- *Formal representation of the system design.* It provides consistency throughout the system and increases flexibility. The dual representation form, graphical for humans and declarative for automatic processing, and the automatic transformation between them are extremely useful.

- *Hierarchical modeling.* Hierarchical decomposition allows large system models to be managed. It also supports reuse: models of processors, boards, etc., are used as many times as required.

- *Automatic model analysis.* Model interpretation transforms the system models into directly useful forms. The key features of the interpreter are

  - *Intelligent diagnostics support.* The program is able to locate the most common errors, such as link swaps, missing processors, and missing links, identifying the nodes by name and type.

  - *Message routing.* Message routing maps are automatically generated. Automatic and manual path definition, multiple paths, and partial map generation are supported.

- *Flexibility.* When new processors or boards are released only modeling them is needed. Special requirements can be satisfied with the model interpreter. This was the case with the TI320C40 cpu; APNA was modified to be able to reconfigure the network according to the connection restrictions.

**References**


[1] F.P. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer, April, 1987, pp. 10-18.

[2] D. Harel, Biting the Silver Bullet, IEEE Computer, January, 1992, pp. 8-20.

[3] B. Abbott, T. Bapty, C. Biegl, A. Ledeczi, G. Karsai and J. Sztipanovits, Experiences Using Model-Based Techniques for the Development of a Large Parallel Instrumentation System, Proc. of the International Conference on Signal Processing Applications and Technology, Boston, MA, 1992.

[4] B. Abbott, C. Biegl and J. Sztipanovits, Multigraph for the Transputer. In: A. Wagner (Ed.), Transputer Research and Application 3. ISBN: 90 5199 030 8. IOS Press, Amsterdam, 1990, pp. 211-222.

[5] M. R. Garey and D. S. Johnson, Computers and Intractability: a Guide to the Theory of NP-completeness, Freeman, 1979.

[6] D. Reed and R. Fujimoto, Multiprocessor Networks: Message-Based Parallel Processing, MIT Press, 1987.

[7] Express 3.0, User's Guide, Parasoft Corporation, 1990.

[8] Transputer Toolset, Vol. 1 Toolset Reference, Parallel C for the Transputer, Logical Systems, 1992.

[9] Logical Systems C for the Transputer, User Manual, Logical Systems, 1990.


*Transputer Research and Applications 6*
*S. Atkins and A.S. Wagner, Eds.*
*IOS Press, 1993*



# Autotransformation of evaluation network as a basis for automatic dynamic parallelizing

S.M.Abramov, A.I.Adamowitch, I.A.Nesterov,
S.P.Pimenov, Yu.V.Shevchuck

Research Centre for Multiprocessor Systems
Program Systems Institute, Pereslavl-Zalessky, Russia

Abstract

**The paper describes a computation model designed to organize parallel computing. The computation is represented as an autotransformation of an evaluation network consisting of processes and processed data. This model can be used as a basis for a programming system with automatic dynamic parallelizing of programs. It is supposed that the source language of the system should be similar in its syntax to the conventional languages C, Pascal or Fortran. The grains of parallelism are the functions of the source language. The programming system is intended for the parallel equipment based on Intel microprocessors, Inmos Transputers and the Texas Instruments parallel DSPs.**

## Introduction

The well-known approach to programming of parallel systems, i.e. multiprocessors, implies that the programmer divides the task into subtasks, which can be run in parallel, maps the subtasks onto processors and provides the synchronizing of simultaneously executed fragments. If these operations are delegated to a computer, we say that parallelizing is being done *automatically*.

Parallelizing can be done by a computer in compile-time by analyzing potential information dependencies in the source program. We believe, however, that *dynamic* parallelizing is even deeper, since it is done in run-time, when the information dependencies become visible.

The use of a functional language as the source language is the natural way to design a system with dynamic parallelizing. Unfortunately, traditional approach to the parallel implementation of functional languages, parallel graph reduction, has a serious drawback. The grains of parallelism in the graph reduction are combinators S,K,I,etc. of combinatory logic. These combinators are tiny reduction steps and system overheads nullify the advantages of the approach. There were made attempts to overcome this drawback by transforming the source program into a new set of function, called serial combinators [5].

In this paper we describe a similar computation model, that was from the very beginning intended for operations with large grains of parallelism. In this model the computation is represented as an autotransformation of an evaluation network consisting of processes and processed data. Opposite to common practice, general features of the source language of the system are derived from the model's characteristics.

In terms of the source language, the grains of parallelism are the functions. Although we regard the source language as strictly functional, the function bodies may be written in an imperative language. We hope that this feature will be attrac-

tive for programmers dealing with numerical analysis, who are usually repelled by syntax of functional languages.

## What is a multiprocessor

By a multiprocessor we mean a network of monoprocessor elements. The design and topology of the network are irrelevant. That does not mean, however, that the structure of the network does not affect the performance of the system. There is no doubt that a tightly-coupled network with high throughput is better than a weakly-coupled one with low throughput..

At a logical level the network is a set of point-to-point connections via which the monoprocessor elements are combined into a graph with arbitrary topology. If the monoprocessor elements are combined into a multiprocessor via a common bus or shared memory, the bus or the memory will be used for the simulation of point-to-point connections.

A monoprocessor element is a conventional processor equipped with its own memory and, optionally, with co-processors, accelerators and I/O devices. A personal computer is a good example of a monoprocessor element.

It is supposed that every monoprocessor element runs a copy of a control program that carries out the duties of an operating system and supports the autotransformation of the evaluation network.

## History

In 1966 V.F.Turchin designed Refal programming language [1,4], a functional language for symbol manipulation. Since then there exists in Russia the school of functional programming, which differs from the traditional ones based on the Lisp methodology. The basic mean of Refal is pattern matching. To represent expressions and data there are used bidirectional lists. In 1988 within the context of refinement of Refal and its implementations there were suggested the *vector representation of lists* and *bulldozer garbage collection* [3].

At the same time as a part of the supercompiler project [2] RL language was designed combining the syntax of Lisp and the data representation of Refal. Since supercompilation requires large computing resources, our group started to implement RL on one of Russian multiprocessors. This research resulted in the definition of the main principles of autotransformation of an evaluation network, which can form the basis for automatic dynamic parallelizing of programs.

## List-like structures

The evaluation network consists of list-like structures of a certain type. For the first time similar structures were used to represent the object expressions of Refal [3]. In this paper we consider them as an extension of the lists used in Lisp and allied languages.

To illustrate our discussion, let us take a data structure, consisting of the word "word" and attached information `info`. For example, `info` may be a dictionary article. The structure of the article is irrelevant. In Lisp notation (Fig.1a) this data structure is the list of five elements and the first one is a list in itself. The list contains (Fig.1b) five reference levels for the representation of the word and unknown number of reference levels for `info`. A reference level consists of two cells. A cell may store either a scalar value, a character in our case, or a reference to another



level. The list must be accessible, so there exists a cell, containing a reference to it. We call this cell a *holder* of the list. We bring your attention to the fact that our data structure contains only two logical levels, but even to represent the word we need five reference levels.

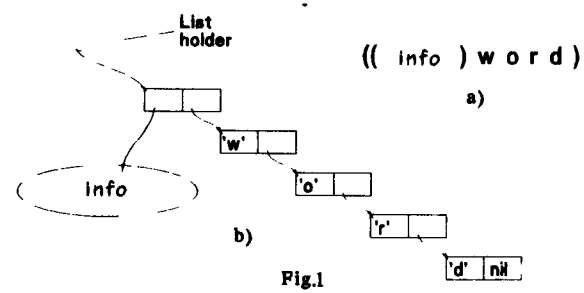


Fig.1

Such lists are possible in our computation model. But they should be considered as a special case of the list-like structures described below. List-like structures also consist of cells. A cell is an elementary memory unit used for list construction. The cell size is sufficient to store any scalar value or a holder of another list. Every cell is tagged with a descriptor of the stored data. The area of the memory that has cell structure is called *structured area.*

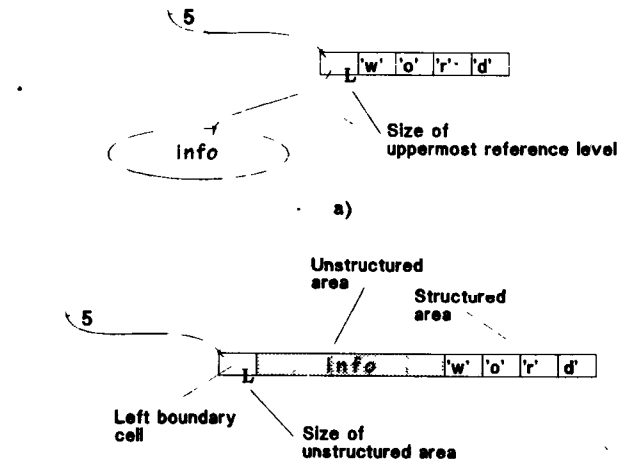


Fig.2

Let every reference level consist of an arbitrary number of cells. Thus there is no necessity to have additional reference levels for the representation of one logical level (Fig.2a). As some overheads we have to keep the size of the uppermost refer-

ence level in the holder of the list. As a result, the data of one and the same logical level are accessed from the list by indexing. Linear arrays of cells is a good way to store strings, arrays of numbers and even more complex heterogeneous structures. The reference level which consists of a structured area only is called an *indexed list.*

The `info` structure may contain data irrelevant to list manipulations, e.g. masks, bit fields, a large text, etc. The memory occupied by these data is not tagged and has no cell structure. These memory areas are called *unstructured areas.*

An unstructured area (Fig.2b) consists of a *left boundary cell* and an adjacent memory for the storage of arbitrary data. The left boundary cell keeps the size of the adjacent area, which must be multiple of the size of a cell. The unstructured area can be regarded as one big cell with a knapsack for data. The reference level containing unstructured areas is called *nonindexed list.*

We want to bring your attention to the fact that the word "list" has several meanings in the context of this article. By "list" we usually mean a separate reference level or a reference level with its holder (Fig.3a). But occasionally we use the word "list" either for a hierarchic list-like structure with several reference levels (Fig.3b) or for a single list holder (Fig.3c).

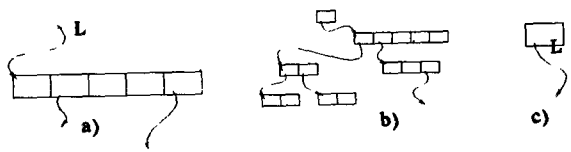


Fig.3

The list-like structure built of nonindexed lists is the most general form of representation of any objects in the described computation model.

The fact that the starting point of our discussion is a Lisp data structure does not mean, however, that we advertise programming in Lisp style. The list-like structure, introduced above, is a novel form of data representation, and it influences the style and methods of programming to a great extent. Programming with the use of nonindexed lists is closer to conventional programming with the use of arrays, structures and pointers, than to the programming in Lisp and allied languages.

## Processes

Specially organized list-like structures, containing an executable code, play an outstanding role among the objects of the evaluation network. These list-like structures are called *processes* and can be activated under certain conditions. Usually, there are many processes in an evaluation network. The processes are surrounded by lists of data, and the purpose of the processes' activity is to analyze and transform the data. While doing this, the processes interact, terminate, spawn new processes. Their activities result in transformation of the evaluation network. Since the processes are part and parcel of the network, we deal with *autotransformation.*

The discussion of the inner structure of the processes is beyond the scope of this paper. We just mention that the process has all the accessories for traditional von-Neumann execution, namely stack, code and data segments, stored in unstructured areas, etc. On the other hand, the process is an ordinary list-like structure that may be used as data for another process.



From the point of view of a process the world consists of the its own *inner world* and the *outer space.* Inside the process the computation is performed in von-Neumann manner, while in the outer space, where a shared pool of data and other processes are residing, the laws of the described model govern.

Not all sorts of activities are permitted for the processes in the evaluation net work. The behaviour of the processes should be restricted to prevent the damage of the evaluation network and to avoid the production of undesirable structures in it. Besides, the result of evaluation must be determinate, i.e. it must not depend on the order of processes execution. Correct constructing of the evaluation network and reasonable behaviour of processes are provided by the source language of the system.

## The TL language

There are many candidates to get the position of the source language, the majority of modern functional languages among them. A hypothetical programming language TL, described here, reflects all the notions and primitives of the described computation model most clearly. In spite of the fact that TL contains statements characteristic of high-level programming languages, it is a low-level language in respect to the described model. Strictly speaking, it is not even a functional language, its functionality and determinacy being the result of a set of restrictions.

A TL program consists of modules containing function definitions. The modules are linked by export-import relations. Common and global variables are absent, therefore the functions can interact only via parameters and returned values. A function can have an arbitrary number of parameters and an arbitrary number of returned values. As parameters and values there can be used characters and numbers as well as list holders, through which the function gets access to the shared pool of list-like structures in the outer space. Side effects of the functions are strictly prohibited. The function is not allowed to modify the lists accepted as parameters, but it may create its own ones, allocating the free memory and rewriting it.

We do not specify the manner in which the function forms its values. The function body may contain inner variables, loops, and other statements, characteristic of the conventional programming languages. In this respect TL isn't a concrete programming language, but rather a family of languages suitable for programming in the described model.

Function calls and function terminations, return of values and manipulations with some specific parameters are peculiar operations, since they result in transformation of the evaluation network. These operations are carried out by the control program. Calls to the control program are named *system calls.* In TL syntax they look like a library procedure calls or may be covered by special syntax constructions.

## Function Invocation

In the roster of the system calls, resulting in the transformation of the evaluation network, the call `spark` occupies the first place. This call organizes a combined invocation of several TL functions. We will regard its simplest form, an invocation of a single function.

`spark(...)`

To organize an invocation of a TL function with N parameters and M returned values, a process with respective number of *inputs* and *outputs* is created. Inputs and outputs are special cells within the process, which provide interface between the inner world of the process and the outer space. A process can have additional inputs for its own purposes. Since we are not discussing here the inner structure of a process, we draw it as a rectangle with the inputs a,b,... at the bottom and outputs x,y,... at the top (Fig.4).

The inputs of the process accept the parameters of the called function. According to TL semantics, the following entities can be fed to a process input:

a) a scalar value[1], e.g. character 'A';
b) a list holder[2];
c) a reference to an output of another process[3].

The outputs of the process are regarded as *producers* of the returned values. Outputs can be attached to different *consumers*, among which there can be:

a) an element of an outer list[4];
b) an input of another process[5].

One output can be attached to several consumers[6].

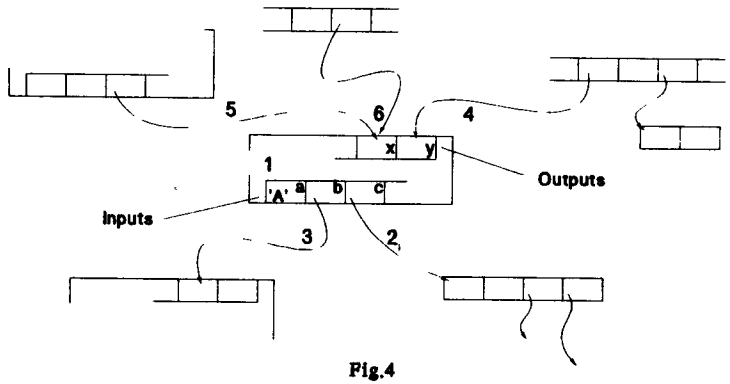


Fig.4

Producers and consumers are the basic notions of an evaluation network. Producers are usually outputs of the process. On the other hand, almost any cell, including a process input, can be a consumer. Saying that a producer is attached to a consumer we mean that the producer has to form a value and to transfer it to the specified consumer. We say that a cell, which is a consumer, stores an *unevaluated value* in contrast to *real* values, such as numbers and references. A reference from a consumer to a producer is an unusual one, that is why we call it a *link*.

The newly created process is announced to be *ready* for execution. This means, that the process is inserted into a queue, from which the processes are fetched for execution by the control program. The parent process continues its execution.

During execution there may arise the conditions under which the execution cannot be continued. In this case the process is *suspended*, and when the conditions disappear, the process is *resumed*, i.e. announced to be ready again.



**Other network transformations**

Manipulations with unevaluated values are the most important of all the other network transformations. Such cells as process inputs or elements of the outer lists are in many ways similar to inner variables and may be a part of different expressions, arithmetic in particular. If during the computation of an arithmetic expression the cell stores a real value, there arise no problems. If the cell stores an unevaluated value, the process, computing the expression, should be suspended until the value is evaluated. However, there are some operations over unevaluated values that can be performed without suspending the process. These operations include:

a) lists constructing;
b) passing parameters to a new process;
c) assignment to another input or list element;
d) sending of the value to an output;
e) etc.

In this chapter we give a brief description of the rest of system calls resulting in transformation of the evaluation network. In the text below we use the following notation: a,b - process inputs or elements of outer lists, x - an output, i - an inner variable or a constant.

`send(a,x)`
`send(i,x)`

The value of the first argument is copied to all the consumers of output x. The links between consumers and output x are destroyed, and output x becomes *evaluated* (Fig.5a). The evaluation of all the outputs of the process results in the natural process termination.

If the consumer a stores an unevaluated value (Fig.5b), the semantics of the call does not alter. All the consumers of output x become the consumers of the producer linked to input a. This is the way to copy an unevaluated value.

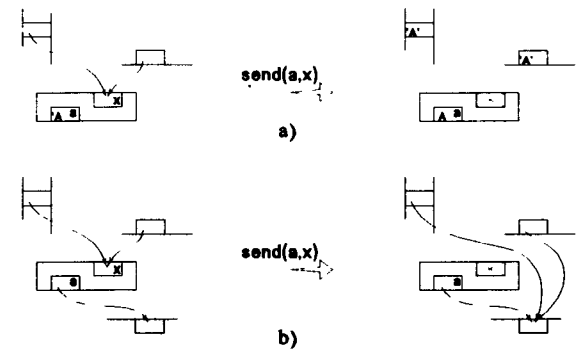


Fig. 5

`wait(a)`

If the consumer a stores an unevaluated value, the call causes its caller to suspend the execution. The process will be resumed when the consumer a gets a real value as a result of some send call.

`drop(a)`

If the consumer `a` stores an unevaluated value, the call destroys the link between the consumer `a` and the producer of the value (Fig.6a). If the producer is a process output, and the destroyed link was the last one to this output, the output becomes evaluated (Fig.6b).

The destruction of links means a rejection of the unevaluated value. In most cases this happens implicitly, during the `assign` and `exit` calls, but sometimes we have to reject the value explicitly.

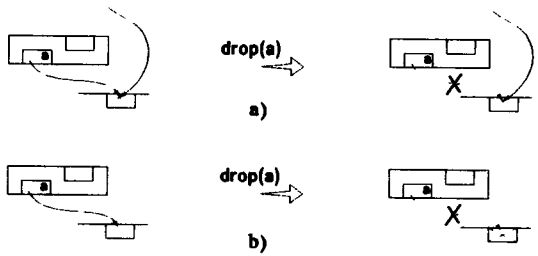


Fig. 6

`assign(a,b)`
`assign(a,i)`

If `a` stores an unevaluated value, the link between it and the producer of the value is destroyed. After that the value of the second argument is copied into `a` (Fig.7a).

If `b` stores an unevaluated value (Fig.7b), `a` becomes a consumer of its producer. The semantics of the call, similar to the `send` call, does not alter.

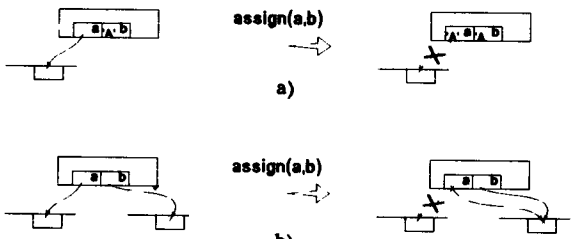


Fig. 7

`exit(i)`

This call terminates the process. Generally speaking, this termination should be regarded as abnormal, since natural termination happens as the result of evaluation of all process outputs. Accordingly, during the execution of `exit` call, the value of `i`, tagged as `error` is sent to all unevaluated outputs of the caller and may be used in processes-consumers to analyze the situation.



## Lazy Evaluation

The implementation of function calls in the described model brings the idea of the possibility of lazy evaluation. But further behaviour of the system contradicts this. The created processes are immediately announced to be ready and can be executed before the values of their outputs are demanded. This strategy is used to maximize the number of ready processes and can be called *pre-evaluation* strategy.

Typical lazy algorithms based on generation of infinite sequences cannot be executed in the pre-evaluation mode. To implement such algorithms there is provided the possibility to organize a function invocation with forbidden pre-evaluation. The processes created in this case are called *lazy* processes. A lazy process is created in a suspended state and stays in it until the value of one of its outputs is demanded. If after the evaluation of all the demanded outputs the process does not terminate naturally, it is suspended again.

## Streams

Describing the `send` system call we have said that the producer-consumer link is destroyed during the transfer of the value from the producer to the consumer. But semantics of `send` may be extended.

If the link between the producer and the consumer is destroyed during the transfer of the value (Fig.8a), they are called *disposable*. As opposed to them the *stream* producers and consumers can transfer values repeatedly, over and over again. If a stream producer of one process is linked to a stream consumer of another process (Fig.8b), we have a pair of processes linked by the *data stream*. The data are transferred on the initiative of the process-producer. If a stream producer is a part of a lazy process, we deal with a *lazy stream*, where the data are transferred on the initiative of the process-consumer.

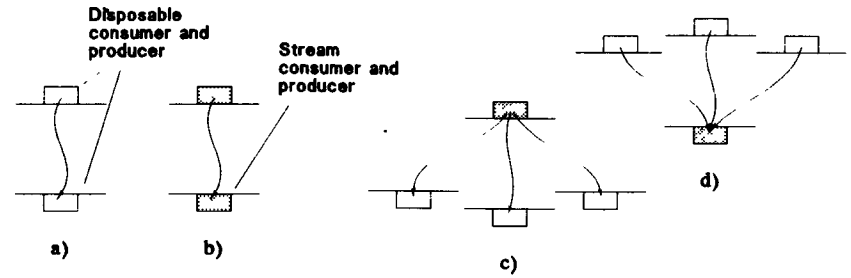


Fig. 8

If several disposable producers are linked to one stream consumer (Fig.8c), the process-consumer plays a role of information collector. The order of the information consumption is not determinate, but the nondeterminacy of this kind is quite acceptable for some tasks.

The construction with several disposable consumers linked to one stream producer (Fig.8d) seems to be meaningless.

### Memory management

The memory of a monoprocessor element not occupied by the control program and available for dynamic allocation of the evaluation network objects must be situated in the continuous address space. The memory is allocated consequently from low addresses to higher ones. If an object isn't used any more, the allocated memory is not freed explicitly, the references to it are just forgotten. As more and more references are forgotten, there appear areas of the memory which do not belong to any object currently on use. Bulldozer garbage collection eliminate such areas by shifting the objects being used into the low address memory and correcting the corresponding references. The algorithms, programming languages and compilers must provide the possibilities for relocation of code and data when the control is passed to the operating system.

### Parallelizing

Since evaluations in a correctly built network are determinate, i.e. the result of evaluations does not depend on the order of execution of the processes, newly created processes may be transferred for execution to any idle or underloaded monoprocessor element. To distribute the processes over the multiprocessor we use one of the variants of diffusion scheduling. The principles of diffusion scheduling are well-known [6]. The distribution of processes is analogous to temperature dissipation in an nonuniformly heated crystal. From time to time every monoprocessor element calculates the value that characterizes the workload of the element. If the workload of a neighbour monoprocessor element is considerably smaller, the processes are transferred there for execution. There are a lot of modifications of the diffusion algorithm, but we believe it to be inappropriate to discuss them without proper experiments and measurements.

### Plugging

Any process, being a part of evaluation network, is linked by mutual references with the objects around it. When a process is transferred to another monoprocessor element, the references become interprocessor ones. To support the interprocessor references we suggest a mechanism called *plugging*.

Let us consider a transfer of a process with one disposable input and one disposable output from a monoprocessor element A into a monoprocessor element B (Fig.9a).

When a process is transferred, the references to other objects of the evaluation network are broken. A couple of *plugs* is created at the rupture (Fig.9b). The broken end of the reference leading to a list or to a producer is attached to the *outlet* plug. The end leading to a list holder or to a consumer is attached to the *inlet* plug. The process input a becomes a consumer of the unevaluated value produced by the inlet plug. Each plug in the couple contains the number of the monoprocessor element, where the complementary plug is located, and the unique couple identifier, which is used for mutual recognition.

In some sense plugs are similar to small processes. An inlet plug, for example, may be regarded as a lazy process with one output. The complementary plugs can communicate by means of message passing via communication network, that combines monoprocessor elements into a multiprocessor.



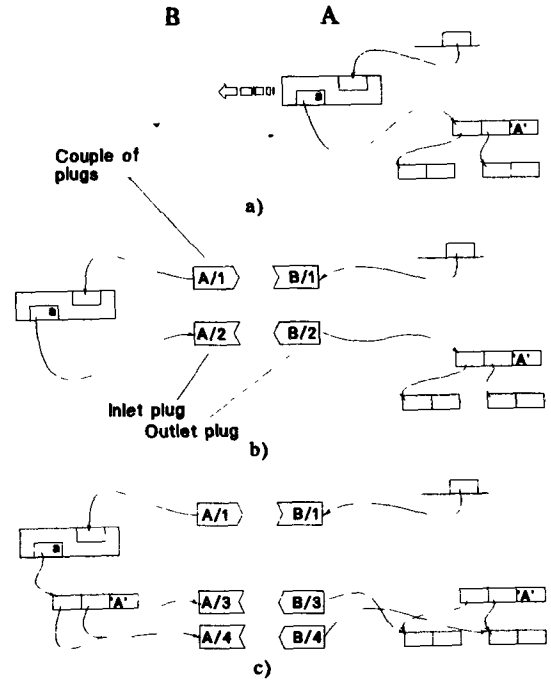


Fig. 9

If a process needs a list-like structure left in A, the inlet plug notifies the outlet plug with a special message. In reply the outlet plug transfers to B the copy of the uppermost reference level of the demanded list-like structure. On completing the operation the plugs are eliminated and new couples of plugs appear at the new ruptures (Fig.9c).

### Dynamic routing

The messages are passed through communication network by a communication subsystem of the control program. Using the universal dynamic routing algorithms, the communication subsystem is able to establish the shortest path between the given monoprocessors in a network with arbitrary topology. The path is established every time a message is being passed. This results in the increased fault-tolerance of the system, since the unexpected hardware malfunctions are not fatal to the operation of the system, at least not to the message passing.

### Conclusion

By now we have an experimental version of the system based on the described model for the Intel 80x86 family. Besides, we are about to finish the development of our own programming environment for Inmos Transputers, with the aim to port the system there. We also keep in mind Texas Instruments DSPs.

**Acknowledgements**

We are grateful to excellent programmer R.F.Gurin, who implemented the first release of the control program mentioned above. We also thank O.V.Manakova, without whose help this paper wouldn't come into existence.

**References**

1. V.F.Turchin. Refal-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, 1989

2. V.F.Turchin. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, Vol.8, No.3, July 1986, pp.292-325.

3. S.M.Abramov, S.A.Romanenko. How to Represent Ground Expressions by Vectors in Implementations of the Language Refal. Preprint, Inst. Appl. Mathem., USSR Academy of Sciences, No.186, 1988 /In Russian/

4. R.F.Gurin, S.A.Romanenko. The Refal Plus Programming Language. Intertech, Moscow, 1991 /In Russian/

5. B.Goldberg. Multiprocessor execution of functional programs. Ph.D. thesis, Yale University, Dept. of Computer Science, 1988. Available as technical report. YALEU/DCS/RR-618.

6. S.L.Peyton Jones. Parallel Implementations of Functional Programming Languages. The Computer Journal, Vol.32, No.2, 1989, pp.175-186.

*Transputer Research and Applications 6*
*S. Atkins and A.S. Wagner, Eds.*
*IOS Press, 1993*



# Does Teamwork Pay? A Comparison of Processor Allocation Approaches in An Artificial Intelligence System

Rodney S. Tosten and Jared L. Colflesh

Department of Computer Science
and Mathematics

Campus Box 402
Gettysburg College
Gettysburg, PA 17325

Email: rtosten@cc.gettysburg.edu

**Abstract.** Approaches to designing supervisor/worker parallel systems replicate several workers that process data groups individually and independently of each other. Even though this approach yields a speedup, this paper demonstrates that by grouping the individual workers in teams, one can further reduce the computing time required to complete the processing of a program. To study this processor allocation problem, the theorem proving procedure resolution refutation is used in the non-team approach. For the team approach, a new parallelized version of resolution refutation is presented and used. This parallelized version differs from others since all intermediate sentences generated in the proof procedure are maintained.

## 1. Introduction

Suppose that a robot is standing at the front door of a building and wants to enter the room 201 on the second floor. For the robot to conclude that it can enter the room, it must assume such information as the elevator is working and the room door is unlocked. The robot, however, does not need to assume that room 202 is unlocked or even that the moon is made of green cheese. For this robot to successfully exist in a given world, its reasoning system must derive the necessary conditions that must hold in the world in order for the robot to accomplish its desired goals. From a set of assumptions and a set of known facts, termed premises, each dealing with the robot's world, the robot's reasoning system must construct the subset of assumptions that the robot requires to derive the desired conclusion. In the example, the reasoning system should derive that the elevator is working and the room door is unlocked to conclude that the robot can enter room 201 Even though in this example only one assumption subset exists, there may be situations where several different