Computación y Sistemas Vol. 4 No. 2 pp.143 -157
© 2000, CIC - IPN. *ISSN 1405-5546* Impreso en México

Parallel Computation in Abstract Network Machine

Andrei Tchernykh[1], Andrei Stepanov[2], Antonio Rodríguez Díaz[1], Isaac Scherson
[1]CICESE Research Center, Ensenada, Mexico,
[2]The Institute for High Performance Computer Systems of the RAS,
Moscow, Russia, anstepanov@mtu-net.ru
[3]Information and Computer Science, University of California-Irvine,
Irvine, CA, USA,
E-mail: chernykh, arodrig , isaac@ics.uci.edu

Article received on February 15, 2000; accepted on August 23, 2000

Abstract

This paper describes a parallel Abstract Network Machine (ANM) which uses DI-structure (dynamic incomplete structures) and associative networks for representation of information. A computational process consists of asynchronous local transformations of the network with a single mechanism - network unification. All computational mechanisms are oriented towards the processing of incomplete information. The ANM is able to perform partial evaluation when given input is incomplete and to automatically synthesize a residual parallel program as a result of the transformation. The technique for transforming, optimizing, specializing multipurpose programs to particular problems, and for parallelizing programs is described. Some general problems of declarative parallel computation without concepts of a shared memory, value assignment, sequential or parallel control flow are discussed.

Keywords:DI-structure, implicit parallelism, declarative programming, partial evaluation, incomplete information processing, graph reduction.

1 Introduction

Parallelism has become a standard technique in the design of high performance computers. Despite the impressive progress achieved in the design of sequential von Neumann machines, their computing power is limited in the light of certain applications. Parallel computing emerged as an alternative and viable medium for the solution of many important problems. As a matter of fact, many conventional machines such as PCs and workstations contain some degree of paralellism. Such a tendency represents a departure from the sequential model of computation. On other hand, parallel computing itself has not been a big success. The difficulty lies with a gap beween the view needed to use a particular machine effectively and the view needed to develop parallel software successfully, that is between a model of parallel computation and a parallel machine model (Skillicorn, 1994).

Though the motivations and emphasis of individual research in parallel processing vary, most research work is quite focused. We attempt here to emphasize several aspects associated with the data-parallel paradigm; the control-parallel paradigm based on a sequential control flow with explicit concurrence, the explicit asynchronous control with shared memory, the implicit data-driven parallel computations, and the processing of incomplete information.

The most traditional and obvious way is to consider these paradigms as different ways to utilize concurrency to increase a computer performance, to exploit very large scale integration in the design of computers, to create new classes of very high level programming languages, and so on. We consider the concepts and relationships that exist both within and between these areas of research as different ways to depart from von Neumann principles of computation (Von Neumann, 1946).

The von Neumann model includes three basic principles: simple operations on elementary operands, linear common memory, and sequential centralized control. In its simplest form such a computer has a CPU, a store, and a connecting tube (bus) that can transmit a single word between CPU and memory. The program is stored in memory as a serial sequence of instructions. The program is executed by fetching successive instructions from memory and executing them in the processor. The course of computation is given by the flow of control in the program. It is not possible to execute any instruction until all previous instructions in the program have been executed. If operands are available, some program instruction could be executed, but is not executed until their turn comes in the program. This is the main obstacle in the utilization of the natural parallelism of algorithms.

Just those principles determined the main features of computers: organization of computation, architectures, advantages and shortcomings of languages, the style of programming with word-at-a-time thinking, mentality of programmers and so on. In a sequential programming environment, the different actions occur in a strict single-instruction-execution fashion. Flynn (Flynn, 1972) in his taxonomy viewed the von Neumann model as a Single stream of Instructions controlling a Single stream of Data (SISD).

The simplicity, lucidity, flexibility of those principles provided a swift development of computers. For sequential computation, the model and conventional architectures, complex versions of the von Neumann computer, are very close. At the same time, they become a hindrance for computer development. Memory interleaving, cache memory, instruction and execution pipelining, branch prediction, speculative execution are common hardware optimizations intended to overcome the limitations of the von Neumann machine. At a more abstract level, however, they can be viewed as a single stream of information. An improvement or substitution of any of von Neumann principles demands reconsideration of all aspects of computation: organization of computation process, computer architecture, language, style and method of programming (Backus 1978).

1.1 Data Parallelism

In the data parallel approach, the paradigm of parallelism is squeezed into the base frame of the von Neumann conception. A data-parallel program derives its parallelism from executing the same instructions on many processors at the same time, but on different data. Regular data structures such as vectors and matrixes, and parallel operations on them are added to sequential computations. A sequential control-flow (one global control unit), a shared, with a single address space, memory inherited from common ancestors are not changed.

One of the greatest advantages of data-parallel programming is its reduced need for synchronization. The parallelism becomes possible only as a result of independent processing of the data elements. Data-parallel programs are naturally suited to SIMD computers. This paradigm is typically associated with fine-grained parallelism and is available on a wide variety of computers. Even so, data-parallelism does not solve all the problems of parallel computing. A main drawback of SIMD computers is that different processors cannot execute different instructions in the same clock cycle.

SIMD	von Neumann	MIMD	Asynchronous	Data-flow	ANM
parallel operations					
parallel data structures					incomplete data structure
sequential control flow		parallel control flow	explicit asynchronous control	implicit data-driven control	transformational principle of computation
shared memory (single address space)				logically distributed memory (queues)	network representation of program/data
operation					n-unification
operand					incomplete value

Table I. Models of computation

1.2 Control Parallelism. A Sequential- Parallel Approach with Synchronization

In the sequential-parallel approach explicit parallel control operators augment the sequential control flow. These operators allow more than one thread of the control to be active at the same time, and provide the possibility of independent calculations. An algorithm may consist of a number of tasks or processes which themselves are purely sequential, but which are executed concurrently on many processors, and which communicate through shared variables. Explicit synchronization facilities are introduced to provide thread synchronization and regulate shared memory cell accesses. It is the common and widely used solution of parallelizing problems. Algorithms for problems requiring control parallelism usually map well onto MIMD parallel computers because control parallelism requires multiple instruction streams. In programming languages for shared address space computer paradigm that kind of parallelism is represented by well-known primitives, such as *fork-join*, *parbegin-parend*, primitives for mutual exclusion and synchronization, parallel loops, pragmas, and many others. Such parallel languages are successors of traditional sequential languages. Extensions of C, Fortran, Pascal and others have been developed for various parallel computers. Also message-passing computers are typically programmed in conventional sequential languages augmented by message-passing primitives such as, for instance *MPI_Send, MPI_Recv*. Accordingly it is possible to keep principles and a style of a sequential programming unchanged, corresponding compilers can be enhanced by parallelizing preprocessors and so on. However the level of such parallel programming languages decreases in comparison with their precursors. It is necessary to control and to describe in an explicit form not only a sequence of operators but their parallelism as well. This approach has some shortcomings especially when one tries to reveal the maximal inner parallelism of problems for massively parallel computers. Programs may be efficient, but tend to be difficult to understand, debug, and maintain, especially when a program is turned into a dish of spaghetti by side effect and goto. The main problem concerns the gap between parallel control and shared memory conception on one hand, and solving the resource contention problems on the other.

1.3 Explicit Asynchronous Computation

In contrast to the above control parallelism where the sequential control is augmented by parallel one, in asynchronous computation the sequential control is *substituted* by the asynchronous control. Program fragments are initially regarded as parallel, independent, and unordered. Any constraint on their execution is formulated as explicit or implicit readiness conditions. Various modifications of asynchronous computation involve different organization of the data exchange between program fragments. In explicit asynchronous computation the notions of a single address space and execution of operators with operands remain unchanged. The data is exchanged through a common memory shared by a given group of statements. The control is implicitly asynchronous and explicit control operators are used for establishing dependencies between operations. Each informational operator (guarded operator) has one or two control operators associated with it. The first one is the *trigger operator,* which is a Boolean function may be rather complicated that determines the "readiness" of the informational operator.

The model assumes that trigger operators are executed concurrently in waiting mode. When the value of a trigger operator becomes "true" the corresponding informational operator is activated. When the execution is finished the second control operator generates a "true" value and writes it into the memory. So, the information about the termination of the operation becomes available for the other trigger operators. Thus, though potentially the computations are considered to be independent and parallel, conditions for readiness of operations have to be explicitly determined.

The use of such an asynchronous computation was formally introduced and tackled by Dijkstra, Kotov, and Narinyani and many others in the seventieth and eighties (Dijkstra, 1975, Kotov, Narinyani, 1969).

1.4 Data-Flow

The data flow (DF) parallel computation (Gao, Bic, and Gaodiot, 1995) is also based on the asynchronous principle, with one essential distinction. The exchange of information between program fragments is performed through isolated direct paths, which are usually queues. Control is decentralized, each statement determining independently its degree of readiness to initiate computation. Notice that all concepts of the von Neumann model, except of an operation and operand, are substituted, in this model (Table 1). Distributed memory allows to simplify the readiness conditions of the operations, and to do them implicitly. The operation is triggered automatically when all its operands are available. Data availability is achieved by channeling results from previously executed instructions into the operands of waiting instructions. This channeling forms a flow of data, triggering instructions to be executed. Many instructions are executed simultaneously, that leads to the possibility of a highly concurrent computation.

The data flow model of computation is closely associated with the principle of single assignment. In traditional computer systems one memory location (memory cell) is assigned to each variable and different values can be assigned to a variable during the computation. In the principle of single assignment a variable (name) may only take a single value during computation. So, during the use of one variable in the program there exist two periods. The first period is when one location is reserved for the variable in the memory, but it has not acquired a value, hence it cannot be read from the memory, but its value can be written. During the second period the variable has already taken the value and has been written in the reserved memory location. The value can be read many times, but it cannot be changed any more.

These models have many attractive properties for the parallel processing and solve many problems of parallel computation (Table 2). The program writing is clear. There is no side effect. In the same way it is possible to write a "high level" as well as a "low level" program. DF programs express in a simple manner the natural parallelism of algorithms, allow the exploitation of parallelism at different levels, and expose various forms of parallelism.

However, an absence of a shared memory concept allows no satisfactory description of the complex parallel processes based on the conception of a common resource. Moreover, the pure data flow execution scheme employs a by-value data access mechanism and this causes explicit data copying, hence the complicated data structures are difficult in use. This problem is eliminated by introducing by-reference data-access that overcomes that shortcomings and allows to use matrices, vectors, lists and others more efficient (Amamiya et al., 1983).

calculated. This scheme seems much less attractive in comparison with the process of computation in an analog device. One could see that the process of forming the potential of the input signal *x* forces forming, after a short delay, the potentials of *G(x), K(x)*, and they, in their turn, force forming *H(x)*. The functions *G, K,* and *H* work in parallel.

This example shows that there is one more approach for increasing the level of the parallelism, namely a manipulation with an incomplete information. We already mentioned that most parallel models (Skillicorn, 1991) leave unchanged two von Neumann concepts of sequential computation: an operator and operand (Table 1), and a principle of so-called computational act or strict semantics. The latter means that to start the execution of an operator or a function all operands required by it must be fully computed and be accessible, all components have to be evaluated before a structure is used. We know that the value is usually determined gradually during its calculation. For example, in an arithmetic unit a sign of a number is usually calculated first, then its order, and its mantissa. The main restriction is that the number can be used only after the moment when it is fully computed and is written to a memory cell, to a data-flow token, or, for example, when the bit of readiness or flag is set. Though in a conditional expression ***if** F(x)>0* ***then** F1* ***else** F2* we are interested only

Model	**Control Flow (control driven)**	**Data-flow (data-driven)**	**Reduction (demand-driven)**
Basic Definition	**Conventional computation;** control indicates what and when a operator should be executed	**Eager evaluation;** operators are executed when all of their operands are available	**Lazy evaluation;** operators are executed when their result is required for another computation
Advantages	Full control Complex data and control structures are easily implemented	Very high potential for parallelism High throughput Free from side effects	Only required instructions are executed High degree of parallelism Easy manipulation of data structures
Disadvantages	Difficult in programming Difficult in preventing run-time error	Time lost waiting for unneeded arguments High control overhead Difficult in manipulating data structures	Does not support sharing of objects with changing local state Time needed to propagate demand tokens

Table 2. Control, data and demand driven models

1.5 Parallelism Based on the Processing of Incomplete Information

Let us consider parallel evaluation of the function *F(x)=H(G(x),K(x))*. The evaluation of the function *H* can be started only after both values *G(x)* and *K(x)* are fully computed. For given *x*, *G* and *K* can be processed concurrently. Notice that it is impossible for the value *F(x)* to be used in another evaluation before it has been fully

in the sign but not in the value of *F(x)*, to start evaluation of *F1* or *F2* we have to wait the termination of calculation of the value *F(x)*.

In conventional computers the value becomes available instantly as a result of one indivisible computational act. This principle forces the computation to be sequential in this point. The value becomes one more bottleneck of parallel computing. To overcome the limitation of this principle, and to widen this neck it is necessary to introduce into practice both incomplete operands and facilities for using them in calculations. For instance, *nonstrict, lazy,* *lenient semantics* (Wei and Gaodiot, 1988, Amamiya and Hasegawa, 1984, Amamiya et al., 1983) or *I-structures* (Arvind et al., 1989) are approaches to increase parallelism in such a way.

An I-structure is a conventional structure with some constrains on its construction and destruction. Each element of the structure can be in three states: *empty,* when data has not been requested, and no data is available; *deferred,* when data has been requested, but is not available yet, and *present,* when data is available. *"Presence bits"* are associated with each cell of storage. According the single assignment semantics writing into a cell with presence bit set causes an error. Reading when presence bit is off causes a *"deferred read"*. I-structures may be consumed before they are entirely produced. This allows increased parallelism via "pipelining" between the producer and the consumer, between G and H in the above example. *Latency* is tolerated via split-phase operations. I-structures are appropriate for the case when the structure is not updated.

In this paper we focus on the issue of dynamic incomplete structures (DI-structures) used in the Abstract Network Machine (Stepanov et al., 1993). We show that incomplete values, relations (functions) with unknown (flexible) arity, and incomplete structures with unknown numbers of elements can be represented and manipulated in the ANM as DI-structures. The machine follows the transformational style for the fine grain parallel computation that is similar a graph reduction widely known in a functional programming.

The incomplete-data-driven strategy of the associative network transformation with a single rule of the transformation is introduced. The decision to undertake computation is based on the increment or giving a more precise description to elements of DI-structures rather than on the availability of fully complete data as in a data-flow. Each new portion of information increasing the DI-structure description forces the computation that follows a data-driven computation (Treleaven et al., 1982).

The paper is organized as follows. The next section briefly describes the main concepts of the ANM, representation of information by DI-structures, and a basis of parallel computation on associative networks. Section 3 discusses the ANM model, operations, and the process of computation. Sections 4 and 5 review some experiments with the ANM in the declarative parallel programming system PARNET.

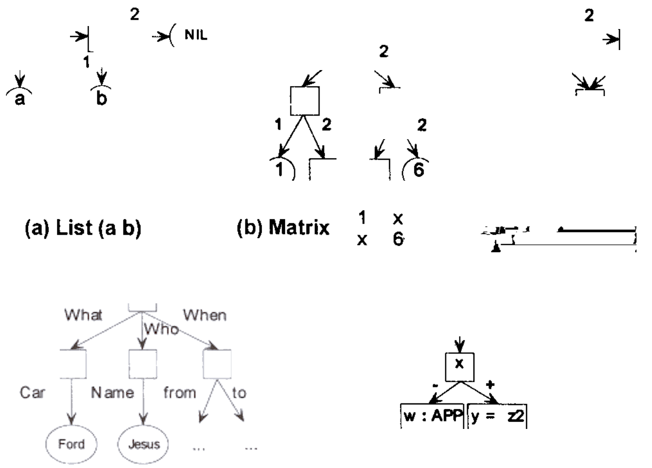
2 Abstract Network Machine

2.1 Network Representation of Information

An *associative network* is a directed graph of a special kind with labeled edges. Nodes of the network are called *objects*. Three types of the objects namely *atoms,* DI-structure, and *empty* objects are introduced.

The atom represents an object that has no an internal structure at a programmer level of consideration. For example, the atom can be integer, character, or Boolean value. The atom is defined completely by its representation. In Figure 1 the atoms are shown in circuits. Two atoms with the same representation are not distinguished. The atom is a static object of the network, its description can not be changed in the course of computation. It is sufficient to have only two atoms *T* (true) and *F* (false). In this case the characters, numbers, and other atoms might be represented by their binary codes, and therefore become DI-structures.

The DI-structures (DIS) and empty objects are referred to as dynamic objects. A description of the DI-structure is a set of elements (associative memory). Each element is a pair *[A X]*, where *A* is the element's name (some identifier, or a positive integer that is used for a key of the associative memory), and *X* is the element's value (some other object of the network).



(a) List (a b) (b) Matrix $\begin{matrix}1 & x\\ x & 6\end{matrix}$

(d) Fact (relation, structure) of buying (e) Logical object

Figure Associative network example:

In Figure 1 the DI-structures are shown by squares and their elements are shown by arrows. The name of the element is written on the arrow, and the object, corresponding to the element's value, is located at the end of the arrow. There is a limitation on the DI-structure description: it cannot contain elements with the same name. No other limitation exists, in particular, loops and cyclic structures are allowed.

An empty object cannot be classified as the atom or DI-structure, because there is no any information about it, except the fact of its existence. Objects and elements of a special kind intended for the implementation of some built-in mechanisms such as arithmetic or logic are introduced. For instance, a logical object *X* in Figure 1e has two elements with the build-in names "+" and "-". The value of "+" ("-") element is a set of declarations about network objects that are valid if *X* is true (false).

The elements of the DI-structure can be interpreted in different ways, related with such terms as a property, inclusion, relation, parameter, octant, role, slot, element of structure etc. (Stepanov et al., 1996). For the ANM all these interpretations are indistinguishable, since all of them are united by one property, which is the only that is used: the name of the element unambiguously determines the element's value. An important feature of the system is its homogeneity in respect to the representation of the objects and relations between them: on formal level they are not distinguished.

2.2 The Representation of Incomplete Information

Atoms are considered to be complete static values, whereas DI-structures are considered as incomplete dynamic values. The concept of fully complete DIS makes sense only on the programmer level of consideration. The number of the elements can be increased in the course of computation. Hence, the description of the DIS is considered to be incomplete in any moment.

Several degrees of the DI-structure incompleteness could be distinguished. We mention three of them. (1) The fact that the object exists is only known. (2) Some elements are available, but the full description from the programmer point of view is incomplete, for instance some rows of the matrix could be unavailable. (3) The total structure is known, while some of its elements are unknown. Latter is a similar to the *I*-structure.

DI-structure keeps some amount of information about a single "real" object of a problem to solve. This information can not be "replaced" by another one in the course of computation, the object can only accumulate it by adding new elements. Elements of DI-structures cannot be updated. DI-structures are never supposed to be completely defined. The ANM does not know the maximum number of elements. Hence, it is impossible to make conclusions, based on the fact that some element is last among those of the current DI-structure. There is no such last element, at any moment one more can be added.

2.3 N-Unification, S-Similarity

There are three kinds of declarations: those describing DI-structures by elements *[A X]*, those stating the identity of two objects *X=Y* (*n-unification)*, and those stating the similarity of an object to a scheme *X:S* (*s-similarity)*. It is possible to express recursive declarations, and to make conditional declarations like ***if*** *X=nil* ***then*** *Y:S, Z=W* ***else*** *X=Z*.

The **n-unification** *X=Y* is considered as a statement that two objects *X* and *Y* are in fact one object, or *X* is identical with *Y*. The n-unification leads to the reduction of the network "gluing" these two objects. If objects are DI-structures, such a gluing leads to pooling their elements. The very simple transformation rule is correctness preserving. If in a resulting set of elements two elements have the same name, one of these elements is removed and a new operation of the n-unification of the values of these elements is generated.

In Figure 2 shows two examples of the n-unification of objects *x* and *y*.

```
x[A x1, B x2, C x3];   |      x[A x1, B x2, C x3];
y[A y1, C y2];         |  ⇒   x1=y1; x3=y2;
x = y;                 |
                              (a)

x(x1,x2,x3);           |      x(x1,x2,x3);
y(y1,y2,y3);           |  ⇒   x1=y1; x2=y2; x3=y3;
x = y;                 |
                              (b)
```

Figure 2. Examples of the transformation rule for n-unification[1].

To n-unify an object with a copy of other object the *s-similarity* is introduced.

The s-similarity *X:S* declares that the object *X* is an instance of the *scheme S*. The scheme is a description of the DI-structure *S*, thereto it can contain local declarations about other objects of this scheme. S-similarity leads to copy of the DI-structure *S*, its n-unification with *X*, and execution of the local declarations. Any network (DI-structure) can be extracted from the main memory and later be used as a scheme.

A scheme is an analogue of a functional definition, and s-similarity resembles a functional call. The essential difference of the transformation by s-similarity in comparison with a graph reduction lies in the manner of the parameter updating.

In functional programming a program is represented by a directed graph, nodes are functional calls or data items, and edges represent the arguments to that function. Execution proceeds by reductions, which transform the graph to a simpler (normal) form, with no more function calls, and amounts to repeatedly rewriting the function representation. Arguments of the copy of the functional definition are updated by arguments of the functional call in a graph reduction, whereas s-similarity leads to the n-unification of the copy of elements of DI-structure *S* with the appropriate elements' values of the object *X*. It gives such features as a more deep processing of incomplete information, evaluation with partially completed parameters and bi-directional flow of information: from input to output and

[1] The simplified version of the DI-structure notation x[1 x1, 2 x2, 3 x3] is x(x1,x2,x3), when elements with integer names are used. The name of the element corresponds to its position number in a list of elements.

from output to input. Moreover, elements of *S* and *X* can be mutually complementary (Tchernykh et al., 1997).

3 ANM Model

3.1 Operations

The ANM model includes the main network memory, scheme memory, operation memory, and mechanisms to execute operations. The process of computation consists in executing the operations from the operation memory. The execution of an n-unification leads to gluing of two objects that, in its turn, may generate several operations of the n-unification and s-similarity that are placed at the operation memory. The s-similarity forces a copying of a scheme from the scheme memory into the network memory followed by the n-unification between the object and the copy of the entry of the scheme. Besides, all operations from the local scheme memory are copied and added to the contents of the operation memory.

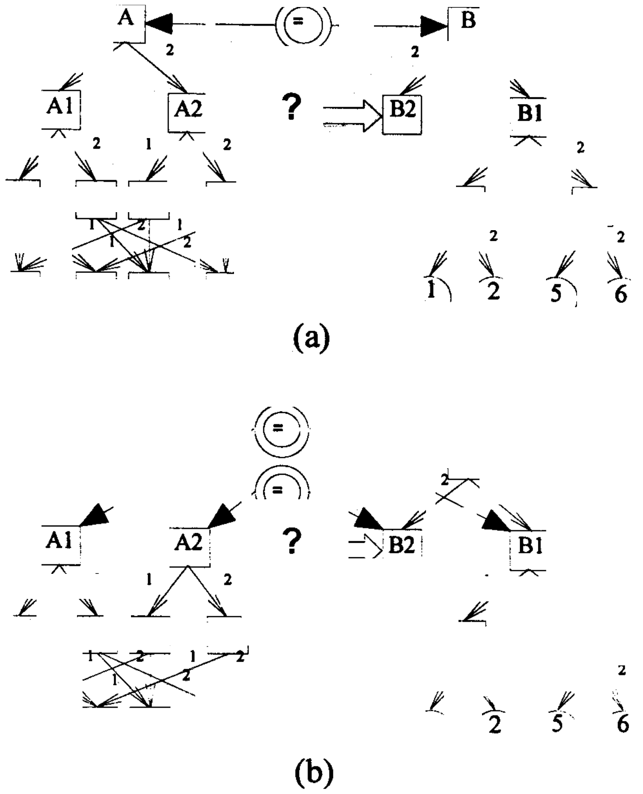


Figure 3 (a, b). Transformation principle of computation (2x2 Matrix Transposition)

The operations are executed until the operation memory contains no more operations. It is possible to execute the operations in any order, including concurrently. Any operation can be postponed for any interval.

3.2 Process of Computation

In Figure 3 several snapshots of the main network memory in the course of the unification of objects *A* and *B* are shown.

The DI-structure *A* could be interpreted as a relation of the transposition of two *2x2* arbitrary matrices *A1, A2*, and *B* as a description of some relation between given matrix *B1((1, 2),(3, 4))* and *B2*.

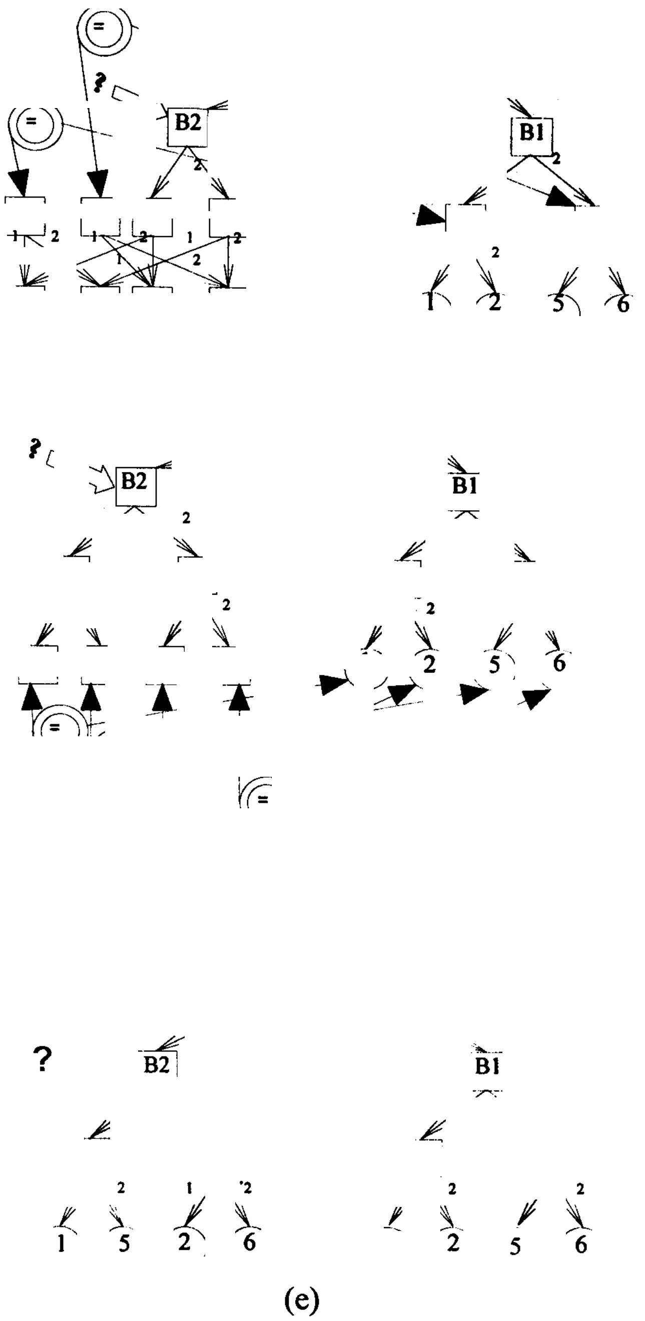


Figure 3 (c, d, e). Transformation Principle of Computation (2x2 Matrix Transposition)

The n-unification $A=B$ declares that A and B represent the same relation. The object marked by the question mark is expected to be a result of the matrix $B1$ transposition. The process consists of five steps. Notice that one n-unification is executed on the first step, two on the second, and four operations can be executed in parallel on the fourth step.

3.3 Programming

To calculate matrix transposition of different matrices a corresponding scheme, say *TRANSP,* to be applied. The scheme should describe a relation between two matrices $A1$ and $A2$. For simplicity we consider only matrices of fixed range, say again 2x2. The matrices are described by means of DI-structures. The object $A1$ has two elements (as many as there are rows), and each element in their turn has two elements: $A1((a_{11}, a_{12}), (a_{21}, a_{22}))$. The element a_{ij} is represented by an empty object. In the process of computation it may turn out to be an atom (in our example) or a DI-structure, for instance, if it is submatrix. The matrix $A2$ is described as $A2((a_{11}, a_{21}), (a_{12}, a_{22}))$.

sch TRANSP(A1, A2);
A1((a_{11}, a_{12}), (a_{21}, a_{22}));
A2((a_{11}, a_{21}), (a_{12}, a_{22}));
end

To transpose a given matrix *B1(('2', '5'), ('8', '1'))* we have to declare that the objects *B1* and *B2* is in the relation *TRANSP* by *(B1, B2):TRANSP*. In this example all elements of *B1* are known and *B2* is an empty object (marked as a result). To solve the task, DI-structure *W(B1, B2)* (Figure 4) has to be placed into the main network memory, and the s-similarity *W:TRANSP* into the operation memory. The DI-structure *W* together with this operation serves as a request for the ANM to transpose the matrix *B1*.

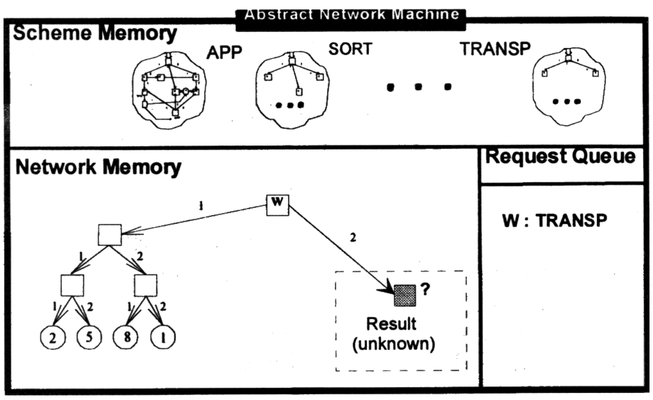


Figure 4. Matrix transposition invocation

The scheme *TRANSP* is copied from the scheme memory into the main network memory, and the operation of the n-unification $W=W1$ is put in the operation memory. It initiates the process of the network transformations shown in Figure 3.

One can see that the *TRANSP* is not an algorithm for yielding *A2* by *A1*. Both matrices take part in the relation *symmetrically,* so it is not necessary to give a complete matrix as its first argument, and the empty object, as its second one. With equal success *B1* can be calculated if *B2* is complete. More than that: both matrices may be completed partially. The system will make all inference it can, based on the supplied information. For instance, given *B1(('2', '5'))* (the second row is unknown), *B2(('2', -), ('-5', -)),* and *(B1, B2):TRANSP*, the structures of both matrices are evaluated completely, supplement each other. Nothing wrong happen if we even provide both complete matrices. Successful termination of the process confirms that the matrices are in this relation. If they are not in the relation, computation will be stopped because of a contradiction. Contradiction occurs if, for instance, two different atoms are trying to be unified. In this case we will know that the declaration *(B1, B2):TRANSP* is invalid for a given matrices.

4 Partial Evaluation and Optimization

Partial evaluation (Jones. 1996) is an automatic program transformation technique to partially execute a program, when only some of its input data are available, and to specialize it with respect to partial knowledge of the data. It provides a unifying paradigm for a broad spectrum of work in program optimization, compiling, and interpretation (Jones et al., 1993, Bjorner et al., 1988).

Consider a program p with two inputs, inp_1 and inp_2. When specific values are given for the two inputs, we can run the program, producing a result. When only one input value inp_1 is given, we cannot run p, but can partially evaluate it, producing a version p_{inp1} of p specialized for the case where inp_1 is given. The specialized version p_{inp1} of p is called a residual program.

The motivation for doing partial evaluation is speed: program p_{inp1} is often faster than p. Specialization is clearly advantageous if inp_2 changes more frequently than inp_1. Each time inp_1 changes one can construct a new specialized p_{inp1} that faster than p, and then run it on various inp_2 until inp_1 changes again.

The partial evaluation includes two kinds of activity: partial execution of a program when a part of its inputs is not given ("dynamic") and forming "a residual program" consisting of "delayed operators". When partial evaluation terminates the residual program is ready for processing the dynamic part of the inputs.

The transformational style of computation in ANM, together with incomplete-oriented style of information processing, leads to the "effect" of partial evaluation. Computation ANM performs can be considered to be partial. Any network obtained is a residual network ready for further. transformations when the incomplete information becomes more complete or dynamic inputs become static. Three computation mechanisms: evaluation, partial evaluation, and generation of a residual program are based on a mechanism of n-unification. Moreover in the ANM the incompleteness of information is not restricted to dynamic inputs but is understood as DI-structure incompleteness.

Partial evaluation is shown to be a universal technique for a transforming, optimizing, translating, and even program parallelization (Ershov, 1982, Tchernykh et al., 1997). Although simple in concept, it has important applications to scientific computing, logic and functional programming, compilation, computer graphic, pattern matching and others (Bjorner et al., 1988; Jones et al., 1989; Consel and Danvy, 1991; Jørgensen, 1992; Jones et al., 1993; Sesyoft and Sondergaard, 1995).

Parallelism. The ANM bridges partial evaluation and parallel computation by means of the transformational style of computation. In this paper two issues of a partial evaluation parallelism are underlined, namely the parallelism of a partial evaluation process, and the parallelism of a residual program.

PARS, as a declarative language (Stepanov and Lupenko, 1991), is based on the principle of representing a program in terms of what is to be evaluated rather than how the evaluation is to be performed. A PARS program seems to be a data description only, not a program description in traditional sense. The program is a set of declarations where their order is not important; relations represented by DI-structures are essentially multi-directional. Their arguments can become definite in arbitrary order, information flows through a relation in all directions.

Programs are not annotated to denote neither parallelism nor sequence. Nevertheless several known kinds of parallelism can be distinguished in ANM in an implicit form: pipeline parallelism, functional parallelism (parallelism of s-similarities), parallelism of structures, function (scheme) argument parallelism, and data-flow parallelism (Stepanov, 1991).

Let us evaluate the following expressions in the *network* representation:

z:F(x); ***if*** *z=nil* ***then*** *y:G(z)* ***else*** *y:Q(z),*

where the list *x* is statically bound. The sequential partial evaluator evaluates the program in three steps and yields the value of *y* equals *G(z)* or *Q(z)*. Function *F(x) is* evaluated on the fist step. The evaluation of the conditional expression *z=nil* can be started on the second step, only after *z* has been fully evaluated. In such an evaluation the list *z* is considered as a monolithic coarse grain value.

ANM. is able to evaluate fine-grained structures and incomplete structures (in the example, lists *x, z* and *y* can be represented as lists of DI-structures). Hence, in the beginning of the *F(x)* evaluation, when we get to know that *z* is not the empty list, *y:Q(z)* can be evaluated not waiting for the moment when the evaluation of the list *z* is completed. Hence evaluations of *F* and *Q* can be done in parallel. Evaluation of the elements of the list *z* in *F(x)* overlaps with the evaluation of *Q(z)*. This kind of parallelism is frequently referred as pipeline parallelism to distinguish the parallelism of the data structures.

The next additional source of parallelism originates from evaluating several arguments of a function (or relation) in parallel. For instance, for the function *F(G(x), H(x))* the corresponding schemes *F, G* and *H* are copied and evaluated in parallel. Thereto pipeline parallelism can be detected in, because the function *F* is able to operate on data partially generated by *G* and *H.*

The ANM allows a program to be specialized not only with given values of its input or partially complete values, but also with respect to some knowledge about them. For instance, a typical partial evaluator will return the term *if z=nil then G(y) else Q(y)* unchanged, if the list *z* is dynamic. Partially static information such as "*z* is a list with unknown length and/or unknown elements, but not empty" can be represented and manipulated in the ANM. For such the case the term can be reduced to *Q(y)*. Programs in the ANM can be specialized with respect to any more complete description of dynamic input. Likewise, Consel and Khoo have implemented the parameterized partial evaluation that allow a program to be specialized also with respect to some abstract properties of an input (Consel and Khoo, 1991).

Specialization. Let us consider example of a program specialization of with respect to a partial knowledge of the input data. The set of schemes for three TRM1, TRM2, and TRM3 versions of *NxN* matrix transposition are shown in Figure 5.

The programs are different enough in style and the codes seem at first sight to be less elegant and efficient than equivalent *imperative* parallel programs on a high level language. Here, it should be emphasized that we do not discuss the syntax of the PARS language, its convenience and expressive power for this domain. We have intentionally chosen "poor" assembler-level *PARS* version to illustrate the internal network representation of programs and a computation process. For instance we use a lower level build-in relation *(x,i,y):*ATTRIB* instead a common operator y=x*[i]*.

```
sch TRM1(I,J,N,M1,M2);
  (M1,I,M1i):*ATTRIB, (M1i,J,M1ij):*ATTRIB,
  (M2,J,M2J):*ATTRIB, (M2J,I,M1ij):*ATTRIB,
  J+1=J1, I+1=I1;
  if N >J then (I,J1,N,M1,M2):TRM1;
  if N >I then (I1,J,N,M1,M2):TRM1;
end
```

(a) TRM1

```
sch TRM2(M1,M2,N); (N,'TRM21',(M1,M2)):Dcycle
end
```

```
sch TRM21(I,J,(M1,M2));
  (M1,I,M1i):*ATTRIB,(M1i,J,M1ij):*ATTRIB,
  (M2,J,M2J):*ATTRIB,(M2J,I,M1ij):*ATTRIB;
end
sch DCycle(N,S,C); (N,'DC',*Entry):Cycle
end
sch Cycle(I,S,C);
  ((I,C),S):*UNFLDBR, I-1=I1;
  if I>1 then (I1,S,C):Cycle;
end
sch DC(I,(N,S,C)); (N,'DC1',(I,S,C)):Cycle end
sch DC1(J,(I,S,C)); ((I,J,C),S):*UNFLDBR end
```

(b) TRM2

```
sch TRM3(M1,M2,N,K);
(K,'1',N,M1,M2):TRM31;
  if N>K then (M1,M2,N,K1):TRM3, K+1=K1;
end
sch TRM31(I,J,N,M1,M2);
  (M1,I,M1i):*ATTRIB,(M1i,J,M1ij):*ATTRIB,
  (M2,J,M2J):*ATTRIB,(M2J,I,M1ij):*ATTRIB;
  if N>J then (I,J1,N,M1,M2):CTRM31, J+1=J1
end
```

(c) TRM3

Figure 5. Schemes for Matrix Transposition

Though the task is far from the domain that deals with complicated data structures, and can be efficiently solved in a base frame of imperative paradigm this example demonstrates a programming style of calculation *M2* by *M1* without algorithmic description.

Operation level parallelism profiles of evaluation processes of the programs TRM1, TRM2, and TRM3 for the case when all 3x3 matrices are static are shown in Figure 6. Note that unit execution time operations without communication and memory access delay is considered. The processes are not very mach alike. They are different in a number of operations, speedup and other characteristics. Completion times of the tasks are 16, 27, 17 unit time slices for the TRM1, TRM2, and TRM3 respectively. The total number of executed operations is 254, 230, and 129, and the maximum number of operations executed in parallel is equal 36, 20, 18 for TRM1, TRM2, and TRM3 respectively.

A partial evaluation can be applied to optimize these processes. Let us consider their specialization with respect to partial knowledge of the input matrices. Several degrees of the incompleteness of the matrix are considered: a) The only fact of the matrix existence is known; b) Some elements of the matrix are defined, but the matrix as a whole structure is unknown; c) The complete structure or *N* is known, while some of its elements or all elements are not given.

In the last case *M1* has a structure *M1((_,_,_),(_,_,_),(_,_,_))* that represents an empty 3x3 matrix. The processes of the programs' evaluation for the case when *N* is only available is shown in Figure 7.

In general, operations of a task could be virtually "divided" into two subsets. The first one includes operations performed useful actions for a specific problem. The second subset includes the rest operations that are the operations, needed to organize computation (iterations, functional calls, argument substitutions, and so on) and to access the elements of data structures, that is, they are preparatory operations for useful calculations. For instance, in matrix multiplication we consider operations of the multiplication, addition, related to the calculation of the elements of the resulting matrix as useful, and operations of incrementing indexes and their comparison with *N* in iterations as preparatory.

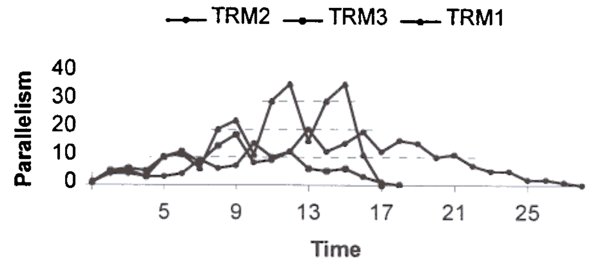


Figure 6. Operation level parallelism profile for matrix transposition parallel process

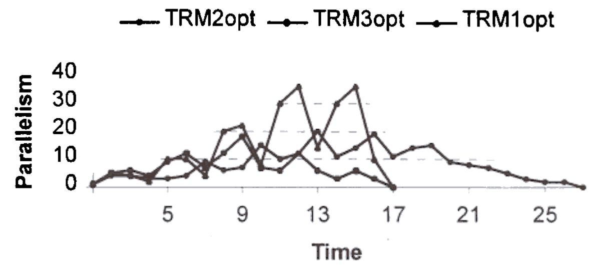


Figure 7. Operation level parallelism profile for matrix transposition parallel process when N is given only

The processes shown in Figure 7 include the execution of preparatory operations only because elements of the input matrices are not given. When the processes are terminated, the residual schemes are synthesized. These schemes are networks "touted" on incomplete elements of input matrix and elements of a matrix-result. These schemes yield the result as soon as the concrete elements of matrices are given. Figure 8 shows the parallelism profile of the execution of these residual schemes. Figure 9 shows the code of these schemes. Notice that the residual schemes and their profiles obtained are equal in all experiments.

The data of evaluation and their performance analysis are shown in Table 3.

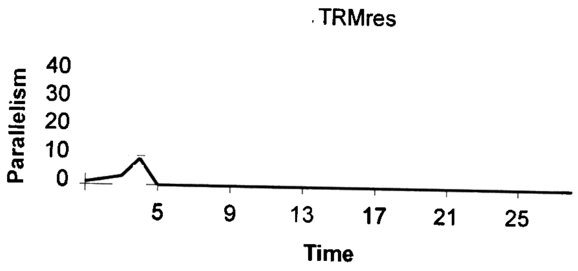


Figure 8. Operation level parallelism profile for matrix transposition parallel process by residual program

The size of a scheme, time of its sequential execution T_1, time of execution on p processors T_p, are estimated for original and residual programs. To measure the quality of programs S_p (the parallelizability), E_p (the efficiency), C_p (the cost) are used.

PARS Program		TRM1	TRM2	TRM3
Source	size	1020B	2014B	1508B
	T_1	254	230	129
	T_p	16	27	17
	P	36	20	18
	S_p	15.87	8.52	7.59
	E_p	44.10	42.59	42.16
	C_p	576	540	306
Partial Evaluation	T_1	242	218	117
	T_p	16	26	16
	P	36	20	18
	S_p	15.13	8.38	7.31
	E_p	42.01	41.92	40.63
	C_p	576	520	288
Residual	size	647	647	647
	T_1	15	15	15
	T_p	4	4	4
	P	9	9	9
	S_p	3.75	3.75	3.75
	E_p	41.67	41.67	41.67
	C_p	36	36	36
R	size	1.57	3.11	2.33
A	Operation	16.9	15.3	8.6
T	Time	4	6.75	4.25
I	Processor	4	2.22	2
O	Cost	16	15	8.5

Table 3. TRM parallelization and performance analysis

The parallelizability is used to refer to a particular case of speedup when the ratio between the time T_1, taken by a parallel computer executing a parallel program on one processor, and the time T_P taken by the same parallel computer executing the same parallel program on p processors is considered.

```
sch TRMOpt(M1,MRes);
M1((a11, a12, a13),(a21, a22, a23), (a31, a32, a33));
MRes((a11, a21,, a31),(a12, a22, a32), (a13, a23, a33));
end
```

Figure 9. Residual scheme

Execution of the program is analyzed for unbounded parallelism on a machine where the number of processors can grow as the size of the problems grows.

From the Table 3 one can see that these very simple examples demonstrate a high level of fine grain parallelism. The yielding speedups of the programs are substantial 15.87, 8.52, and 7.59 times for the TRM1, TRM2, and TRM3 respectively. Hence the ANM can expose a vast amount of a fine grain parallelism of declarative programs, which are free of any explicit description of a parallel or sequential control structure. Especially it gains when a problem involves complicated irregular data structures (such as lists, trees, graphs, etc.) and detection and description of the parallelism manually makes difficulties.

Partial evaluation can optimize programs by considerable reduction of calculations. In the examples the number of operations in the residual programs is decreased in as much as 16.9 times for TRM1, 15.3 times for TRM2, and 8.6 times for TRM3. A hidden inherent parallelism irrespective of a style of an algorithm description and a sequence of declarations in a program code is revealed. For these programs the same optimal scheme shown in Figure 9 is synthesized after partial evaluation. Reducing computational resources can be yielded by this technique too. The maximum number of processors used for evaluation of the original programs is reduced in the residual program in 4 times for TRM1, 2.22 times for TRM2, and 2 times for TRM3.

5 Experiments

An experimental programming system PARNET (Stepanov et al., 1996) provides facilities to conduct experiments with DI-structures. It includes: an ANM emulator, parallel garbage collector, scheme synthesis facilities, declarative programming language PARS (Stepanov and Lupenko, 1991), parallel computation process emulation facilities, scheduler, subsystem for the collection and visualization of modeling data, monitoring subsystem, archives of schemes support subsystem, and integrated development environment.

Experiments for tasks from various fields were carried out. These domains include LISP paradigms, operations on matrices, linear equations, sorting and combinatorial problems, graphs, electronic circuit's simulation (Stepanov et al., 1993), automatic program parallelization, and others.

Table 4 shows ratios between the T_1, T_p, P, C_p of original programs and residual programs for several algorithms: two variant of insertions sort *InsSort1* and *InsSort2*, matrix multiplication *Mmult*, quick *QSort* and merge *Msort1* sorts.

The results show the reduction of waste calculations. For instance, the number of operations in the residual programs is decreased in as much as 24.39 times for *InsSort1,* and 31.62 times for *InsSort2*. Such an optimization impacts on the decrease of parallelism, and hence, on an economy of computational resources. The number of processor elements is decreased in 27.20 and 12.55 times for InsSort1 and InsSort2 respectively, without limitation of their parallelism. However, the speedup of the programs is increased in 2.25 and 2.68 times, and the cost is decreased by 61.20 and 33.60 times.

PARS		InsSort1	InsSort2	Mmult	QSort	Msort1
Program		(N=24)	(N=24)	(N=3)	(L=24)	(N=24)
R	size	21.95	31.86	0.276	0.21	0.038
A	operation	24.39	31.61	15.49	1.16	29.98
T	time	2.25	2.68	4.57	1.46	1.97
I	processor	27.20	12.55	4.44	0.79	19.49
O	cost	61.20	33.60	20.32	1.15	38.33

Table 4 Parallelization and performance analysis

Notice that parallelism of residual programs is revealed automatically and no pointed out by a programmer. It should be emphasized that the process of partial evaluation is also parallel.

The partial evaluation can be used for the specialization of programs with respect to knowledge of a problem to be solved and part of the input data. Given programs are specialized with respect to the size of the problems.

The strategy is expected to help to avoid unnecessary computation and to make computation more effective.

We have considered the programs, which are special, in that they are for the most part *data-independent*, meaning that the parallelism of operations to be evaluated is independent of the actual values being manipulated. The programs are "well specialized" because they tend to be mostly *size* dependent. In a given examples a parallelization is mostly depended on a vector or matrix range that is static (known) input.

Table 4 demonstrates also results for the specialization and parallelization of the list quick sort algorithm. *Qsort* is well-known *data-dependent* algorithm. For such programs partial evaluation can not eliminate waste computations in a great degree. The program is specialized with actual value of list length, but mostly computation of the task depends of "pivot" values. The number of operations is decreased only in 1.16 times, speedup is increased in 1.46 times, the cost is decreased in 1.15 times.

6 Conclusions and Related Works

We describe the Abstract Network Machine based on parallel computation with DI-structures, and the declarative program parallelism extraction method based on parallel dynamic partial evaluation.

We show that partial evaluation plays an important role in the parallel computation process. This approach is intended for a broad spectrum of activity such as automatic transforming, optimizing, specialization of programs with respect to the partial knowledge of the input, and for their parallelization.

We demonstrate through the analysis of the program examples the way to partially overcome some shortcomings and non-effectiveness of declarative programs, and show that the method is particularly effective on numerically oriented scientific programs, since they tend to be mostly data-independent. It is also suitable for the problems where irregular data structures such as lists, trees, graphs etc. are involved and manual optimization and revealing of the parallelism makes difficulties. In some cases the technique provides for reduction of waste computation, and exposes "the hidden natural parallelism" of the programs irrespective of a programming style and a sequence of declarations in a program.

As in many declarative languages the concepts of a variable, value assignment, explicit sequential or explicit parallel control flow are not used (Cole, 1992, Skillicorn, 1994). The network transformation resembles a parallel graph reduction (Darlington et al. 1987) known in functional programming.

Many authors consider graph reduction machines and mechanisms for implementation of functional languages, lazy evaluation, lambda calculus (Peyton et al. 1987, Kumar et al., 1995). Known reduction models represent a program as a λ-graph (Gupta et al., 1989), task, or partial task graph (Kumar et al., 1995). In the ANM a program is represented as a graph of DI-structures named an associative network. The essential difference of the transformation of the associative network in comparison with graph reduction lies in the network-unification but not substitution of the argument with the appropriate value in a course of the reduction.

The incomplete-data-driven strategy of the associative network transformation progresses an idea of the data-driven computation (Treleaven et al., 1982) for the case of fine grain data structures. The decision to undertake computation is based on the increment of the number of elements of the DI-structure or getting a more precise elements' description rather than on the availability of complete data as in a data-flow. Each new portion of information increasing the DI-structure description forces the computation.

The uniform representation of data and programs resembles functional programming. The implementation of the arithmetic resembles data-flow. Dealing with incompleteness increases parallelism, in a similar way as in *nonstrict function, lazy evaluation, lenient cons* (Amamiya and Hasegawa, 1984, Amamiya et al., 1983, Wei and.Gaodiot, 1988) or *I-structures* (Arvind etc., 1989). The changeable direction of the information flow (an argument of a relation becomes input or output depending on which information is ready first) is also used in logical programming.

Described approach to automatic program parallelization by means of dynamic partial evaluation fundamentally differs from the approaches taken by parallelizing compilers or partial evaluators (Surati and Berlin, 1994, Tchernykh, 1986). Our work bridges partial evaluation and parallel computation through the transformational approach to computation. It should be emphasized that all useful features of the ANM result from the single mechanism of the network-unification using just few concepts.

Work on modeling of the ANM with resource, topology, timing, and other constraints is in progress so that the actual parallelism, efficiency, overhead and other performance factors may be assessed. Many problems remain to be solved before such a machine can be of practical use in real environment.

Acknowledgement

Part of this work was supported by CICESE (Centro de Investigaci6n Científica y de Educaci6n Superior de Ensenada) under project #631101, by CONACYT (Consejo Nacional de Ciencia y Tecnología de México) under grant #1902P-A., and by Russian Fund of Basic Research.

References

Amamiya M, Hasegawa R, Mikami H. "List Processing with a Data-Flow Machine". *Lecture Notes in Computer Science*, 147, 1983, 165-190.

Amamiya M, Hasegawa R. "Data-flow Computing and Eager and Lazy Evaluations". *Computing,* 2(2), 1984, 105-129

Arvind, Nikhil R. Pingali K. I-structures: "Data structure for parallel computing", ACM Transactions on Programming Languages and Systems, 11(4): 598-632, 1989.

Bjorner I.D., Ershov A.P., Jones N.D. "Partial Evaluation and Mixed Computation", *North-Holland,* 1988.

Cole M. "Parallel Software Paradigms". In L.Kronsjo, D.Shumseruddin, ed. Advances in Parallel Algorithms. Halsted Press, New York, 1992, 1-25.

Consel C. Khoo S. "Parameterized partial evaluation". In *ACM SIGPLAN Conference on Programming Language Design and Implementation,* p. 92-106, 1991

Darlington J., Cripps M., Field T., Harrison P., Reeve M. "The design and implementation of ALICE: a parallel graph reduction machine". In S.S.Trakkan, editor, Selected Reprints on Dataflow and Reduction Architectures, IEEE Computer Society Press, 1987.

Dijkstra E.W. "Guarded commands, nondeterminacy, and formal derivation of programs". *Comm. ACM, 18, 1975, 8, 453-457.*

Ershov A.P. "Mixed computation: potential applications and problems for study". *Theoretical Computer Science,* 18, 1982.

Flynn,M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 94, 1972.

Gao Guang R., Bic Lubomir, Gaudiot Jean-Luc eds. "Advanced Topics in Dataflow Computing and Multithreading". *IEEE Computer Society Press,* Los Alamitos, CA, 1995.

Gupta J.P., Winter S.C., and Wilson D.R.. "CTDNet - A mechanism for the concurrent execution of lambda graphs". *IEEE Trans. Software Eng.* 15, 1989, 1357-1367

Jones N., "An Introduction to Partial Evaluation". ACM Computing Surveys, vol. 28, N3, p.480-503, 1996

Jones N., Gomard C., and Sestoft P. "Partial Evaluation and Automatic Program Generation". Prentice-Hall, 1993

Jones N., Sestoft P., and S□ndergaard H. "MIX: a self-applicable partial evaluator for experiments in compiler generation". *LISP and Symbolic Computation,* 2(1):9-50, 1989

Jørgensen I. "Generating a compiler for a lazy language by partial evaluation". In *ACM Symposium on Principles of Programming Languages*, p. 258-268, 1992

Kotov V.E., Narinyani A.S.. "On transformation of sequential programs into asynchronous parallel programs". *Proc. IFIP Congress, Edinburg, 1968, North-Holland , 1969, 351-357.*

Kumar P., Gupta J.P., Winter S.C. "CTDNET III - An Eager Reduction Model with Laziness Features". In J.R.Davy, P.M.Dew, editors, Abstract machine Models for Highly Parallel Computers. Oxford, 1995, 103-117.

Peyton Jones S.L., Clark C., Salkild J., and Hardie M., "GRID – A high-performance architecture for parallel graph reduction" Processing of 1987 Functional

Programming Languages and Computer Architecture Conference. Springer-Verlag LNCS 274, pp. 98-112, 1987.

Sesyoft P., Sondergaard H. editors. "*Special Issue on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'94)*". *(Lisp and Symbolic Computation, vol. 8, no. 3)*, 1995

Skillicorn D.B. "Foundation of Parallel Programming", *Cambridge International Series on Parallel Computation,* 6, 1994.

Skillicorn D.B. "Model for Practical Parallel Computation", *International Journal of Parallel Programming*, 23 (2), 1991, 133-158.

Stepanov A., Tchernykh A., Lupenko A., Tchernykh N. "Parallel Computations on Associative Networks". *MPCS'96 Second International Conference on Massively Parallel Computing Systems, IEEE Computer Society Press*, 1996

Stepanov A.M. "Parallel Computation on Associative Networks", *Preprint, AS USSR.' Institute of Precise Mechanics and Computer Technology*; 2, Moscow, 1991, 53 (In Russian)

Stepanov A.M., Tchernykh A.N., Tchernykh N.G. "Parallel Computations on Associative Networks in Application to the Logic Modeling Problem". *Non Conventional Supercomputers*, Moscow, 2, 1993, 23p.

Stepanov A.M., Lupenko A.I. "Programming for ANM". *Preprint, Institute of Precise Mechanics and Computer Technology of RAS*; 3, Moscow, 1991, 53p.

Surati R. and Berlin A. "Exploiting the Parallelism Exposed by Partial Evaluation". MIT A.I. Memo No. 1414a, May, 1994

Tchernykh A., Stepanov A., Lupenko A., Tchernykh N. "Extraction and Optimization of the Implicit Program Parallelism by Dynamic Partial Evaluation" - *pAs'97 The Second Aizu International Symposium on Parallel Algorithms/Architecture Synthesis,* p. 322-338, *IEEE Computer Society Press*, 1997

Tchernykh A. Dynamic "Partial Evaluations and Automatic program parallelization" *Preprint,* Institute of Precise Mechanics and Computer Technology of RAS; Moscow, 1, 1986, 41p.

Treleaven P.S., Brownbridge D.R., Hopkins R.P. "Data-driven and Demand-driven Computer Architecture", *Computing Surveys*, 14, 1, 1982, 93-143.

Wei Yi-Hsiv, Gaodiot J. "Lazy evaluation of FP Programs: A Data-Flow Approach". *Proc. of the Int. Conference on Fifth Generation Computer Systems*, 1988.

Von Neumann, John. 1946. "The Principles of Large-Scale Computing Machines", reprinted in Ann. Hist. Comp., Vol. 3, No. 3, pp. 263-273

***Andrei Tchernykh** graduated from the Sevastopol Instrument Making Institute, USSR, in 1975. He received a Ph.D. degree in computer science from the Institute of Precise Mechanics and Computer Technology of the Russian Academy of Sciences RAS, Russia in 1986. From 1975 to 1995 he was with the Institute of Precise Mechanics and Computer Technology of the RAS, Scientific Computer Center of the RAS, and at Institute for High Performance Computer Systems of the RAS, Moscow, Russia. He took part in parallel supercomputer ELBRUS project development and implementation. Since June 1995 he has been working at Computer Science Department at the CICESE Research Center, Ensenada, Baja California, Mexico. He participated in and led number of Russian, and CONACyT research projects.). Dr. Tchernykh has been a member of the Program Committee for several professional conferences and served as general co-chair for International Conference on Parallel Computing Systems. He is a coordinator of the Parallel Computing Laboratory of CICESE. His main interests include parallel and cluster computing, incomplete information processing, partial evaluations, and declarative parallel programming.*

***Andrei Stepanov** graduated from the Moscow Lomonosov State University in 1958. In 1967 he received his Ph.D. degree. His work was connected with developing new computer architectures and CAD. Since 1990 up to now he has been working as the Head of Parallel Computation Laboratory at Scientific Computer Center of RAS. His scientific interests are connected with artificial intelligence, non-traditional computations and new computer architectures.*

***Antonio Rodriguez Diaz.** is associate professor in Universidad Autonoma de Baja California (UABC). He works in his Ph.D. thesis in the area of non-deterministic dynamic scheduling for parallel computation.*

***Isaac D. Scherson** is currently a Professor in the Deptartments of Information and Computer Science and Electrical and Computer Engineering at the University of California, Irvine. He received BSEE and MSEE degrees from the National University of Mexico (UNAM) and a Ph.D. in Applied Mathematics (Computer Science) from the Weizmann Institute of Science, Rehovot, Israel. He held faculty positions in the Dept. of Electrical and Computer Engineering of the University of California at Santa Barbara (1983-1987), and in the Dept. of Electrical Engineering at Princeton University (1987-1991). Dr. Scherson has been a member of the Technical Program Committee for several professional conferences and served as Workshops Chair for Frontiers'92. He is the editor of the book that resulted from the Frontier's 92 workshop and a co-editor of an IEEE Tutorial on Interconnection Networks. He was the Guest Editor of the Special Issue of The Visual Computer on Foundations of Ray Tracing (June 1990.). Dr. Scherson is a member of the IEEE Computer Society and of the ACM. Since July, 1992, he has contributed to the IEEE as member of the IEEE Computer Society Technical Committee on Computer Architecture. His research interests include concurrent computing systems (parallel and distributed), scalable server architectures, scheduling and load balancing, interconnection networks, performance evaluation, and algorithms and their complexity.*