PARALLELIZATION 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.

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.

Conference: The paper is selected from XIVth International Conference "Knowledge-Dialogue-Solution" KDS 2008, Varna, Bulgaria, June-July 2008

Introduction

Production systems (or rule-based systems) are used to develop knowledge-based systems [1]. The set of rules describes the problem-solving method of the system application. Production systems are preferable to algorithmic languages as rules usually require terminology of the application domain and are assigned by its ontology [2] which allows to get a problem-solving method understood by the user.

In confluent production systems, the output does not depend on the order of rules of the logical inference. It means that all the rules are independent of each other, i.e. the confluent production language does not need extra language constructions for writing parallel programs and thus for constructing parallel systems to solve application tasks in knowledge-based systems. This differs them from other classes of production systems and parallel programming systems based on logical languages [3-5].

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 and design knowledge-based systems 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 must 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.

[1] This paper was made according to the program № 14 of fundamental scientific research of the Presidium of the Russian Academy of Sciences, the project 06-I-П14-052

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, $S_1(X_1), \ldots, S_k(X_k)$ are simple formulas, $X, X_1, \ldots, X_k$ are vectors of terms];

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

The formula before the symbol «→» is a production antecedent, the formula 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, $v_2, \ldots, v_{i-1}$ can be free variables of term t_i. The sequence of descriptions can be empty. All variables $v_1, v_2, \ldots, v_m$ 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, $w_1, w_2, \ldots, w_n$ are variables, rule is of the kind (1). Variables $w_1, w_2, \ldots, w_n$ 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($zw_1, zw_2, \ldots, zw_n$) where NameSch is a scheme name, $zw_1, zw_2, \ldots, zw_n$ 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 «¤» $\in \{\cup, \cap, \backslash\}$ 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 antedecent is met: $IF(\pi_j) \cap THEN(\pi_i) \neq \varnothing$ where $IF(\pi_j) = \{o_1', \dots, o_a'\}$ – a set of terms of a domain included in the antedecent of the rule π_j, $THEN(\pi_i) = \{o_1'', \dots, o_b''\}$ – 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_l, $i \neq j$. 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.

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) ∩ IF(k) ≠ ∅
  then iChangedLoopAr[k] = 1.
4. Data = Data ∪ 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 = ∅ & 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 loading 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.

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 one has to know not only the characteristics of the graph but also architecture and system constraints imposed by the computing environment. There may be the following constraints:

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.

Hardware constraints in the form of relatively small number of free processes, memory allocation according to processes and temporary delays in the course of data passing between processes on the one hand and multilevel module calls on the other hand constrain module execution in parallel. Therefore sequential module execution can be considered optimal. In the course of the rule computation there can be a number of module calls, then the main process does not launch new rules for computing any more, completes the rest and then passes control to each called module one by one.

Bibliography

[1]. Gavrilova T.A., Khoroshevsky V.F. Intellectual System Knowledge Bases. (In Russian) – SPb.: Piter, 2000.

[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]. M. Wallace, St. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997. http://citeseer.ist.psu.edu/wallace97eclipse.html http://citeseer.ist.psu.edu/update/38822

[4]. NESL - A Parallel Programming Language. http://www.cs.cmu.edu/~scandal/nesl.html

[5]. Boon S. Ang, Derek Chiou, Larry Rudolph and Arvind. The START-VOYAGER Parallel System. Massachusetts Institute of Technology, Laboratory for Computer Science. http://citeseer.ist.psu.edu/ang98startvoyager.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