June 1987

Report No. STAN-CS-87-1166
Also numbered KSL 87-43

Parallel Execution of OPS5 in QLISP

by

Hiroshi G. Okuna and **Anoop Gupta**

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory
Report No. KSL 87-43

June 1987

Parallel Execution of OPS5 in QLISP

by

Hiroshi G. Okuno* and Anoop Gupta**

*
KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

and

Electrical Communications Laboratories
Nippon Telegraph and Telephone Corporation
3-9-11 Midori-cho, Musashino
Tokyo 180 Japan

**
CENTER FOR INTEGRATED SYSTEMS
Computer Science Department
Stanford University
Stanford, California 94305

submitted to 21st Annual Hawaii International Conference on System Sciences (HICSS-21).

Abstract

Production systems (or rule-based systems) are widely used for the development of expert systems. To speed-up the execution of production systems, a number of different approaches are being taken, a majority of them being based on the use of parallelism. In -this paper, we explore the issues involved in the parallel implementation of OPS5 (a widely used production-system language) in QLISP (a parallel dialect of Lisp proposed by John McCarthy and Richard Gabriel). This paper shows that QLISP can easily encode most sources of parallelism in OPS5 that have been previously discussed in literature. This is significant because the OPS5 interpreter is the first large program to be encoded in QLISP, and as a result, this is the first practical demonstration of the expressive power of QLISP. The paper also lists the most commonly used QLISP constructs in the parallel implementation (and the contexts in which they are used), which serve as a hint to the QLISP implementor about what to optimize. We also discuss the exploitation of speculative parallelism in RHS-evaluation for OPS5. This has not been previously discussed in the literature.

Table of Contents

Parallel Execution of OPS5 in QLISP

Abstract

Production systems (or rule-based systems) are widely used for the development of expert systems. To speed-up the execution of production systems, a number of different approaches are being taken, a majority of them being based on the use of parallelism In this paper, we explore the issues involved in the parallel implementation of OPS5 (a widely used production-system language) in QLISP (a parallel dialect of Lisp proposed by John McCarthy and Richard Gabriel). This paper shows that QLJSP can easily encode most sources of parallelism in OPS5 that have been previously discussed in literature. This is significant because the OPS5 interpreter is the first large program to be encoded in QLISP, and as a result, this is the first practical demonstration of the expressive power of QLISP. The paper also lists the most commonly used QLISP constructs in the parallel implementation (and the contexts in which they are used), which serve as a hint to the QLJSP implementor about what to optimize. We also discuss the exploitation of speculative parallelism in RHS-evaluation for OPS5. This has not been previously discussed in the literature.

1. Introduction

There are several different programming paradigms that are currently popular in Artificial Intelligence, examples being production systems (or rule-based systems), frame-based systems, semantic-network systems, logic-based systems, blackboard systems. Of the above, production systems have been widely used to build large expert systems [10, 14]. Unfortunately, production systems run quite slowly, and this has especially been a problem for applications in the real-time domain. Production systems must be speeded-up significantly if they are to be used in new increasingly complex and time-critical domains. In this paper, we focus our attention on a specific production-system language, OPS5, that has been widely used to build expert systems and whose performance characteristics have been extensively studied. We also focus on parallelism as a means to speed-up the execution of OPS5.

The parallel execution of the OPS5 production-system language has been studied by several groups [4, 8, 11, 13]. Their general approach consisted of two steps: (i) the design of a dedicated parallel machine suitable for execution of OPS5; and (ii) the mapping of the OPS5 compiler and run-time environment on to the parallel hardware. In these implementations, the second step (the mapping step) involves parallel encoding of OPS5 using hardware specific and operating-system specific structures. In this paper, we explore how this mapping step may be done in a high-level parallel dialect of Lisp, called QLISP. The main advantages of encoding using a high-level programming language are: (i) Increase in portability, since the code does not depend on machine specific features; (ii) Greater flexibility and expressive power of the high-level language results in faster turn-around time, fewer errors, and more readable and modifiable code. The main disadvantage, of course, is that the encoding may not be as effiiient as h&ad-coded hardware-specific encodings. We normally do not wary about such issues for uniprocessors -- language compilers for uniprocessors are good enough -- but the disadvantage is significant for parallel implementations where the technology is not as far advanced. There is one more strong motivation for doing a parallel implementation of OPS5 while remaining within Lisp (unlike most previous parallel implementations). This is that OPS5 is often used as an embedded system within larger AI systems, and the fact that the rest of these systems are encoded in Lisp, If OPS5 is also encoded in Lisp, then it makes the task of interfacing much simpler.

There are several parallel Lisp languages, for example, Multilisp [5, 6, 7] and QLISP [3], that are available for speeding up Lisp programs by using multiple processors. Since QLISP is based on the Common Lisp [12], it provides very powerful facilities to the user. Multilisp is based on a functional programming subset of Lisp.

Another distinguishing features of QLISP is that control mechanisms to access shared data or global data are embedded in Lisp primitives. Other parallel Lisp languages use some data structures for locking, such as semaphores. QLISP enables the user to write parallel programs without paying much attention to the consistency of shared or global data One of the main purposes of this research is to explore the expressive power of QLISP by implementing a large program in it. Ours is the first large ("real") program implemented in QLISP, so this constitutes the first practical demonstration of the expressive power of QLJSP. We also list the most commonly used QLISP constructs and the contexts in which they are used, which can serve as a guide for optimizing the implementation of the QLISP language. A language where it is easy to express parallel constructs, but which does not offer better performance isnotofmuchuse.

The approach we take for parallelizing OPS5 is based on that of the Production System Machine (PSM) project at Carnegie-Mellon University [4]. The PSM project studied how the speed-up from parallelism increases as one goes from coarse-granularity (rule-level) to fine-granularity (intra-node) parallelism. We impkment each of their schemes and show that it is relatively easy to encode these parallel schemes within QLISP. We also show some interesting ways in which to exploit conflict-resolution parallelism and speculative parallelism[1] in RHS evaluation using QLISP.

'Ibis paper is organized as follows. Section 2 presents some background information about the OPS5 language, the Rete algorithm used to implement OPS5, and about QLISP. Section 3 describes how we do a parallel implementation of OPS5 using QLISP and the various issues involved. Finally, Section 4 is devoted to a discussion and conclusions.

2. Background

2.1. The OPS5 Production-System Language

An OPS5 [1] production system is composed of a set of *if-then* rules called *productions* that make up the *production memory,* and a database of assertions called the *working* memory. The assertions in the working memory *are called working memory elements.* Each production consists of a conjunction of *condition elements* corresponding to the if part of the rule (also called the *left-hand side* of the production), and a set of *actions corresponding to the then part* of the rule (also called the right-hand *side* of the production). The left-hand side and the right-hand side are separated by the "-->" symbol. The dons associated with a production can add, remove or modify working memory elements, or perform input-output. Figure 2-1 shows two simple productions named *pl* (with three condition elements)and *p2* (with two condition elements).

```
(p pl (Cl "color <x> ^size 12)        (p p2 (C2 ^price 38 ^color <y>)
      (C2 "price 38  ^color <x>)            (C4 ^color <y>)
      (C3 ^color <x>)                  -->
  -->                                  (modify 1 ^price 50) )
      (remove 2) )
```

Figure 2-1: Example of productions

The production system *interpreter* is the underlying mechanism that determines the set of satisfied productions

[1] The parallel computations of a program can be divided into two categories; *mandatory computations* and *speculative computations* [7]. The former means that all computations executed in parallel are necessary, while the latter means that some computations executed in parallel may not be necessary.

and controls the execution of the production system program. The interpreter executes a production system program by performing the following *recognize-act* cycle:

- Match: In this first phase, the left-hand sides of all productions are matched against the contents of *working memory*. As a result a *conflict set is obtained*, which consists of *instantiations* of all satisfied productions. An instantiation of a production is an ordered list of working memory elements that satisfies the left-hand side of the production.
- Conflict-Resolution: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.
- Act: In this third phase, the actions of the production selected in the conflict-resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, the first phase is executed again.

Each working memory element is a parenthesized list consisting of a constant symbol called the *class* of the element and zero or more *attribute-value* pairs. The attributes are symbols that are preceded by tbe operator ^. The values are symbolic or numeric constants. Each conditional element in the LHS consists of a class name and one or more terms. Each term consists of an attribute prefixed by ^, an operator, and a value. An operator is optional and its default value is =. other operators are <, <=, >, >=, <> and <=>. A value is either a constant or a variable. A variable is represented by an identifier enclosed by < and >. A variable can match any value, but all occurrences of the same variable in the LHS of a rule should match the same value. Conditional elements may not contain all pairs of attribute-value present in a working memory element. If a conditional element is preceded by -, it is called a negated condition element. The match for a rule succeeds only if there is no working memory element matching its negated condition element.

The RHS of a production can contain any number of actions. Actions can be classified into:

- **Working memory operations: These are make, remove, and modify.**
- **I/O operations: These are openfile, closefile, and write.**
- **Binding operations: These are** bind and cbind.
- **Miscellaneous operations: These are** default, call, halt, and **build.**

The above action types often take functions as arguments. Some such functions are // (quote), substr, genatom, compute, litval, accept and acceptline.

2.2. The Rete Match Algorithm

Empirical study of various OPS5 programs shows two interesting characteristics; *temporal redundancy* and *structural similarity* [2]. Temporal redundancy refers to the fact that a rule-firing makes only a few modifications to the working memory and most working-memory elements remain unchanged. Structural similarity refers to the fact that all productions are not totally distinct, and that there are many similarities between the condition elements of different productions. The Rete match algorithm exploits these two features to speed up the match phase of the interpreter.

The Rete algorithm uses a special kind of data-flow network compiled from the left-hand sides of productions to perform match. The network is generated at compile time, before the production systems is actually run. Figure 2-2 shows such a network for the two productions shown in Figure 2-l. In this figure, lines have been drawn between nodes to indicate the paths along which information flows. Information flows from the top-node down along these paths. The nodes with a single predecessor (near the top of the figure) are the ones that are concerned with individual condition elements. The nodes with two predecessors arc the ones that check for consistency of variable bindings between condition elements. The terminal nodes are at the bottom of the figure. Note that when two left-hand sides require identical nodes, the algorithm shares part of the network rather than building duplicate nodes.

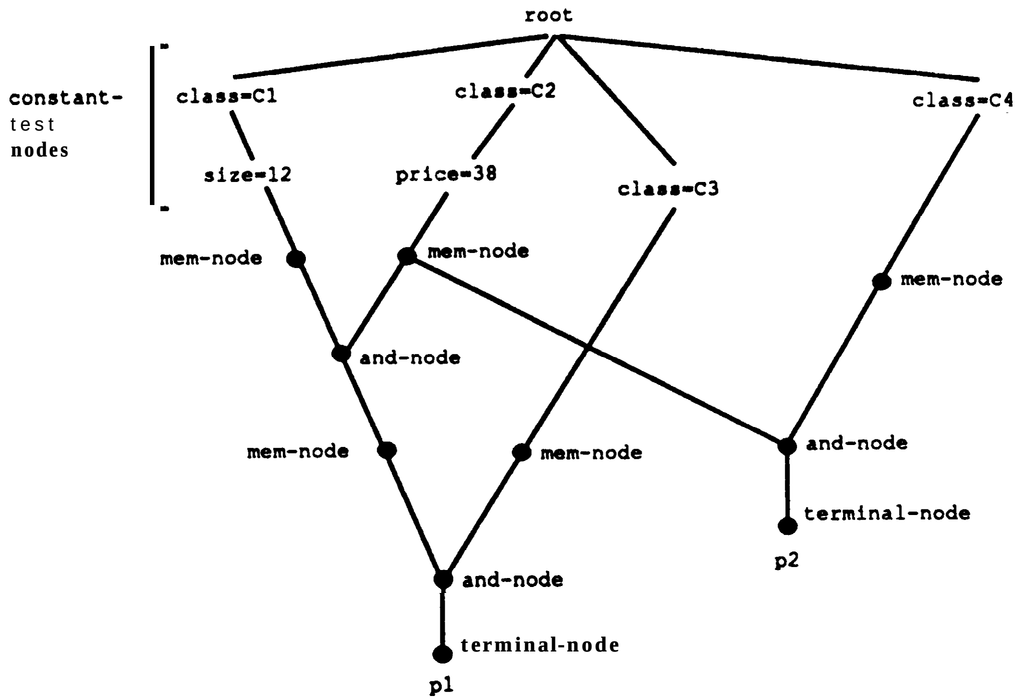

Figure 2-2: The Rete network

To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as state within the nodes. This way, only changes made to the working memory by the most recent production firing have to be processed every cycle. Thus, the input to the Rete network consists of the changes to the working memory. These changes filter through the network updating the state stored within the network. The output of the network consists of a specification of changes to the conflict set.

The objects that are passed between nodes are called tokens, which consist of a tag and an ordered fist of working-memory elements. The tag can be either a +, indicating that something has been added to the working memory, or a –, indicating that something has been removed from it. The list of working-memory elements associated with a token corresponds to a sequence of those elements that the system is trying to match or has already matched against a subsequence of condition elements in the left-hand side.

The data-flow network produced by the Rete algorithm consists of four different types of nodes. These are:

1. Constant-test nodes: These nodes are used to test if the attributes in the condition element which have a constant value are satisfied. These nodes always appear in the top part of the network. They have only one input, and as a result, they are sometimes called *one-input* nodes.

2. **Memory nodes:** These nodes store the results of the match phase from previous cycles as state within them The state stored in a memory node consists of a list of the tokens that match a part of the left-hand side of the associated production. For example, the right-most memory node in Figure 2-2 stores all tokens matching the second condition-element of production p2.

   At a more detailed level, there are two types of memory nodes -- the *α-mem* nodes and the *β-mem* nodes. The a-mem nodes store tokens that match individual condition elements. Thus all memory nodes immediately below constant-test nodes are α-mem nodes. The β-mem nodes store tokens that match a sequence of condition elements in the left-hand side of a production. Thus all memory nodes immediately below two-input nodes are β-mem nodes.

3. **Two-input nodes:** These nodes test for joint satisfaction of condition elements in the left-hand side of a production. Both inputs of a two-input node come from memory nodes. When a token arrives on the left input of a two-input node, it is compared to each token stored in the memory node connected to the right input. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action is taken when a token arrives on the right input of a two-input node.

   There are also two types of two-input nodes -- the *and-nodes* and the *not-nod&*. While the and-nodes are responsible for the positive condition elements and behave in the way described above, the not-nodes are responsible for the negated condition elements and behave in an opposite manner. The not-nodes generate a successor token only if there are no matching tokens in the memory node corresponding to the negated condition element.

4. **Terminal nodes:** There is one such node associated with each production in the program, as can be seen at bottom of Figure 2-2. Whenever a token flows into a terminal node, the corresponding production is either inserted into or deleted from the conflict set.

23. QLISP - Parallel Lisp Language

QLISP is a queue-based parallel Lisp proposed by Dick Gabriel and John McCarthy [3] and is being implemented on an Alliant FX/8 shared-memory multiprocessor by Stanford University and Lucid Inc. QLISP is similar to Multilisp [5, 6, 7], but language constructs incorporate important mechanisms for parallel computation such as spawning and locking. The spawned processes are put in the system queue and given to a processor by the scheduler to evaluate it The key ideas in QLISP were derived by reexamining Common Lisp [12] from the perspective of parallel processing, and by striving to make the minimal number of extensions to Common Lisp. Some QLISP primitives are summarized in the following subsections.

. 23.1. QLET

The qlet form executes its local binding in parallel.

(qlet *predicate* ({ (*var value*) }*) {*form*}*)

The qlet form is a construct to evaluate all *values* in parallel[2]. However, its computational semantic depends on the result of *predicate* which is evaluated first in the qlet form.

- *If the* result of *predicate* is nil, *the* qlet form acts exactly as the let farm.
- If the result of *predicate* is neither nil nor eager, a process far each *value* is spawned and the process evaluating a qlet form is suspended. When all the results of *value* are available, each result is bound to each *var* and the process evaluating a qlet form resumes its computation; that is, the body of a qlet form is evaluated.
- If the result of *predicate is eager,* a special value, *future*[3], is bound to each *var* and the body of a qlet form is evaluated immediately. *future* is associated with a process which evaluates a *value* eventually. In the execution of the body, if the value is not supplied yet, the process executing the body is suspended till the value is available.

Two kinds of parallel fibonacci functions are shown in Fig. 2-3.

The first one calculates a fibonacci number by spawning a process to calculate every fibonacci number of a smaller number. There may occur a combinatorial explosion of processes if n is a large number. For example, the number of spawned processes is 176.21890 and 242784 for n = 10.20 and 25, respectively. The second fibonacci function spawns a process only if the depth of the nesting is less than the value of *cutt-off*. The qlet predicate

[2] Since the pcall farm in MultiLisp evaluates arguments da function in parallel, it will be easily implement by qkt in QLISP

[3] The mechanism of eager is an implicit implementation of the future form in MultiLisp, or the *lary evaluation.*

```
(defun fib (n)
 (cond ((< n 2) 1)
      (t (qlet t ((fl (fib (- n 1)))
                 (f2 (fib (- n 2))) )
           (+ f1 f2) ))))


(defun fib-c (n)
 (labels ((fib-cutoff (n depth)
          (declare (special *cutoff-number+))
          (cond ((< n 2) 1)
              (t (qlet (< depth *cutoff-number*)
                   ((fl (fib-cutoff (- n 1)) (1+ depth))
                    (f2 (fib-cutoff (- n 2)) (1+ depth)) )
           (+ fl f2) )))))
   (fib-cutoff n 0) ))
```

Figure 2-3: Two parallel **Fibonacci functions - Exampk of qkt**

enables the user to **control the** spawning of **processes. Needless to** say, an **appropriate** value for ***cut-off*** should be **determined by the tradeoff between the cost and benifit of spawning.**

23.2. QLAMBDA

The **lambda** form in the Common Lisp creates **a closure which is used to share** variables among several functions or as an anonymous function. The **qlambda form creates** a *process* **closure.**

(qlambda *predicate lambda-list* **{***form***}** ***)**

A process closure **is used not only to share variabks among several process closures but also to control an** exclusive invocation of the **same** process closure. That is, only one application **of a process closure is evaluated** and other applications of the same process closure are suspended **The** evaluation **of** a **process closure depends on the value** of *predicate* which is evaluated at the time of evaluation **of the qlambda form, that** is, **creation** of **a process** closure.

- If the result of *predicate is nil, the* **qlambda form acts exactly as the lambda farm That is, a lexical closure is created**
- If the result of *predicate* is neither **nil nor** eager, **a process closure is created. When it is applied with** arguments, a separate process is spawned **for evaluation. If more than one applications occur, only one** applications are evaluated and others are **blocked. This is an implicit locking mechanism**
- If the result of *predicate is* **eager, a process closure is created and spawned immediately** without waiting for any **arguments.**[4]

A process closure may be used as an anonymous process, of which application is evaluated as a separated process. **The spawn** form is a shorthand form to do it; that is,

(spawn {*form***}*) is the same as ((qlambda t () {***form***}*) .**

In a sequential construct such as **block, all forms may be evaluated in parallel by spawn. A set of functions to** update of the conflict-set is shown in Fig. **2-4. The** global **variable** `*conflict-set-lock*` holds **a qlambda** closure to control the exclusive **access** to the variable `*conflict-set*` **which** holds the list of production

[4]**This curious mechanism can be used** to write **a parallel** Y **operator, that is,** ***for all* f, Y(f)=f(Y(f)), in QLISP. However, other useful applications are** not yet known.

instances. The idea to provide an exclusive access to *conflict-set* is to execute an update operation by using the same qlambda closure. The lock is released when register-cs returns a value immediately or when sort-conflict-set updates the *conflict-set+ or executes a sorting by spawning a subprocess by qlet with the predicate eager.

```
(proclaim (special *conflict-aet-lock* *conflict-set*))

(defun ops-init ()
  (setq *conflict-set-lock*
        (qlambda t (body) (apply (car body) (cdr body))) ))

(defun inaertca (name data rating)
  (funcall *conflict-set-lock*
           (list 'register-ca
                 name data (cons (sort-time-tag data) rating) t )))

(defun removeca (name data rating)
  (funcall *conflict-set-lock*
           (list 'register-ca
                 name data (cons (sort-time-tag data) rating) nil )))

(defun register-ca (name data key flag)
  (cond ((null *conflict-set*)
         (aetq *conflict-set*
               (carete-new-cs-element key nil name data flag) ))
        (t (sort-conflict-aet name data key flag *conflict-set*)) ))
```

Figure 2-4: Locking for Conflict-set

233. CATCH and THROW

A pair of **catch** and throw provides a way to do a **non-local** exit in the **Common Lisp.**

(catch *tag form*) **and** (throw *tag value*)

In **QLISP,** it **provides** not only a means of non-local exit but also a **mechanism** to control subprocesses spawned during the evaluation of form in the **catch form.** If the **catch gets** a value by the normal termination of *form or a* throwing, the catch kills all processed spawned during **the** execution **of the *form.*** *If the value contains* a **future, the** associated processes are not killed Note **that the** execution of a process spawned at **a** value-ignoring position of a sequential construct is **aborted.**

23.4. QCATCH

The qcatch form is similar to the **catch** form, but **the** control of spawned processes is **different.**

(qcatch *tag form*)

If the evaluation of **the *form* terminates normally** *and the* **qcatch gets** a **value, the qcatch waits for** all the processes spawned during the execution of the ***form* to** *terminate.* **Therefore, processed** spawned at a value-ignoring position will be evaluated before terminating the **qcatch** form. If the execution of the *form is* aborted by a **throwing, the qcatch kills all** spawned processes beneath **it.**

23.5. UNWIND-PROTECT

The unwind-protect form **is useful to do some** cleanup jobs **no matter** what **the unwind-protect** form is terminated

(unwind-protect *protected-form* { *cleanup-form* } *)

The unwind-protect form is **very important** in **QLISP world** in **order** to make the data consistent, because processes can be killed by the **catch** even if no throwing occurs.

23.6. Others

The suspend-process and **resume-process** forms are used for the **user to** control the scheduling of processes. **THe wait** and no-wait are used to **control** the **termination** of a process spawned at a value-ignoring position of sequential constructs.

3. Parallel execution of OPSS programs

As stated in Section 2.1, the **OPS5 interpreter** repeatedly **executes** a match -- conflict-resolution -- act cycle. In this section, we discuss how parallelism may be exploited in executing **each** of the three phases. Most of **the discussion focuses on the match phase, as the match phase takes 90% of the time in the interpreter.**

3.1. Parallelism in Match Phase

In this section, we explore how parallelism may be exploited to speed up the **match** phase. We present several different **algorithms.** We start with a **coarse-granularity** algorithm and slowly move towards finer **granularity.** In particular, we explore parallelism at three **levels** of granularity -- **rule-level parallelism,** no&-kvel paralklism, and **intra-node** parallelism **All** of the above algorithms are based on the Rete **algorithm** described in Section 22. What changes from one parallel algorithm to the other is the kinds **of node activations that are allowed to be processed in** parallel. The granularities we choose to discuss here **correspond** to **those** discussed in **[4].**

Before exploring the above schemes **further,** a word about the different kinds of node **activations** in the Rete network. Activations of constant-test nodes (shown in **top-part of network** in **Figure** 2-2) require just a simple test and are fairly cheap to execute. We call these ***ctest*** activations. It is usually not worth it to spawn a **process** to execute an individual **ctest** activation, because the overhead **of** spawning is **larger** than the work saved.

The second **kind** of node activations **are** the memory-no& **activations. These** require that **a token** be added or deleted from the memory node, and can be expensive because a delete request may **require searching** through all **the** tokens stored in that memory node. The third kind are the ***two-input node*** activation& that require searching through the opposite memory-node to find all matching tokens (tokens with consistent **variable** bindings). **These** are also fairly expensive. **We normally lump the processing required by the two-input node and the associated memory** nodes together into a single task/process, because the two are closely **interrelated** (the **two-input activation** examines **the memory node) and separating them incurs a large synchronization overhead. One also has to be careful about the** sequence in which the above node activations **are** executed. For example, the **Rete** algorithm **sometimes** generates conjugate tokens, where exactly the **same token** is **first** scheduled to be added to the memory node and **later** deleted- The **final** result should be that the state of the memory node remains unchanged. However, in parallel implementations it is easily possible that the **scheduler** decides **to** pick the delete request before the add request, and if **not** handled properly, the final state of the **memory** node may have **an** extra token. To process **conjugate** pairs **correctly,** each **memory** node has ***an extra-deletes-list to*** **store a** deleted token whose target token has not arrived yet.

Finally, there are terminal-no& activations that insert **or delete instantiations/tokens** into **the conflict-set.** Here also the problem of conjugate tokens can occur. **The** details for terminal-node activations **are** discussed later in Section 3.2.

For all the parallel implementation discussed in this paper, we use a **common** strategy for handling the **ctest** activations. **(We** present this strategy here, **before** discussing the **differing** strategies for the **remaining** types of activations.) **This** strategy is that multiple activations of the **root node** are **processed** using **separate processes** (i.e., **activations** corresponding to different **changes** to working memory are processed in parallel). However, all successors of the root node or the ctest nodes are evaluated using the following rule. If the successor node is also a ctest node then evaluate it sequentially within the same process, otherwise fork a separate process to do the evaluation. The code far such an evaluation policy is shown in Figure 3-l.

```
(defun match (token root-node)
  (qlet 'eager
        ((foo (doliat (node (successor root-node))
                     (cond ((c-teat? node) (c-teat token node))
                           (t (qlet 'eager
                                  ((foo (eval-node token node))) ))))))))

(defun c-teat (token node)
  (cond ((do-c-teat token node)
         (eval-node-list  token  (successor  node)) )))

(defun eval-node-list (token node-list)
  (cond ((null node-list)
        (t (let ( (node (pop  node-list) ) )
             (qlet (cond ((lock-node-p node) 'eager)
                         (t t) )
                   ((foo (eval-node token node))
                    (bar (eval-node-list token node-list)) )))))))

(defun eval-node (token node)
  (cond ((funcall (function node) token (arguments node))
         (eval-node-list token (successor node)) )))
```

Figure 3-1: QLISP code to evaluate Rete nodes in parallel.

3.1.1. Rule-level Parallelism

Rule-level parallelism is a very natural farm of parallelism in production systems. Here the march for each individual rule is. performed in parallel. In the context of our Rete-based implementation, this requires that we introduce *lock nodes* at points where a ctest node leads into a memory-node. All lock nodes before memory-nodes of the same rule use an identical lock, and those before memory-nodes of distinct rules use distinct locks. Figure 3-3 shows how the original Rete network of Figure 2-2 is modified to exploit rule-level parallelism. (Identical locks are shown grouped together in figure.) The locks are implemented using qlamhda closures, and the code for one such lock node is shown in Figure 3-2. As discussed earlier, a QLISP closure ensures that only one process can be actively executing inside the closure. The proposed locks then ensure that all activations corresponding to a single rule are executed in sequence, which is the desired semantics far rule-level parallelism.

```
(qlambda-closure  successor-node)    ;;; structure of lock node
(qlambda t (token node) (funcall (eval-node token node)))   ,,*= qlambda closure
```

Figure 3-2: Code for the lock node.

Fiially, we need to provide locks before the tokens enter the conflict-set, since the conflict-set is a global data structure and multiple processes should not be modifying it at the same time.

Using rule-level parallelism, previous studies [4] show that only about S-fold speed-up can be obtained. This is (i) because the number of rules that require significant processing is small and (ii) because even amongst these *affected* rules there is a large variation in the processing requirements. To reduce this variation in the processing times, we now discuss exploiting parallelism at a finer granularity where the processing for a single rule can be done in parallel.

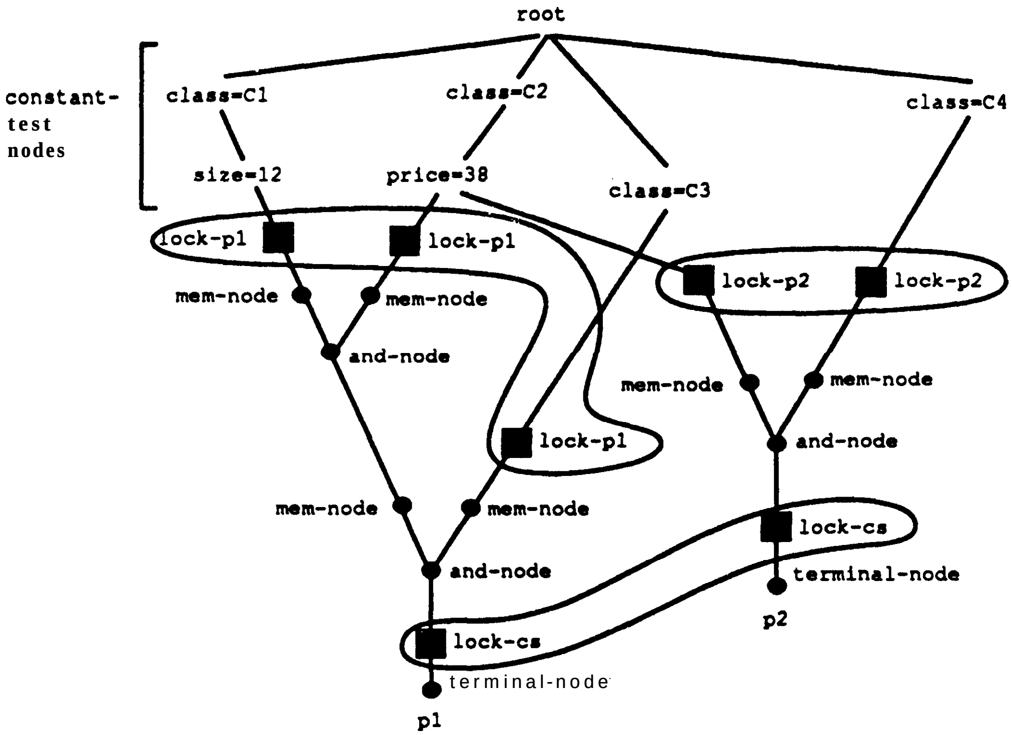

Figure 3-3: Modified Rete Network for Rule-level parallelism

3.12. Node-level Parallelism

When using node-level parallelism [4], any distinct two-input nodes can be evaluated in parallel[5]. To implement node-level parallelism, lock nodes are placed before each two-input no& and its associated memory nodes as shown in Figure 34. The structure of a lock node is the same for node-level and rule-level parallelism. However, the value of the qlet predicate are different for evaluating different types of node activations. The predicate is t for evaluating a memory-node and a tow-input node, but it is 'eager for evaluating successor nodes below a two-input node. That is, the execution of a two-input node is terminated by a future and the lock is released.

Note that if some two-input no& generates multiple tokens, the next two-input node becomes a bottleneck This is because only one activation of a given two-input node can be processed at the same time.

3.13. Intra-node Parallelism

The intra-node parallelism [4] exploits maximal parallelism present in the Rete algorithm If multiple tokens arrive at a two-input node, then these multiple activations of the two-input node are processed in parallel. However, we have to be very careful about how we access the memory nodes: (i) it is not desirable to have multiple processes modifying the same memory node; and (ii) the correct operation of the Rete algorithm requires that the opposite memory-node should not be modified while processing a two-input node activation. To ensure the correct operation, we adopt the solution proposed by Gupta in [4]. We use a common hash-table for all tokens stored in the memory nodes of the Rete network. Tokens are put into hash-table buckets based on the node-id of the associated

[5]According to the result of the simulations of PSM, the speedup of node-parallelism is about 5-fold.

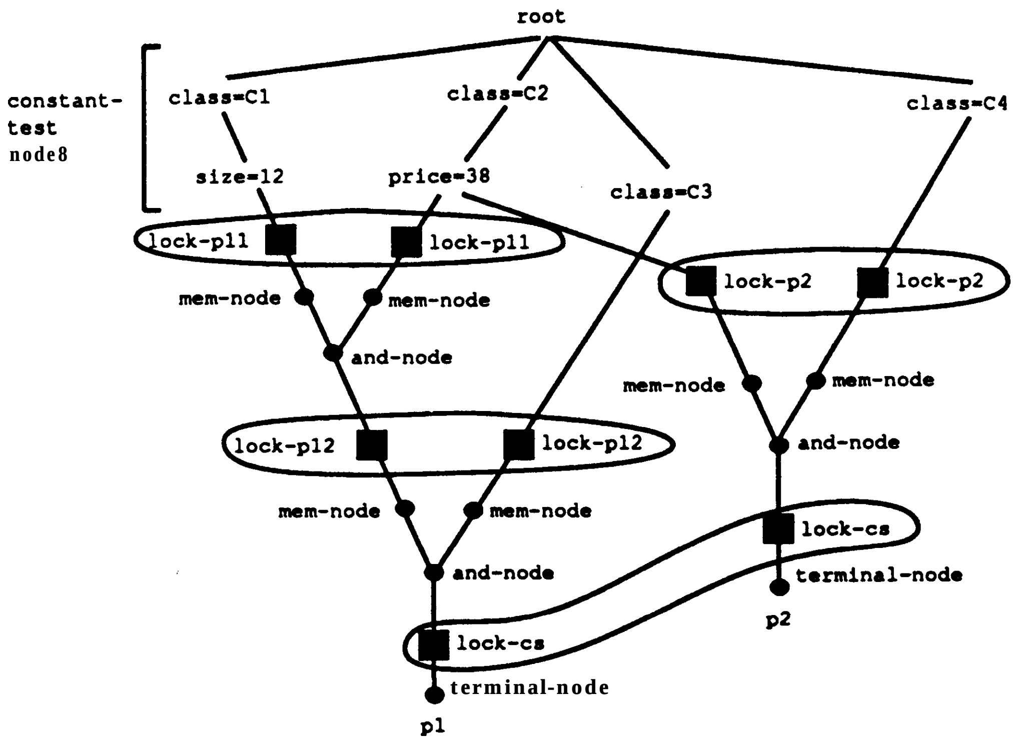

Figure 3-4: Modified Rete Network for Node-level parallelism

two-input node and some values that are tested from the token. The buckets in this hash-table are controlled by locks that are implemented as qlambda closures. Figure 3-5 shows the structure of this hash table. This scheme works because the probability that multiple t&ens would hash to the same bucket is considered small. If they do hash to the same bucket then they have to be processed sequentially.

In the above scheme, the Rete network reverts back to its original structure as shown in Figure 2-2 (except that locks are needed for executing the terminal nodes). All the remaining locks that were earlier associated with the Rete network are no longer present. Locking has now moved to hash-table buckets.

lock	left-hash-table		right-hash-table	
	token-list	extra-deletes-list	token-list	extra-deletes-list

Figure 3-5: Hash table for memory nodes

3.2. Conflict-Resolution Parallelism

During the conflict-resolution phase one of the several production instantiations in the conflict-set is selected for execution. The method by which this production instantiation is selected is called the tit-resolution strategy. OPS5 provides for two conflict-resolution strategies -- LEX (lexical) and MEA (means-ends-analysis). The two differ in the way a key is constructed for sorting various instantiations. The key for LEX consists of the sorted time-tag values of the working-memory elements in the instantiation. The key for MEA consists of the time-tag of the first working-memory element in the instantiation, followed by the sorted time-tag values of the remaining working-memory elements in the instantiation.

To perform conflict resolution, normally, the conflict-set is maintained as a sorted list of production instantiations. Executing conflict-resolution in parallel imposes the following requirements:

- We must allow multiple instantiations to be inserted into a deleted from the conflict-set in parallel.
- We must allow for conjugate pairs of instantiation& that is, where the delete request fa an instantiation is received before the add request.
- We would like to have the highest priority instantitation available to the RHS evaluation process as soon as possible, although the rest of the conflict-set data structure is not completely sorted.

To handle the first requirement, we build an asynchronous systolic priority queue structure in software [9] using QLISP. In this structure, inserts and deletes are input at the head of the priority queue. These then asynchronously filter down until they find the right position in the sorted queue. A delete may annihilate an already present element if it is already present. If a delete does not find a corresponding element already there (conjugate token problem), it locates itself at the right location in the queue with a special flag, and waits fa the corresponding add request to come by later. An insert behaves similarly. The key point is that the highest priority instantiation is always . available at the head of the queue, even if elements are still percolating down in the lower priority regions of the queue. The data structure that we use for a single instantiation in the priority queue is shown in Figure 3-6 and some related code is shown in Figure 24.

```
conflict-set-element =
 (key next-element positive-instance-list negative-instance-list)

  where next-element = (qlambda-closure . conflict-set-element)
     key = (sorted-time-tagof-Instance-clement . rating-of-production)
     positive-instance-list = (positive-instance . ..)
     extra-deletes-list = (extra-deletes-instance . ..)
     positive-instance = ((flag . simplified-form) production . instanceclement-list)
     extra-deletes-instance = (production. instance-element-list)
```

Figure 3-6: Representation of a production instance

The time to calculate the maximum element in the above scheme is O(k), where k is the number of changes to the conflict-set per recognize-act cycle. Since k is around 5 for most systems this is not a problem The time to finish sorting, however, can be much larger. This time is $O[N \times k]$, where N is the total number of elements in the conflict-set, which can be much larger. This is not optimal fa sorting, but it is good for getting the highest priority element. The highest priority element is used in the speculative execution of the RHS.

33. Speculative Execution of RHS

In the normal execution of a rule-based system, one would wait until conflict-resolution finishes completely before starting to execute the RHS of the highest priority rule. However, in a parallel implementation, this may imply too sequential a behavia. Even if RHS execution takes only 10% of the time, this limits the maximum speed-up to 10-fold. As a solution, we propose the speculative evaluation of RHS in this paper. By speculative evaluation of RHS we mean the following. While the match and conflict-resolution are still going on, we make a guessaboutthehighestprioritynlle. (This in our case is simply the rule currently at the head of the conflict-set.) We start evaluating the RHS of this rule, i.e., gathering up the changes it would make to working memory in a list (without actually changing the waking memory). If our guess is proved wrung, that is whenever there is a change in the rule at the head of the conflict-set, we simply create a new process to evaluate the RHS of this new rule. We currently do not abort the previously evaluating RHS because aborting is not easy to implement in QLISP. Furthermore, it is possible that the evaluated RHS of the non-highest rule may come in useful on a later cycle.

The OPS5/QLISP system provides a new action command sfcall, side-effect-free call which execute a user-defined routines written in QLISP or in Lisp. These user-defined routines should not refer any global data which may be modified by other routines, because the system assumes that simplification should be valid at any time and independent from any global context. The algorithm of simplification is sketched below:

1. Check the type of operations.
2. If a working memory operation, calculate all arguments and make a token.
   - If make, make a token of add and replace the original action with it.
   - If remove, make a token of delete and replace the original action with it.
   - If modify, make a token of delete and a token of add and replace the original action with them.

   However, if an action contains a fun&n such as accept, acceptline, these functions are not executed. Only omitted attribute-value pairs are supplied and the original action is replaced with a new action which has all attribute-value pairs.
3. If a side-effect-free call sfcall, do it
4. Otherwise, process next action.

This simplification is quite similar to the argument evaluation for a Lisp function with keyword arguments of the Common Lisp. The simplification routine is invoked when the maximum production instance of conflict-set is changed and stores a simplified form to the simplified form slot of the instance. Note that this simplified form is valid for any time, because it is calculated with using only local values which is specified in an instance. Conjugate pairs may create unnecessary processes, but the current implementation does not abort them, because such an aborting mechanism is not easy to implement and the number of conjugate pairs are not expected to be large.

4. Discussion

In this paper, we present the details of an implementation of the OPSS production-system language using QLISP, a parallel dialect of Lisp. We would like to make the following observations:

- The number of modifications needed to the original lisp code for OPS5 were minimal to exploit the different kinds of parallelism. Fa example, to exploit the three kinds of parallelism described for match, less than 100 lines of code (out of a total of about 3000 lines in the original code) had to be modified a added. We believe that such a high-level programming approach provides very powerful and flexible tools for research in parallel programming.
- The QLISP constructs that we used most frequently in our parallel implementation are "(qlet 'eager . ..)" to spawn new processes and "(qlambda t . ..)" process closures fa locks. The code sections that are locked and the processes that are spawned consist of a few lines of lisp code with some but not much recursion or iteration. On average, we expect the individual tasks to take about 1 millisecond of

computation time on a 1 MIPS machine. This requires that the process creation overhead. the locking overhead, and the scheduling overhead for the spawned tasks be significantly less than 1 millisecond, if the suggested implementations are to be useful If the overheads are much larger, then all the advantages of parallel execution will be subsumed by the overhead.

- We are currently using a QLISP simulator to obtain some performance numbers. Our implementation is running, and we have just started getting some performance numbers. Unfortunately, the simulator does not model the underlying hardware very accurately, so we still do not have a good idea about the true overheads involved. However, *fa reasons* mentioned in the next point, this may not be a big problem in practice.
- The parallel constructs provided by QLISP (qlet, qlambda, . ..) take a predicate that controls whether a parallel process is actually spawned *a* not. This convenient run-time method of controlling the granularity at which parallelism is exploited is a very powerful mechanism. It makes it extremely easy to modify code to adjust to different implementations with differing overheads. It is also convenient to adjust the granularity depending on the load present on the parallel machine.
- As stated in the beginning of this paper, another advantage of implementing OPS5 in QLISP, instead of in Pascal or C, is that it is easy to embed the OPS5 system within other AI systems (which normally use Lisp). Furthermore, if there are complex functions in the RHS of rules, then these functions can also use the parallel constructs available in QLISP, which is not possible in previously proposed parallel implementations of OPSS.
- As a final means for improving performance fa existing OPSS systems we are planning to directly compile OPSS into QLISP code, instead of using an interpreter as we currently do.

Acknowledgments

The authors would like to thank Prof. Edward Feigenbaum fa supporting this work at Knowledge Systems Laboratory at Stanford University. We would also like to thank members of the QLISP group at Stanford We would especially like to thank Joe Weening fa help with the QLISP simulator.

The computing facilities used in doing this work and writing this paper are supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Anoop Gupta is also supported by a faculty grant from Digital Equipment Corporation.

References

1. Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

2. Forgy, C. L. On the Efficient Implementations of Production Systems. PhD thesis, Tech&al Report CMU-CS-79, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, February, 1979.

3. Gabriel, R.P. and McCarthy, J. Queue-based multiprocessor Lisp. Conference Record of the 1984 ACM Symposium on Lisp and Functional programming, *ACM,* Austin, Texas, August, 1984.

4. Gupta, A. Parallelism in Production Systems. PhD thesis, Technical Report CMU-CS-86-122, Department of Compukr Science, Carnegie-Mellon University, Pittsburgh, March, 1986

5. Halstead, R MultiLisp. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, August, 1984.

6. Halstead, R. "Multilisp: A Language fa Concurrent Symbolic Computation". ACM *Transaction* on *Programming Languages and Systems 7,* 4 (October 1985).

7. Halstead, R "Parallel Symbolic Computing". *IEEE Computer 19,* 8 (August 1986), 35-43.

8. Hillyer, B. K. and Shaw D. E. "Execution of OPS5 production Systems on a Massively Parallel Machine". *Journal of Parallel and Distributed Computing 3,* (1986), 236-268.

9. Leiserson, C. E. Systoric Priority Queues. Conference on Very Large Scale IntegrationL Architecture, Design, Fabrication, January, 1979. Also Available as a CMU Computer Science Department technical report CMU-CS-79-115, April, 1979.

10. John McDermott. "R1: A Rule-Based Configurer of Computer Systems". *Artificial Intelligence 19,* 1 (1982), 39-88.

11. Raja Ramnarayan, Gerhard Zimmerman, and Stanley Krolikoski. PESA-1: A Parallel Architecture for OPS5 Production Systems. Hawaii International Conference on System Sciences, January, 1986.

12. Steele, G.L.. *COMMON LISP : The Language.* Digital Press, Burlington Massachusetts, 1984.

13. Stolfo S. J. and Miranker D. P. "The DADO Production System Machine". *Journal of Parallel and Distributed Computing 3,* (1986), 269-296.

14. Gregg T. Vesonder, Salvatore J. Stolfo, John E. Zielinski, Frederick D. Miller, and David H. Copp. ACE: An Expert System for Telephone Cable Maintenance. International Joint Conference on Artificial Intelligence, 1983.