July 1984.

[KHM89] Kranz, D. A., Halstead, R. H., and Mohr, E. "Mul-T: A High-Performance Parallel Lisp". 1989

[KLP79] Keller, R. M., Lindstrom, G. and Patil, S. "A loosely-coupled applicative multiprocessing system". *Proc. 1979 AFIPS NCC*.

[Kranz86] Kranz, D. et al., "Orbit: An Optimizing Compiler for Scheme", *Proc. SIGPLAN '86 Symp. on Compiler Construction*, June 1986, pp. 219-233.

[MaMi84] Mago, G. and Middleton, D. "The FFP Machine - A Progress Report". *Proceedings of the International Workshop on High-Level Language Computer Architecture*. May 1984, pp. 5.13-5.25

[McGr84] McGraw, J. R., et al. "SISAL: streams and iteration in a single-assignment language". *Language Reference Manual, Ver.1.2*, Lawrence Livermore National Laboratory M-146, 1984.

[PCSH87] Peyton Jones, S. L., Clack,, C., Salkild, J., and Hardie, M. "GRIP - A high-performance architecture for parallel graph reduction". *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*. Springer-Verlag LNCS 274, pp. 98-112.

[Sar89] Sarkar, V. "Partitioning and Scheduling Parallel Programs for Multiprocessorrs". *Research Monographs in Parallel and Distributed Computing,* Pitmand, London and The MIT Press, Cambridge, Massachusetts, 1989.

[SkGl83] Skedzielewski, S., and Glauert, J. "IF1 - An intermediate form for applicative languages". Draft 9, 1983.

[SKL88] Swanson, M. R., Kessler, R. R., and Lindstrom, G. "An Implementation of Portable Standard Lisp on the BBN Butterfly". *1988 ACM Symp. on Lisp and Functional Programming*, Snowbird, Utah, July 1988, pp. 132-142.

[Tra91] Traub, K. R. "Multi-thread Code Generation for Dataflow Architectures from Non-Strict Programs". *Lecture Notes in Computer Science 523*, Springer-Verlag, 1991.

[WaSl81] Warren, F. and Sleep, M. R. "Executing functional programs o a virtual tree of processors". *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 187-194. New Hampshire, October 1981.