ABSTRACT NETWORK MACHINE:

PARALLEL COMPUTATION ON ASSOCIATIVE NETWORKS WITH INCOMPETE DATA STRUCTURES

ANDREI TCHERNYKH

CICESE Research Center, Ensenada, Mexico.
chernykh@cicese.mx.

ANDREI STEPANOV

Institute of Informatics Problems of the RAS, Moscow, Russia.
ipiran@ipiran.ru

ALEXANDER LUPENKO

Institute of Informatics Problems of the RAS, Moscow, Russia.
ipiran@ipiran.ru

NATALIA TCHERNYKH

The Institute of Multiprocessor Computer Systems of the RAS, Moscow, Russia.
ntcherny@cicese.mx

Colaboración especial para la Revista Dyna Nro. 140.

RESUMEN: En este articulo se describe un modelo de computación paralela denominado máquina abstracta de red (ANM). Nos centramos en el papel que juega la evaluación parcial en la transformación, optimización y especialización de programas ANM, y el revelado automático de su paralelismo inherente La técnica se basa en computación paralela sobre redes asociativas implementadas en ANM. ANM usa redes asociativas tanto para representar tanto información completa como información incompleta. ANM sigue el estilo de transformación paralela de la computación basada en un proceso de unificación de red, el cual es una variante del mecanismo de reducción de grafos para computación paralela granular y para el procesamiento de información incompleta. Se da una breve descripción de esta aproximación, así como también una comparación con el esquema tradicional de evaluación parcial. Adicionalmente, se puntualiza sobre algunos de los problemas asociados con esta clase de computación paralela.

PALABRAS CLAVES: Modelos Paralelos, Paralelismo Implícito, Programación Declarativa, Evaluación Parcial, Procesamiento de Información Incompleta.

ABSTRACT: In this paper, we describe a model of parallel computation named Abstract Network Machine (ANM). We focus on the key role played by partial evaluation for the transformation, optimization, specialization of ANM-programs, and for automatic reveling of their inherent parallelism. The technique is based on parallel computation on associative networks implemented in ANM. The ANM uses associative networks for representing both complete and incomplete information. It follows parallel transformation style of computation based on a network unification, which is a modification of a graph reduction mechanism for fine grain parallel computation and processing of incomplete information. A brief description of this approach is given, as well as a comparison with the traditional approach to partial evaluation. We point out some of the problems associated with this kind of parallel computation.

KEYWORDS: Parallel Model, Implicit Parallelism, Declarative Programming, Partial Evaluation, Incomplete Information Processing.

1. INTRODUCTION

The main problem and main discomfort for designers of software applications for parallel and distributed systems is the identification of potential internal parallelism of the problem and its solution, and to use explicit parallel programming techniques in order to meet it most adequately.

There are several trends in parallel computing to relieve programmers' mind.

The first one is to find better human-parallel-computer interface that offers more help for the programmer on the stage of the identification of potential parallel activity, partitioning of a problem into parallel tasks, and producing a completely explicit description of the tasks available, communication and synchronization between them. It means also increasing the level of programming languages. It is considered that the fewer details about computational process, about computer resources needed to solve a problem, the programmer has to specify in his program, the higher the level of the languages is. In the languages with explicit concurrency the programmer is obliged to describe not only precise sequence of computational actions but their parallelism as well. That is why the majority of the parallel programming languages has a lower level than corresponding sequential languages.

The languages with an essentially higher level got some progress as well. Here we underline, first of all, the non-procedural languages such as Pure Lisp, Prolog, FP, SISAL, data-flow, and another declarative languages. A program written in one of these languages is a set of declarations (statements, functions, facts, etc.) describing notions of the object domain in terms of some formal system, for example, first order predicate logic. The computational process is a kind of an inference process based on rules adopted in this formal system. This leads to giving up the notion of algorithm (description of What and How to do something is substituted by description of only What to do). It is now argued that such languages could play an important role in a development of a parallel application. But it is necessary to emphasize that the real situation in this research area is far from ideal in practice.

Implicitly parallel paradigms are based on the premise that it is best to write programs in a declarative form and then allow a compile-time or run-time analysis to extract all available parallelism. The enormous diversity of architectures, together with the specific problems of parallelism, makes the development of an efficient parallel program difficult. Moreover, the compile-time analysis of a program does not necessarily reflect the problem the program is solving. In addition, declarative programs also impose an extra cost in the form of different overhead and waste computation. Parallelism can be extracted during run-time through recursion, nesting, and looping. However, because the depth of recursion or the number of times a loop is iterated is usually data dependent, the compiler frequently cannot determine the parallelism.

In the next sections, we show that processing of incomplete information and, in particular, partial evaluation can be seen as a way out of this problem.

1.2. Processing of incomplete information

Another trend is not directly connected with parallel computation models, but the impact of it on software development methodologies is close. We mean processing of an incomplete information and, first of all, partial evaluation (*Bjorner et. al*., 1988; *Ershov*, 1982). The partial evaluation method was shown to be a universal technique for a program transforming, optimizing, translating and even parallelizing.

Let's consider a function $F(x)=H(G(x),K(x))$ (Figure1). Given x, the functions G, K can be evaluated in parallel. Evaluation of the function H can be started only after the values $G(x)$ and $K(x)$ are fully determined. It is impossible for the value $F(x)$ to be used in another evaluation before it has been fully determined (Figure1a). This process, though good because it is parallel to some extent, seems much less attractive, when compared with the process of computation in an analog device. One could notice that the process of forming the potential of input signal *x* forces forming, after a short delay, the potentials of *G(x), K(x),* and they, in turn, force forming *H(x).* The functions *G, K, H* work in parallel (Figure 1b).

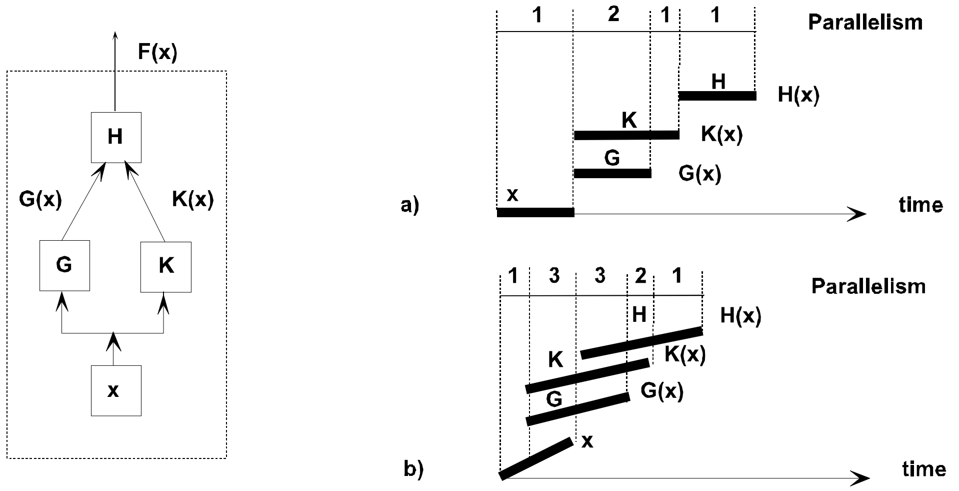


Figure 1. Parallelization of function *F(x)=H(G(x),K(x)).*

We can see that there is one more approach for increasing the level of the parallelism: a manipulation with an incomplete information. Most of parallel models (*Skillicorn*, 1991) leave unchanged the two concepts of sequential computation: operator and operand (Tabla 1), and the principle of the so-called computational act. Latter means that to start the execution of an operator all operands required by it must be fully computed and accessible. In spite of the fact that a value of an operand is determined gradually, for example, in a arithmetic unit a sign of a number is usually determined first, then its order and mantissa, it can be used only after the moment when this value is fully determined and is written to a corresponding memory cell, to a data-flow token, or when a bit of readiness is set.

It seems like the value is calculated instantly as a result of one "indivisible" computational act. This “forces” the computation to be sequential. Such a value becomes one more “bottleneck”. 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 computation. For instance, *nonstrict function, fuzzy arithmetic, lenient cons* (*Amamiya and Hasegawa*, 1984, *Amamiya et. al.*, 1983, Wei and Gaodiot, 1988) or *I-structures* (*Arvind and Thomas*, 1980) are approaches to increase parallelism in such a way.

1.3. Partial evaluation

Partial evaluation 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. If all inputs are defined, then the result of partial evaluation is output data of the program and an empty residual program, since initial algorithm is carried out completely. In this case functioning of a partial evaluator is similar to that of a usual interpreter. Partial evaluation may be considered to be a generalization of the usual evaluation (*Ershov*, 1982). Introduction of such a generalization appeared to be very fruitful and allow studying of various programming concepts from the new point of view. Although simple in concept, partial evaluation has been used in such areas as compilation, scientific computing, computer graphic, pattern matching and others (*Bjorner et. al.*, 1988; *Jones et. al.*, 1989; *Consel* *and Danvy*, 1989; *Consel and Danvy*, 1991; *Jørgensen*, 1992; *Jones et. al*., 1993; *Sesyoft and Sondergaard*, 1995; *Lawall and Danvy*).

Such a computation is based on three separated mechanisms: evaluation usually implemented in a hardware, partial evaluation, and residualizing, which are either implemented as a generation extension of a language or as a partial evaluator.

Let's consider another way to implement partial evaluation. In (*Ershov*, 1982) a concept of a *program transformation* by a *transformation machine* was introduced. Transformations, i.e. a substitution of some syntactic or semantic construction of a program by another one (possibly simplified) is considered as an instruction set of such a machine. The instructions not only do some program evaluations, but also change (simplify) the program itself. If the initial input data are completely determined, then a sequence of transformations reduces the program to the only operator of the result output. If a part of the input data is not available (*dynamic*), then at some step no transformation can be applied and the program becomes a residual program, which is ready to continue evaluations, as soon as dynamic inputs are available. Thus, in the transformation machine, there is *no difference* between evaluation and partial evaluation. Moreover, the process of transformations appears to be parallel and non-deterministic, since under some circumstances the possible transformations can be performed in an arbitrary order. Unfortunately, when we deal with transformations of sequential programs, based on von Neumann computation with an assignment operator and side effect, highly elaborate global transformations appear. The transformational principle of evaluation seems to be much more natural when applied to declarative programs. Because of absence of an assignment operator, GOTO, and side effect, the transformations become entirely local. The process of transformation is, in fact, similar to the process of a reduction widely known in functional programming (*Kumar et. al*., 1995; *Gupta et. al.*, 1989), when all data are available.

The results obtained in partial evaluation and transformational computation are mostly associated with fundamental properties of *sequential computation.* We believe that they can be also included in *parallel computation* by a natural way.

The evaluation of an "incomplete" input data plays a key role in partial evaluation. On the other hand the possibility to manipulate incomplete values like a *I-structure*, *compound object* is an additional source of parallelism (*Amamiya et. al*., 1983, *Wei and.Gaodiot*, 1988, *Stepanov et. al*., 1996).

The evaluation of *incomplete information* is considered in the PARNET project (Stepanov, 1991). Partial evaluation, when the incompleteness of the information is only restricted to the dynamic part of program inputs, is a particular case of such evaluation. In the PARNET the incompleteness is understood in a broader sense. For instance, a relation with unknown or flexible arity, underdetermined program, soft function, or a matrix with unknown elements and/or unknown range, and an incomplete structure can be represented and manipulated.

In the present paper we describe a model of computation named Abstract Network Machine (ANM). We consider the key role played by partial evaluation for the parallel transformation, optimization, specialization of declarative ANM-programs, and for their automatic parallelization. The technique is based on parallel computation on associative networks implemented in ANM. The ANM uses *associative networks* for representing both fully determined and incomplete information (program-like and data-like). It follows the parallel transformation style of computation and is, in fact, a run-time parallel partial evaluator. ANM does not separate three mechanisms: evaluation, partial evaluation, and residualizing. All of them are an effect of a single transformation mechanism, namely network unification, which is modification of a graph reduction mechanism for fine grain parallel computation and processing of incomplete information.

We discuss the results of the several experiments and point out some of the advantages and problems associated with this kind of parallel computation.

This paper is organized as follows. Section 2 briefly describes the main concepts of ANM, the principles of parallel computation on associative networks, and the concept of transformational computation. Section 3 discusses programming in PARS and the process of parallel partial evaluation in the ANM. The remaining sections present examples of using the ANM for the automatic parallelization and optimization of declarative programs.

2. ABSTRACT NETWORK MACHINE

2.1. Network representation of information

The basis for data representation in ANM is the *associative network,* which is a directed graph of a special kind with labeled edges. The nodes of the network are called *objects*. The tree types of objects are introduced: a) *simple* objects or *atoms*; b) *compound* objects, and c) *empty* objects (Figure 2).

Atoms represent those objects that have no internal structure at a given level of consideration. For example, the atom can be integer, character, or Boolean values. The atom is fully defined by its representation. Two atoms with the same representation are not distinguished. The atom is a static object of the network, its description can not be changed during the computation, whereas the compound and empty objects are referred to as dynamic objects.

A description of the compound object is a set of attributes. Each attribute is a pair *[A X]*, where *A* is the attribute's name (some identifier, or a positive integer), and *X* is the attribute's value (some other object of the network).

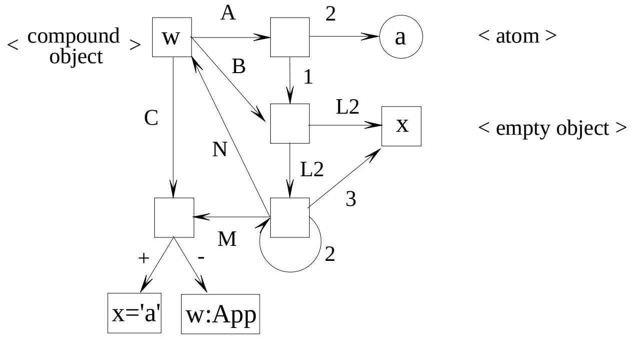


Figure 2 Associative network.

The attributes can be interpreted in different ways. We give here some examples of such interpretations:

a) The attribute *X[A Y]* of the object *X* is a *property* of *X*, where *A* is the name of the property, *Y* is the value of the property. The description of the compound object is the list of its properties;

b) The attribute *X[A Y]* of the object *X* is a relation of an *inclusion* of *Y* as a part of *X*. *Y* is an element of a structure *X*. *A* is a "local name" (or a number) of the *Y* in the structure of *X*;

c) The compound object represents a fact of an existence of some *relation* between objects. The attribute *X[A Y]* of such an object signifies, that *Y* is one of the arguments of the relation *X*, and *A* is the name (or the number) of this argument.

Other ways of their interpretation, related with such terms as a parameter, octant, role, slot, etc. are existed. From the ANM point of view all these ways of interpretation are indistinguishable, since all of them are united by one property: the name of the attribute unambiguously determines the attribute's value. Hence, there is one limitation on the compound object description: it cannot contain attributes with the same names. The number of the attributes can be dynamically increased in the course of computation. Hence, the description of a compound object is considered as incomplete in any moment. Several degrees of object incompleteness can be underlined, namely, 1) the fact that its existence is only known, 2) some attributes are available, but the full description from the point view of programmer (e.g. all elements of the concrete range matrix) is unknown, and 3) the total structure is known, while some of its elements are unknown (that is similar to I-structure). So the term "incompleteness" is understood in a wide sense. For example, a relation with unknown (flexible) arity, or a matrix with unknown elements and/or unknown range can be represented by compound objects and manipulated in ANM.

The empty object cannot be classified as the atom or as the compound object, because we do not know anything about it, except that it exists. It does not have any description.

An important feature of the system is homogeneity in respect to the representation of the objects and of the relations between them: on formal level they are not distinguished.

Special objects and attributes are introduced for the implementation of some built-in mechanisms such as arithmetic or logic.

There are three kinds of assertions: those describing objects *[A y]*, those stating the identity of two objects *x=y* (network-unification), and those stating the similarity of an object to a scheme *x:S* (scheme-similarity). It is possible to express recursive assertions, and to make conditional assertions.

The *n-unification* establishes the identity of two objects. It leads to the transformation of the associative network "gluing" these two objects and merging their attributes. The very simple transformation rule is correctness-preserving. If during this merging two unified attributes appear to have the same names, one of these attributes is removed and a new n-unification of their attribute's values is generated.

S-similarity. The concept of a program or procedure as a control-flow description is not used in ANM. The corresponding analogue is a scheme. The scheme is an isolated network that could be understood as a description of some standard data structure or relation. The s-similarity *x:S* declares, that the object *x* is an specific instance of *S*. Hence, the object *x* and the copy of the scheme are identical, that is $x=S^1$.

It resembles to a process of a reduction widely known in functional programming, where a common implementation technique is to represent a functional program by a graph, which is evaluated by successive transformations to its nodes. The evaluation of a function-node amounts to rewriting (reduction) its representation, according to the definition of the function. That is copying of the function definition and substitution of the arguments in the definition on corresponding values presented when the call is made. In such a ``graph reduction'' implementation, the order in which the transformations are applied to the nodes is irrelevant. The transformations can therefore occur concurrently, and as a result it is often claimed that functional languages are inherently parallel without the need for explicit language constructs for expressing parallelism.

The essential difference of the transformation by the s-similarity lies in the n-unification but not substitution of the argument with the appropriate value. N-unification leads to reduction of the network "gluing" the “argument-object” and the “value-object” together and merging their attributes. Both objects can be defined partially. The system makes all inference it can, based on the supplied information. Descriptions of the objects could complement each other or could be equal. The object-relation and its definition by the scheme can also have different arity (in our case, different number of attributes). It could also be related with several definitions, hence it can be s-similar to several schemes, which complement each other.

Just this difference, in fact, determines main features of the ANM and ability for partial evaluation inherent in parallel computation associative networks.

The operations can be conditional and stored in sets that are values of IF and THEN attributes:

if *x* then *y=z*, *x=w* else *w:S*, *y=k*;

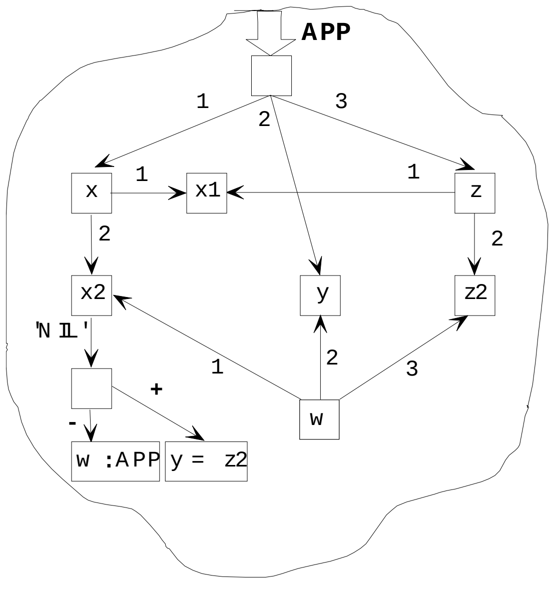


Figure 3. Representation of the relation ‘Append’.

3. ANM MODEL

The ANM model consists of the main network memory, the scheme memory, the operation memory (request queue), and a set of processing units for executing the operations.

3.1. Transformational principle of computation

The computational process consists of the local transformations of the network. The execution of the n-unification leads to gluing of two objects and merging their attributes. It can determine the description of the objects more precisely, that, in its turn, can generate several operations of the n-unification and s-similarity. The s-similarity *x:S* forces copying the scheme *S* from the scheme memory into the network memory followed by the n-unification of the object *x* with the object *S'*. *S'* is a copy of the entry of the scheme. Besides, all the operations contained in the local scheme memory, if they exist, are copied and added to the contents of the operation memory. It is possible to execute the operations in an arbitrary order, for example, in parallel. Any operation can be delayed for any period of time. More details can be found in (*Stepanov et. al.*, 1996; *Stepanov,* 1991; *Tchernykh et. al.*, 1997).

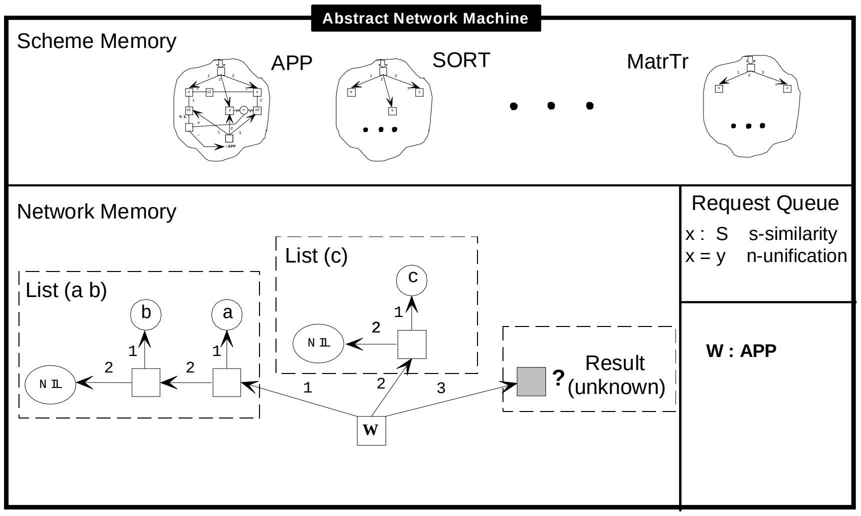


Figure 4. Task.

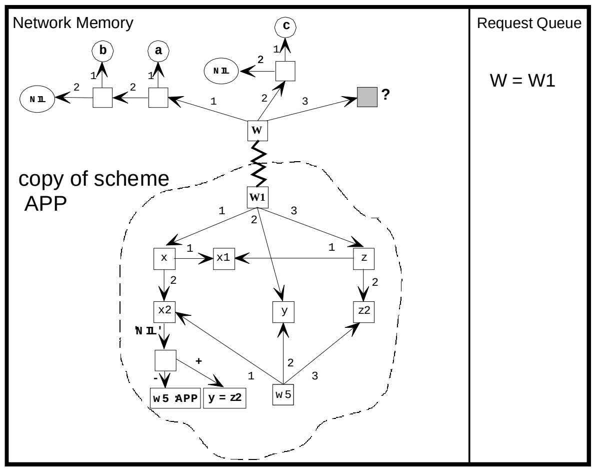
The program APPEND (Figure 1A in Appendix) describes the relation APPEND. "Entry" of the corresponding scheme *APP* (Figure 3) is a compound object, marked wide arrow, which describes the relation, with three attributes, whose names are *1, 2, 3,* and whose values are three objects *x, y, z*. That objects represent three lists (two ones are the lists to be appended, a third one is the resulting list).

In contrast to a procedure or function, an ANM scheme represents a relation. If it is described adequately, it could be multidirectional. In the case of APPEND, if the first and resulting lists are given, the second list will be built by the same scheme. In other words, a list subtraction will be performed.

For appending any particular lists the compound object *W* (Figure 4) is placed into main network memory. The structure of this object may include first list, second list, and resulting list (empty object, marked as the result).

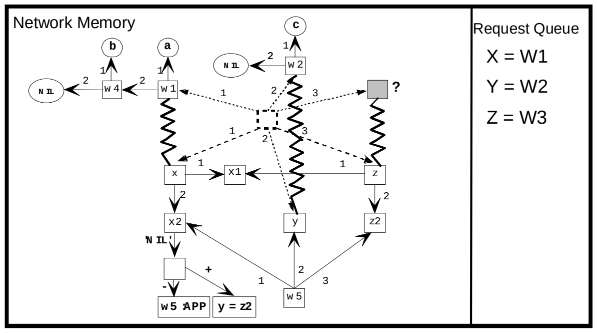
The operation of the s-similarity is placed into the operation memory. This operation points to the compound object *W* and the scheme *APP*. It means that *W* is the instance of the standard structure described by scheme *APP*. We say, *W* is similar to the *APP*. The object *W* and s-similarity *w:APP* is a task for the ANM to append given lists.

The ANM processing unit reads this s-similarity operation from operation memory and executes it. The scheme *APP* is copied from the scheme memory into main network memory, and the operation of the n-unification (*W=W1*) to be put to the operation memory is generated.

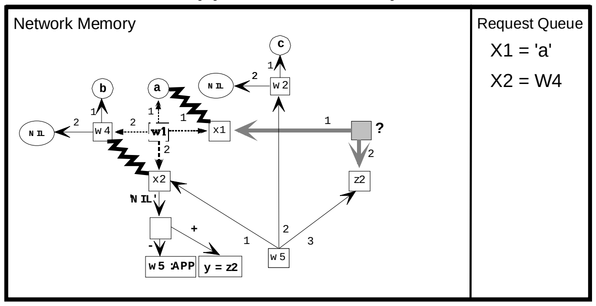


(a) After the 1-st step

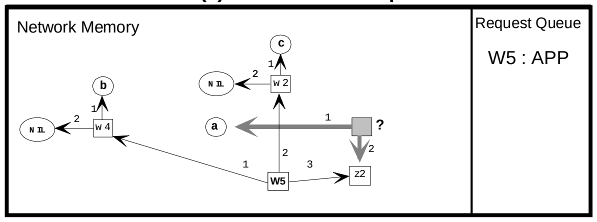
Figure 5a . Transformational principle of computation.



(b) After the 2-nd step

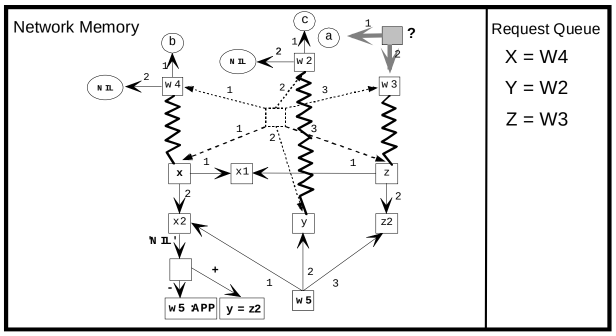


(c) After the 3-rd step

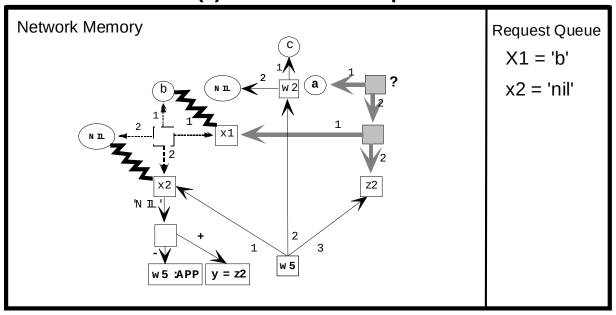


(d) After the 4-th step

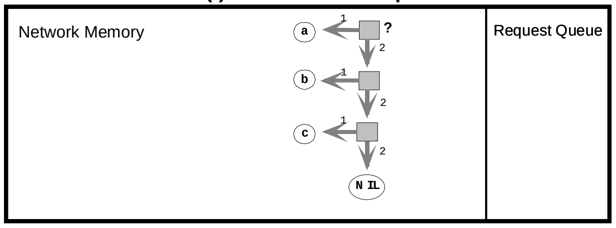
Figure 5b, c, d. Transformational principle of computation.



(e) After the 6-th step



(f) After the 7-th step



(g) Result

Figure 5e, f, g. Transformational principle of computation

It initiates the process of the network transformations (Figure 5). In Figure a zigzag line represents a operation of the n-unification.

The result is a compound object that represents the structure of the resulting list (Figure 5g). We can see that this result becomes more complete already after the third step (result is marked by the gray lattice). It could be used for another evaluation before the moment it is fully computed. Dashed lines show useless objects (garbage).

4. EXPERIMENTAL PROGRAMMING SYSTEM "PARNET"

On the base of ANM the experimental programming system PARNET was developed and implemented. It includes: ANM emulator, parallel garbage collector, scheme synthesis facilities, parallel computation process emulation facilities, subsystem of modeling data collection and visualization, declarative programming language PARS, monitoring subsystem, archive of schemes support subsystem, integrated development environment.

4.1. Structures and relations

As mention above the concept of a program is not used in ANM. An analogue of the program (procedure) is a scheme. The scheme is an isolated network, which is some standard structure description. Any relation can be expressed in terms of some standard structure. ANM treats all the schemes as standard structures regardless to how we interpret them as structures or as relations. Any network obtained in the course of computation can be used as a scheme.

4.2. Operations

An "operation" in ANM is, in fact, a request to the ANM to perform some actions. The operations can be executed in arbitrary order, or in parallel. There are only two kinds of operations: network unification, named *n-unification* $X=Y$, and scheme similarity, or simply s-similarity $X{:}S$

The n-unification can be considered as a declaration, that two objects X and Y are in fact one object, in other words, that they are identical. The execution of the n-unification leads to "merging" of these two objects. S-similarity declares, that the object X is an instance of some standard structure S. It leads to copying S and the unification of X with the copy of scheme entry S'.

Facilities to express conditional and recursive assertions are introduced. If we consider some structure as a relation, we can consider the s-similarity as an analogue of a procedure call in usual computers.

4.3. Programming language PARS

PARS (*Stepanov* and *Lupenko*, 1991) deals with objects. Unlike the traditional languages, no actions with objects are described. It 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 text seems to be a data description only, not a program description in a traditional sense. The program in PARS is a set of assertions where their order is not important; relations represented by objects are essentially multi-directional when their arguments can become definite in arbitrary order, and information flows through a relation in all directions.

5. PARALLEL PARTIAL EVALUATION IN ANM.

The transformational style of computation, together with an incomplete-oriented style of the information processing, leads to the effect of partial evaluation. All computations ANM performs can be considered to be "partial". Any network obtained is, in a sense, "a residual network" that is ready for further transformations when the incomplete information becomes more complete, or dynamic inputs become static. In fact, three computation mechanisms: evaluation, partial evaluation, and generation of a residual program are substituted by a single mechanism of n-unification.

5.1. Notation

We follow notation of (*Mogensen and Sestoft*, 1996) writing p for the program text, and $[p]$ for the function computed by p, or $[p]_L\ d$ to denote the result of running program p with input d on an L-machine, or $[p]_L\ [d_1, d_2]$ if we evaluate the program with respect to the two parts of the input.

A partial evaluator is a program *peval*, which performs partial evaluation, hence given a program p and a part d_1 of its input, produces a residual program p_{d1}:

$[peval]_L\ [p, d_1] = p_{d1}$ .It satisfies
$[p_{d1}]_T\ d_2 = [p]_S\ [d_1, d_2]$ for all d_2 or
$[[peval]_L\ [p, d_1]]_T\ d_2 = [p]_S\ [d_1, d_2]$,

where L is a language in which *peval* is written, S is a source language, and T is a target language.

5.2. Compiling by Partial Evaluation

A compiler *comp* generates a target program p' in the target language T from the source program p in the language S. The effect of running p on input d equals compiling source into target form $[comp]_L\ p = p'$, and then running the p':

$output = [p']_T\ d = [p]_S\ d$
$[p]_S\ d = [[\ comp\ ]_L\ p\ ]_T\ d$ for all d

An interpreter *int* for the language S, written in the language I, satisfies for any S-program s and input data d:

$[s]_S\ d = [int]_I\ [s, d]$

Let's partially evaluate the interpreter *int* with respect to a program s and unknown input d to that program.

$[peval]_L\ [int , s] = int_s$,
$[int_s]_I\ d = [int]_I\ [s, d]$,
$[int_s]_I\ d = [s]_S\ d$ for all d

Hence, the residual program int_s is equivalent to the source program s.

Formally, if the input and output languages of the *peval* are identical, S-*to*-T-compiler written in C is considered as an application of a T-partial evaluator written in L to the S-interpreter written in T which has the program s in S-language with unknown input d.

It suggests the possibility of a program *conversion* (*Jones*, 1996). Compiling S into a proper subset of S; translating direct style programs into continuation-passing style; translating lazy programs into equivalent eager programs (*Jørgensen*, 1992); and other transformations are provided by partial evaluation when *int* is a self-interpreter for S.

Moreover, this way of the transformations could be used when the models of computation or paradigms of the S and T languages are very different: recursive-iterative, declarative-imperative, sequential-parallel.

The parallelization is considered as a T-partially evaluation of the interpreter *int*, which is written in parallel language T, with respect to a sequential program s in the language S and unknown program input d.

$[peval]_L\ [int , s] = int_s$,
$[int_s]_T\ d = [int]_T\ [s, d]$,
$[int_s]_T\ d = [s]_S\ d$ for all d

The result of such a parallel program *int* specialization with respect to the sequential source s, is a target program written in the parallel language T, with the same input-output function as the sequential source program.

In later sections we have specified that S=T=*PARS*, and *peval* performs partial evaluation on declarative language *PARS*.

Traditionally, for a source program, two steps of evaluation could be applied to get a result of the program: *partial evaluation of a source program* and *evaluation of a residual program*. On the first step a partial evaluator *peval* is used to specialize this program (source to source) with respect to the part of its available inputs, i.e. static data. The second step is to run the residual program after compiling with the rest of the input data, and to yield the result. Offline partial evaluation includes the binding-time analysis phase and specialization phase.

In the PARNET system the same technique is realized as a cycled phases of a runtime system application, because of *evaluation=partial evaluation* for the PARS language. Any phase is to specialize the program with respect to known on this phase information. So, for the PARS language (ANM-code) interpreter *int* is an empty program, in the case where we don't want to transform the style of the program or to have any *conversion*. Hence,

$[peval]\ [\ \varnothing\ , s\ ] = \varnothing_s = s^1$,
$[s^1]_S\ d = [s, d] = [\ s\ ]_S\ d$ for all d

Because of the uniform representation of the program *s* and the data *d*, evaluation of the [*s*, *d*] in ANM gives the same result as running $[s]_S$ *d.*

5.3. Parallelism

Two issues of parallelism are underlined, namely parallelism of a partial evaluation process, and parallelism of residual programs.

In the ANM several known kinds of the partial evaluation process parallelism can be distinguished in an implicit form: pipeline parallelism, parallelism of *s-similarities* ("functional parallelism"), parallelism of structures, scheme ("function argument") parallelism, and data-flow parallelism (*Stepanov*, 1991).

6 PARALLELIZATION OF DECLARATIVE PROGRAMS

6.1. Performance measurements

In this paper, we consider unit-time operations without communication and memory access delay. To measure the quality of parallel programs S_p (the parallelizability), E_p (the efficiency), C_p (the cost) are used. The term parallelizability (Quinn, 1994) 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. This definition can be misleading since a parallel program can contain extra operations to facilitate parallelization and can exaggerate the speedup, because it masks the overhead of the parallel program. But it can simplify the comparison of means intended to improve the program performance. Let

$$S_p = T_1 /T_p;\ E_p = S_p /P;\ C_p = P*T_p;$$

where T_1 is equal to the total number of operations taken by the program; T_p is the running time (unit-time steps) of the program when we use *P* processors; *P* is considered as the maximal number of operation running in parallel. Execution of the program is analyzed for unbounded parallelism on a machine where the number of processors is infinite and can grow as the size of the problems grows.

To measure the quality of optimization we use K^o (optimization coefficient), S^o_p (speedup coefficient), R^o_p (optimization resource coefficient), and C^o_p (optimization cost coefficient). K^o is the ratio between the numbers of operations taken by the computer executing the program before and after optimization. S^o_p is the ratio of the residual over the source time taken by the parallel computer. R^o_p is the ratio of the number of processors needed to execute the residual program over the one for the source program. C^o_p is the ratio between the costs of evaluating the programs before and after optimization. Thus,

$$K^o = T_1 /T_{1o};\ S^o_p = T_p /T_{po};$$
$$R^o = P/P_o;\ C^o_p = C_p /C_{po};,$$

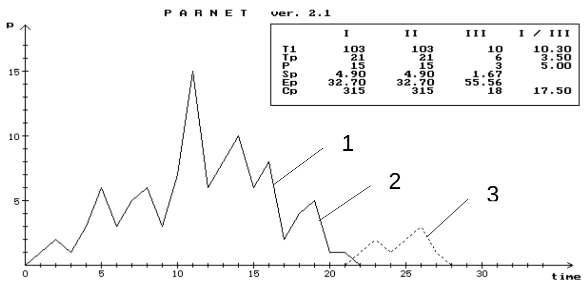
where $T_{1,}$ T_{1o} , T_p , $T_{po,}$ P, $P_{o,}$ $C_{p,}$ C_{po} are the total number of operations, running time, number of processors, and a cost of algorithm before (source) and after optimization (residual).

6.2. Optimization and specialization of a program

Let's consider an example of a matrix transposition. The scheme *Mtransp* in Figure2A in Appendix is the corresponding PARS-scheme for the transposition of square matrices of the range *N.* This example demonstrates a programming style without algorithmic description of obtaining *M2* by *M1*. It is the description of the bi-directional relation between *M1* and *M2*.

Figure6 shows the parallelism profiles for solving this problem for *N*=3 (It is a very small size, hence we can evaluate all the operations in such the simple program by hands if we want, and show a parallelism profile in more details). The parallelism profile 1 shows a calculation with available input data *N* and *M1*. Obviously, that the process, though parallel in high degree, is not optimal and includes waste computations.

How could we optimize this code? To do this, the ANM is able to use the partial evaluation technique. For example, our scheme, describing the standard relation between square matrices of the range *N*, can be specialized with respect to the actual range of matrices that we are going to use in several experiments. Profile #2 in Figure6 shows the process of *Mtransp* evaluation when only *N* is given. The residual scheme is automatically synthesized after the process is finished. The process of the task solving for the given matrix *M1* by the residual scheme is shown by the curve 3



```
          I        II       III    I / III
T1       103       103       10     10.30
Tp        21        21        6      3.50
P         15        15        3      5.00
Sp      4.90      4.90     1.67
Ep     32.70     32.70    55.56
Cp       315       315       18     17.50
```

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

This parallelism profile and the figures given in Table 1 (*Mtransp*) clearly show that revealing of “pure, useful” parallelism and economy of computational resources (as much as twice) are achieved by this technique. The number of operations, the running time, and the cost of the residual program are considerably smaller than those of the source *Mtransp* in 4.96, 4.25, and 8.50 times, respectively.

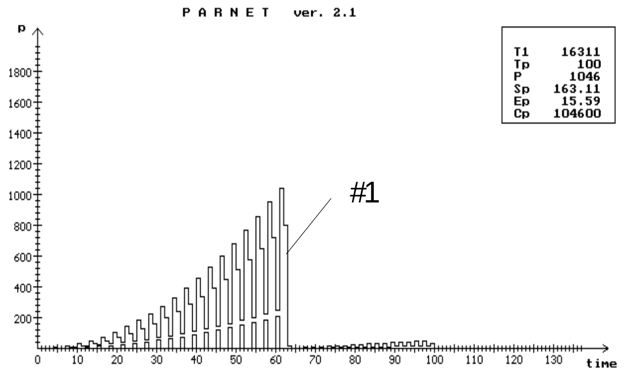
One can see that the scheme *Mtransp* is not an algorithm for obtaining matrix *M2* by *M1*. Both matrices take part in the relation *symmetrically*, so it is not necessary to give a fully definite matrix as its first argument, and an empty object, as its second one. With the equal success we can obtain *M1* by *M2* and *M2* by *M1*. Moreover, both matrices may be determined partially. Being based on the supplied information, the system will make all inference it can. Given *M1((-,'2',’3',('4',-,'6'),(-,'8'))* (some elements are missed and we don't describe the corresponding attributes), *M2((‘1’,-,‘7’),(-,‘5’),(-,-,‘9’)),* and *(M1, M2):TRANS*, the structure of both matrices will be defined completely (will complement each other). We can even supply both fully determined matrices. If it is a case, and computation is terminated, that confirms that the matrices are in the relation. If they are not in the relation, computation will stop because of contradiction (it will be found, that, for example, the ANM has to unify two different atoms).

*Data-independent task***.** Let’s consider the possibility of parallelization of programs that are sequential in nature (*Tchernykh et. al*., 1997). Figure 3A in Appendix is the set of schemes for two programs *InsSort1* and *InsSort2* described the same insertion-sorting algorithm in two different styles.

InsSort1 is based on a recursive description that is used to take an element of a vector, to sort recursively all the rest elements of the vector, and to insert the element to the sorted vector.

The scheme *InsSort2* is based on a similar recursive algorithm that is used only to insert each element of the vector *V1* to the sorted vector *V2*. Initially, *V2* is an empty vector

The schemes are different and their texts seem at first sight to be less elegant and efficient than equivalent *imperative* parallel programs. However, no elegant, “assembler” PARS level makes no secrete of the internal declarative representation of information in ANM Besides, the gain and optimization can be obtained by parallel partial evaluation.



```
T1     16311
Tp       100
P       1046
Sp    163.11
Ep     15.59
Cp    104600
```

Figure7 Parallel evaluation process of scheme InsSort1 (fully determined input data, N,V1).

There are no explicit facilities to describe the parallel or sequential control flow in PARS. Any n-unification and s-similarity assertions and the descriptions of objects in the scheme can be written in different order. For example, in the scheme *InsSort2* (Figure 3A in Appendix) the operations (n,i,p):*EQ, (V1,n,an): *ATTRIB, (n1,'1',n):*SUM, can be reordered.

Figures 7 and 8 show processes of computation for *InsSort1* and *InsSort2*. The pictures show the not-very-much-alike parallelism profiles of the programs' execution when all the input data *N* and *V1* are given.

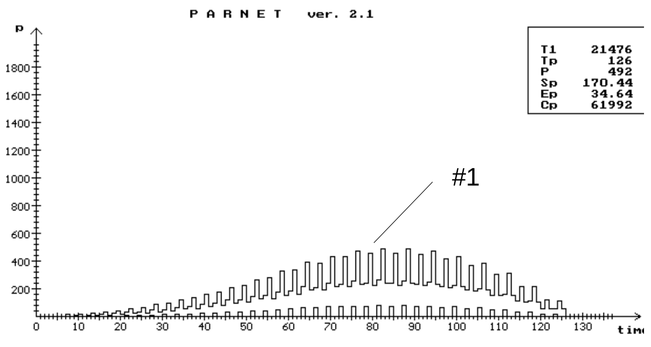


Figure 8 Parallel evaluation process of scheme InsSort2 (fully determined input data, N,V1).

If only *N* is available (vector *V1* is undetermined), the residual program ready for further execution can be synthesized. This program can yield the result as soon as the elements of *V1* are available.

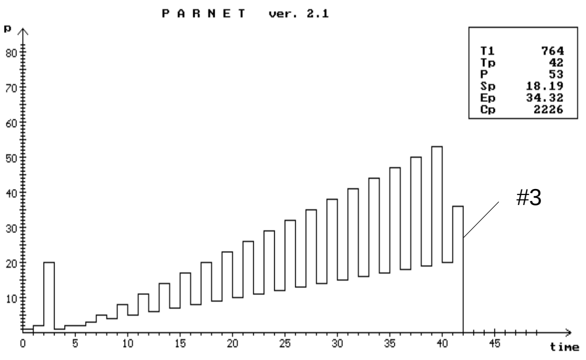


Figure9 Parallel evaluation process of the residual scheme.

Figure 9 is a parallelism profile of execution of the residual program with the same vector *V1*. The time required to sort a dataset of n elements for sequential insertion sort is Θ(n^2). The residual program has time complexity for worst-case Θ(2n+2) = Θ(n). Figure 10 shows the relative timings for each program.

The Table 1 summarizes evaluation of *InsSort1* and *InsSort2*. The figures show the different parallelism of the programs execution when all the input data *N* and *V1* are given. If only *N* is available (the vector *V1* is undetermined), the residual program ready for further execution may be synthesized in the end of evaluation of the source scheme. This scheme is network touted on incomplete elements of the input vector *V1* and elements of the vector-result *V2*. This "program" can yield the result as soon as the concrete elements of vector are given. The Table 1 also shows evaluation of the residual program parallelism with the same vector *V1*.

These very simple examples serve to show that partial evaluation: (a) 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; (b) can optimize programs by considerable reduction of calculations; (c) reveals in some cases an inherent parallelism irrespective of a style of describing an algorithm and a sequence of assertions in a text of a program.

Partial evaluation allows us to eliminate waste calculations. The number of operations is decreased in as much as 24.39 times for *InsSort1*, and 31.62 times for *InsSort2*. The programs speedup is increased in 2.25 and 2.68 times, the cost is decreased by 61.20 and 33.60 times. The equal residual parallel programs are synthesized (Table 1, *InsSort1/2-residual*).

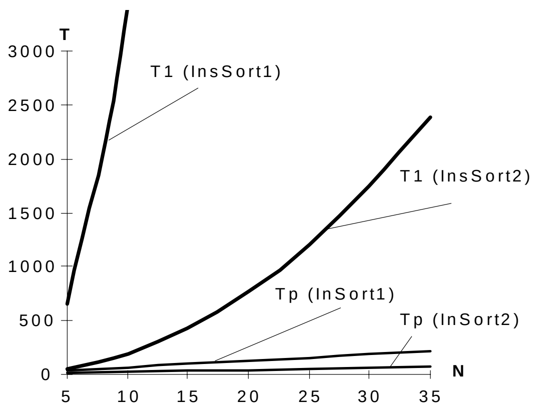


Figure10 Execution time of insertion sort.

The same way can be used for adaptation of standard programs to a particular problem. Corresponding optimal schemes are automatically synthesized in ANM. An increase in parallelism and economy of computational resources can be achieved by this technique.

The strategy is expected to help to avoid unnecessary computation and to make computation more effective. It should be emphasized that the process of partial evaluation is also parallel.

*Data-dependent tasks***.** 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 value being manipulated. The programs "are well specialized" because they tend to be mostly *size* dependent. In our particular case parallelization is mostly determined by vector or matrix range that is static (known) input data.

Let’s consider the task that is mostly data-dependent. The Table 1 gives some figures for the specialization and parallelization of the list quick sort algorithm *Qsort*. One can see that partial evaluation can not eliminate waste computations in *Qsort* in a great degree. The program is specialized with actual value of list length, but mostly computation of this 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.

Nevertheless the program parallelism automatically exposed is high when the input data are determined fully. In our example, when the list length equals 24 and the number of processor elements equals 47, the speedup is 20.88, with efficiency 44.43. The number of operations, the running time, and the cost of the program are 1441, 69, and 3243, respectively.

7. RELATED WORK

Much work has been done to implement declarative languages in parallel form. Most of them tend to resort to imperative features for some purposes, particularly for the description of the parallelism. Programs use the language’s control features to describe the parallelism explicitly. The multithreading technique to exploit the concurrency available in a declarative program in implicitly parallel languages like *pH* (*Arvind et. al*. 1996) is used.

The philosophy underlying the ANM project is closely coupled with the data flow architecture and the languages like Id, pH. “Let the programmer concentrate on the algorithms; let the compiler worry about efficient implementation” (*Arvind et. al*. 1996).

The transformation machine was considered by Ershov (*Ershov,* 1982). We believe the possibility of partially determined data evaluation in such a parallel machine will help to overcome some shortcomings of the declarative programming. Parallel partial evaluation, automatic parallelization, transformation, optimization, and other useful features could be available in such a parallel machine.

Many authors consider graph reduction machines and mechanisms for implementation of functional languages, lazy evaluation, lambda calculus (*Darlington et. al.,* 1987; *Peyton et. al.* 1987, *Kumar et. al.,* 1995, *Sesyoft,* 1993).

Using partial evaluation for the automatic program parallelization matches the goal of the partial evaluation-based compiler for the Supercomputer Toolkit (*Surati and Berlin,* 1994). The central focus of this study is to expose and to find a way to exploit extremely fine-grained parallelism. Traditionally, parallelization techniques include the control-flow and data-flow analysis, vectorizing, static and dynamic scheduling. Much recent effort has been directed towards creating parallelizing systems, partial evaluators, as stand-alone or as optimizing subsystem of different systems, online and offline type. In the Supercomputer Toolkit project a compiler of the Scheme dialect of Lisp was written, which coupled partial evaluation with static scheduling techniques to exploit parallelism by automatically mapping it into a coarse-grain parallel Supercomputer Toolkit architecture.

The realizing of partial evaluation as parallel computation in a transformation machine makes our approach to automatic program parallelization fundamentally different from the approaches taken by parallelizing compilers or traditional partial evaluators. 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.

Our approach is related to various works on partial evaluation and parallel computation. Examples of treating partially static structures was considered by Mogensen (*Mogensen*, 1988; *Mogensen and* Sestoft, 1996) and Arvind (*Arvind and Thomas*, 1980). The specialization of a program with respect to some abstract properties of an input was implemented by Consel and Khoo in the parameterized partial evaluation (*Consel and Khoo*, 1991).

Table 1 Performance analysis.

PARS Program		**InsSort1 (N=24)**	**InsSort2 (N=24)**	**Mtransp (N=3)**	**Mmult (N=3)**	**QSort (L=24)**	**Msort1 (N=24)**
Source	**size**	2700 Byte	3887 Byte	1508 Byte	3007 Byte	2904 Byte	6296 Byte
	T_1	27287	35370	129	1131	1671	46168
	T_p	126	150	17	32	101	118
	P	1496	690	18	120	37	916
	S_p	216.56	235.80	7.59	35.34	16.54	391.25
	E_p	14.48	34.17	42.16	29.45	44.72	42.71
	C_p	188496	103500	306	3840	3737	108088
Partial evaluation	**T_1**	26182	34265	117	1062	256	44631
	T_p	76	150	16	30	77	118
	P	1495	682	18	114	11	895
	S_p	344.50	228.43	7.31	35.40	3.32	378.23
	E_p	23.04	33.49	40.63	31.05	30.22	42.26
	C_p	113620	102300	288	3420	847	105610
Residual	**size**	123 Byte	122 Byte	678 Byte	10,883 Byte	13,768 Byte	165,011Byte
	T_1	1119	1119	26	73	1441	1540
	T_p	56	56	4	7	69	60
	P	55	55	9	27	47	47
	S_p	19.98	19.98	6.50	10.43	20.88	25.67
	E_p	36.33	36.33	72.22	38.62	44.43	54.61
	C_p	3080	3080	36	189	3243	2820
R	**size**	21.95	31.86	2.22	0.276	0.21	0.038
A	**operation**	24.39	31.61	4.96	15.49	1.16	29.98
T	**time**	2.25	2.68	4.25	4.57	1.46	1.97
I	**processor**	27.20	12.55	2.00	4.44	0.79	19.49
O	**cost**	61.20	33.60	8.50	20.32	1.15	38.33

8. CONCLUSION

We propose the parallel model of parallel computation named Abstract Network Machine based on parallel computation on associative networks, and the declarative program parallelism extraction method based on parallel dynamic partial evaluation. It allows writing programs in a declarative form and after that, on a partial evaluation phase, to extract available parallelism. We show that partial evaluation plays an important role in the parallel computation process. This approach is intended for automatic transforming, optimizing, adapting some universal declarative programs to a specific problem, and for their parallelizing.

Results of the experimental programming of problems from various problem domains such as LISP programming paradigm, operations on matrices, linear equations, sorting and combinatorial problems, graphs, simulation of electronic circuits (*Stepanov et. al*., 1993), and others, show that it is a suitable approach to the automatic program parallelization, and to partially overcome the shortcomings and non-effectiveness of declarative programming. It is particularly effective on numerically-oriented scientific programs, since they tend to be mostly data-independent. It is also useful when solving problems where complicated data structures (such as lists, trees, etc.) are involved and handmade revealing of the parallelism is difficult. In many cases this technique permits automatic reduction of waste computation to a great extent, and exposes "the hidden natural parallelism" in contradistinction of "the artificial parallelism" that can be revealed by a reconstruction of the program.

We describe an abstract machine, general approach to automatic program parallelization by partial evaluation, and its experimental application to the simple declarative PARS programs. Many problems remain to be solved before such a machine can be of practical use in real environment. Also the real-size application programs raise a number of critical issues and resource problems, which need to be solved before the approach becomes truly practical.

ACKNOWLEDGMENTS

The authors are pleased to acknowledge *John Willian Branch* for his help during the final preparation of the paper. Part of this work was supported by *CICESE* (Centro de Investigación Científica y de Educación Superior de Ensenada) under project #631109, by *CONACYT* (Consejo Nacional de Ciencia y Tecnología de México) under grant #32989-A, and by *Russian Fund of Basic Research.*

REFERENCES

Amamiya, M. et al. List Processing with a Data-Flow Machine. *LN in Computer Science*, 147, 165-190, 1983.

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

Arvind and Thomas R. I-structures: An efficient data type for functional languages. *Technical Report LCS/TM-178*, MIT, 1980.

Arvind, A. et al. A multithreaded Substrate and Compilation Model for the Implicitly Parallel Language pH. Computation Structures Group Memo 382, MIT, 1996.

Bjorner, I. et al. Partial Evaluation and Mixed Computation, *North-Holland*, 1988.

Consel C. and Danvy, O. Partial evaluation of pattern matching in string. *Information Processing Letter*, 30, (2):79-86, 1989.

Consel, C. and Danvy, O. Static and dynamic semantic processing. In *ACM Symposium on Principles of Programming Languages,* pages 14-23, 1991

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

Darlington, J. et al. 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, 1987.

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

Gupta, J.et al. CTDNet- A mechanism for the concurrent execution of lambda graphs. *IEEE Trans. Soft.Eng.* 15, 1357-1367, 1989.

Jones, N. et al. MIX: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation,* 2(1):9-50, 1989

Jones, N. et. al. Partial Evaluation and Automatic Program Generation. Prentice-Hall, 1993

Jones, An Introduction to Partial Evaluation. *ACM Computing Surveys*, 3, (28), p. 480-503, 1996

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

Quinn. M. Parallel Computing: theory and practice, New York, McGraw-Hill, 1994.

Kumar, P. et. al. CTDNET III - An Eager Reduction Model with Laziness Features. In Davy, J. and Dew, P. Editors, Abstract machine Models for Highly Parallel Computers, p. 103-117, 1995.

Lawall, J. and Danvy, O. Continuation-based partial evaluation. FTP version.

Mogensen, T. Partially static structures in self-applicable partial evaluator. In *I..Bjorner, A..Ershov, N..Jones. Partial Evaluation and Mixed Computation*, p. 325-347, North-Holland, 1988

Mogensen, T. and Sestoft, P. Partial evaluation. An article for Encyclopedia of Computer Science and Technology, FTP version, 1996

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

Sesyoft, P. "Deriving a Lazy Abstract Machine" J. Functioning Programming 1 (1), 1993

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

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

Stepanov, A. et. al. Parallel Computations on Associative Networks in Application to the Logic Modeling Problem. *Non Conventional Supercomputers*, Moscow, 2, 23p. (In Russian), 1993.

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

Stepanov, A. and Lupenko, A. Programming for ANM. *Preprint, AS USSR. Institute of Precise Mechanics and Computer Technology*; 3, Moscow, 53. (In Russian), 1991.

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

Stepanov, A. et. al., N.Tchernykh. 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

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

APPENDIX

```
Sch App(X,Y,Z);
if NIL(X) then Y=Z else
X=(X1,X2),Z=(X1,Z2),(X2,Y,Z2):App;
end
```

Figure 1A. Append.

```
sch Mtransp(M1,M2,N,K);
(K,'1',N,M1,M2): Mtransp1, (N,K,P):*Greater,
if P then (M1,M2,N,K1):Mtransp,
(K,1,K1):*SUM;
end

sch Mtransp1(I,J,N,M1,M2);
(M1,I,M1i):*Attrib, (M1i,J,M1ij):*ATTRIB,
(M2,J,M2J):* ATTRIB,(M2J,I,M1ij):*
ATTRIB, (N,J,P):*Greater,
if P then          (I,J1,N,M1,M2):Mtransp1,
(J,1,J1):*SUM;
end
```

Figure 2A. Square matrix transposition schemes.

```
sch InsSort1(V1,V2,i,n);
(n,i,p):*EQ, (V1,n,an):*ATTRIB,
(n1,'1',n):*SUM,
If P then V1=V2 else (V1,V11,i,n1):InsSort1,
(V11,an,V2,i,n1):InsVa1;
end

sch InsVa1(V1,a,V2,i,n);
(i,n,p1):*GREATER, (i,'1',i1):*SUM,
(V1,i,ai):*ATTRIB, (V2,i,x):*ATTRIB,
If p1 then x=a else (a,ai,P):*GREATER,
(V1,a,V21,i1,n):InsVa1,
(V1,V22,i,i1,n):InsRestV1;
If P then x=ai, V2=V21 else x=a, V2=V22;
end

sch InsRestV1(V1,V2,i,j,n);
(V1,i,ai):*ATTRIB,(V2,j,ai):*ATTRIB,(i,'1',i1):*
SUM,(j,'1',j1):*SUM,(i,n,P):*EQ,
If not P then (V1,V2,i1,j1,n):InsRestV1;
end
```

(a)

```
sch InsSort2(V1,V2,i,n);
(n,i,p):*EQ; If P then (V1,i,a):*ATTRIB,
(V2,i,a):*ATTRIB else
(V2,'1','0',V1,i,n,V2):MergeV;
end

sch MergeV(V1,i1,n1,V2,i2,n2,V3);
(i2,n2,P1):*GREATER, (i2,'1',i21):*SUM,
(n1,'1',n11):*SUM,
If P1 then V1=V3 else (V2,i2,a):*ATTRIB,
(V1,i1,n1,a,V11,j):InsEl,
(V11,i1,n11,V2,i21,n2,V3):MergeV;
end

sch InsEl(V,i,n,a,V1,j);
(i,n,p1):*GREATER, (i,'1',i1):*SUM,
If p1then (V1,i,a):*ATTRIB, j=i1
else (V,i,vi):*ATTRIB, (V1,i,x):*ATTRIB,
(a,vi,P):*GREATER,
(V,i1,n,a,V21,j):InsEl, (V,i,n,V22,i1):InsRestEl;
If P then x=vi, V1=V21 else x=a, V1=V22;
end

sch InsRestEl(V,i,n,V1,j);
(i,n,p1):*GREATER, (i,'1',i1):*SUM,
(j,'1',j1):*SUM;
If not p1 then (V,i,a):*ATTRIB,
(V1,j,a):*ATTRIB, (V,i1,n,V1,j1):InsRestEl;
end
```

(b)

Figure 3A. Two schemes for i.