Implementing and analyzing the multi-threaded LP-inference

S Yu Bolotova¹, E V Trofimenko¹ and M V Leschinskaya³
¹ Associate professor, The Department of Applied and System Software, Voronezh State University, Voronezh, Russia
² Master, The Department of Applied and System Software, Voronezh State University, Voronezh, Russia

E-mail: Bolotova.svetlana@gmail.com

Abstract. The logical production equations provide new possibilities for the backward inference optimization in intelligent production-type systems. The strategy of a relevant backward inference is aimed at minimization of a number of queries to external information source (either to a database or an interactive user). The idea of the method is based on the computing of initial preimages set and searching for the true preimage. The execution of each stage can be organized independently and in parallel and the actual work at a given stage can also be distributed between parallel computers. This paper is devoted to the parallel algorithms of the relevant inference based on the advanced scheme of the parallel computations "pipeline" which allows to increase the degree of parallelism. The author also provides some details of the LP-structures implementation.

1. Introduction
Production systems became an essential part of various applications in a number of areas such as engineering, medicine, IT and many others [1]. The development of the expert systems as well as the theory of learning and problem solving solutions is dependent to a large extent on the production systems. Despite the fact that they're well known in the areas of computer science and mathematical research, they still attract significant interest which is proved by many recent papers [2–5].

At the same time production systems might run very slowly due to the high computational requirements. It’s quite common for them to perform an intensive exchange with external memory and exponential number of computational operations. Optimizing the performance of the production systems can make them more reliable and bring attention of those experts who currently avoid working with such systems due to the serious requirements they demand [1].

The purpose of this paper is to focus on the possible ways of acceleration of the production system execution. Though there might be various ways to do so the authors chose an approach based on a parallel computing. As part of the practical implementation the paper presents the architecture of a program solution for the extended production systems and describes an object-oriented class LPStructure containing all the properties and methods that are necessary for the implementation. It heavily uses the multi-threading at its core part for searching logical reductions and solutions of the production logical equations [6]. The class was developed using C++ language and STL library and built as a dynamic C-library, called LPStructure.dll.

This article supplements the previous paper [7] by adding the practical aspects of the program implementation of the parallel LP-structures.
2. A general description of the algorithm

The considered software system implements the relevant parallel backward chaining based on the solution of the productional logical equations and for this purpose it uses mathematical algorithms for finding the truth preimage and accelerating backward inference [6]. Hereafter we use the notations and definitions from the mentioned article.

LP-structure (lattice-production structure) — is the lattice with an additional binary relation set on it, which has a number of production-logical properties [6]. The relevant backward inference strategy is aimed at minimization of a query number to the external information source (either to a database or an interactive user). The inference based on equation solving starts with the creation of all minimal initial preimages in the LP-structure for the atoms that correspond to the values of an examined object. Using this constructed set it is enough to find the preimage that contains only true facts. If founded it makes possible to make a conclusion about the corresponding value of the object. The effective way to achieve that, is to prioritize viewing of the preimages containing the values of the most relevant objects [6]. These are first of all the objects, whose values are present in a maximal number of the constructed preimages. A negative answer to a unique query eliminates all subsequent queries about the elements of a facts subset. Along with a significantly reduced number of queries, when using LP-inference, the preference is given to testing the sets of facts of a minimal cardinality.

However, experiments have shown that the process of simultaneous construction of the minimal initial preimages for large knowledge bases and their "deep" structure may require an excessive amount of computational resources of the computer. Considering this relevant LP-inference method was modified to use parallel computing algorithms. Parallel relevant LP-inference allows speeding up the process of constructing sets of facts that are required in the inference, and their further processing [8].

As an illustration we shall give simplified descriptions of algorithms of a relevant LP-inference and a parallel LP-inference.

Input:
- initial facts — true (set $T$), false (set $F$);
- set of rules $R$;
- hypothesis $b$.

Output:
- true initial preimage $X^0 = \{X_1^0, ..., X_n^0\}$ for $b$
  (or $X^0 = \text{null}$).

// Relevant LP-inference
$X^0 = \text{null}$
$\{X\} = \text{getPreImages}(b)$
while $X^0 = \text{null}$ and $\{X\} \neq \emptyset$ do
  $k = \text{getRelevantIndex}(\{X\}, T)$
  Ask $(x_k)$
  foreach $X_j \in \{X\}$ do
    if $X_j \subseteq T$ then
      $X^0 = X_j$
      break
  end
  if $X_j \not\subseteq F \neq \emptyset$ then
    $\{X\} = \{X\} \setminus X_j$
end

2
The function \( \text{Ask}(x) \) asks a user (or an external database) about the truth of an atomic fact \( x \) and according to the answer modifies sets \( T \) and \( F \). Function \( \text{getPreImages}(b) \) solves the equation with a right-hand side \( b \), i.e., it constructs set \( \{X\} \) of all minimal initial preimages for atom \( b \).

The equation solving is carried out by splitting relation \( R \) into "layers" \( R_t, t \in T \), each containing no more than one solution [6]. The layer has the maximum possible set of relationship pairs with the unique right-hand sides, and two layers differ in one pair at least. In each separate layer the process of the equation solving is reduced to the problem of finding the set of initial nodes of the graph \( G_{R,t} \), corresponding to a layer.

The work for different layers is organizing independently and in parallel. The primary application thread creates secondary threads (which are limited by the \( \text{MaxThreads} \) — the maximum number of threads), passing them a block of data. Hereinafter the "threads" are threads of execution [8]. Created thread searches the solution of production logic equation in a separate layer. When the sufficient number of preimages is calculated (or all preimages), the LP-inference program module immediately checks its "truth", sending queries to the external information source, if necessary. If one of the preimages is not true, the program stores the information about false facts (it reduces the possible number of rules, required for the next equation solution) and calculates the next preimage cluster which also significantly speeds up the process. After this calculation the thread exits.

In the case of large amount of running threads, in accordance with the Amdahl's law, the efficiency of parallel computing is declined. Frequent creation and termination of threads with a short lifecycle and switching their contexts increase the amount of resources. Therefore, in implementing a thread pool mechanism is used [8]. The maximum pool size is limited by parameter. Its value must be greater than the number of processors in order to achieve the maximum simultaneity.

The main thread takes a working thread from the pool and passes it the necessary data for processing. When the number of active threads reaches its maximum, the request is queued. If the number of active threads is less than the maximum, a new thread is created. When the number of active threads is maximized, request is put to the queue and starts waiting for the release of one of the threads.

For higher efficiency the processes of preimages calculation and their further analysis are performed asynchronously. Until a solution is found, another preimages cluster is stored in the dynamic queue \( Q \).

The main thread takes the preimages clusters from the queue and (if one of the secondary threads is free) initiates the parallel analysis of their items relevance. The most relevant elements are placed in the priority queue \( P \), based on a binary heap. The maximum queue size is specified by the \( \text{maxRelevanceQueueSize} \) parameter. Such data organization is used to sort investigated elements according to their relevance, to change their order by adding new elements, if necessary, and to access the element with maximum value in the constant time. The process of identifying the relevant objects of constructed clusters continues until a solution is found.

The general structure of the parallel algorithm of the relevant inference can be represented by the following scheme (figure 1).

![Figure 1. Algorithm structure scheme.](image-url)
The parallel algorithm of the relevant inference is presented below.

//The parallel relevant inference
X₀ = null
while X₀ = null do
asynchronous call \( \{X\} = \text{getPreImagesCluster}(b) \)
if \( \{X\} \neq \emptyset \) then
    // Add the cluster to queue Q
    addClusterToQueue(Q, \{X\})
end
// If the queue is not empty
while not clustersQueueIsEmpty(Q) do in parallel
    // Extract the cluster from queue
    \{Y\} = extractClusterFromQueue(Q)
    foreach \( Y_j \in \{Y\} \) do
        if \( Y_j \cap F \neq \emptyset \) then
            \{Y\} = \{Y\} \setminus Y_j
        end
    end
while X₀ = null do
    // If P has the free place
    asynchronous if \( \text{relevanceQueueSize}(P) < \text{maxRelevanceQueueSize} \) then in parallel
        // The relevant object index k ant its relevance
        foreach \( Y_j \in \{Y\} \) do
            if \( Y_j \cap F \neq \emptyset \) then
                \{Y\} = \{Y\} \setminus Y_j
            end
        end
        k = getRelevantIndex(\{Y\}, T, relevance=0)
        // Insert object to the priority queue
        InsertToQueue(P, k, relevance)
    else
        Waiting() // Wait until the queue is freed
    end
asynchronous while \( \text{relevanceQueueSize}(P) \neq 0 \) do
    // Element with a maximum relevance
    k = ExtractMaxFromQueue(P)
    Ask \( y_k \) // Check the fact truth
    foreach \( Y_j \in \{Y\} \) do
        if \( Y_j \subseteq T \) then
            X₀ = Y_j
            break
        end
        if \( Y_j \cap F \neq \emptyset \) then
            foreach \( y_m \in \{Y_j\} \) do
                //Element in P - lower relevance
                if elemIsMemberOfQueue(\( y_m \), P) then
                    \( \ldots \)
The function `getPreImagesCluster(b)` launches the process of computing the fixed number of minimal initial preimages for the atom $b$. As already mentioned, this process is carried out in separate threads to provide the greatest efficiency.

Function `getRelevantIndex({Y}, T, relevance = 0)` finds the relevant element and its relevance `relevance` for each cluster in a separate thread. Then the element and its relevance are placed in the priority queue. If this element is already there, the total relevance is calculated and the queue is rebuilt.

At the same time the `Ask(y_k)` function asks the external source about the truth of the most relevant fact, extracted from the priority queue. If the negative request answer is obtained the preimages, which contain an $y_k$ element, excluded from consideration, and already investigated elements relevances are modified.

As an illustration we shall give the computing model as an acyclic graph $G = (V, R)$, where the set of operations is represented as a set of vertices $V = \{1,...,|V|\}$, and the information dependencies between operations — as a set of arcs $R$ [9]. The arc $r = (i, j)$ means that the operation $j$ uses the result of another operation $i$. The operations of the algorithm that are not connected by an arc can be parallelized.

A possible way to describe the parallel execution of the algorithm solution search in a separate layer is shown in figure 2.

Hereafter is an algorithm of `getPreImagesCluster(b)` function, where $R'$ is a set of layers $\{R_i\}$. As already mentioned, the solution of each layer is calculated in a separate thread. The maximum number of threads in the pool is limited by `MaxThreads` parameter. The number of active threads is stored in the `countUsedThreads` variable. If the pool has a free thread, then it schedules this thread for execution by passing it a pointer to a `FindEquationSolution(R_i)` function, that finds a solution in the next layer $R_i$.

The algorithm of `getPreImagesCluster(b)` function is presented below. Denote the set of layers as $\{R_i\}$. As already stated, the solution of each layer is calculated in a separate thread. The number of currently active threads is stored in the `countUsedThreads` variable. If a pool has a free thread, the new thread is started and it calls the `FindEquationSolution(R_i)` function, which finds a solution of the equation in another layer $R_i$.

```plaintext
// Finding the solutions of the equation for each layer in a separate thread
foreach R_i \in R' do in parallel
    if countUsedThreads < MaxThreads then
        BeginThread() // Start the thread
        countUsedThreads++
        FindEquationSolution(R_i)
        ExitThread() // End the thread
        countUsedThreads--
    else
        Waiting() // Wait until the thread is freed
end
```
Function $\text{getRelevantIndex}(\{Y\}, T, \text{relevance})$ finds the index $k$ of the any of the most relevant and not checked for truth facts contained in the current cluster elements $\{Y\}$ and its $\text{relevance}$. The function uses two mentioned above indices of relevance to minimize the number of calls of the function $\text{Ask}(x_k)$. The process of identifying relevant objects is very expensive, so it is also parallelized. For each number of facts the system calculates the number of preimages, containing these facts using multiple threads (number of threads is limited by $\text{MaxThreads}$ parameter). The number of current active threads is stored in the $\text{usedThreadsNumber}$ variable. The fact, that belongs to the maximum number of preimages, increments its relevance score by 1. When accessing shared resources threads are synchronized using the critical sections mechanism. We also present computing graph for this algorithm (figure 3).

A simple variant of the function $\text{getRelevantIndex}$ is given below.

```
//Parallel relevance counting
int getRelevantIndex (\{Y\}, T, relevance)
if ( \text{usedThreadsNumber} < \text{MaxThreadsCount} ) or
( \text{countUsedThreads} < \text{MaxThreads} ) then
    BeginThread() // Start the thread
```
if (usedThreadsNumber < MaxThreadsCount) then
    usedThreadsNumber++
    usedSelfThreads = true
else
    countUsedThreads++
    usedSelfThreads = false
end
k=MostRelevantFind(relevance) // The relevant atom index
ExitThread() // End the thread
if usedSelfThreads then
    usedThreadsNumber--
else
    countUsedThreads--
end
return k
else
    Waiting() // Wait until the thread is freed
end

The MostRelevantFind(relevance) function finds the index $k$ of the most relevant object and its relevance.

The benefits of parallel LP-inference are confirmed experimentally. When processing the large knowledge bases the use of parallel algorithms speeds up LP-inference by up to 30%.

3. Implementation

The proposed algorithms were implemented as a dynamic C-library called Parallel LPStructure.dll. Below are some practical considerations for the LP structures. In accordance with [2], it is proposed to use a flexible bit vector which length can be adjusted by a "client" program to store lattice elements. The dimension of such a vector is specified by two parameters — the length in bytes of one cluster (eItemSize) and the number of these clusters (eLengthItems).

The value eItemSize determines the size of the cluster — a part of the vector that is processed by a single logical operation of the C++ language. This value is stored as a static class constant and can be changed with recompiling. By setting eItemSize to 1, it is possible to restructure a bit vector to a sequence of bytes, which results in memory saving and makes the vector structure more transparent in terms of understanding the details of the algorithms. Setting eItemSize to 8 can significantly improve the speed of operations on vectors, since many cycles in the program will be 8 times shorter, and 64-bit operations will be more efficient for computers with the appropriate memory capacity.

The number of clusters for a bit vector is stored in the eLengthItems integer field. The constructor of the LPStructure class receives a corresponding integer value for setting this field as a parameter. This fact allows the "client" program to dynamically adjust the length of the bit vectors when creating an LP-structure, based on the lattice size and thus it allows to adjust the amount of the available knowledge base accordingly.

In the ParallelLPStructure class Elem and Pair types are defined through renaming the types of single and a tuple of integers. The type Elem assumes storing the address of the lattice element, namely, the bit vector corresponding to the element. The type Pair contains a contiguous pair of Elem values, so that it is intended to store a pair of elements (more precisely, their addresses) from some binary relation on the lattice. Of course, types Elem and Pair could also be implemented in a form of classes with overloaded operators (|, &, <=), corresponding to operations on the lattice. However, the decision to use the simplest types turns out to be more consistent in terms of the adopted strategy of saving memory.

When using STL libraries to implement LP structures, it's possible to use effectively parameterized vector and set container classes, as well as their combination. It is especially suitable to use set<Elem>
defined as ElemContainer for the representation of sets of lattice elements. To store the generic binary relations defined on the lattice, it’s convenient to define the Pair-Container type as set <Pair>. Some algorithms use paired vectors — PairVector (vector<Pair>) to store some intermediate data. Taking into account all the previously mentioned types, the canonical binary relation might be described by the vector of sets vector<ElemContainer> named ElemContainerVector.

The value of aMaxThread specifies the maximum number of worker threads during execution. It is stored as a static constant field of the object-oriented class ParallelLPStructure.

Hereby we’ve listed the main types used in the ParallelLPStructure class.

4. Experiments results

To illustrate the parallel relevant LP-inference capabilities we created about five hundred knowledge bases that differed significantly in the scope of the rules sets. As a result of testing we obtained the timing indicators of the normal and parallel relevant backward inference. The maximum number of threads was also limited in the range from 1 to 20 in increments of 3. The results of the examinations were processed in the package Statistica 6.

In this package the one-factor and two-factor analysis of variance were carried out. The homogeneity of group dispersions is verified using the Levene test, the results of which are shown in figure 4.

![Figure 4. The Levene test results.](image)

The results show that the variances do not differ (P << 0.05). Consequently, one of the conditions for applying the parametric variant of the variance analysis is not fulfilled. However, this method is very stable in case of deviation from the specified requirement, therefore it is possible to continue the study [10].

To verify the normal distribution of the analyzed data, the graphs of the normal probabilities and the histograms shown in figures 5 and 6 were plotted.

The figures show that the distribution is close to normal. In addition, the sample sizes allow a further research. The result of the variance analysis is shown in figure 7.

Since the error value for the null hypothesis shows that there is no connection between the number of threads and the execution time is P << 0.05, we can conclude that the time is statistically significantly different depending on the number of threads.

A posteriori analysis using the Tukey method (figure 8) showed that a statistically significant difference in the execution time exists between pairs of parallel relevant inference with 5 threads and 15 threads, as well as parallel relevant inference with 8 and 15 threads.

Two-factor analysis of variance (figure 9) revealed a significant impact of the number of threads and the number of facts, as well as their mutual impact on the duration of implementation (P << 0.05).

Nonparametric one-factor variance analysis of Kraskel-Wallis (figure 10) demonstrated the statistically significant differences between the compared groups (P << 0.05).

For clarity, we also constructed a sweep diagram (figure 11) and a dependence graph of the algorithm execution time and the threads number (figure 12).

The diagram shows the differences between the averages. It can be concluded that the use of 8 threads yields the highest time efficiency. Further increase in the number of threads reduces efficiency, increasing execution time.
Figure 5. Normal probability graphs.

Figure 6. The histogram.
**Figure 7.** The result of variance analysis.

| Effect          | SS   | Degr. Of Freedom | MS   | F     | p     |
|-----------------|------|------------------|------|-------|-------|
| Intercept       | 1290,503 | 1              | 1290,503 | 632,6348 | 0,00000 |
| Threads         | 40,327 | 5              | 8,065 | 3,9538 | 0,001742 |
| Error           | 599,727 | 294            | 2,04  |       |       |

**Figure 8.** The posteriori analysis using Tukey method results.

| Cell No. | Threads | (1) 2,2824 | (2) 2,0739 | (3) 1,8466 | (4) 1,4901 | (5) 2,0687 | (6) 2,6826 |
|----------|---------|------------|------------|------------|------------|------------|------------|
| 1        | 1       | 0,978206   | 0,647656   | 0,061704   | 0,975810   | 0,726543   |
| 2        | 2       | 0,978298   | 0,968346   | 0,317385   | 1,000000   | 0,271218   |
| 3        | 5       | 0,647656   | 0,968346   | 0,812921   | 0,971366   | 0,040124   |
| 4        | 8       | 0,061704   | 0,317385   | 0,812921   | 0,327529   | 0,262113   |
| 5        | 10      | 0,975810   | 1,000000   | 0,971366   | 0,327529   | 0,262113   |
| 6        | 15      | 0,726543   | 0,271218   | 0,040124   | 0,000440   | 0,262113   |

**Figure 9.** The results of two-factor analysis of variance.

| Effect          | SS   | Degr. Of Freedom | MS   | F     | p     |
|-----------------|------|------------------|------|-------|-------|
| Intercept       | 1290,503 | 1              | 1290,503 | 12592,53 | 0,00  |
| Facts           | 550,367 | 9              | 61,152  | 396,71  | 0,00  |
| Threads         | 40,327 | 5              | 8,065   | 78,70   | 0,00  |
| Facts*Thread    | 24,764 | 45             | 0,550   | 5,37    | 0,00  |
| Error           | 24,596 | 240            | 0,102   |         |       |

**Figure 10.** The results of nonparametric Kruskal-Wallis one-way analysis of variance.

| Depend.: Time   | Code | Valid N | Sum of Ranks | Mean Rank |
|-----------------|------|---------|--------------|-----------|
| 1               | 1    | 50      | 8124,500     | 162,4900 |
| 2               | 2    | 50      | 7609,000     | 152,1800 |
| 5               | 5    | 50      | 6967,000     | 139,3400 |
| 8               | 8    | 50      | 5860,500     | 117,2100 |
| 10              | 10   | 50      | 7664,500     | 153,2900 |
| 15              | 15   | 50      | 8924,500     | 178,4900 |
A similar conclusion can also be drawn from the graph of the algorithm execution time and threads number dependence. On the X axis the number of rules in the knowledge bases is marked, on the Y axis — the execution time in seconds, and the colors of the curves reflect the number of threads with parallel algorithm implementation. For the case of a parallel relevant inference with 8 threads and the relevant inference, a t-criterion is used (figure 13). Its results indicate the presence of statistically significant differences between the mediums (P << 0.05).

Figure 11. The span diagram.

Figure 12. The graph of the dependence of the execution time of the algorithm from the number of threads.
All the demonstrated results indicate a decrease in the solution finding time in case of using multithreading by 20–25% on average. In this situation, we are talking about a statistical indicator, since even with normal output there is some probability of accidental achievement of the best result "from the first time."

5. Conclusion

It’s quite common for the algorithms that process large knowledge bases to consume significant computational resources which makes it a big challenge for the researches to find a way to improve the efficiency of the mathematical algorithms and program implementations. The described library uses the theory of LP-structures to speed up the backward inference production systems along with implementing this theory using mutli-threading approaches. The effectiveness of parallel LP-inference is proved by experimental results: the execution time of the considered processes decrease on average by 30%.

Application of the methods described is demonstrated on the simple architecture of production systems in order to focus on the methods proper. A similar approach can also be applied to production systems having a more complex structure; many models in informatics have production nature, and the structures of information representation are often hierarchical. Besides, the methodology presented can be applied to logical systems of a more general form, including non-classical.

References

[1] Gupta A 1987 Parallelism in production systems (Pitman)
[2] Liberatore P 2008 Artificial Intelligence 172(23) 265–299
[3] Maciol A 2008 Expert Systems with Applications 34(3) 1825–1836
[4] Poli R and Langdon W B 2006 Artificial Intelligence 170(11) 953–982
[5] Katerinenko R S and Bessmertnyi I A 2009 Programming and Computer Software 37(4) 197–199
[6] Makhortov S D 2009 Mathematical Foundations of Artificial Intelligence: Theory LP-structures for the construction and studying of knowledge models of production type ed Vasenin V A (Moscow: Publ. MCCME)
[7] Bolotova S Yu and Makhortov S D 2015 Modern Informatization Problems in the Technological and Telecommunication Systems Analysis and Synthesis: Proceedings of the XX-th International Open Science Conference ed Kravets O Ja (Yelm: Science Book Publishing House LLC) pp 339–347
[8] Richter J 1999 Programming applications for Microsoft Windows (Microsoft Press)
[9] Voevodin V V and Voevodin VI V 2002 Parallel computing (St. Petersburg : BHV-Petersburg)
[10] Vukolov E A 2008 Bases of the statistical analysis. Workshop on statistical methods and operations research using the packages STATISTICA and EXCEL (Moscow: Forum)