Distributed Jess on a Condor pool

DANA PETCU*, MARIUS PETCU**
*Computer Science Department
Western University of Timisoara
B-dul Vasile Parvan 4, 300223 Timisoara
Institute e-Austria Timisoara
**Aurel Vlaicu University, Arad
ROMANIA
http://web.info.uvt.ro/~petcu

Abstract: - This Distributed rule-based systems, like a distributed version of Jess, are needed for real applications. The proposed architecture for such system is based on several instances of Jess distributed in a Condor pool and a central manager allowing the cooperation between these instances. Efficiency tests are performed on classical benchmarks.

Key-Words: - Distributed rule-based engines, cluster simulations

1 Introduction
The suites of benchmarks, like the one reported in [10], shows that the rule-based running on the current hardware need hours to give the solutions when the number of rules to be fired are of thousand order. In this context, parallel or distributed version of those systems will be welcome. The strategies to improve the speedup using multiple processors (see Section 2) were already discussed in the last twenty years.

We selected Jess as current representative rule-based systems. Java Expert System Shell [9] is a rule-based programming environment written in Java. Inspired by Clips expert system shell [2], it has grown into a complete environment of its own, being today one of the fastest rule engines available. Jess uses the Rete algorithm [7] to compile rules, an efficient mechanism for solving the difficult many-to-many matching problem. One can build with Jess software that has the capacity to reason using knowledge supplied in the form of declarative rules. Jess is used in agent frameworks, expert systems from medicine to cryptography, intelligent tutoring, robotics, and so on [5].

The strategy adopted here to build a distributed rule-based system is one based on client-server model: a manager, the server, registers different Jess instance, the clients, and manages the cooperation between them. The manager can be the same or slightly different for most of the available rule-based systems. Jess has been taken as an example for this study mainly due to its openness to the environment (communication via sockets, ability to create Java objects and call Java methods). The structure of Distributed Jess is detailed in Section 3.

The preliminary tests on Distributed Jess installed on a Condor pool of computers and presented in Section 4 show its efficiency in the case of using a cluster environment.

2 Distributed rule-based systems
A rule-based system consists in a working memory, a set of rules and an inference engine. The working memory is a global database of data (elements) representing the system state. A rule is a condition-action pair. The inference engine is based on a three-phase cyclic execution model of condition evaluations (matching), conflict-resolution and action firing. An instantiation is a rule with a set of working memory elements. A conflict set is the set of all instantiations. In a sequential environment, conflict-resolution selects one instantiation from the conflict set for firing. In a distributed environment, multiple instantiations can be selected for firing simultaneously. Firing an instantiation can add, delete or modify elements in the working memory.

Encompassing a high degree of parallelism, the time performance of rule-based systems can be improved through parallel or distributed processing. The interest in parallel and distributed rule-based systems has been raised after 1984 and the first parallel implementations were already available at the beginning of the last decade. Amaral in [1] presents a comprehensive synthesis of the efforts made before 1994 in providing parallel firing systems.

Concerning the Jess predecessor, Clips, there are several parallel and distributed implementations produced mainly before 1994. For example, in the parallel version reported in [8] and developed in
1994 to run on Intel hypercubes, some added
commands allows parallel calls, a complete version
of Clips runs on each node of the hypercube, only
rule-level parallelism is supported, and parallel
commands enable the assertion and retraction of
facts to and from remote nodes working memory.
Unfortunately, none of those parallel or distributed
versions of Clips constructed around 10 years ago
are anymore available in the public domain.

The interest in distributed rule-based engines has
appear again in last five years in connection with
Jess. Recently Jess was used in conjunction with a
Java implementation of an actor model [4] to write
distributed artificial intelligence applications. In the
distributed computational environment each Jess is
an active independent computational entity able to
communicate with other Jess instances.

The OKEANOS middleware [12] provides an
infrastructure for mobile agents which access
computational services and communicate by passing
messages. The agents are implemented in Java and
contain rule-based knowledge interpreted by Jess.
They consist of two parts. One part is responsible
for managing incoming and outgoing messages and
for transforming messages into Jess-based
declarative rules. The other part, the Jess engine
interprets the incoming rules, new facts are added to
the knowledge base, or existing facts are retracted
from it, depending on the content of the incoming
messages. The resulting knowledge base of a Jess-
agent determines its state and is characterized by its
dynamic change.

Recently, a parallel version of Jess for clusters
based on jPVM was build to support using a
decentralized architecture. It has been used to
improve the speedup of P-system simulators (details
in [11]).

The parallel matching approach parallelizing
only the match phase leads to a limited speedup by
the sequential execution of rules [13]. The multiple
rules firing approach parallelizing the match phase
and the act phase by firing multiple rules in parallel
is more promising, but supplementary costs are due
to synchronization needs. Special techniques like
copy-and-constraints, compatible rules, analysis of
data dependency graph have been used with success
to increase the parallelism. Those techniques are
general and do not exploit the parallelism specific to
the application domains, as it is the case here. The
task-level parallelism approach based on the
functional decomposition of the problem into a
hierarchy of tasks can lead to better results that the
above mentioned ones [13], but the techniques tend
to be ad hoc.

In this context we are interested to build
distributed version of Jess based on task parallelism.

3 Distributed Jess
Jess has the special feature to allow socket
connections. A Jess instance can be only client to
some service.

We assume that the user wants to adopt the
SPMD programming style: he or she writes one
piece of Jess code for all the Jess instances. The
different behaviors of Jess instances are possible due
to the existence of a global variable id defining the
identifier of a specific process.

Figure 1. The centralized architecture of Distributed
Java: Octopus, the server, and several Jess instances, the
clients (in the particular case of 10 clients)

In order to obtain an id, the Jess instance try to
connect to a server, the Octopus, which grants ids in
the order of connection (Figure 1):

Jess>(socket server.domain no_port
Octopus)
Jess>(defglobal ?*id*=(integer
(sym-cat (readline Octopus))))

if Octopus resides on server.domain and listen at
no_port. The Octopus creates a thread for treat each
client requirements and registers all the clients.
The communications between different instances
are done via the socket connection with the Octopus.
The sender writes:

Jess>(printout Octopus id_dest text_send)

the Octopus read the id of the destination and the
text_send, identify the destination and send to the
socket connection of the destination the information. The receiver must read the information from the socket:

```
Jess> (bind ?income_text
    (readline Octopus))
```

The task of matching pairs of sends and receives and the task of dealing with their order are on the user agenda.

A more sophisticated variant will be build in the near future. It will allow the deposit of the transition messages to the Octopus. An income message will be stored into a structure from memory shared by Octopus’ threads, and the receive process at the destination will have two phases: receive request (a specific printout on the socket) and then the above described reading from the socket.

A more complicated scheme of communications was described for Parallel Jess in [11]. In this case each Jess has its local Octopus which deals with sending and receiving messages (acting at the Jess instance requests). The central Octopus of Distributed Maple is replaced by the PVM systems which manages the inter-Octopus communications.

In what follows we discuss a particular case, the one in which the Jess instances can be launched via a Condor batch system.

Condor [3] is a specialized batch system for managing compute-intensive tasks. It provides a queuing mechanism, scheduling policy, and resource management of non-dedicated machines. When the user submits a job, Condor finds an available machine on the network and runs the job on that machine (with checkpoints and job migration if needed).

When we use the SPMD programming style and \( p \) Jess instances of the Jess resident on a shared file system of a Condor pool of computing machines, the user task of launching the Jess instances is simplified. He or she must do only

```
condor_submit todo.cmd
```

where `todo` can be similar to

```
universe = java
executable= jess/Main.class
arguments = jess.Main
    -f to_do_in_Jess.clp
jar_files = jess.jar
output = jess.$(Process).output
error = jess.$(Process).error
log = jess.$(Process).log
queue 16
```

in the case of \( p=16 \). The Octopus can run for example on the submitter machine. Then in the `to_do_in_Jess` the DNS name of that machine must be specified as `server.domain`.

## 4 Efficiency test

Several tests are needed to measure the speedup of Distributed Jess when it uses multiple processors. Here we use the classical benchmark [10]. The particular test problem taken into account is the Miss Manners problem.

Miss Manners is the problem of finding an acceptable seating arrangement for guests at a dinner party, by attempting to match people with the same hobbies, and to seat everyone next to a member of the opposite sex. The classical solution employs a depth-first search approach to the problem. The variables of the problem are the number of guests and chairs, the maximum and minimum numbers of hobbies (e.g. 128 guests and 128 chairs, max hobbies and min 2 hobbies).

A Jess version of the solution can be found at [6]. The data are generated randomly, the number of guests of opposite sex being equal (for each guest: name, sex and list of hobbies). In one of the easiest case, e.g. maximum 3 hobbies and minimum 2 hobbies the depth-first search is building a solution relative fast. But the time to obtain the solution is increasing exponentially with the number of guests and chairs. For example, for a sample of initial data, on a PIV at 2.2 GHz with 512 Mb RAM, the problem for 64 guests is solved in 7 seconds, the one for 128 guests in 110 seconds, while the one for 256 guests in 1801 seconds. If the problem is more complicated, e.g. the minimum number of hobbies is 1, the depth-first search explores several branches of the search tree until it reach a solution. For example, for another sample of initial data generated with maximum 2 hobbies and minimum 1 hobbies the problem for 64 guests is solved in 2103 seconds, while the one for 128 guests in 7361 seconds.

The steps proposed here to split the computational effort into \( p \) tasks in order to obtain a solution for the above mentioned problem are the followings ones. We assume that \( 2p \) divides the number of guests. In the preprocessing phase of the initial data, the data set is split into \( p \) equal fragments and it is decides if the are at least \( p \) special guests having the same sex and the maximal number of hobbies. If the answer is yes, they are distributed each to a distinct data fragment (interchanges are possible). If no, take the ones with the closest number to the maximal number of
hobbies. Each task receives a data fragment and the start and end number of seats. The special guest selected in the first phase is seating on the first chair assigned to the task which treats the data fragment. The special guest is communicated to the task which treats the left neighbor data fragment. Same rules are applied for the depth-first search on each task, but on the different data fragment. The task search is complete only if at least one hobby of the guest seating on the last chair assigned to task is on the list of the special guest communicated by the task treating the right neighbor fragment of data. The pairs guest-chair provided by each task are collected.

The Jess source proposed in [6] was modified according these steps. A batch file was also written to launch the Jess instance. Each instance loads the modified Jess source and according its instance identifier treats a fragment of the data and send and/or receive the information about the special guests sitting on the chairs nearby the ones treated by the Jess instance.

The cluster environment used in the experiments consists in a heterogeneous pool of 16 PCs with Intel Celeron, PIII and PIV, at 0.6 to 3 GHz, and from 0.128 to 1 Gb RAM, connected in a local network with a peak speed for communications of 100 Mb/s (Figures 2 and 3).

<table>
<thead>
<tr>
<th>No. instances</th>
<th>Mean run time</th>
<th>No. fired rules per instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2301 seconds</td>
<td>33406 rules</td>
</tr>
<tr>
<td>2</td>
<td>206 seconds</td>
<td>8510 rules</td>
</tr>
<tr>
<td>4</td>
<td>32 seconds</td>
<td>2206 rules</td>
</tr>
<tr>
<td>8</td>
<td>10 seconds</td>
<td>590 rules</td>
</tr>
</tbody>
</table>

Table 2. Running time improvement (and speedup) by using several nodes of the Condor pool

<table>
<thead>
<tr>
<th>No. instances</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>206 s</td>
<td>113 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Time S2</td>
<td>1.82</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Time S4</td>
<td>1.68</td>
<td>2.91</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Time S8</td>
<td>1.42</td>
<td>2.51</td>
<td>3.33</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Time S16</td>
<td>1.22</td>
<td>1.83</td>
<td>2.2</td>
<td>3.67</td>
<td>-</td>
</tr>
</tbody>
</table>

described tasks, the speedup value is expected to be acceptable. Table 2 gives a proof for this fact.

4 Conclusions

Several approaches to construct parallel rule-based engines were discussed in this paper. Following one such path, we have initiated the development of Distributed Jess, enabling the cooperation between Jess instances running in a cluster environment. At this stage Distributed Jess exists as a demo system with the functionalities described above already implemented.
References: