Synchronization in a Parallel Matcher

Charles Forgy

October, 2008

Outline

- Background
  - Why bother with parallelism
  - Rete and parallelism
- Options for synchronization
- PST's new engine
- Making better use of parallelism
- Impact of future hardware

Why Bother With Parallelism?

- “... we find ourselves as an industry turning from increased frequency to parallelism.” From the forward to the Intel Technology Journal, Vol 11, Issue 3.
- "We are in a parallel revolution, ready or not, and it is the end of the way we built microprocessors for the past 40 years." From Dave Patterson's keynote address to Usenix 2008.

Do Systems Need More Speed?

- Some application areas do today (e.g., Event Processing).
- If faster systems were available, new application areas would open up.

Rete and Parallelism

- Rete was designed from the beginning to be a parallel algorithm (though some optimizations work against that).
- A Rete matcher is organized as a network of agents that cooperate by passing messages.
- As long as a few sequencing rules are followed, the order in which the messages are processed is unimportant.
- In fact, there have been several parallel Rete engines built.

Outline

- Background
  - Why bother with parallelism
  - Rete and parallelism
- Options for synchronization
- PST's new engine
- Making better use of parallelism
- Impact of future hardware

Synchronization

Synchronization is arguably the number one issue in building a parallel matcher.

- To achieve a reasonable balance of work for the processors, the match must be broken into a fairly large number of tasks.
- Synchronization overhead can kill performance.
- Synchronization must be handled without impact on the rest of the system (beware of spin locks).

What is a Thread?

- A program counter.
- A stack and related information (in machine registers).
- Plus the process state that is shared with the other threads

Why Synchronization is Needed

- To provide safe access to shared, writable state.
  - Avoid it as much as possible.
    - Make shared objects immutable.
    - Keep writable state local to threads.
  - Where it must be used, Java provides a number of good mechanisms.
    - See java.util.concurrent.
    - In today's JVM's synchronized blocks are efficient.
- To allocate tasks to threads.
  - This is mainly what I want to talk about.

Kinds of Synchronization

- Context switch.
- Application-managed threads.
- Busy waits.

Context Switch

- Uses the operating system to suspend and restart threads.
- Treats a thread like a process, requiring
  - Saving and reloading the processor registers.
  - Switching to a different stack.
  - (Possibly) flushing the caches and TLB's.
- Very expensive.
  - “The actual cost of context switching varies across platforms, but a good rule of thumb is that a context switch costs the equivalent of 5,000 to 10,000 clock cycles, or several microseconds on most current processors.” -- Brian Goetz, *Java Concurrency in Practice*.

Busy Wait or Spin Lock

- The thread does not relinquish the processor to another thread, but rather goes into a loop to wait for the necessary condition to proceed.
- A spin lock provides the lowest overhead for the thread that uses it, but it uses a processor that might be used productively by another thread.

Application Managed Threads

- The application is given a pool of threads which it assigns to tasks at times of its choosing.
- If implemented correctly, this can have substantially lower overhead than context switches.

Outline

- Background
  - Why bother with parallelism
  - Rete and parallelism
- Options for synchronization
- PST's new engine
- Making better use of parallelism
- Impact of future hardware

The Requirements

- A Java rule engine.
- For shared-memory multiprocessors.
- Compatible with other Java code.

The Recognize-Act Cycle

```
void recognize_act() {
    while (true) {
        updateConflictSet();
        if ( stoppingCondition() )
            return;
        executeDominantInst();
    }
}
```

Recognize-Act Using a Parallel Matcher

```
void recognize_act() {
    while (true) {
        allowMatcherThreadsToRun();
        waitForMatcherThreads();
        if ( stoppingCondition() )
            return;
        executeDominantInst();
    }
}
```

A Matcher Thread

```
while (true) {
    waitForMainThread();
    performMatch();
    allowMainThreadToRun();
}
```

Wait and Allow Methods

- The “wait...” and “allow...” methods can be implemented in various ways.
- The most direct would be counting semaphores from the java.util.concurrent package.
- Ultimately, whichever primitive is used, the primitive will employ context switches.
- Using busy waits is not appropriate for an engine that is to be used as a component in a larger application.

Parallelism in the Match

- The match threads are managed by the fork-join framework in jsr166y.forkjoin.
- This framework is quite efficient and is suitable for tasks as small as 1000 instructions.

A Task in the Matcher

- Fork-join works best when it is given a few large tasks which are recursively decomposed into smaller tasks.
- In this matcher, a task is a set of tokens, and a set of nodes to pass the tokens to.
- A task can be decomposed into smaller tasks by breaking either set into subsets.

Results

- The new engine is significantly faster than OPSJ for complex event processing tasks.
- Synchronization overheads are too high to make it appropriate for simple business rule applications.
  - Recall that a context swap takes 5k to 10k instruction times, and two context swaps are required per recognize-act cycle.

Outline

- Background
  - Why bother with parallelism
  - Rete and parallelism
- Options for synchronization
- PST's new engine
- Making better use of parallelism
- Impact of future hardware

Why Don't Business Rules Need Parallelism?

- On most cycles, the fired rule changes only a few working memory objects.
- Each change typically affects only a few rules.

How We Could Change This

- Switch from tuple-oriented to set-oriented conditions.
  - Set-oriented conditions are already supported by many of today's languages.
- Allow multiple rules to fire on each cycle.

Example of Tuple-Oriented Conditions

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

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

Example of Set-Oriented Conditions

```
rule reverse_all_edges
if {
    s: stage(s.value=="duplicate");
    collect l: line;
} do {
    foreach( line lin : lList ) {
        insert(new edge(lin.p1, lin.p2));
        insert(new edge(lin.p2, lin.p1));
        delete(lin);
    }
    delete(s);
    insert(new stage("detect_junctions"));
}
```

Firing Multiple Rules

- There have been many proposals for extending rule engines to fire multiple rules per cycle.
- I believe this will be successful only if the parallel engines are at least as easy to use as current sequential rule engines.

Outline

- Background
  - Why bother with parallelism
  - Rete and parallelism
- Options for synchronization
- PST's new engine
- Making better use of parallelism
- Impact of future hardware

Busy Waits

- If we have many cores, there will be less pressure to keep them productive all the time.
- X86 processors have a pair of instructions, MONITOR and MWAIT, that allow very efficient busy waits.
  - The processor can stop executing instructions and enter a special wait state.
  - This feature is particularly important when hyperthreading is used.

Fork-Join Parallelism

- Intel has a research project to handle fork-join parallelism in hardware.
- Kumar, Hughes, Nguyen, “Architectural Support for Fine-Grained Parallelism on Multi-core Architectures,” *Intel Technology Journal*, vol 11, issue 3, August 2007.

Conclusions

- Parallelism in rule engines is useful today for complex problems.
- Parallelism will become more widely applicable in the near future.
  - Systems can incorporate more powerful rules without becoming harder to write.
  - Changes in hardware will substantially reduce the cost of synchronizing threads.

Thank You