Making Parallelism Accessible In OPSJ

Charles Forgy

Production Systems Technologies, Inc.
Draft of September 2009

1. Introduction

Over the years, there have been many attempts to build parallel engines for production systems. These developers of these systems have consistently discovered that, although it is not difficult to implement a parallel engine, the amount of speed-up that is achieved is limited. The problem—if it can be called that—is that the existing single-threaded engines can handle typical rule-based programs fairly efficiently. In rule-based programs as they have traditionally been written, only a small number of objects in the system's working memory are modified when a rule fires. Rule engines take advantage of this slow rate of change of working memory by saving information from one cycle to the next. As far as possible, the rule engines process only the changed objects, reusing much of the information they already have concerning the other objects.

This limit on available parallelism has caused many researchers to look at rule languages and ask whether the languages can be changed to permit more work to be performed on each cycle. (See for example, [ACH94] [AMA94] [HER93] [NEI91] [WU93].) This paper provides an overview of some research in this area being performed by Production Systems Technologies to extend the parallel version of OPSJ.

1.1. Traditional Rules and Rule Engines

To see why parallel engines do not provide much speed up with traditional rule-based systems, consider the standard recognize-act cycle that is used by forward-chaining systems:

```
     void recognize_act_cycle() {
     while ( !isStoppingCondition() ) {
          Instantiation inst =
               performMatchAndConflictResolution();
          if ( inst == null )
               setStoppingCondition(true);
          else
               fire(inst);
     }
}
```

The rule engine spends the bulk of its time in the **`performMatchAndConflictResolution()`** method. This method can be parallelized effectively, but with current technology, significant amounts of time have to be devoted to synchronization. Regardless of the form of parallelism that is used in this method, it is necessary to synchronize the various threads in order to ensure that the correct rule is selected for execution.

This means that on every cycle, there is a flurry of parallel activity during the match and conflict resolution steps, then all the parallel tasks synchronize, and one thread executes the actions of the selected rule. This is a very difficult pattern for synchronization primitives to handle efficiently. While the one thread is executing the RHS actions, there are basically two choices for handling the other threads: The threads can be left executing an active, non-halting synchronization primitive such as a spin lock, or they can be "parked" and managed by the operating system.

Both of these choices have disadvantages. Using a non-halting synchronization primitive allows the threads to start up immediately when the matcher tasks become available on the next cycle, but they waste processor cycles by doing the active waiting. Parking the threads allows the operating system to allocate the processor cycles to other tasks or programs, but restarting a parked thread is extremely slow. Current single-threaded engines can execute 10,000 to 100,000 non-trivial rules per second. This equates to 10 to 100 microseconds per cycle. Since current operating-system-based synchronization mechanisms typically require several microseconds per thread, synchronization overhead can overwhelm the gains from parallel execution.[1] What is needed, then, is a way to use fast synchronization primitives without wasting too much processor time.

1.2. How Rule Architectures Could be Changed

To increase the opportunities for parallelism in rule-based systems, there are at least three approaches that could be tried:

1. The architecture could be changed so that more changes are made to working memory on each recognize-act .
2. The architecture could be changed to allow multiple threads of execution in the rules (i.e., to allow multiple instances of the recognize-act cycle to execute in parallel) with all the rules operating on the same working memory.
3. The system could be changed to allow instances of the recognize-act cycle to operate in parallel, but with a separate working memory for each instancs.

The parallel OPSJ system makes use of approaches 1 and 3. Approach 2 is simple to implement in a rule engine, but it results in a system that is very difficult to write rules for.

The design that is presented below is very simple because it attempts to provide basic mechanisms, and not to impose specific solution architectures on the end user. That is, it attempts to provide a small set of building blocks that can be put together as the target application dictates.

2. Existing Language Features

The parallel language features are being added to the existing parallel, event-processing, version of OPSJ. Before describing the features for parallelism, it is necessary to describe two features of the existing language.

The first of these features is "consuming" conditions: A consuming condition is one that is marked with the keyword **`CONS`**. When a rule that contains one of these conditions fires, the working memory object

[1]It should be noted that this does not apply to all existing rule-based programs. In particular, event-processing systems that handle high event rates are good candidates for parallelization. However, the techniques described in this paper will provide higher levels of parallelism even for them.

that matches the condition cannot match the condition again unless the object is modified. This feature essentially provides a more flexible form of refraction. For a very basic example of its use, the following rule is a common example of how not to count objects using refraction:

```
     rule incorrectCount
if {
     cnt: Counter;
     obj: TargetObject( . . .specific features. . . );
} do {
     cnt.value += 1;
     update(cnt);        // ERROR: Causes an infinite loop
}
```

The problem with this, of course, is that every time the Counter is updated, all refraction information is lost. To do this correctly in OPSJ, the rule can be written using a consuming condition:

```
     rule correctCount
if {
     cnt: Counter;
     obj: [CONS] TargetObject( . . .specific features. . . );
} do {
     cnt.value += 1;
     update(cnt);
}
```

This feature is particularly useful in event processing because rules are not permitted to modify events.

The other important feature is garbage collection of working memory objects. The OPSJ engine keeps track of how many rules reference each object in working memory. This count can be affected in many ways, including the action of **CONS** and the expiration of an event from a sliding window. When the count goes to zero, the engine automatically removes the object from working memory.

These two features are important because they allow the system to handle bookkeeping tasks that would otherwise be extremely troublesome in a parallel program.

3. Increasing the Amount of Work on Each Cycle

The term “data parallelism” is used to describe programs that perform a single operation in parallel on multiple sets of data. Adding two vectors pairwise to create a third vector is a typical example. Two common extensions to rule languages provide a form of data parallelism in rule programming. These extensions might be called “immediate rules” and “set-oriented rules.” Both of these increase the amount of work performed on each recognize-act cycle and both are supported in the parallel OPSJ system.

Set-oriented rules are rules that contain one or more conditions that match a set of objects. Consider the following normal (non-set-oriented) rules:

```
     rule reverse_edges
```

```
if {
    f1: stage (f1.value=="duplicate");
    f2: line (var x=f2.p1, var y=f2.p2);
} do {
    insert(new edge(x, y));
    insert(new edge(y, x));
    delete(f2);
}

rule done_reversing
if {
    f1: stage (f1.value=="duplicate");
    !f2: line;
} do {
    delete(f1);
    insert(new stage("detect_junctions"));
}
```

These rules are taken from the Waltz benchmark program. All they do is to replace each **line** object with two directed **edge** objects. Multiple rule firings are needed because these rules can change only one **line** at a time. Using a set-oriented rule, these rules can be replaced by one rule, and, more significantly, the processing performed by these rules can be handled by a single firing of the set-oriented rule. The rule in parallel OPSJ would be written as follows:

```
     rule reverse_all_edges
if {
    f1: stage (f1.value=="duplicate");
    f2set: collect line();
} do {
    for ( line f2 : f2set ) {
          int x = f2.p1;
        int y = f2.p2;
              insert(new edge(x, y));
    insert(new edge(y, x));
    delete(f2);
         }
    delete(f1);
    insert(new stage("detect_junctions"));
     }
```

The keyword **collect** indicates that the condition is to be bound to the set of all objects that match the following pattern. In this rule, the set contains every **line** object in the system.

Immediate rules permit the engine to fire multiple rules on a single cycle. Specifically, when an immediate rule is selected for firing, the engine fires every current instance of the rule. To implement this functionality, **performMatchAndConflictResolution()** must be changed to return a list rather than a single object, and the recognize-act cycle is changed as follows (where changes are shown in blue):

```
     void recognize_act_cycle() {
```

```
     while ( !isStoppingCondition() ) {
          Instantiation[] inst =
                    performMatchAndConflictResolution();
               if ( isempty(inst) )
               setStoppingCondition(true);
          else
               fireallrules(inst);
     }
}
```

In OPSJ, an immediate rule is declared using the keyword **irule** instead of **rule**. Using an immediate rule, the two rules for processing line objects could be rewritten as:

```
     irule reverse_edges
if {
    f1: stage (f1.value=="duplicate");
    f2: line (var x=f2.p1, var y=f2.p2);
} do {
    insert(new edge(x, y));
    insert(new edge(y, x));
    delete(f2);
}

rule done_reversing
if {
    f1: stage (f1.value=="duplicate");
    !f2: line;
} do {
    delete(f1);
    insert(new stage("detect_junctions"));
     }
```

These rules will require two recognize-act cycles to replace all the lines with edges, regardless of how many lines there are.

3.1. Issues With Immediate Rules

Immediate rules must be used with care. Since all the instantiations are fired logically in parallel, the engine cannot perform the checks that are available in standard rule-at-a-time execution. For an example of the problems this can cause, consider the following rule from the same system:

```
     rule make3junction (10)
if {
    f1: stage (f1.value=="detect_junctions");
    f2: edge (var base_point=f2.p1, var pa=f2.p2,
              f2.joined==false);
    f3: edge (f3.p1==base_point, f3.p2!=pa, var pb=f3.p2,
              f3.joined==false);
    f4: edge (f4.p1==base_point, f4.p2!=pa, f4.p2!=pb,
```

```
                var pc=f4.p2, f4.joined==false);
} do {
    make_3_junction(base_point, pa, pb, pc);
    modify(f2); f2.joined=true;
    modify(f3); f3.joined=true;
    modify(f4); f4.joined=true;
}
```

This rule recognizes when three edges meet at a common point, and creates an object to describe the junction. Since each edge is marked as joined when it is used, it might seem that this rule can be replaced with an immediate rule:

```
     irule incorrect_make3junction (10)
if {
    f1: stage (f1.value=="detect_junctions");
    f2: edge (var base_point=f2.p1, var pa=f2.p2,
                f2.joined==false);
    f3: edge (f3.p1==base_point, f3.p2!=pa, var pb=f3.p2,
                f3.joined==false);
    f4: edge (f4.p1==base_point, f4.p2!=pa, f4.p2!=pb,
                var pc=f4.p2, f4.joined==false);
} do {
    make_3_junction(base_point, pa, pb, pc);
    modify(f2); f2.joined=true;
    modify(f3); f3.joined=true;
    modify(f4); f4.joined=true;
     }
```

Unfortunately, doing this introduces a bug into the system. The problem is that the three edge conditions do not impose any constraints on the edges except that they must be different. Since there are six ways that a set of three edges can be assigned to the variables f1, f2, and f3, this rule will have six instantiations for each set of edges. This did not matter in the original program because when one of the six fired, the joined flags for the edges were changed, and the other five instantiations were deleted before they could fire. With the immediate rule, however, all six instances of the rule will be fired, and six copies of the junction object will be created.

To solve this problem, the action part of the rule can be changed to repeat the tests of the joined flags:

```
     irule corrected_make3junction_2 (10)
if {
    f1: stage (f1.value=="detect_junctions");
    f2: edge (var base_point=f2.p1, var pa=f2.p2,
                f2.joined==false);
    f3: edge (f3.p1==base_point, f3.p2!=pa, var pb=f3.p2,
                f3.joined==false);
    f4: edge (f4.p1==base_point, f4.p2!=pa, f4.p2!=pb,
                var pc=f4.p2, f4.joined==false);
} do {
    if ( !f2.joined && !f3.joined && !f4.joined ) {
       make_3_junction(base_point, pa, pb, pc);
       modify(f2); f2.joined=true;
```

```
        modify(f3); f3.joined=true;
        modify(f4); f4.joined=true;
    }
}
```

As this illustrates, immediate rules must be used with care, though they are sometimes essential.

4. Multiple Communicating Knowledge Sources

A very natural way to support multiple recognize-act cycles is to implement multiple communicating rule sets, where each rule set has its own recognize-act cycle. In OPSJ programs, rules are grouped into components called knowledge sources, where each knowledge source consists of

1. Some number of rules.
2. A working memory and its contents.
3. The state necessary to implement the recognize-act cycle. This primarily consists of the conflict set plus the state held by the matcher.

Every knowledge source can have its own thread of execution, and the knowledge sources execute asynchronously. Standard OPSJ has always supported multiple knowledge sources in a single program. However in the past this ability has not been used much, because it has not been convenient to get multiple knowledge sources to work together. The new parallel system attempts to make communicating knowledge sources reasonable to use.

To permit these knowledge sources to communicate, only one method needed to be added to the knowledge source's API:

```
void sendPacket(KnowledgeSource dest, Packet wmes)
```

A Packet contains a list of objects (and certain other information that will be described in the next section). When the destination knowledge source receives the complete packet, the objects in the packet are inserted into the knowledge source's working memory.

To support Packets, the recognize-act cycle is extended as follows:

```
void recognize_act_cycle() {
while ( !isStoppingCondition() ) {
     insertCompletePackets();
     Instantiation[] inst =
          performMatchAndConflictResolution();
     if ( isempty(inst) )
          setStoppingCondition(true);
     else
          fireallrules(inst);
}
}
```

The method **`insertCompletePackets()`** waits until is has received a complete packet, and then inserts all the objects at once.

In addition, the overall control for the knowledge source is extended so that, when there are no rules ready to fire, the knowledge source can perform a Wait operation. This allows knowledge sources to

remain in the system, consuming no resources until something arrives for them to process.

It is not legal in OPSJ for two knowledge sources to contain the same mutable object in their working memories. For this reason, the objects in the packet are copied, unless the engine can be sure that the object will not be changed by another knowledge source. In particular, if the object is an event, it does not have to be copied because events are immutable.

4.1. Buses

Parallel OPSJ adds a feature called a bus to make it more convenient to connect knowledge sources. A bus is a very basic object that simply receives packets and immediately resends the packets to all the receiving knowledge sources that have registered with the bus. Buses can be many-to-many; that is, a bus can receive packets from multiple knowledge sources, and resend those packets to multiple destination knowledge sources. Buses obviously add no new abilities to the system, but they can significantly simplify the creation of systems with many knowledge sources.

5. Blackboard-Like Knowledge Sources

The architecture described in the last section provides a powerful basis for building networks of intelligent agents, but it has one significant drawback: it does not allow the agents to be reactive. Rather, it requires each knowledge source to know which other knowledge sources (or at least, which buses) are interested in its packets. To provide this missing reactivity, the parallel OPSJ system also supports two architectural features from blackboard systems:

1. A mechanism that permits one knowledge source to monitor the contents of another knowledge source's working memory
2. A structured working memory.

5.1. Monitoring Other Knowledge Sources

To support the monitoring of other knowledge sources, the system adds a new kind of rule called a probe. A probe is a standard rule except that the engine is modified to execute every satisfied probe at the beginning of every recognize-act cycle. The recognize-act cycle is:

```
void recognize_act_cycle() {
while ( !isStoppingCondition() ) {
     insertCompletePackets();
     InstGroup inst =
               performMatchAndConflictResolution();
          if ( isempty(inst) )
          setStoppingCondition(true);
     else {
          fireallprobes(inst);
          fireallrules(inst);
```

```
            }
      }
}
```

The method **performMatchAndConflictResolution()** now returns an object that lists all satisfied probes as well as the rule or rules selected for execution.

With this feature, one knowledge source can monitor another knowledge source simply by attaching the necessary probes to the monitored knowledge source. For instance, if knowledge source KS2 wanted to be told of every derived event of type A that was created by knowledge source KS1, it could attach the following rule to KS1:

```
      probe KS2_probe
if {
      ev: [CONS] DerivedEvent( ev.type == A );
} do {
      sendPacket(KS2, new Packet(ev));
}
```

Note the use of **CONS** in this rule. In general, all probes should use **CONS**; doing this ensures that the engine can garbage collect the event object as soon as every rule has processed it.

5.2. Structured Working Memories

In blackboard systems, there are frequently a few knowledge sources (sometimes only one knowledge source) maintaining the global state used by the entire set of knowledge sources. To allow structuring the data in such an architecture, blackboard systems impose a level structure on the blackboard, where each level holds only a certain kind of data. With this level structure in place, each knowledge source can monitor only those levels that contain the kinds of data it is interested in.

Parallel OPSJ also supports this. In parallel OPSJ, a working memory can be declared to have a certain number of levels, and each condition in each rule can specify which level it is sensitive to. In addition, the **insert()** operation is extended to allow a level to be specified. For backward compatibility, when no level is specified in a condition or in an **insert()** operation, level 0 is assumed. For an example of how this is used, suppose that events in a system were divided into two levels: **PRIMITIVE** and **DERIVED**. Then the rule from the last section would have been written as follows.

```
      probe KS2_probe
if {
      ev: [CONS] DerivedEvent( ev.type == A ) :: DERIVED;
} do {
      sendPacket(KS2, new Packet(ev));
      }
```

Because insert operations require a level, the Packet objects actually have to contain two pieces of information for each object: the object itself plus the level it is to inserted into. The rule above specifies no level, so level 0 is used by default. Thus this rule is equivalent to:

```
     probe KS2_probe
if {
     ev: [CONS] DerivedEvent( ev.type == A ) :: DERIVED;
} do {
     sendPacket(KS2, new Packet(0, ev));
     }
```

5.3. Blackboard Control Features

The design presented in this section deliberately omits any blackboard-specific control features. If a complete blackboard implementation is desired, the blackboard control can be implemented in a knowledge source (or multiple knowledge sources) using regular rules. For an overview of the possibilities in this area, the reader is directed to [COR03].

6. Extensions

The design for communicating knowledge sources will be used first on shared-memory multiprocessor machines. However, the extensions were all designed to be suitable for use in computer networks. All that is necessary is the implementation of another version of **`sendPacket()`** that uses interprocess communication. The send operations will take longer to complete of course, but nothing else will change.

7. Bibliography

ACH94: Acharya, Anurag, Scalability in Production System Programs, 1994
AMA94: Amaral, Jose Nelson, A Parallel Architecture for Serializable Production Systems, 1994
HER93: Hernandez, Mauricio A. and Stolfo, Salvatore J., Parallel Programming of Rule-based Systems in PARULEL, 1993
NEI91: Neiman, Daniel E., Control Issues in Parallel Rule-Firing Production Systems, 1991
WU93: Wu, Shiow-yang, and Browne, James C., Explicit Parallel Structuring for Rule-Based Programming, 1993
COR03: Corkill, Daniel D., Collaborating Software: Blackboard and Multi-Agent Systems & the Future, 2003