THE DUTCH PARALLEL REDUCTION MACHINE PROJECT*

H.P. BARENDREGT, M.C.J.D. VAN EEKELEN AND M.J. PLASMEIJER,

Computer Science Department
University of Nijmegen
Nijmegen, The Netherlands

P.H. HARTEL, L.O. HERTZBERGER AND W.G. VREE,

Computer Science Department
University of Amsterdam
Amsterdam, The Netherlands

In November 1984 three research groups at the universities of Amsterdam, Nijmegen and Utrecht started a cooperative project sponsored by the Dutch Ministry of Science and Education (Science Council). The first phase lasting until the end of 1987 is a pilot study and has as aim to answer the following question. Is it *possible and realistic* to construct an efficient parallel reduction machine? The present paper gives an outline of the problems concerning parallel reduction machines and of our research towards their solutions.

1. Introduction

The Dutch Parallel Reduction Machine Group investigates the feasibility of building a parallel reduction machine suited for the efficient execution of programs written in a functional language. An (abstract) machine which performs reductions is called a *reducer*. The principle of reduction is simple, so there is no doubt that reducers can be built, in software and hardware, sequential and parallel. The *desirability* to do so depends on the advantages of functional languages as a new programming tool, on the possibility to implement them efficiently and to what cost. With these factors in mind, the Dutch PRM-group has concentrated its research mainly on the basic problems of efficient implementation.

First we introduce functional programming languages, discussing advantages, disadvantages and implementation issues. Then we address the important topic of the underlying reduction model. Furthermore we discuss the sequential and parallel implementation of the model we have chosen and the architecture of the experimental parallel reduction machine.

2. Functional Programming Languages

In 1936 two computational models were introduced, one by A. Turing and one by A. Church.

* This work is sponsored by the Dutch ministry of Science and Education, dienst wetenschapsbeleid

Turing described a class of machines (later to be called *Turing* machines). He defined the set of computable functions as those that are implementable on his machines. Based on the concept of a Turing machine are the present day *Von Neumann* computers. *Imperative* programming languages such as FORTRAN, PASCAL etc. as well as all the assembler languages are based on the way a Turing machine is instructed: by a sequence of statements that modify the internal state of the machine.

Church on the other hand invented a formal system called *lambda calculus* and defined the notion of computable function via this system. He did not give indications for implementing his system. *Functional* programming languages, like MIRANDA[1] , HOPE etc., are related to the lambda calculus. An early (although somewhat hybrid) example of such a language is LISP. *Reduction* machines are specifically optimised for the execution of these functional languages.

Also in 1936 Turing proved that both computational models are equally strong in the sense that they define the same class of computable functions.

Rather than giving a precise definition of what lambda calculus or a functional programming language is, we give some examples of functional programs. For a more formal treatment see [BAR86].

A *functional program* consists of an expression E representing both the input and algorithm (this is according to a reduction model similar to the lambda calculus). This expression E is subject to some fixed rewrite rules. *Reduction* consists of replacing a part P of E by another expression P' according to the given rewrite rules. Such an expression P is called a *redex* i.e. a *red*ucible *ex*pression. In schematic notation: E[P] --> E[P'], provided that P --> P' is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called *normal form* E* of the expression E will be the output of the given functional program.

```
Example.    (7 + 4) * (8 + 5 * 3)  -->  11 * (8 + 5 * 3)
                                   -->  11 * (8 + 15)
                                   -->  11 * 23
                                   -->  253.
```

In this example the reduction rules consist of the "tables" of addition and of multiplication on the numerals. Note that the "meaning" of an expression is preserved after reduction: 7 + 4 and 11 have the same interpretation. This feature of the evaluation of functional programs is called *referential transparency*. Also symbolic computations can be done by reduction.

Reduction systems for functional languages usually satisfy the *Church-Rosser* property, which implies that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example may be reduced also as follows:

```
(7 + 4) * (8 + 5 * 3)  -->  (7 + 4) * (8 + 15)
                       -->  11 * (8 + 15)
                       -->  11 * 23
                       -->  253.
```

Or even by evaluating several expressions at the same time:

```
(7 + 4) * (8 + 5 * 3)  ⇒   11 * (8 + 15)
                       -->  11 * 23
                       -->  253.
```

This gives the possibility of parallel execution.

---

[1] MIRANDA is a trademark of Research Software Limited

2.1. Advantages and Disadvantages of Functional Languages

The most important advantages of functional languages are the following.

- Some algorithms can be described easier in functional languages than in imperative languages. This is caused by the high expressive power of functional languages. For example one may use infinite data structures or yield a function as result of a function.
- The referential transparency makes it possible to use natural mathematical proof methods for verifying a functional program. It also makes program transformations easier: a programmer can start with a straight forward solution of a problem and via transformations convert this solution to a more efficient one. Therefore it is less difficult to develop reliable software.
- Due to the Church-Rosser property, alternative evaluation orders, such as parallel evaluation, will never produce a wrong result, although special care has to be taken in order to avoid non-termination.

However, functional languages also have some disadvantages.

- Functional languages only describe algorithms that transform data into other data. Several applications have interactive aspects and depend on internal states. Such applications like process control, real time programming, data base managers and operating systems can only be implemented on a reduction machine with appropriate interfaces. (On a Von Neumann computer these interfaces are also needed. But for these machines they can be programmed in the same imperative way as the machine itself.)
- Until recently, functional programs ran much slower than equivalent imperative programs and therefore they could not be used as a general programming tool.

In application areas where there is only interaction with the environment but little or no algorithmic transformations, a reduction machine is indeed not better than a conventional machine. However as soon as complex programs are needed, the advantages of functional programming will pay. Practical experience [KOO87a] confirms that functional languages are a good tool to develop reliable software. Current research [BOU85] shows that even in unexpected areas (such as specifying digital and analog circuits) functional languages form a powerful alternative.

2.2. Implementing functional languages on sequential machines

The descriptive power of functional languages is not present in the well-known imperative languages. One of the reasons is that this power is not easy to implement. In particular, problems are caused by:

- the lazy evaluation (call-by-name) which makes among other things infinite data structures possible;
- the possibility to yield a function as result.

Although the use of recursion instead of program loops is elegant, it does not help to increase the execution speed. As a result, a couple of years ago, functional programs ran that slow that no one would program a serious application in a functional programming language, despite of the advantages of these languages. Refining the set of compilation techniques also used for imperative languages, implementers of functional languages generally have learned how to reduce the additional overhead. In general this is done by:

- removing tail recursion where possible, thus transforming function calls to loops;
- using graphs to "share" an argument called by name such that it is evaluated at most once [WAD71, TUR79];

- using stacks where possible (for example with purely arithmetical expressions) instead of creating objects (graph elements) on the heap [JOH84].

Using the techniques described above code for a sequential machine can be generated, of which the efficiency is comparable to the efficiency of code produced for imperative languages.

The following techniques are still being developed:

- efficient heap management (clever garbage collection; destructive update [HUD86b]). Statistical properties of graph reduction concerning heap management are reported in [HAR87b].
- using strictness analysis [MYC81] to detect those arguments in which a function is strict, i.e. predict which arguments of a function can be evaluated before the function is called in order to speed up performance;

Although most of these techniques are well applicable in the general case, it takes a lot of effort to obtain the full gain of them in special cases. Strictness analysis is not fully worked out yet, theoretically and current work does not always include algorithms which are applicable in practice [REN86,87].

New sequential machine architectures have been designed specially geared to functional languages [CLA80, RICH85, KLU79]. As yet, there has been no significant progress in efficiency from this field of research. Until now, by the time a prototype of a special purpose reduction machine was finished, its speed was surpassed by improved implementations on general purpose machines. The NORMA machine design [RICH85] is based on Turner's combinators [TUR79]. The complete design itself has been done using a functional language. However, its performance is poor and show that these combinators do not form the right reduction model.

3. Reduction model.

History has shown that the design of a parallel machine architecture is rather difficult. This holds in particular for a parallel *reduction* machine. One of the problems is caused by the circumstance that some of the optimisations which are needed for an efficient sequential implementation, such as graphs, introduce inefficiencies in a parallel environment. Both theoretical and practical understanding of reduction is necessary to solve this inconsistency. For this reason a reduction model is necessary that contains the essential concepts of functional languages as well as the essential concepts of their implementation. Furthermore we need a language based on this reduction model which we will use as an intermediary between functional languages and reducers. Then it will be possible to reason about differences between languages and their implementations, to prove correctness and to port declarative programs to different (parallel) machines.

We recognize that parallelism is a difficult problem. In our project we have chosen to exploit coarse grain parallelism via the following of steps. The first one is to construct an efficient sequential reducer (see section 3.1 and 3.2). The next one is to use annotations in order to indicate parallelism (see section 4).

We have seen that functional languages have very little in common with the familiar Turing machine model of computation. Some implementers use directly the λ-calculus as the reduction model for these languages [PEY87]. However, if one wants to have the reduction model also close to the implementations, λ-calculus is not a good choice for the following reasons:

- Most implementations are not really based on λ-calculus but on combinatory logic [TUR79, JOH84, COU85].
- As discussed above, graphs are an essential part in any implementation.

- Patterns in functional languages contain essential information. By translating them to conditionals we might loose some of the information which is essential for the implementation (strictness analysis [NOC87]).
- Functional languages are still being further developed. Several researchers investigate how to incorporate language concepts that have no natural place in λ-calculus such as multi-programming, non-determinism and unification [HUD86a , DEG86].

We have developed two reduction models: Clean [BRU87] and an extension Lean. Clean is an intermediate language between functional languages and their implementations that incorporates all the aspects mentioned. It is an extension of Term Rewriting Systems [O'DO85, KLO85] to Term Graph Rewriting Systems (TGRS). This has the advantage that a lot of theoretical properties from the TRS world are inherited and provide a sound foundation for a TGRS theory. A formal description of the basis and theoretical properties of the reduction model can be found in [BAR87a]. For instance, in [BAR87a] it is proven that all hyper-normalizing reduction (evaluation) strategies in the TRS world, a class to which all well-known normalizing strategies belong, are also normalizing in the TGRS world.

Together with the research group of the University of East-Anglia (UK) which is taking part in the British Flagship project, we are working on a more general language Lean [BAR87b] based on Graph Rewriting Systems (GRS). We hope that this will lead to a more general intermediate language between all kinds of *declarative* languages (such as PROLOG) and (parallel) machines. Preliminary versions of such a language are implemented [GLA87, JAN87]. The research shows that GRS's are a powerful mechanism. We expect that GRS's will be able to incorporate also possible future developments in functional languages such as non-determinism and unification.

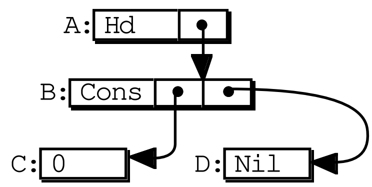
3.1. Clean, a functional language based on graph rewriting systems

The object that is manipulated in Clean is a connected, possibly cyclic, directed graph called the *program graph*. Each node in the graph has an unique *identifier* associated with it (corresponding to a machine address) and it contains a *symbol* and a possibly empty sequence of identifiers which define directed arcs to nodes in the graph. Programming with pictures is rather inconvenient so we have chosen for a linear notation for graphs. In the most extensive form of this notation (the canonical form) graphs are represented by giving the list of the nodes out of which the graph is built.

Clean canonical notation

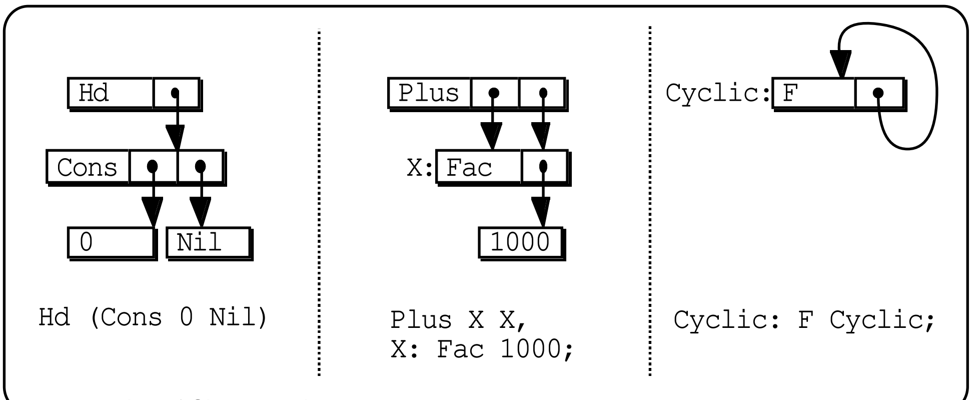
```
A: (Hd B),
B: (Cons C D),
C: (0),
D: (Nil);
```

Graphical equivalent



graph example

In order to obtain a more readable form we may substitute the contents of a node for a reference to this node and we only explicitly notate the identifiers of nodes if we need them to express sharing. Brackets are left out if they are redundant.



examples of Clean graphs

A *Clean program* consists of a set of *rewrite rules* and an initial program graph indicated by Start.Each rewrite rule specifies a possible transformation of the program graph. For instance:

Hd (Cons a b)	-> a	;
Fac 0	-> 1	|
Fac n	-> *I n (Fac (-I n 1))	;
F (F x)	-> x	;
Start	-> Hd (Cons (Fac 1000) Nil)	;

A Clean graph is rewritten to its normal form by a sequence of applications of individual rewrite rules. For a rule to be included in the sequence, there must be a match between the graph (*redex pattern*) specified at the left-hand-side of the rule and some subgraph (*redex*) of the program graph. A redex pattern matches a redex if both graphs have an isomorphic structure and contain the same constant symbols. If a particular rule is applied to a matching redex, the graph is rewritten according to the right-hand-side of that rule. In general, this right-hand-side also consists of a graph (*contractum pattern*). An instantiation of this pattern (the *contractum*) is constructed and a new program graph is finally constructed by taking all arcs pointing to the root node of the redex and redirecting them to the root node of the contractum. This has the effect of "overwriting" the root of the redex with the root of the contractum.

In general there will be several possible redexes in the graph. It may even be the case that one and the same redex can be reduced according to more than one rule; a situation that is called ambiguity. An algorithm which repeatedly rewrites the graph making choices out of the available redexes and out of all the possible matches of those redexes is called a reduction *strategy*. Given a set of rules (including a start rule), an initial graph and a reduction strategy we have a system with a dynamic behaviour, a reducer. Although it is sometimes only implicitly defined, every implementation of a rewriting system must rewrite according to a given strategy. In principle a Clean program is reduced with a strategy which is called the *functional* strategy, because it resembles very much the way in which reducing is performed normally in lazy functional languages. A formal description can be found in [GOO87]. Alternative strategies may also be used. Via annotations one can give compiler directives to indicate strict arguments or to indicate subgraphs that may be reduced in parallel (jobs for instance, see also section 4.2).

3.2. Using clean to implement functional languages

In [KOO87b] it is shown that functional languages like SASL [TUR79], Miranda [TUR85], OBJ2 [FUT85] and Tale [BAR86] can easily be compiled to Clean code. There are several ways of doing this. For instance, code schemes used by Turner and Hughes [TUR79, HUG82] can be expressed directly in Clean. To demonstrate that compilation is relatively easy we are developing Miranda [TUR85] compilers (one written in Modula2, one written in Miranda) which generate Clean code.

A sequential Clean compiler is developed on a VAX/750 running UNIX BSD 4.2 [BRU87]. We incorporated some of the general implementation techniques mentioned in section 2. Tail recursion is removed, stacks are used where possible. Sharing is controlled by the code generated by the Miranda compiler. Presently we work on two classes of optimisations:

1. those that are understood and are ready to be implemented (improving the use of stacks, pattern matching and heap management);

2. those that require research before they can be implemented (like strictness analysis; we think that Clean is more appropriate to do this than pure λ-calculus [NOC87]).

When the first group of optimisations are implemented we will get a state-of-the-art Clean compiler which will be ported to other sequential architectures.

3.3. Related work

Various Lisp-compilers have reasonable performance, although in general they run not so fast as the current implementations such as Clean. Moreover, Lisp does not support infinite data-structures nor higher order functions.

The Categorical Abstract Machine [COU85] which is based on categorical combinators, is very fast for eager evaluation, but it only supports lazy evaluation as an exception. The same holds for the Hope [BUR80] compiler which uses the special purpose language FP/M [BAI84] as an intermediate language.

The Swedish lazy-ml compiler [JOH84] is on one hand based on a low-level graph manipulation language (the G-machine [KIE85]) and on the other hand it is based on combinatory logic. The performance of the code generated by this compiler is good.

4. Parallel implementation of reduction

Clean has been designed in such a way that it can be fitted by annotation to parallel architectures. This means that the programmer has to mark expressions that can be evaluated in parallel. The annotation may include information about the distribution of the expressions over a network of processing elements.

Parallel implementations of functional languages can be divided into *fine* grain and *coarse* grain. Fine grain parallelism requires only strictness analysis to determine needed redexes, which will constitute the grains. Coarse grain parallelism requires the determination of the grain size of an expression. This size is a measure for the cost of evaluating the expression. Several approximations of the grain size of an expression have been proposed [HUD85, HUG82, KEL79]. However, this function is non computable and to our knowledge there exists no satisfactory heuristic. Because the hardware that is necessary for a fine grain parallel machine is very complex [MAG79], we have chosen for a coarse grain approach. The programmer has to design process- and data-structures in such a way that they can be annotated to match the coarse grain architecture [VRE87].The grain annotation is covered in the next paragraphs.

Clean is being implemented on the Distributed Object Oriented Machine [ODIJ85] developed at the Philips Laboratories, the Netherlands, and on our Experimental Parallel Reduction Machine.

4.1 Local memory architecture

The basic model of the kind of parallel machine architecture that we will investigate is a collection of reduction processors, each equipped with its own local memory, interconnected by a (fast) data communication network. A fundamental property of this model is that the access time of a processor to its own memory is much shorter than the access time to the other memories. These two access times may differ by one to two orders of magnitude. To account for this property we have decided not to support a global address space i.e. when a subgraph happens to reside in a particular local memory, the nodes of this subgraph can only point to nodes within the same local memory. A separate mechanism is provided to transport a subgraph from one local memory to another.

An experimental parallel reduction machine has been constructed, based on a local memory architecture [HAR86]. The main objective of this effort is to collect statistical data of the communication cost of job based subgraphs (see section 4.2). These measurements can only be efficiently performed on dedicated hardware. The reduction processors have been realized with commercial shared-bus microprocessor systems (M68000 with VME-bus). Communication of graphs has been implemented with the aid of fast parallel bus-interconnections between dual-ported memories (VME to VMX). To exploit the full advantage of these fast connections, the construction of special graph collection processors is envisaged. At present, experiments are being conducted (graph collection in software) to obtain preliminary performance figures of the proposed parallel reduction model on the experimental machine. The combination of fast sequential reduction (Clean) and fast hardware supported communication of coarse grain jobs will in our opinion enable efficient parallel reduction.

4.2. Job based parallel reduction

The architectural features discussed above limit the kind of expressions that can be efficiently reduced in parallel. Only certain coarse grain parts of the program that we call "jobs" are allowed to be copied to another processor for parallel evaluation. A job is an expression with the following properties:

a) it is *self-contained* (i.e. a subgraph containing no references to other points of the graph);

b) its evaluation is needed to compute the final result;

c) the cost to evaluate the expression outweighs the cost involved in transportation.

Job property c) guarantees that parallel execution of a set of jobs will be faster than their sequential execution. Property b) makes sure that the result of a job is essentially used in the whole computation and so no actual processing will be wasted. Finally property a) allows jobs to be evaluated in a separate address space and avoids the need for global garbage collection.

The restriction of parallel reduction to jobs defines a minimum granularity on which the data communication of the architecture can be based. Small quantities of data (like a single node or a redex containing only pointers to its arguments [WAT84]) will never be transported. Overhead incurred by transmission protocols can be spread over the cost of transporting a whole subgraph.

A disadvantage of job based parallel reduction on a local memory architecture is that sharing of expressions cannot be exploited globally across jobs. Within a job all sharing can be maintained. Summarizing, parallel job reduction employs copying reduction on the global job level and sharing reduction within jobs.

4.3. Sandwich reduction strategy

To avoid the major disadvantage of copying (duplication of work), a special reduction strategy has been devised on the job level. This strategy guarantees that a job will be a primary redex when it is transported to another processor. Thus, before copying takes place, a job contains no secondary redexes and hence no work can be duplicated while shipping the job.

In practice the sandwich strategy has been implemented by a single special annotation given by the programmer:

$$\text{sandwich } G \ job_1 \ job_2 \ . \ . \ . \ job_n$$

where $job_i = F_i \ a_1 \ a_2 \ . \ . \ . \ a_m$.

The sandwich construct is the only means in the language to create jobs. An expression is sequentially reduced to normal form until a sandwich expression is needed. The reduction of (G job_1 job_2 ... job_n) is then suspended until the parallel evaluations of job_1 job_2 . . . job_n have been completed. However, before the jobs are submitted for parallel evaluation, all arguments a_1 a_2 . . . a_m of each job_i are sequentially reduced to normal form, rendering job_i a primary redex. Now copying the normal forms of the a_i, in order to ship the job, cannot result in extra work. If a_1 a_2 . . . a_m would have been reduced in parallel, then any redex shared between the a_i would have been copied. This would result in the duplication of work. The strategy has been called "sandwich strategy" because it contains one level of eager evaluation between two levels of lazy evaluation.

We have constructed a simulation of a parallel reducer [LEE85,HAR87c]. A number of divide-and-conquer algorithms, such as the fast Fourier transform, have been programmed to make effective use of the sandwich strategy.

If the volume of data processed by an algorithm is characterised by a number "n" then the algorithms that have a computational complexity > O(n), can be made to benefit significantly from parallel evaluation with the sandwich strategy.

The speed-up that may be achieved for a particular value of "n" depends on some architectural properties, in particular the ratio between the sequential execution speed of a processing element and the transfer speed of jobs and results. For a sufficiently large value of "n", a near linear speed-up results as more processing elements are used. In this case, the speed-up is only dependent on the number of processing elements.

For the algorithms that have a computational complexity $\leq O(n)$, the speed-up that may be achieved is also dependent on the ratio defined above.

The speed-up for divide-and-conquer algorithms is also valid for other architectures with local memory [MCB87, KLU83].

Experiments on our experimental parallel reduction machine have been conducted in order to measure the architecture constant and to study its effect on divide-and-conquer algorithms [HAR87a].

4.4. Related work

Most proposals for parallel reduction are either based on pure copying reduction [KLU83, MAG79, TRE82], or on pure sharing reduction [DAR81, HUD85, KEL79, PEY86, WAT84]. To our knowledge mixed reduction based on both copying and sharing has not been proposed.

Two contemporary and similar research projects are the Flagship machine [WAT84] and the Grip project [PEY86]. In contrast to our proposal both machines feature a global address space. Like the Alice machine [DAR81] the Flagship machine is a packet rewrite machine and will have rather fine grain communications. The Grip machine is based on super combinator reduction and will use conservative parallel strategies. To compensate for the lack of a global address space we spent much effort in the optimisation of job based graph-communication. These projects are in a phase too early to make comparison of results possible.

5. Future research

In cooperation with the research group of the University of East-Anglia the reduction model will be investigated further in order to include modularization, general type system, unification, general I/O etc. Also we will investigate the possibility for the user to define his own special (parallel) strategy for his program, for instance via high level specification of (parallel) reduction strategies and a formalism for mixing several strategy schemes during evaluation [EEK86]. This research must make the language Lean [BAR87b] based on this GRS a good intermediate language between declarative languages and (parallel) machine architectures. All this must be accomplished without loosing the basic elegance, the practical usability and the theoretical framework of the model. In the near future we will improve the practical applicability of Clean and Lean by increasing the efficiency and by adding separate compilation, typing and strictness analysis.

The efficiency of the experimental parallel reduction machine can be increased by the construction of special hardware components for specific tasks such as graph compaction and packet transport and we will try to extend the class of problems for which the sandwich strategy is applicable. Ideas concerning load distribution are presently being tested by experiments on the experimental parallel machine.

6. Conclusions

- We have designed a reduction model, a Graph Rewriting System (GRS) which includes the most relevant aspects of functional languages and the way they are implemented. With this model we can derive *theoretical* properties which are valid for parallel reducers.
- We have developed the experimental language Clean based on this GRS. It is used as an intermediate language between functional languages and (parallel) machine architectures. With Clean *practical* properties of reducers can be examined and compared. Functional languages can be translated in a natural way into Clean.
- The efficiency of the Clean implementation is good (comparable to PASCAL).
- For divide-and-conquer algorithms it is possible to obtain significant increase of performance on a parallel reduction machine. Reduction is not essential for this speed-up, but it makes parallel programming much easier.
- The sandwich annotation is a good way to implement parallel reduction on a local memory architecture.

Finally we give a status quo answer to the question stated in the abstract :

It is possible and realistic to construct an efficient sequential reduction machine on existing sequential hardware. For a particular class of algorithms it is possible and realistic to implement a parallel reduction machine on existing parallel architectures (direct- and shared-connection machines). Further speed-up can be achieved by introducing special hardware components. This may eventually lead to the construction of a special purpose reduction machine.

7. Acknowledgements

The following people of the Dutch parallel reduction machine project have made valuable contributions to the work presented in this paper: Jan Bonneveld, Raymond Boute, Tom Brus, Maarten van Leer, Marc van Leeuwen, Hans Mulder, Betsy Pepels, Gerard Renardel de Lavalette, Arthur Veen and Zhao Jing-Wei. We are grateful for the collaboration with the University of East-Anglia.

8. References.

[BAI84] Bailey, R., "FP/M Abstract Syntax Description", Internal Document, Imperial College London, 1984.

[BAR86] Barendregt, H.P., Leeuwen, M. van, "Functional Programming and the Language Tale", (Eds. J.W. de Bakker, W.-P. de Roever and G. Rozenberg), Springer LNCS 224, pp. 122 - 207, 1986.

[BAR87a] Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R., "Term Graph Reduction", University of East-Anglia and University of Nijmegen, Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, the Netherlands, LNCS vol 259, pp. 141-158, June 1987.

[BAR87b] Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R., "Towards an Intermediate Language based on Graph Rewriting", University of East-Anglia and University of Nijmegen, Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, the Netherlands, LNCS vol 259, pp. 159-175, June 1987.

[BOU85] Boute, R.T., "Functional description of digital systems", in: IFIP WG 10.1 Working conference on Methodologies for Computer System Design, North-Holland, Amsterdam, 451-467.

[BRU87] Brus, T.H., Eekelen, M.C.J.D. van, Leer, M.O. van, Plasmeijer, M.J., "Clean, A Language for Functional Graph Rewriting", University of Nijmegen, Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland , Oregon, USA, LNCS 274, pp. 364-384, September 1987.

[BUR80] Burstall, R.M., MacQueen, D.B., and Sanella, D.T., "Hope: An Experimental Applicative Language", Proceedings of the 1980 LISP Conference, pp. 136-143, Stanford, California, August 1980.

[CLA80] Clarke, T.J.W., Gladstone, P.J.S., MacLean, C.D., and Norman, A.C., "SKIM - the S,K,I Reduction Machine", Proceedings of the 1980 ACM Lisp Conference, pp. 128-135, August 1980.

[COU85] Cousineau, G., Curien, P.L., Mauny, M., "The Categorical Abstract Machine", Proc. Conf. on Functional Languages and Computer Architecture, Nancy, pp. 50 - 64, September 1985.

[DAR81] Darlington, J., Reeve, M., "ALICE: A multiple-processor reduction machine for the parallel evaluation of applicative languages", ACM Conference on functional programming, languages and computer architecture, pp 65-76, New Hampshire, Oct. 1981.

[DEG86] De Groot, D. & Lindstrom G. (eds), "Logic Programming: Functions, Relations and Equations", Prentice Hall 1986.

[EEK86] Eekelen, M.C.J.D. van, Plasmeijer, M.J., "Specification of rewriting strategies in Term Rewriting Systems", University of Nijmegen, Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, LNCS vol. 279, October 1986.

[FUT85] Futatsugi, K., Goguen, J., Jouannaud, J-P. and Meseguer, J., "Principles of OBJ2", Proceedings of 12th ACM Symposium on Principles of Programming Languages (POPL '85), pp. 52-66, ACM, 1985.

[GLA87] Glauert, J.R.W., Kennaway, J.R., Sleep, M.R., "DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction", ICL Technical Journal, May 1987.

[GOO87] Goos, J., Van Latum, F., "The use of an abstract-interpretation TRS in specifying and proving reduction strategies", Master Thesis, University of Nijmegen, March 1987.

[HAR86] Hartel, P.H., Vree, W.G., "A Load Distribution Network for a Multi Processor Reduction Machine", Internal Report D-6, Dutch Parallel Reduction Machine project, University of Amsterdam, April 1986.

[HAR87a] Hartel, P.H., Vree, W.G., "Parallel graph reduction for divide-and-conquer applications", Internal Report D-15, Dutch Parallel Reduction machine project, University of Amsterdam, Feb 1987.

[HAR87b] Hartel, P.H., Veen, A.H., "Statistics on graph reduction of SASL programs, March 1987, to appear in Software Practice and Experience.

[HAR87c] Hartel, P.H., "Experimental Parallel Reduction Machine User Manual", Internal Report D-14, Dutch Parallel Reduction Machine Project, University of Amsterdam, Oct 1987.

[HUD85] Hudak, P., Goldberg, B., "Distributed execution of functional programs using Serial combinators", IEEE Transactions on computers, Vol. C-34, 10, pp 881-891, Oct. 1985.

[HUD86a] Hudak, P., and Smith, L., "Para-Functional Programming: A Paradigm for Programming Multi-Processor Systems", 12th A.C.M. Symp. on Principles of Programming Languages, pp. 243 - 254, Jan. 1986.

[HUD86b] Hudak, P., "Arrays, Non-Determinism, Side-effects and Parallelism: A Functional Perspective", proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, LNCS vol. 279, October 1986.

[HUG82] Hughes, J., "Graph reduction with super-combinators", rep PRG-28, Oxford University, June 1982.

[JAN87] Jansen, T., "Interpreting Lean", M. Sc. thesis, University of Nijmegen, May 1987.

[JOH84] Johnsson, T., "Efficient Compilation of Lazy Evaluation", Proc. of the ACM Sigplan '84, Sigplan Notices, Vol. 19, No 6, June 1984.

[KEL79] Keller, R.M., Lindstrom, G., Patil, S., "A loosely-coupled applicative multi-processing system", Proc. Nat. Comp. Con., Vol. 48, pp 613-622, 1979.

[KIE85] Kieburtz, R.B., "The G-machine: A fast, graph-reduction evaluator", Proc. Conf. on Functional Languages and Computer Architecture, Nancy, pp. 1 - 16, September 1985

[KLO85] Klop, J.W., "Term rewriting systems", to appear in the Handbook of Logic in Computer Science, editors S. Abramsky, D.M. Gabbai, T.S.E. Maibaum.

[KLU79] Kluge, W.E., "The Architecture of a Reduction Language Machine Hardware Model", GMD, Schloß Birlinghoven, D-5205 St. Augustin 1, Interner Bericht ISF-79-3, 1979.

[KLU83] Kluge, W.E., "Cooperating reduction machines", IEEE Transactions on computers, Vol. C-32, 11, pp 1002-1012, Nov. 1983.

[KOO87a] Koopman, P.W.M., "Interactive Programs in a Functional Language: A Functional Implementation of an Editor", Software Practice and Experience,Vol 17(9), pp 609-622.

[KOO87b] Koopman, P.W.M., Nocker, E.G.J.M.H., "Compiling Functional Languages to Functional Graph Rewriting Systems", University of Nijmegen, to appear.

[MAG79] Mago, G.A., "A network of microprocessors to execute reduction languages", Part I and II. International journal of computer and information sciences, Vol. 8, 5, pp 349-385 and 6, pp 435-471. 1979.

[LEE85] Lee, Th.G.A. van der, Oolman, H., "Tracing and debugging in SASL", Report 66, Department of Informatics, Nijmegen University, April 1985

[MCB87] McBurney, D.L. and Sleep, M.R., "Transputer-based experiments with the ZAPP architecture", Proceedings of Parallel Architectures and Languages Europe (PARLE), part I, Eindhoven, the Netherlands, LNCS vol 258, pp. 242-259, June 1987.

[MYC81] Mycroft, A., "Abstract interpretation and optimising transformations for applicative programs", PhD. thesis, Univ. of Edinburgh, 1981.

[NOC87] Nocker, E.G.J.M.H., "Strictness Analysis for Functional Graph Rewriting Systems", University of Nijmegen, to appear.

[O'DO85] O'Donnell, M.J., "Equational Logic as a Programming Language", Foundations of Computing Series, MIT Press, 1985.

[ODIJ85] Odijk, E.A.M., "DOOM: a Decentralized Object-Oriented Machine", Doc. Nr. 0125, Esprit 415 internal report, Philips, Eindhoven, 1985.

[PEY86] Peyton Jones, S.L., "Using future bus in a fifth generation computer", Microprocessor and microsystems, Vol. 10, 2, pp 69-76, March 1986.

[PEY87] Peyton Jones, S. L., "FLIC - a Functional Language Intermediate Code", Dept. of Comp. Sc., University College London, internal working paper.

[REN86] Renardel de Lavalette, G., "Strictheidsanalyse", Department of Philosophy, University of Utrecht, Logic Group Preprint Series, no. 10, May 1986.

[REN87] Renardel de Lavalette, G., "Strictness analysis for a language with polymorphic and recursive types", University of Utrecht, to appear.

[RICH85] Richards, H., "An Overview of the Burroughs NORMA", Proceedings of the Workshop on Implementation of Functional Languages, Programming Methodology Group University of Göteborg and Chalmers University of Technology, Report 17, pp. 429-438, February 1985.

[TRE82] Treleaven, P.M., Hopkins, R., "A recursive computer for VLSI", Proceedings of the 9-th international IEEE/ACM symposium on computer architecture, special issue of Computer architecture news, pp 229-238, Austin, Texas, 1982.

[TUR79] Turner, D.A., "A new Implementation Technique for Applicative Languages", Softw.

Pract. and Experience, Vol 9 (1), pp. 31 - 49, January 1979.

[TUR85] Turner, D.A., "Miranda: A non-strict functional language with polymorphic types", Proc. Conf. on Functional Languages and Computer Architecture, Nancy, pp. 1 - 16, September 1985.

[VRE87] Vree, W.G., "The grain size of parallel computations in a functional program", Proc. Int. Conf. on Parallel Processing and Applications, L'Aquila, Italy, September 1987.

[WAD71] Wadsworth, C.P., "Semantics and Pragmatics of the Lambda-calculus", Ph. D. thesis, Oxford University, 1971.

[WAT84] Watson, I., "A packet based data-driven rewrite-rule machine", Department of computer science, University of Manchester.