# Yet Another Parallel Hypothesis Search for Inverse Entailment

Hiroyuki Nishiyama and Hayato Ohwada

Faculty of Sci. and Tech. Tokyo University of Science†,
2641 Yamazaki, Noda-shi,
CHIBA, 278-8510, Japan
hiroyuki@rs.noda.tus.ac.jp, ohwada@rs.tus.ac.jp,

**Abstract.** In this study, we design and implement a powerful Inductive Logic Programming (ILP) system to conduct a parallel hypothesis search for inverse entailment. One of the most important parts of ILP is speeding up the hypothesis search, and a number of parallel hypothesis exploration methods have been proposed. Recently, the Map-Reduce algorithm has been used for large-scale distributed computing, but it is difficult to apply such cloud technology to the ILP hypothesis search problem. By contrast, we designed a method that can dynamically distribute tasks for the hypothesis search, and implemented a network system in which modules autonomously cooperate with each other. We also conducted a parallel experiment on a large number of CPUs. Results confirm that the hypothesis search time is shortened according to the number of computers used, without reducing the optimality of the generated hypothesis.

## 1 INTRODUCTION

Recently, various analytical approaches to a large amount of data have been developed. For example, the Map-Reduce model consists of a large number of workers and one master, and uses on-line distributed-computing for quick analysis. However, Map-Reduce has limited efficiency. With Inductive Logic Programming (ILP) [2], which uses artificial intelligence, we can induce the cause (rules) by searching for relationships in information in the results (resulting events). However, searching for relationships in information is not efficient distributed processing in the Map-Reduce scheme, because new relationships are found after searching for relationships in background knowledge to generate rules. Therefore, the application of ILP for large data is not realistic because ILP requires much learning time. To solve this problem, various studies have been conducted in an effort to speed up ILP [1, **?**,**?**]. Although the problems were partially solved, the process was not sufficiently sped up based on the number of processors provided, and the quality of the generated rules was not optimal, resulting in difficulty in its use as a practical tool.

Considering these issues, we designed and implemented a new parallel processing system for ILP. With this system, the worker itself has an autonomous

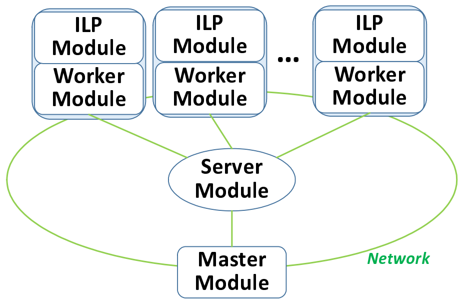

**Fig. 1.** System configuration of the ILP parallel computation system

function, unlike Map-Reduce. When a worker has no task (e.g., immediately after start-up or immediately after completion of a task), the worker accepts a divided task from another worker and starts the task. When the workload (the number that the worker must search for relationships in information) reaches a fixed quantity, the worker requests other workers to process the divided task. After the request for the first task issued by a master is implemented for one worker, autonomous process distribution is started among workers, and all existing workers are engaged (saturation of the task). Because the divided task continues to be repeated among workers, all workers finally complete all processing at approximately the same time, and the master receives the processing result (generated rules).

Our parallel processing system has the following features.

- 1: The master does not need to consider the division of the process in advance, as the master of the Map-Reduce scheme does.
- 2: All workers work until the end (no free time).

The first feature indicates that the proposed system does not require predivision processing of the master, which is used in conventional Map-Reduce. The second feature means that speedup can be achieved. In our study, we implement the proposed parallel ILP system and demonstrate its effectiveness by conducting parallel-learning experiments using drug discovery data.

## 2 SYSTEM DESIGN

### 2.1 System Configuration

Our ILP parallel-processing system consists of four modules (Fig.1). We designed each module as follows.

- Server module for negotiation between modules
  The server module mediates negotiation messages (e.g., ”request,” ”accept,” and ”commit”) regarding the task shared by workers. This module broadcasts to connected workers and the master. This function is used during negotiation to find an acceptor of the divided task.

- Master Module
  The master module sends a task (main task) request to the worker module (worker) and collects results (generated rules) from the worker module when all divided tasks of the worker modules are finished. By connecting with the server module, this module can negotiate with worker modules and collect the negotiations between worker modules to recognize the processing situation of all worker modules.
- Worker Module
  The worker module manages an ILP module and negotiates with other worker modules via the server module. When this module has no task, it accepts the shared task from other worker modules, transfers the task data to its own ILP module, and starts learning. In addition, this module sends a request message for the shared task to other workers when the ILP module needs to divide the task.
- ILP Module
  Basically, the ILP module has the same functions as in traditional ILP. Receiving task data from the worker module, the ILP module starts the learning process. It searches for generating rules; thus, the amount of search processing that this module must perform increases. If the amount of search processing exceeds a threshold, this module divides the search processing and generates a divided task. It then sends a request for the divided task to the worker module. In addition, when all search processing (its own learning process) is finished and the requesting task exists (waiting for an ”accept” response), this module cancels the request of the task via the worker module and starts the task itself. When all search processing is finished and the requesting task no longer exists, this module declares the end of the task to the worker modules. As a result, this module starts other divided tasks from other worker modules.

## 2.2 Communication Protocol Between Modules

In our system, a master module communicates with worker modules and worker modules communicate with each other via the network (Fig. 1). We prepared two communication protocols. One is the ”Negotiations Protocol between Modules” to find a worker module that can execute a task (main task or divided task). The other is the ”Task Content Transmission Protocol” to send task contents to a worker. The details of each protocol are as follows.

- Protocol 1. Negotiations Protocol between Modules
  This protocol communicates via the server module. All messages are broadcast to all worker modules and a master module. This protocol is used for negotiation messages (e.g., ”request,” ”accept,” and ”commit”) to find a worker module that can execute a task.
- Protocol 2. Task Content Transmission Protocol
  After a worker module is determined by protocol 1, protocol 2 is used for transmitting the task contents to the module. Communication between mod-

ules is carried out by peer-to-peer communication (not via the server module).

As a result, all worker and ILP modules perform a task by repeatedly requesting shared tasks (i.e., saturation of the divided task). In the saturation state, all worker modules send ”request” messages regarding the shared task to other worker modules and await an ”accept” message. Conversely, each worker module is ready to receive a request task from all other worker modules. Thus, a worker module that has finished its own task sends an ”accept” message to another worker module and receives a new shared task.

Shared tasks are requested and received until a learning rule is generated. As a result, worker and ILP modules operate without spare time.

## 3 IMPLEMENTATION AND EXPERIMENT

In the present study, we implemented a rule generation engine for ILP [2] and each module described above using the Java programming language. In addition, in this implementation experiment, we temporarily set the threshold (number of searches to be conducted) at 200 for the division request in the ILP module. We used two 6-CPU computers and two 4-CPU computers in our parallel processing experiment.

### 3.1 Speedup Experiment

We used two 6CPU computers to conduct an experiment to measure speedup and used middle-scale data on drug discovery analysis as training data. One by one, we increased the number of CPUs from 1 to 12, and performed 12 experiments. In our system, a worker module (and an ILP module) used one CPU. The experiment results are presented in Table 1. We obtained a speedup of 7.4 using a 12-CPU unit for this exercise. However, this problem was small-scale and did not provide enough speedup.

### 3.2 Large-Scale Parallel Experiment

Using four computers, we performed application experiments on a large-scale example of a drug discovery problem. In this experiment, we used 1 CPU, 12 CPUs (two 6-CPU computers), and 20 CPUs (two 6-CPU computers + two 4-CPU computers) in a total of three experiments. The experiment results (Table 2) confirmed that speedup corresponded to the number of CPUs used. For this drug-discovery problem, it conventionally took at least 15h using a parallel-processing system; however, with our proposed method, one learning process required less than 1h. Thus, experiments can be easily reconducted by adjusting the parameters and can achieve better research results.

**Table 1.** Parallel experiments using 12 CPUs (two 6CPU computers)

| The Number of CPU | Processing time(sec.) | Speedup |
|---|---|---|
| 1 | 724 | 1.000 |
| 2 | 387 | 1.871 |
| 3 | 269 | 2.691 |
| 4 | 218 | 3.321 |
| 5 | 192 | 3.771 |
| 6 | 174 | 4.161 |
| 7 | 146 | 4.959 |
| 8 | 131 | 5.527 |
| 9 | 116 | 6.241 |
| 10 | 107 | 6.766 |
| 11 | 102 | 7.098 |
| 12 | 98 | 7.388 |

**Table 2.** Parallel experiments using 20 CPUs (two 6-CPU computers and two 4-CPU computers)

| The Number of CPU | Processing time(sec.) | Speedup |
|---|---|---|
| 1 | 56372 | 1.000 |
| 12 | 5723 | 9.850 |
| 20 | 3563 | 15.821 |

## 4 DISCUSSION

For the above parallel processing experiments, we measured the communication time and processing steps between workers to investigate how the workers (worker module and ILP module) depend on each CPU. For this measurement, we used six workers and one 6-CPU computer. In this reconducted experiment, we implemented measuring functions on each module. Thus, the processing time is longer than in the experiment results using 6 CPUs (Table 1).

We measured the time required for learning and for receiving the task data of each worker. Table 3 lists the processing time of each worker. The end time for all tasks is 205.69sec.

**Table 3.** Learning time and receiving time of each worker in the parallel-processing experiment using six workers (6 CPUs) (sec)

| Worker ID | 1 | 2 | 3 | 4 | 5 | 6 | Average |
|---|---|---|---|---|---|---|---|
| Learning time | 195.62 | 194.98 | 196.95 | 193.52 | 194.22 | 190.86 | 194.36 |
| Receiving time | 5.47 | 5.49 | 4.62 | 6.22 | 8.05 | 8.08 | 6.32 |
| Total operation time | 201.10 | 200.47 | 201.58 | 199.74 | 202.28 | 198.94 | 200.68 |

**Table 4.** Learning time and receiving time of each worker in the parallel-processing experiment using six workers (6 CPUs) (%)

| Worker ID | 1 | 2 | 3 | 4 | 5 | 6 | Average |
|---|---|---|---|---|---|---|---|
| Learning time | 95.10 | 94.79 | 95.75 | 94.08 | 94.42 | 92.79 | 94.49 |
| Receiving time | 2.66 | 2.67 | 2.25 | 3.02 | 3.92 | 3.93 | 3.07 |
| Total operation time | 97.77 | 97.46 | 98.00 | 97.11 | 98.34 | 96.72 | 97.57 |

Table 3 indicates the processing time of each worker. We confirmed that each worker spent an average of 94In a previous study [3], much time was necessary for sending and receiving between workers. In the present study, we successfully shortened the time required for sending and receiving, because the implementation of the proposed system used the object communication function and serialize function of Java API.

## 5 CONCLUSIONS

In the present study, we designed and implemented a parallel system of Inductive Logic Programming (ILP) to realize a parallel hypothesis search for inverse entailment. The hypothesis generation of ILP requires search processing, and the result of the hypothesis generation influences the next hypothesis generation. Thus, ILP cannot be parallelized using the Map-Reduce method that was developed based on cloud technology. To address these issues, we designed a method that can dynamically distribute tasks for search processing, and implemented a network system in which modules autonomously cooperate with each other. In addition, we enabled parallel experimentation using a large number of workers (CPUs). Results confirmed that we shortened the processing time depending on the number of computers used, without reducing the quality of the rule generated.

## References

1. Andreas Fidjeland, Wayne Luk and Stephen Muggleton: Customisable Multi-Processor Acceleration of Inductive Logic Programming, *Latest Advances in Inductive Logic Programming*, pp. 123-141, 2014.
2. Fumio Mizoguchi and Hayato Ohwada: Constrained Relative Least General Generalization for Inducing Constraint Logic Programs, *New Generation Computing 13*, pp. 335-368, 1995.
3. Hayato Ohwada, Hiroyuki Nishiyama and Fumio Mizoguchi: Concurrent Execution of Optimal Hypothesis Search for Inverse Entailment, *Inductive Logic Programming, Lecture Notes in Computer Science Vol. 1866*, pp. 165-173, 2000.
4. Ashwin Srinivasan: Parallel ILP for distributed-memory architectures, *Machine Learning, Vol. 74, Issue 3*, pp 257-279, 2009.