International Journal

INFORMATION THEORIES & APPLICATIONS

2009 Volume 16 Number 4

International Journal
INFORMATION THEORIES & APPLICATIONS
Volume 16 / 2009, Number 4

IJ ITA is official publisher of the scientific papers of the members of the ITHEA® International Scientific Society

IJ ITA welcomes scientific papers connected with any information theory or its application.

IJ ITA rules for preparing the manuscripts are compulsory.
The **rules for the papers** for IJ ITA as well as the **subscription fees** are given on *www.ithea.org* .
The camera-ready copy of the paper should be received by http://ij.ithea.org.
Responsibility for papers published in IJ ITA belongs to authors.

General Sponsor of IJ ITA is the **Consortium FOI Bulgaria** (www.foibg.com).

International Journal "INFORMATION THEORIES & APPLICATIONS" Vol.16, Number 4, 2009

Printed in Bulgaria

Edited by the **Institute of Information Theories and Applications FOI ITHEA®**, Bulgaria,
in collaboration with the V.M.Glushkov Institute of Cybernetics of NAS, Ukraine,
and the Institute of Mathematics and Informatics, BAS, Bulgaria.

Publisher: **ITHEA®**
Sofia, 1000, P.O.B. 775, Bulgaria. www.ithea.org, e-mail: info@foibg.com

ISSN 1310-0513 (printed) **ISSN 1313-0463 (online)** **ISSN 1313-0498 (CD/DVD)**

PARALLELIZATION METHODS OF LOGICAL INFERENCE FOR CONFLUENT RULE-BASED SYSTEM[1]

Irene Artemieva, Michael Tyutyunnik

***Abstract**: The article describes the research aimed at working out a program system for multiprocessor computers. The system is based on the confluent declarative production system. The article defines some schemes of parallel logical inference and conditions affecting scheme choice. The conditions include properties of a program information graph, relations between data objects, data structures and input data as well.*

***Keywords**: logical Inference, parallel rule-based systems*

***ACM Classification Keywords**: D 3.2 – Constraint and logic languages, I 2.5 Expert system tools and techniques.*

Introduction

Development of computer architecture and network technologies, new theoretical and applied problems requiring a lot of computations, slow running speed of sequential computer systems and theoretically limited growth of their efficiency resulted in the necessity of using multiprocessor and multicore computer systems thus making parallel computations take centre stage in modern programming and computing technologies. This, in return, gave impetus to development of programming languages and language processors used to create applications for such systems.

At present, there are several directions for development of languages and language processors. First, high-level programming languages developed for sequential machines are extended via special tools for parallel computations and their parallel versions are designed (e.g., parallel versions of Fortran, C/C++, Modula-3 [3-6]). Developers of parallel programs write problem-solving algorithms using these means.

Second, there are special communication libraries and interfaces for organizing interprocessor interaction (e.g., PVM, MPI, MPL, OpenMP, ShMem [7-11]). They can be used in programs written in high-level languages. Developers of parallel programs organize parallel processes in a program aimed at solving problems on a multiprocessor computer system by means of these libraries and interfaces.

Third, language processor designed specifically for multiprocessor computer system implements automatic and semi-automatic parallelization of computations. In this case, methods of organization of parallel computations are hidden from developers of programs solving applied tasks (e.g., BERT 77, PIPS, VAST/Parallel [12-14]). Automatic parallelization of computations found practical use in packages of applied programs (e.g., ATLAS, DOUG, GALOPPS, NAMD, ScaLAPACK [15]) aimed at solving problems in specific areas.

[1] This paper was made according to the program № 2 of fundamental scientific research of the Presidium of the Russian Academy of Sciences, the project 09-I-П2-04

Fourth, there are programming languages or systems specialized for certain architecture or problems (e.g., KL1 and languages for programming on vector or matrix computers). These languages use parallelization of nested data, enable description of pipeline parallelism and parallelism at the level of problems.

Using high-level languages extended by means of organization tools for parallel computations or special communication libraries and interfaces urges developers of parallel programs to have special knowledge enabling creation of efficient parallel problem-solving algorithm. During parallelization of programs describing sequential problem-solving algorithm, compiler usually finds only a few fragments of a code for parallelization. Specialized languages are either inaccessible to a broad spectrum of specialists or applicable only to specialized architectures.

Problem-solving method may be represented either as an algorithm or as a set of rules (productions). There are many classes of applications using rules. Such representation of method is often used to create knowledge-based systems requiring thorough search for solutions and uses a lot of data stored in system knowledge bases. Representation of problem as a set of rules is a more natural way compared to an algorithm and does not urge program system developers to have special knowledge about organization of computing which is controlled by a language processor of rule-based language.

By now, representation of solution method as a set of rules is supported by various rule-based systems and the Prolog language. There are also parallel language processors for the Prolog language [16] and other rule-based languages. However, results of problem solutions, where the Prolog language (which is claimed to be rule-independent) is used, depend on the writing order of these rules, i.e. the language does not possess inherent parallelism. Results of problem solutions in rule-based systems which are not confluent also depend on the implementation order of these rules during the logical inference. Thus, when creating knowledge-based systems for multiprocessor computer systems with the usage of the Prolog language or rule-based non-confluent system, developers face the same problems as when they use languages of other classes.

In rule-based confluent systems, the result of the logical inference does not depend on the implementation order of these rules, i.e. such systems possess inherent parallelism. A language processor – a production language compiler allocates computations according to a process. When generating an object code, a language processor analyzes characteristics of the source program, a set of constraints imposed by the computing environment, characteristics of input data defined by the user and selects the most applicable scheme of the parallel logical inference.

The aim of this article is to describe schemes of the parallel logical inference implemented by the language processor of the confluent production system and conditions affecting the scheme choice.

Production System Language Characteristics

The research on ontologies [2] and design knowledge-based systems [1] on their basis makes it possible to formulate requirements for the language of the confluent production system.

1. The language must allow to represent a problem-solving method as a set of solving methods for subtasks described by the modules. Each module must have its interface, i.e. data description, that can be used by another module or that are required for its operation/performance. There must be an explicit condition of a module call, i.e. among the rules there can be rules the right part of which is a module call.

2. The language must allow to use operations on numeric data and sets. The language must allow to use limited logical and mathematical quantifiers that are analogs of loops in the rules.

3. The language must admit rules that are dependent on parameters (rule scheme). The scheme assigns a set of rules, i.e. it can be considered as an analog of a subprogram in the algorithmic language.

The language admits rules of two kinds:

(1) (prefix) $P(X) \rightarrow S_1(X_1)\&\ldots\&S_k(X_k)$, where $P(X)$ is a formula, $S_1(X_1)$, …, $S_k(X_k)$ are simple formulas, X, X_1, …, X_k are vectors of terms];

(2) (prefix) $P(X) \rightarrow$ NameMod(S_1, …, S_k) is a rule for module call, where $P(X)$ is a formula, NameMod(S_1, …, S_k) is a module name, X is vector of terms, and S_1, …, S_k are arguments of the module.

The rule before the symbol «→» is a production antecedent, the rule after the symbol «→» is its consequent. The antecedent is a logical expression made up of relations, functional terms, atomic formulas, generalized formulas according the following rules.

Each rule must meet the main antedecent for variables: $V(\{S_1(X_1), \ldots, S_n(X_n)\}) \cup V(P(X)) \subseteq V(\text{prefix})$, where $V(O)$ is a set of variables included into O (O can be any construction).

Prefix is a sequence of descriptions of variables $(v_1{:}t_1)(v_2{:}t_2)\ldots(v_m{:}t_m)$ where $(v_i{:}t_i)$ is a description of a variable, v_i is a variable, t_i is a term for all i=1,...,m. Term t_1 does not contain free variables. For i=2,..., m only variables v_1, v_2,...,v_{i-1} can be free variables of term t_i. The sequence of descriptions can be empty. All variables v_1, v_2,..., v_m are dually different.

The following construction can be called a scheme:

(3) NameSch([.CONST]w_1, [.CONST]w_2, …, [.CONST]w_n): rule where

NameSch is a scheme name, w_1, w_2, …, w_n are variables, rule is of the kind (1). Variables w_1, w_2, …, w_n are formal parameters of the scheme. The scheme body is a rule and must contain formal parameters. «[.CONST]» means that «.CONST» can be absent.

The following construction can be called a scheme concretization:

(4) NameSch(zw_1, zw_2, …, zw_n) where NameSch is a scheme name, zw_1, zw_2, …, zw_n are terms that are actual parameters of the scheme.

The scheme is an analog of a procedure in programming languages, the scheme concretization is an analog of a procedure call.

A domain symbol (a term of the domain ontology) – name n; variable v; sets I, R, S; empty set $\varnothing$; set $\{t_1,t_2,\ldots,t_k\}$ where $t_1,t_2,\ldots,t_k$ are terms; intervals $I[t_1,t_2]$, $R[t_1,t_2]$ where t_1 and t_2 are terms; expression t_1 ¤ t_2 where t_1 and t_2 are terms, sign «¤» $\in \{+, -, *, /\}$, t_1 ¤ t_2 where t_1 and t_2 are terms-sets, sign «¤» $\in \{\cup, \cap, \setminus\}$ is a sign of operation on sets; $\mu(t)$ is a power of set where t is a term-set, $(Z_n\,(v : t_1)\ t_2)$ is a quantifier term where $Z_n \in \{+, *, \cup, \cap\}$, v is a variable (index of a quantifier term), t_1 is a term-set that assigns a range of v, t_2 is a term (the body of a quantifier term) that contains operation index v; $f(t_1,\ldots,t_k)$ is a functional term where f is a functional symbol (a term of the domain ontology), $t_1,\ldots,t_k$ are terms (arguments of functional term) are terms;

$p(t_1,\ldots,t_k)$ or $\neg p(t_1,\ldots,t_k)$ where $p(t_1,\ldots,t_k)$ is an atomic formula, p is a predicate symbol (a term of the domain ontology), $t_1,\ldots,t_k$ are terms (arguments of atomic formula); t_1 @ t_2 where t_1 and t_2 are terms, sign «@» is a sign of mathematical relation or relation on sets are simple formulas.

Logical expression f_1 @ f_2 where «@» $\in$ {&, V}; quantifier formula $(Z_n\ (v : t)\ f)$ where $Z_n \in$ {&, V}, v is a variable (index of a quantifier formula), t is a term-set that assigns a range of v, f I a formula (the body of a quantifier formula) that contains index v are formulas.

Information Graph Definition

A language processor is a compiler that translates a text in the production language into the object code in the algorithmic high-level language. The object code implements the logical inference assigned by the production rules and contains calls of modules of the run period support environment. Before the code generation the language processor makes the program information graph and analyzes its characteristics.

An aligned cyclic graph the vertices of which are rules and arcs of which indicate information relations between rules, i.e. arcs connect those rules that exchange data, is called the information graph. The arc between two vertices exists if the following antecedent is met: $IF(\pi_j) \cap THEN(\pi_i) \neq \varnothing$ where $IF(\pi_j) = \{o_1', \ldots, o_a'\}$ – a set of terms of a domain included in the antecedent of the rule π_j, $THEN(\pi_i) = \{o_1'', \ldots, o_b''\}$ – a set of terms of a domain – arguments of the module called in the consequent of the rule π_i, or a set of terms of a domain included in the consequent of the rule π_I, $i \neq j$. The rule π_j will be called dependent on π_i. The information graph is created for each module. So, the information graph of a program is an array of information graphs of its modules.

Let us consider the structures used for representing the information graph of each module and different properties of a module that can be defined on basis of its graph. The element (i,j) of the incidence matrix IncMatrix with dimensions $\mu(\Pi_m) \times \mu(\Pi_m)$ where $\mu(\Pi_m)$ means the number of module rules is equal to 1 if the j-th rule is a direct child of the i-th rule, and is equal to 0 otherwise. LoopMatrix stores the information about the graph vertices that are included in a loop: elements correspondent to the rules not included in a loop are equal to 0, and included – 1. The element of an array iLoopAr with the number i is equal to -1, if the rule i is not included in any of the loops, is equal to 0 – the rule is included in one of loops but is not a loop entry; if the rule is a loop entry, the element is equal to the number of direct children of this rule. The number of direct children of the rule i is a value of each element with the number i of an array iParentsAr.

Parallelization of Rules Using Set of Active Rules

Since the language considered in the paper is confluent, the parallelization scheme where all the rules are sent one by one to slave processes and are executed in parallel for so long as they change the condition of the run-time environment is the most natural one. It is evident that when computations are organized in such a way, a set of empty executions of rules will be done because there are no input data for them at the initial moment of execution. One can use a set of active rules (SAR) to eliminate empty calls of rules. SAR is a set of rules which have input data at the moment of computation. Let us describe the parallelization scheme inside the module using SAR.

Let us introduce the designations. R_m – a set of all the rules of the program module, R' – a set of rules that are being executed by dependent processes; R'' – a set of rules that have been executed by dependent processes; FormSAR – a set of operations to create a set of rules that are being used at the current moment of execution; ComputeRule(π) – a set of operations to start dependent process to compute rule π; GetResult(π, M) – a set of operations to receive computed data M of rule π from the executed process; Synchronize(M) – a set of operations to synchronize data received from the executed process with data stored in the main process; Compute(π, M) – a set of operations to compute rule π, i.e. to search for new values M objects included in the consequent of rule.

Let us describe the processes of parallelized logical inference by means of the designations.

Begin of main process;

FormSAR; $\Pi' = \varnothing$; $\Pi'' = \varnothing$;

While SAR $\neq \varnothing$ do:

select π from SAR;

SAR = SAR \ {π};

ComputeRule(π);

П' = П' $\cup$ {π};

End of while;

While $\neg$(МАП = $\varnothing$ и П' = $\varnothing$) do:

If П'' $\neq \varnothing$ then

GetResult(π'', M);

Synchronize(M);

FormSAR;

П'' = П'' \ {π''};

end of if;

While SAR $\neq \varnothing$ do:

select π from МАП;

ComputeRule(π);

SAR = SAR \ {π};

П' = П' $\cup$ {π};

End of while;

End of while;

End of main process.

The dependent process begins to execute the rule during StartProcess(π) command of the main process.

Begin of dependent process;

Compute(π, M);

П'' = П'' $\cup$ {π};

П' = П' \ {π};

End of dependent process.

Let us comment on the algorithms. In the beginning, a set of active rules is formed by the main process. The set contains the rules for all the objects of which included in the rule condition in the input data are specified. Then in the loop each rule from SAR is sent for computation to the dependent process. The main process waits for the execution result of at least one of the dependent process. After that, the results of the rule execution are synchronized with the current values of data and SAR is updated: this set is completed with the rules or the objects of which new values appear. If SAR is not empty, rules from SAR are sent to free slave process for computation. The program ends when there are no dependent process executing rules and SAR is empty.

As opposed to the common scheme or parallelization, this scheme solves the problem of execution of empty rules. However, it has a drawback: the system incurs additional expenses due to forming of SAR during computations.

Using the Information Graph at the Parallelization of Logical Inference

This paper suggests using the "client-server" architecture when a separate process is a dispatcher (main process), other processes are handling processes (dependent processes) for constructing a parallel production system. The main process inputs and outputs data synchronizes them and exchanges data with dependent processes; it prioritizes rules and provides each process with a subprogram to process a rule. Each dependent process executes a subprogram that implements the logical inference for the rule, i.e. it searches for all substitutions at which the condition of the rule applicability is true and for each substitution it performs actions defined by the consequent of the rule and passes the received data to other processes. Below there are schemes of the workflow of the main process and dependent process.

Before the main process starts to work, iCurParentsAr – a copy of iParentsAr – is created, at the same time if the information graph contains loops, we must change values of elements of the array iCurParentsAr in the following way: if iLoopAr[i] > 1, then iCurParentsAr[i] = iLoopAr[i], i.e. substitute rules-loop entries for the number of parents that do not belong to loops. Elements of the array iCurParentsAr change their values during the calculations. The array element i gets equal to -1 if the rule i is being processes, -2 – if the rule i has been processed.

Scheme 1a (main process):

Calculations Begin: $\mu(P_f) = \mu(P)$; $P_w = \varnothing$. Block 1.

LOOP: While exist iCurParentsAr[i] = 0 or $P_w \neq \varnothing$, do: Block 2; Block 3. Loop End.

Block 4. Calculations End.

Block 1 (*Start all rules which appropriate to root vertices*):

For every free process j from P_f do:

For every element i from array iCurParentsAr: If iCurParentsAr[i] = 0,

then Send(Q_i, i, j); iCurParentsAr[i] = -1; $P_f = P_f \setminus \{j\}$; $P_w = P_w \cup \{j\}$.

Block 1 End.

Here Send(Q_i, i, j) is a procedure that sends data set Q_i into process j and informs process j of the necessity to calculate rule i; $\mu(P)$ is the number of slave processes for calculations; $\mu(P_f)$ is the number of free processes. Data set Q_i contains all the values of the objects included in the condition of rule i.

Block 2 (*Receive and synchronize results*):

1. Recv(Z, i, j); $P_w = P_w \setminus \{j\}$; $P_f = P_f \cup \{j\}$.
2. iCurParentsAr[i] = -2.
3. For all vertices k such as

iCurParentsAr[k] > 0, do:

3.1. If IncMatrix[i,k] = 1,

then iCurParentsAr[k] = iCurParentsAr[k] – 1;

3.2. If iLooptAr[i] > 0 & iCurParentsAr[k] = -2 & iLoopAr[k] > 0 & THEN(i) $\cap$ IF(k) $\neq \varnothing$

then iChangedLoopAr[k] = 1.

4. Data = Data $\cup$ Z;

Block 2 End.

Here Recv(Z, i, j) is a procedure that receives from process j data set Z that are results of computing rule i. Data set Z contains values of the objects included in the consequent of rule i. Data is a data set that contains all the values of all the objects included in the rules.

Block 3 (*Assign rules ready for computing to free processes*):

For every free process j do:

For every element i from array iCurParentsAr:

If iCurParentsAr[i] = 0

then Send(Q_i, i, j); iCurParentsAr[i] = -1.

If $P_w = \varnothing$ & not exist elements k from array iCurParentsAr such as iCurParentsAr[k] = 0 then:

For every element t from array iCurParentsAr:

If iCurParentsAr[t] = -2 & iLoopAr[t] > 0 then:

For all vertices s:

If LoopMatrix[t,s]=1 & iCurParentsAr[s]>0

then iCurParentsAr[s] = 0;

Goto Block 3.

Block 3 End.

Block 4 (*Make rules which appropriate to loop vertices ready for repeated calculations*):

If exist elements i from array iChangedLoopAr such as iChangedLoopAr[i] = 1then:

For all rules do:

If k – (distant) child of rule i

then iChangedLoopAr[k] = 1.

For every element t from array iChangedLoopAr:

If iChangedLoopAr[t] = 1

then iCurParentsAr[t] = iParentsAr[t];

If iLoopAr[t]>1 then iCurParentsAr[t]=0.

Goto Block 3.

else:

Block 4 End.

Here iChangedLoopAr is an integer array where the element with number i is equal to 1 if loop rule i has been computed and then appear new values for the objects included in its antecedent; otherwise the element with number i is equal to 0. This structure is filled in the course of computing the rules and is used to construct a children list, the children must be computed again.

Scheme 1b (slave process):

Calculations Begin: wRecv(Z, i); wCalc(i, Z, Q); wSend(Q, i); Calculations End

Here wRecv(Z, i) is a procedure that receives from the main process data set Z that contains values of the objects included in the antedecent of rule i; wSend(Q, i) is a procedure that sends into the main process data set Q that are the results of computing rule i; wCalc(i, Z, Q) is a procedure that computes rule i with the help of the logical inference.

Tuple Passing at Incomplete Rule Computation

The previous scheme rigidly specifies that the dependent rule cannot be computed until the rules it is dependent on have been computed. This scheme does not have such a restriction – the next rule waits for at least one tuple – the result of the application of the rule it depends on but not the termination of all the rules-parents. If each next tuple appears at the beginning of the rule application, there can be a situation when all the rules are processed parallel regardless of how they are connected informationally. However, due to the restrictions connected with the

number of processes of cluster computer free for computation, the number of rules that work parallel cannot exceed the number of free processes.

Let us complete the above schemes with a series of new operations that will allow to load free processes with those rules the only parents of which are being computed.

Let iParentsIdAr be an integer array the dimensions of which coincide with the number of module rules, the element with number i being equal to the number of the parent if vertex i has the only parent. SendPFrom(p_{from}, Q_i, i, j) is a procedure that sends data set Q in process j and informs process j of the necessity to calculate rule i, process j must receive from process p_{from} a set of next tuples for the objects included in the antecedent of rule i. Data set Q contains all the values of the objects included in the antecedent of rule i. SendPTo(p_{to}, p_i) is a procedure that sends in process p_i that applies rule i the message about the necessity to send to process p_{to} tuples for those objects included in the antecedent of rule processed by p_{to}.

Block X1 (*Assign additional rules to calculations using tuples passing*):

For every free process j from P_f do:

  For every element i from array iCurParentsAr:

    If iCurParentsAr[i] = -1 then

      For every element k from array iParentsIdAr:

        If iParentsIdAr[k] = i then

          iParentsIdAr[j] = -1;

          SendPFrom(p_{from}, Q_k, k, j);

          iCurParentsAr[k] = -1; $P_f = P_f \setminus \{j\}$; $P_w = P_w \cup \{j\}$.

          SendPTo(p_{to}, p_i);

Block X1 End.

Scheme 2a (main process):

Calculations Begin:

$\mu(P_f) = \mu(P)$; $P_w = \varnothing$.

Block 1.

Block X1.

LOOP: While exist iCurParentsAr[i] = 0 or $P_w \neq \varnothing$, do:

  Block 2.

  Block 3.

  Block X1.

LOOP End.

Block 4.

Calculations End.

Scheme 2b (slave process):

Calculations Begin:

flag = 0; $p_{to\_} = 0$;

wRecv_(p_{from}, Z, i);

If $p_{from} > 0$ then flag = 1;

Block 1.

If flag = 1 then:

  wRecvPFrom(p_{from}, Z_i);

  If $Z_i \neq \varnothing$ then $Z = Z \cup Z_i$;

  else flag = 0;

wCalc(i, Z, Q);

If wRecvPTo(p_{to}) = 1then $p_{to\_} = p_{to}$;

If $p_{to\_} > 0$ then wSend_($p_{to\_}$, Q);

Block 1 End.

If flag = 1 then Goto Block 1.

wSend(Q, i);

Calculations End.

Block X1 loads all the free processes with the rules that can receive tuples from their computed parents and informs the processes that compute parents of the necessity to pass tuples to other processes.

The scheme of the workflow of the dependent process is a modified scheme 1b. To describe it we use the following procedures.

wRecv_(p_{from}, Z, i) is a procedure that is an extended version of procedure wRecv. wRecv_ receives from the main process data set Z that contains the values of the objects included in the antecedent of rule i and receives the number of process p_{from} from where next tuples can come. If p_{from} = 0, tuples from other processes will not be sent. $Z \equiv \{O_1, \ldots, O_z\} \equiv \{\{k^1_1, \ldots, k^1_{k1}\}, \ldots, \{k^z_1, \ldots, k^z_{kz}\}\}$.

wRecvPFrom(p_{from}, Z_i) is a procedure that receives from slave process p_{from} data set Z that contains the value of the objects included in the antecedent of rule i. $Z_i \equiv \{O_1, \ldots, O_z\} \equiv \{\{k^1_1, \ldots, k^1_{k1}\}, \ldots, \{k^z_1, \ldots, k^z_{kz}\}\}$.

wRecvPTo(p_{to}) is a function that receives from the main process the number of slave process p_{to} to which it is necessary to send tuples. The function returns 0 if the number of the process from the main process has not been received, and it returns 1 if it has.

wSend_(p_{to}, Q) is a procedure that sends to slave process p_{to} data set Q that is the current result of the computed rule i. Data set Q contains all the values of the objects included in the consequent of rule i. $Q \equiv \{O_1, \ldots, O_q\} \equiv \{\{k^1_1, \ldots, k^1_{k1}\}, \ldots, \{k^q_1, \ldots, k^q_{kq}\}\}$.

Applying this scheme we can launch in parallel the process that will process the rule dependent on data not when the rule-parent has been performed, but when attributions for the objects found in the course of calculations start to come. Thus, if the dependence on data allows, one can launch all the rules in parallel as the correspondent attributions appear. This implies that the period of applying all the rules can be shorter than the period of applying the rules using the first scheme.

Parallelization of Computations for Rules with Quantifiers

The scheme considered below makes it possible to parallelize computations for a single rule with quantifier. The body of rule with quantifier must be executed as many times as many values the index of quantifier construction receives, i.e. rule with quantifier defines the loop the body of which is the rule body. Therefore rules with quantifiers can be easily parallelized.

The parallelization scheme assumes the following actions. As soon as rule with quantifier becomes accessible for computations, the main process organizes loop on all values of the quantifier index. At each step of the loop, rule with a certain current value of the index is sent to the free process for computation. Each new step of the loop sends the rule to execution in parallel with previous steps as copies of the rules do not contain the dependent data. The results of computations of each copy of the rule are processed by the main process as it is shown on scheme 2.

Let us consider an example of how the scheme works. It is as follows:

<{<i, 3, 10>} a(i) & b(v) & v > 10 $\rightarrow$ c(i+v) >.

i variable is the index, the low bound is specified with 3, the upper bound is specified with 10. The body of the rule a(i) & b(v) & v > 10 $\rightarrow$ c(i+v).

Using this scheme, we will have a set of 8 rules:

a(**3**) & b(v) & v > 10 -> c(**3**+v);

a(**4**) & b(v) & v > 10 -> c(**4**+v);

…

a(**10**) & b(v) & v > 10 -> c(**10**+v).

Each copy of the rule with a new value of the index is launched if there is a free slave process. If at the initial moment of the execution there are 8 or more free processes, all the copies of the rules can be computed concurrently.

Let there be n free slave processes available for the system at the moment of computation of the rule with quantifier. If the quantifier index receives m values, where $m \geq n$, using the scheme will reduce the execution time of T rule down to n times and down to m times if $m < n$.

Parallelization of Computations through Partitioning Object Value Area inside Rule

This scheme describes parallelization of partitioning rule computation through partitioning area of existing values of objects included in the rule condition and sending each subarea to the correspondent dependent process where searching for the result will take place. If there are free dependent processes, the main process can divide object value area into subareas the number of which equals the number of free processes. In this case, each process receives a copy of the rule with the correspondent subarea of object values. As subareas do not intersect, all copies ob the rule can be computed concurrently.

Let there be n items of o objects. For each o_i object, there are a set of tuples of values of ar arguments in quantity of mi.

$ar_1(o_1)$ $ar_1(o_2)\ldots$ $ar_1(o_n)$

$ar_2(o_1)$ $ar_2(o_2)\ldots$ $ar_2(o_n)$

… … …

$ar_{m1}(o_1)$ $ar_{m2}(o_2)\ldots$ $ar_{mn}(o_n)$

In the process of searching for values satisfying the rule condition a selection and conditional test in a set of Args of all the values:

$ar_1(o_1)$ $ar_1(o_2)$ … $ar_1(o_n)$,

$ar_1(o_1)$ $ar_1(o_2)$ … $ar_2(o_n)$,

$ar_1(o_1)$ $ar_1(o_2)$ … $ar_3(o_n)$,

… … …

$ar_1(o_1)$ $ar_2(o_2)$ … $ar_1(o_n)$,

$ar_1(o_1)$ $ar_2(o_2)$ … $ar_2(o_n)$,

$ar_1(o_1)$ $ar_2(o_2)$ … $ar_3(o_n)$

etc. Evidently, each string of values can be tested for compatibility with the rule condition in parallel. This, we can partition a set of object values into nonintersecting subsets:

$Args = Arg_1 \cup Arg_2 \cup \ldots \cup Arg_k$.

$Arg_1 \cap Arg_2 \cap \ldots \cap Arg_k = \varnothing$.

Let us exemplify:

R_1: a(v1, v2) & b(v3, v4) & condition(v1) & condition (v2) & condition (v3) & condition (v4) $\rightarrow$ c(v1+v3);

For each object of R_1 rule, there is a set of values-tuples:

Object «a»	
Arg. 1	Arg. 2
a1[1]	a2[1]
a1[2]	a2[2]
…	…
a1[n]	a2[n]

Object «b»	
Arg. 1	Arg. 2
b1[1]	b2[1]
b1[2]	b2[2]
…	…
b1[m]	b2[m]

The scheme of search for all the tuples of objects satisfying the condition of R_1 rule is generally as follows (for execution process):

```
Begin;
For i = 1 to n do:
 if condition(a1[i]) = «true» and condition(a2[i]) = «true» then
  For j = 1 to m do:
   If condition (b1[j]) = «true» and condition(b2[j]) = «true» then
    MakeCalculation(a1[i], a2[i], b1[j], b2[j]);
   If end;
  Loop end;
 If end;
Loop end;
End;
```

It is clear that there is a bypass of the tuples of objects by means of nested loops. To parallelize this rule processing, we can partition the search space for tuples for the first object according to the number of free processes. In other words, if we have k of processes available for usage in rule computation, we send the 1^{st}-$(n/k)^{th}$ tuples of the first object to the first process, the $(n/k + 1)^{th}$-$(2*n/k)^{th}$ tuples of the first object to the second process, ((n-1)*n/k + 1)th-nth tuples of the first object to the k^{th} process correspondingly.

The example for the t^{th} process ($1 \le t \le k$) is as follows:

```
Begin;
For i = 1 to ((t-1)* n/k+1) do:
 If condition(a1[i]) = «true» and condition(a2[i]) = «true» then
  For j = 1 to m do:
   If condition(b1[j]) = «true» and condition(b2[j]) = «true» then
    MakeCalculation(a1[i], a2[i], b1[j], b2[j]);
   If end;
  Loop end;
 If end;
Loop end;
End;
```

The main process distributes tuples of the first object and sends a copy of the correspondent tuples to the correspondent process.

Method of Control of Scheme Choice

Let us describe the method of control of parallelization scheme choice and consider constraints imposed by the environment of program execution and structure of the information graph (IG).

Let us have $\mu(Rm)$ number of rules in m module. It has Popt defined provided that the rule is executed completely and the number of free slave process equals $\mu(S)$. In real tasks, IG can have up to 100 rules and, at the same time, the branchiness can be very low. On the other hand, real clusters provide not more than a few dozens of processes. In this case, $\mu(S) << \mu(Rm)$ but as we are unable to compute more than Popt of rules concurrently, it will be more correct to consider the relation of Popt and $\mu(S)$ numbers. If Popt < $\mu(S)$, there are idle slave process during all the computations. On the other hand, if Popt > $\mu(S)$, there are no idle processes at some points of time. It is clear that graphs have different branchiness at different levels. Hence at one point there are all free slave processes operating, at another one there is only one process with others being idle for a long period of time.

Using the method of sending tuples to processes, the system must have at its disposal the number of slave processes equaling the number of rules while real clusters provide not more than a few dozens of processes.

Additionally, there appear increase expenses for sending tuples to processes and data synchronization system.

We propose to combine two methods of rule parallelization and thus the system of logical inference must use the maximum number of free processes at each point of time. To reach this, the system assigns all ready for execution rules from IG for computation. If there are idle slave processes after that, the system sends computation of rules-children parents of which are being processed. At the same time, there is an exchange of newly computed tuples among processes processing a group of parent-child rules.

Algorithm 1 of rule computation in the main process is given below. The information graph is designed for m module. It has $\mu(Rm)$ number of rules and $\mu(S)$ number of free slave processes. Note that the execution time for each vertex-rule is not known. Consequently, assigning rule ready for execution to a free slave process can only be done dynamically during the logical inference system operation.

1. All IG rule-vertices ready for execution are found. If in the consequent of a rule ready for execution there is a module call, launch of all new rules stops, the main process waits for the results of execution of all current rules and then passes control to the main process of the called module. Transfer to Step 1 of the called module. If in the consequent of a rule ready for execution there is no module call, it is passed to a free process from a set $\mu(S)$ for computation. If there are no free processes after that, transfer to Step 4, otherwise – Step 2.
2. Rule-vertices where their only parent is the vertex that is being processed are chosen from IG. Each found rule is passed to a free slave process for computation; this process can receive resulting tuples from the processes processing the rule-parent. If there are no free processes after that, transfer to Step 4, otherwise – Step 3.
3. If a rule ready for execution contains a quantifier, for each quantifier index there is a copy of the rule with the substituted index value/ the copy is sent to a slave process. All copies of the rule are launched concurrently.

4. If a process has finished the rule computations and sent the results, the main process accepts all resulting tuples and synchronizes them with its data. If all the rules are computed and all slave process are free, transfer to Step 4, otherwise – Step 1.
5. If in IG there are vertices included in loops and, as a result of computations of rules corresponding to these vertices, new values of objects which are included in conditions of loop rules, a subgraph is designed. It contains all the rules which are children of these loop rules. IG is substituted with he designed subgraph and computation starts from Step 1 again. If there are no loops in IG and no values after loop computations, transfer to Step 5.
6. If the module the rules of which were computed at the previous steps has been called from another module, the main process passes control to the process it was called by. Transfer to Step 1. If it is the main module, the computation ends.

Algorithm 2 of a slave process.

1. A slave process accepts the number of a rule and input data from the main process and starts computation.
2. If in the process of computation there is a command to accept tuples from another process, the given process accepts these tuples and synchronizes them with its data.
3. If in the process of computation there is a command to send the computed tuples to another process, all the computed tuples are sent to that process.
4. After the rule computation, all resulting data are sent to the main process.

The algorithms show that all the rules are executed once if there are no loops in the information graph. If there are loops in IG and for one of the computed rules included in the loop there are new values, such rule along with all the rules that are children of the given one is executed again as long as new values appear.

The slave process algorithm describes the computation of the result of the processing of the rule which is search for attributions satisfying the rule condition and execution of the rule consequent. If in the rule consequent there is a module, the main process stops executing the rules of the current module and begins to process the called module according to Algorithm 1.

The search for attributions is search for attributions for all the objects of the rule condition that is passing through the nested loops. Generally, let there be finite sets of values of M, M1, M2 objects and function f: M1 x M2 $\rightarrow$ M. Let us consider the problem of computation of over the whole definition range. This problem is solved by means of the following algorithm.

Begin;

For all elements from M1 do:

    Get x $\in$ M1;

    For all elements from M2 do:

        Get y $\in$ M2;

        Calculate f(x, y);

    Loop end;

Loop end;

End.

A loop is used to design computations for each rule to search for values of each object included in the rule condition. Each loop of a new object is embedded into the loop of the previous object in the rule condition. Thus, already found values are excluded from the search range.

In this case, to avoid rescanning of the value range during the recomputation of a certain rule information about what sections of the value range have been scanned before is connected with each rule, i.e. for each rule each set of values is divided into two parts: 'new' and 'old.' If we designate scanning through a new part of the value range of a function or a relation named fri as N(fri), through the old part – O(fri), through the whole value range – A(fri), Cartesian product set – as "*", union of sets – "+", then all the new acceptable values of λ substitution can be resulted from scanning the following set:

N(fr1) * A(fr2) * A(fr3) * … * A(frk)

+ O(fr1) * N(fr2) * A(fr3) * A(fr4) * … * A(frk)

+ O(fr1) * O(fr2) * N(fr3) * A(fr4) * … * A(frk)

+ … + O(fr1) * O(fr2) * … * O(frk-1) * N(frk).

Thus, during the recomputation of loop rules and rules dependent on them, the logical inference system searches for values only for a new range. Labels are stored in special structures. If the rule is a loop one, in case of its recomputation its labels return to zero.

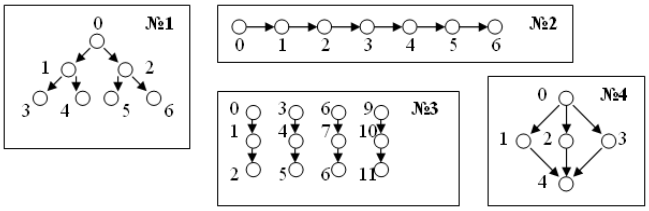
Experimental Study of Parallel Programming System Properties

Some experiments were conducted to evaluate the efficiency of production system operation with parallelization schemes. Each experiment included time measurement for generated programs for cluster with different sets of data. The experiments were aimed at studying the dependency of program execution time on the number of connections among rules and the number of free processes. In the description of the experiments there is average period of program execution time.

Each logical program consists of a set of rules which are connected with each other via data. The experiments cover the programs the connections inside which can be the most frequent. We can assume that all the rest programs are a composition of exemplified programs.

The experiments were conducted on a multiprocessor computer system with 12 processors free for computations. Each processor was given one process. Client-server architecture of the program executing the logical inference requires at least 2 free processes one of which is meant for data control and synchronization; the rest are meant for computations of rules per se. If there are 2 free processes, rules are processed consequently one by one.

To exemplify, we chose the programs the information graphs of which are as follows:



The execution time of each program is shown in Fig. 1. Example 1 demonstrates the maximum execution time of 30.5 sec. with 2 processes (one of them is for control functions, the others – for rule execution), the minimum execution time of 13.2 sec. With 4 and more processes the efficiency grew 2.3 times. Example 2 demonstrates

the maximum execution time of 30.5 sec. with 2 processes, the minimum execution time of 5.3 sec. With 8 and more processes the efficiency grew 5.7 times. Example 3 demonstrates the maximum execution time of 52.5 sec. with 2 processes, the minimum execution time of 8.3 sec. With 12 and more processes the efficiency grew 6.3 times. Example 4 demonstrates the maximum execution time of 21.8 sec. with 2 processes, the minimum execution time of 13.2 sec. With 3 and more processes the efficiency grew 1.6 times.

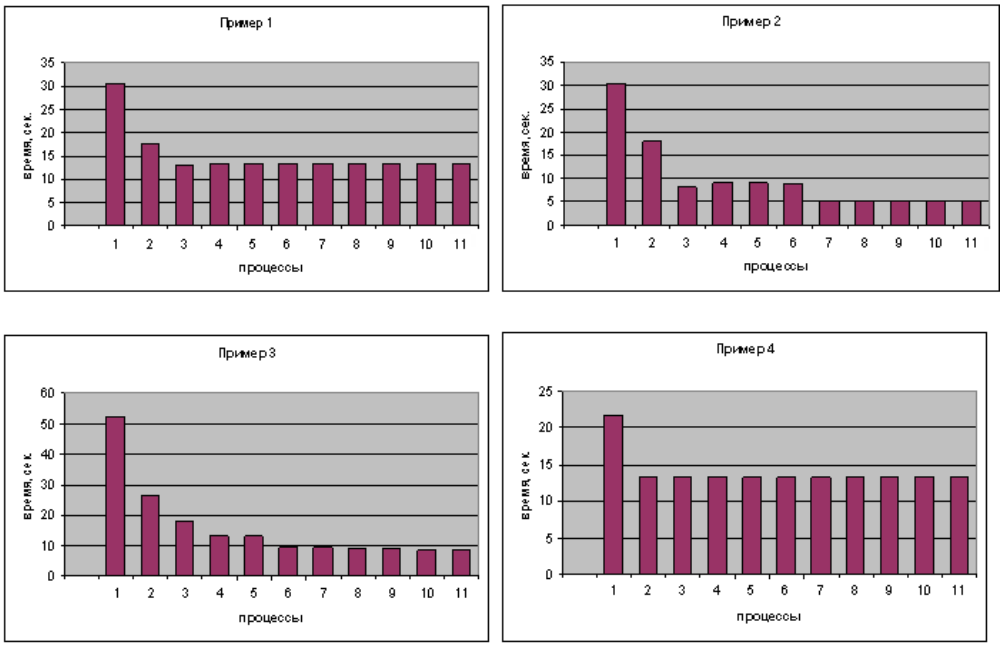


Fig. 1. The dependency of program execution time on the number of connections among rules and the number of free processes.

Apart fro this, there were experiments on actual problems, one of which is the problem of organic synthesis. The dependency of the time for solution of this problem on the number of processes is shown in Fig. 2.

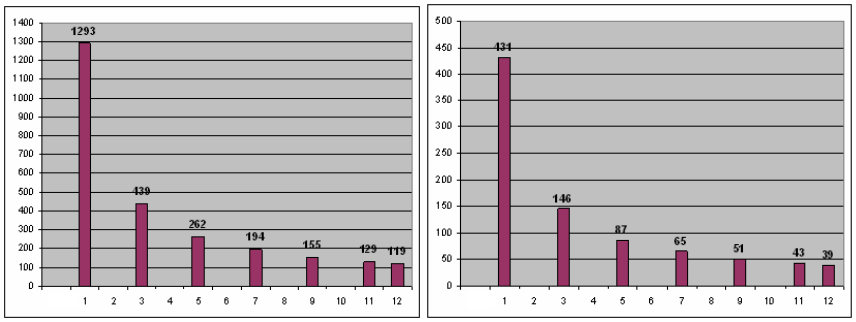


Fig. 2. The dependency of the execution time on the number of processes for the problem of organic synthesis.

The left diagram demonstrates the execution time on one processor of 1293 sec., on 12 processors – 119 sec., the efficiency is 10.8 times. The right diagram presents the program the volume of data of which is three times

less than for the previous program. The maximum execution time – 434 sec., the minimum execution time – 39 sec., the efficiency – 11 times.

Conclusion

The above schemes use such characteristics of the information graph as the number of its vertices, the number of its branches that can be processed in parallel, etc. To choose a scheme of parallelization of the logical inference not only the characteristics of the graph but also architecture and system constraints imposed by the computing environment must be used. They are:

1. The number of free system processes. If the number of the processes is more or equal to the number of the rules of the program, this case is convenient for calculations as one can specify in advance the particular rule for each process. If there are less processes than rules, then the rules are assigned to the processes dynamically.

2. The period of the rule application. In the course of computing a rule there may be a situation when this rule is processed many times longer than any other rule of the logical program. In this case there may be an idle time of the system that awaits the end of calculations of this rule. One of the solutions is to send intermediate results of the rule calculation to other process that process dependent rules.

3. The structure of the program information graph that is assigned by a set of information graphs of modules that are part of the program. The graph characteristics define the number of graph sections that can be executed in parallel. One has to analyze the graph to assign a rule for a free dependent process. The more branches are there in the graph, the stronger is the possibility of parallelizing calculations of rules. As one does not know the exact time of computing a rule, the maximum number of processes P_{opt} that can be performed in parallel with each other is defined in the following. For each vertex of the graph they build a set of vertices into which there are no ways from the given vertex and which are not parents (and far parent as well) of this given vertex. After computing the maximum from the number of the elements of all the sets, one will have the sought P_{opt}.

The closer is P_{opt} to the number of the rules in module $\mu(\Pi_m)$, the more efficient will be the parallelization of the task and quicker will be the calculations, i.e. $E = P_{opt} / \mu(\Pi_m)$ – tends to 1 (E defines the average fraction of the rule calculation by a separate process). From the definition P_{opt} it follows that P_{opt} will be bigger if the graph has more direct children for one parent, i.e. there is wide branchiness of the graph.

For each graph P_{opt} is invariable if the rules are executed completely: first – parents, then – children. However, one can start to pass intermediate results to the dependent rules without awaiting the completion of one rule. In this case if in the rule-parent there is at least one tuple of new values for the object included in the antecedent of the rule-child, the process that maintains the rule-parent sends the tuple to the process assigned to process the dependent rule. Doing this with all the rules and assigning a rule for a process, the system of the logical inference can assign all the rules for execution in parallel with each other. It implies that P_{opt} is always equal to the number of rules $\mu(\Pi_m)$ (and $\mu(P)$ must be equal to $\mu(\Pi_m)$) in module m, $E = 1$.

It is evident from the experiments that execution time of most programs on multiprocessor computer systems is close to theory.

Bibliography

[1] Gavrilova T.A., Khoroshevsky V.F. Intellectual System Knowledge Bases. (In Russian) – SPb.: Piter, 2001

[2]. Kleshchev A.S., Artemjeva I.L. Mathematical models of domain ontologies // Int. Journal on Inf. Theories and Appl., 2007, vol 14, № 1. PP. 35-43.

[3]. Berkeley UPC - Unified Parallel C. http://upc.lbl.gov/.

[4]. David E. Culler, Andrea Dusseau. Parallel Programming in Split-C. Computer Science Division University of California, Berkeley. 1993. www.eecs.berkeley.edu/~yelick/arvindk/splitc-super93.ps.

[5]. Ken Kennedy (director), Vikram Adve. Fortran Parallel Programming Systems // Parallel Computing Research Newsletter. 1994, vol. 2, issue 2. http://www.crpc.rice.edu/newsletters/apr94/resfocus.html.

[6]. Modula-3 resource page. http://www.modula3.org.

[7]. Carl Scarbnick. Building your own parallel system with PVM. http://www.sdsc.edu/GatherScatter/gsnov92/PVMparallel.html.

[8]. The Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi.

[9]. Message passing with MPL. http://math.nist.gov/~KRemington/Primer/mp-mpl.html.

[10]. OpenMP. http://openmp.org/wp.

[11]. ShMem. http://www.fis.unipr.it/lca/tutorial/hpvm/hpvmdoc_83.html#SEC90.

[12]. BERT 77. http://www.basement-supercomputing.com/content/view/25/52/

[13]. The PIPS Workbench Project. http://www.cri.ensmp.fr/~pips/.

[14]. VAST/Parallel. Crescent Bay Software. http://www.crescentbaysoftware.com/vast_parallel.html.

[15]. Douglas Kothe, Ricky Kendall. Computational science requirements for leadership computing. 2007. www.nccs.gov/wp-content/media/nccs_reports/ORNL_TM-2007_44.pdf.

[16]. Paralogic Inc. http://www.plogic.com/index_html.

Authors' Information

Irene L. Artemieva – artemeva@iacp.dvo.ru

Michael B. Tyutyunnik – michaelhuman@gmail.com

Institute for Automation & Control Processes, Far Eastern Branch of the Russian Academy of Sciences; 5 Radio Street, Vladivostok, Russia