CLIPS meets the Connection Machine or How to create a Parallel Production System

Steve Geyer
MRJ, Inc.
10455 White Granite Drive
Oakton, Virginia 22124

Abstract

Production systems usually present unacceptable runtimes when faced with applications requiring tens of thousands to millions of facts. Many efforts have focused on the use of parallelism as a way to increase overall system performance. While these efforts have increased pattern matching and rule evaluation rates, they have only indirectly dealt with the problems faced by fact burdened applications. We have implemented PPS, a version of CLIPS running on the Connection Machine, to directly address the problems faced by these applications. This paper will describe our system, discuss its implementation, and present results.

1 Introduction

As production systems have been used to implement a wider and wider range of applications, the limits of current technology have been stretched. One particularly sensitive limit has been the problem size and how this size impacts the total runtime of a system. Most systems degrade rapidly once their size limits are reached. Indeed, the acceptable runtime is often an important, if not the most important, factor in setting an upper limit on problem size. Many applications have had to wait for technology to mature enough to support the application's minimum acceptable problem size.

Several factors influence the size of a problem. Two common factors are the number of facts manipulated by the application and the number of rule evaluations required to come to a solution. Studies demonstrate that many production system spend 90% of their total time matching facts to rule patterns. In an attempt to create more efficient systems on serial computers, algorithms have been developed to optimize this task. Rete is the most commonly used algorithm [1]. This algorithm can efficiently manage large numbers of simultaneous pattern queries, and as queries are completed, Rete updates the list of rules ready for execution. Rete caches internal data structures to remember partially matched queries and the number of cached entries increases rapidly as facts enter working memory. The memory required by these data structures and the computation necessary to manage them sets a practical limit on how many facts can be placed in working memory.

Many production systems built on parallel hardware have also focused on efficiently matching facts to patterns. Parallel pattern matching does support large rule sets and increases the rate at which facts can be processed. However, if the application is fact driven, the resources consumed in parallel pattern matching can overwhelm the increased resources brought by the parallel architecture. This is especially true if the parallel pattern matching algorithm caches partially matched queries. Special procedures are necessary when designing production systems that will process applications with large numbers of facts.

We are interested in problems requiring tens of thousands to millions of facts. Some examples are simulation and modeling, package/vehicle scheduling, intelligent databases, and low-to-mid level processes for image understanding. In each of these applications areas, many real world problems demand more facts than can be processed by current production systems. To get these systems away from the laboratory and running real world problems will require new techniques. We have developed PPS to explore one possible technique.

This paper is organized as follows: Section 2 presents necessary background material and describes the algorithmic approach taken by PPS. The changes made to CLIPS to create PPS are discussed in Section 3. This section can be skipped by those uninterested in implementation details. Experimental results are presented in Section 4 followed by a discussion of potential enhancements in Section 5. The paper finishes with a summary and conclusions in Section 6.

2 How PPS works

This section describes PPS. It begins with a description of the Connection Machine and explores the features that makes the CM well suited to this problem domain. Next it discusses the choice of CLIPS as a software base and describes the syntax changes necessary to allow CLIPS programs to run on PPS. Finally, the section will discuss the internal changes necessary to CLIPS to allow parallel execution on the Connection Machine.

2.1 The Connection Machine

The Connection Machine, or CM, is a parallel computer architecture that supports between 4 and 64 thousand separate processors. Figure 1 is a pictorial diagram of a CM. Each individual processor has a local memory, an ALU (Arithmetic Logic Unit), and a general inter-processor communication system. All processors share the same instruction stream supplied to them from a front end computer. Individual processors can perform separate operations by executing or ignoring, selectively, the sequences of instructions supplied by the front end. More complete technical information can be found in reference [2]. The CM has several properties that separate it from the other parallel architectures commercially available.

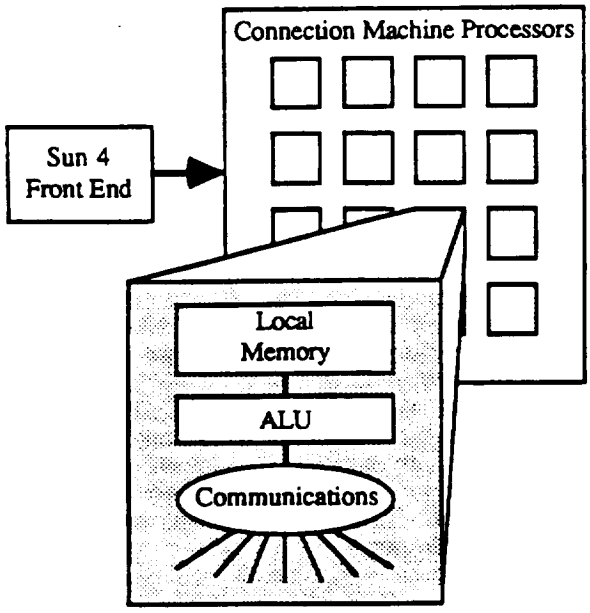

Figure 1. The Connection Machine

By supporting thousands of processors, the CM encourages the programmer to focus on how the data is manipulated and how it interacts with other data. This is in contrast to more conventional multiprocessors where the focus tends to be on the parallel algorithm's flow of control. The CM system software supplies even more flexibility by creating "virtual" processors. The programmer can choose the number of processors necessary to solve a problem and the CM will automatically divide the physical processors into virtual ones. The CM does constrain the number of virtual processors to be a power of two. With the vast number of processors available, it is natural to place each data structure manipulated by a program into a separate virtual processor. Each data structure can then be viewed as having its own processor to perform any computation required.

The CM has a general purpose, hypercube based, communication system that allows each processor to efficiently communicate with any other. Virtual processors generalize this system to allow communication between themselves. Some specialized operators have been created on top of the communication system to perform certain functions very rapidly. Important to PPS are the operations that allow the CM to rapidly replicate data from thousands of virtual processors to thousands of others and a mechanism that allows all active processors to enumerate themselves.

The most idiosyncratic property of the CM is how instructions are supplied to the processors. The CM is a Single Instruction Multiple Data or SIMD machine. While each processor in the CM has its own memory for data storage, it must share its instructions with all others. Each processor has a context flag to control its individual execution of the instructions supplied to all processors. For example, if an *if then else* is reached, all processors calculate the *if* expression together. Those processors failing the *if* test will have their context flag cleared and the remaining processors will execute the *then* clause. The context flag is reversed and those failing the *if* test execute the *else* clause. The context flag is then restored to its original value and execution proceeds. Some efficiency is lost as one or more sets of processors are disabled. The advantage of this approach is that the individual processors and local memory can be made simpler and smaller and hence the CM is able to have thousands of physical processors. For PPS, the SIMD nature of the CM is not limiting and having thousands of physical processors is very important.

The front end processor is responsible for supplying instructions to the CM processors and performing serial computations not well suited to the CM. The front end also supplies the development environment, editors, and the file system. The work done on PPS was performed on a Sun-4 front end.

2.2 Software Considerations

Many reasons support the decision to base PPS on top of CLIPS. Compared to any system we may have built from scratch, CLIPS is a mature system. It was already supporting a user community and was actively being used to write production systems. By starting with CLIPS, we would only need to write and debug those sections of code necessary for parallel evaluation. From a users point of view, PPS only requires small additions to source syntax which allow serial versions of CLIPS, with their debugging tools, to be used to debug productions destined for parallel evaluation. Finally, C source code was supplied with CLIPS without the need for complex negotiations with a vendor.

To avoid losing the advantages gained by basing PPS on CLIPS, it was important to keep the programmers view of PPS very close to CLIPS. Extensions and restrictions from standard CLIPS syntax should be limited in nature and necessary to support parallel evaluation. The major

innovation of PPS is to break facts into serial and parallel groups. The first word of a fact is used to determine the fact's class (in a manner similar to *deftemplate*). The programmer can choose to place certain classes of facts into parallel working memory and they will automatically be processed by the CM. Serial facts are processed by the normal CLIPS mechanisms. The programmer chooses where PPS places facts based on the number of facts in a class and the type of operations performed on these facts.

The first field in a parallel fact is constrained to start with a word which specifies the fact's class. Only classes designated by the programmer will be placed on the CM. Parallel facts must also be of a fixed length and each field of the fact must have its data type specified. Multifield variables are excluded in rule patterns. These restrictions are to lessen the CM memory requirements and to avoid dynamic allocation. Future versions of PPS could lift these restrictions.

The form *deffactfields* creates a fact class and allows a detailed description of the fact's contents. Its syntax is:

(deffactfields *classname*
parallel | serial
(*fieldname type*)...)

The first argument, *classname*, defines this word as a fact class. The next argument is either parallel or serial and it specifies how to process this class. The rest of deffactfields is a list of field names followed by their data type. The standard CLIPS data types have been extended to include integer and boolean. Facts whose first word has never been described with a deffactfields are assumed to be serial facts.

Once a class of facts has been described as a parallel class, the system will automatically place all facts belonging to that class into the parallel working memory. All work required of the parallel working memory is performed on the CM.

2.3 Parallel execution in PPS

As stated earlier, PPS splits the working memory into a serial and a parallel part. When a rule enters PPS, Rete (the standard CLIPS algorithm) is used to compile and process the serial patterns. Parallel patterns are converted to queries of parallel working memory and these queries are attached to the rule body. During execution, Rete manages the serial patterns and when they have matched, the rule is placed in a queue, ready for execution. Upon a rule's execution, a parallel query is performed to collect matching parallel facts, and the rule's body is evaluated in parallel over these facts. In PPS, a single parallel rule evaluation processes all facts that currently match its pattern. The large number of processors available on the CM makes the cost of processing a rule almost independent of the number of facts that it matches. Efficiently processing facts in parallel is very important as the number of facts increases to millions.

There is no certain knowledge that any fact or combinations of facts will actually match the rule pattern. Since it is not known when there is work for a parallel rule to perform, they have to be periodically executed. Currently, PPS uses a simple round robin approach to schedule parallel rules. After each execution, a parallel rule will place itself at the end of the agenda for future execution. This gives other rules an opportunity to execute before reevaluating the current one. Execution terminates when all serial rules have been removed from the activation agenda and no parallel rules are able to find facts or combination of facts not already evaluated. It is assumed that parallel rules, on average, will find many facts to evaluate and this will mask the inefficiencies caused by extra rule evaluations. Section 5 discusses other, more efficient, control strategies.

Since the scheduling scheme used by PPS allows rules to be executed many times, some mechanism is necessary to eliminate the reevaluation of a rule over facts already processed. A global time, based on the number of rule evaluations, is maintained by PPS. Each fact in parallel working memory is timestamped by the rule creating or modifying it. When rule evaluation begins, the timestamp of its previous evaluation is compared to each fact's timestamp. This comparison identifies facts that have entered working memory since the rule's previous evaluation. Only facts, or combination of facts, more recent than the rule's previous evaluation are processed in the current evaluation. This mechanism eliminates the reevaluation of facts by rules.

For PPS to execute efficiently, the CM must be able to query the working memory in parallel, get all matching combinations in parallel, and evaluate the resulting matches in parallel. The CM places each parallel fact into it own virtual processor and is quickly able to query these facts, filter out the uninteresting ones, and create matches. The matches end up in separate processors and all matches are simultaneously available for execution of the rule body. Since the matching and evaluating of rules happen together for all facts, the SIMD nature of the CM has no negative impact on how PPS performs. Instead, it has simplified the writing and debugging of PPS.

What advantage can PPS gain by replacing the Rete algorithm with a potentially expensive database query? When processing a million facts, Rete would have to create millions of intermediate data structures to hold pending queries. These structures would consume megabytes of storage and the management of this storage would place a large computational burden on the system. In its place, PPS requires only a small (32 bit) fixed memory cost per fact. The cost of querying can be justified as long at the average number of matches and subsequent rule evaluation is faster than performing a similar match and evaluation

in another manner (such as Rete). With millions of facts being queried, it is possible that, on average, hundreds or thousands of facts will match each rule execution. Under these conditions, PPS can perform better than alternative methods.

This section has outlined the approach taken by PPS. PPS is more memory efficient than Rete and, under the proper circumstances, PPS will also be more time efficient. The next section will outline that changes necessary to create PPS from CLIPS.

3 Implementation

In order to create PPS, it was necessary to modify the normal CLIPS processing in several places. This section will begin with a short description of how CLIPS compiles and evaluates rules. This is followed by a description of how the rule compiler was modified. Finally, the changes to rule evaluations are outlined.

3.1 Normal CLIPS processing

CLIPS uses *defrule* to create a rule. When CLIPS receives a defrule, it creates two descriptions of the rule being processed. Figure 2 is an example rule with the two descriptions created by CLIPS. The lower left diagram in Figure 2 shows the internal structure of the rule's pattern, the lower right is the internal description of the rule body.

After CLIPS creates the pattern description of the rule, it checks the pattern for internal consistency. The Rete tree builder is then called to create appropriate modifications

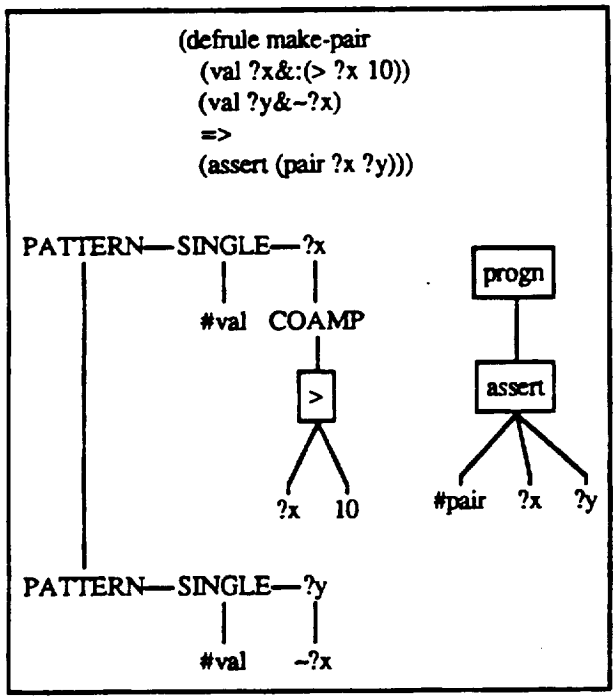

Figure 2. Sample rule and its internal representation

to the discrimination and join network. Finally, the rule body is processed and it is attached to the Rete join network. A more complete description of this process can be found in reference [3].

CLIPS has a built in expression evaluator used to evaluate expression trees. Each box in Figure 2 contains a function that CLIPS will evaluate with the expression evaluator. The lines under each box, connected to other objects, are the arguments required by this function. Each function determines the values of its arguments and then performs its operation. Expressions and the expression evaluator are use both to evaluate the rule body and evaluate conditions inside the Rete algorithm.

PPS interrupts the normal compilation process in two places, the processing of patterns and the processing of the rule body. It also extends the expression evaluator.

3.2 PPS Pattern Compilation

PPS steps in after CLIPS has created the pattern description. It separates the pattern clauses that match serial facts from those that match parallel facts and reorders the clauses to have the serial ones first. If no serial clause is found, one will be created to match the fact *initial-fact*. After the standard internal consistency checks are made on the reordered pattern, the serial clauses are passed to the Rete tree builder for normal processing. The parallel clauses are passed to a pattern compiler which converts rule patterns into equivalent database queries. These queries will be attached to the rule body in a later stage in the processing. From this point the CLIPS processing precedes in a normal manner.

Since PPS uses queries to match the rule patterns, each legal CLIPS pattern must be converted into a database query. These database queries are made up of restricts and joins. Restricts are used to select a subset of the original database based on some conditional test. Joins create new databases containing the possible permutations of the input databases. For example, if two databases had elements (A B) and (1 2 3), the joined database would have the elements (A1 A2 A3 B1 B2 B3).

The pattern compiler examines the rule pattern and creates a database query. Restrict is generated when a pattern limits the value of some field. A join is generated to combine each new pattern clause to the ones already processed. The number of joins will be one less than the number of clauses in the pattern. Where a restriction is placed depends on the required information. If all the information is found in the clause being processed, then the restrict is placed before a join. If the clause needs information from other clauses in the pattern, then the restriction is placed after the join.

In the example found in Figure 2, the rule has a pattern that searches for two facts named *val*. When they are found, the variable ?x is constrained to have a value

greater than 10 and variable ?y must not have the same value as ?x. When a pair of facts matches all these constraints, a new fact named *pair* is asserted into working memory using the fact variables found in the pattern. The results of PPS pattern compilation can be found in the gray area of Figure 3.

The pattern compiler begins by examining the first PATTERN in Figure 2. The first field of this pattern, a SINGLE with a name of *val*, describes the fact's class. The pattern compiler uses this fact class to create a new database (the left hand most db in Figure 3). Processing continues on this clause at the ?x. This variable is found to have an expression constraining its value (found under the COAMP). Since this constraint only uses information found in the current clause, it immediately creates a database restriction to evaluate the constraint (the left hand most restrict in Figure 3). This clause is now finished and the processing is begun on the next PATTERN. Like the previous clause, this pattern expects a fact class of *val*, so another database is created. Since ?y has a constraint dependent on another clause, the creation of the restrict is delayed and the two databases are combined with a join. Processing finishes with a restrict being created to constrain ?y from having the value in ?x (the remaining join and restrict in Figure 3).

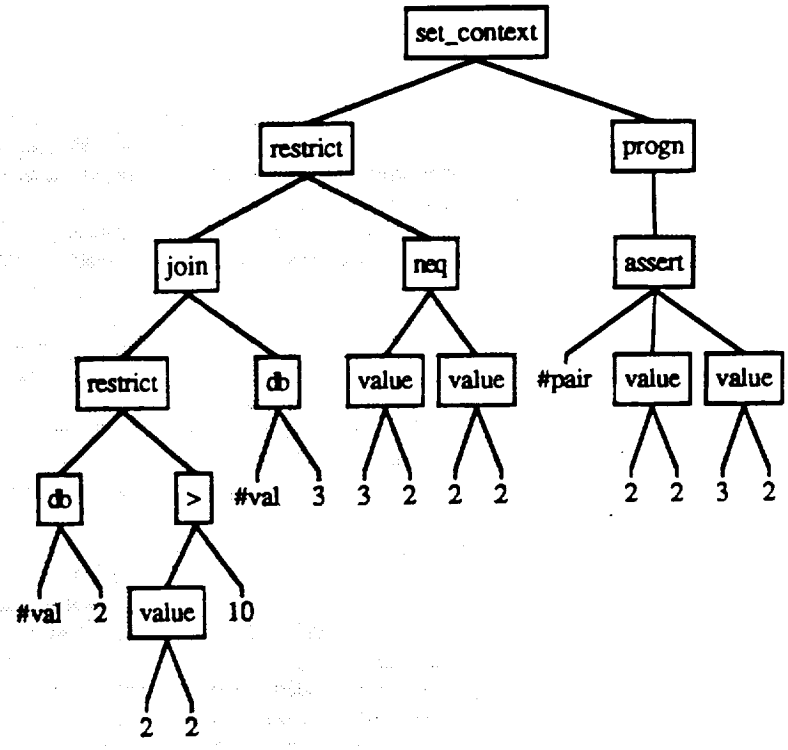

Figure 3. Database query merged with rule body

This query, seen in Figure 3, is equivalent to the original pattern. The execution of this form would be as follows. The db in this expression creates new databases from the original facts. The arguments to db are the name of the fact class and the index number of this pattern. Parallel patterns always start at two or greater since there is always at least one serial pattern in every rule (remember that one is added if none exist). The result of the far lower left db is passed into restrict. Its two arguments are a database and an expression. This restrict limits the database to facts having a value for field 2 of pattern 2 (or ?x) greater than 10. The result of this restrict is passed to the join. Join's arguments are always two databases. The second database entering this join is created from the original facts. Once these databases are joined the resulting database is passed to the top level restrict. This restriction forces the value for field 2 of pattern 3 (or ?y) to be different from field 2 of pattern 2 (or ?x).

Once the database query is created, it is ready to be attached to the rule body. The function *set_context*, seen in Figure 3, takes a database on its left and prepares it for the rule body evaluation. Then the rule body, on the right, is executed. Only one more step is necessary to finish preparing the rule for execution. It will be examined next.

3.3 PPS Expression Compilation

The parallel expression compiler examines expressions to find parallel computations. The standard functions found in the original expression are replaced with parallel equivalents. This module makes use of the data supplied by *deffactfields* to determine what to should be made parallel. Overall, this module uses standard compiler techniques to compile expressions. It keeps track of each source datatype and can convert between different datatypes as appropriate. The only twist is that the CM allows all operands to have variable length and when parallel instructions are emitted, they must include lengths. The final results from the example problem can be see .in Figure 4.

In this final expression tree, various functions have been converted into CM versions. For example, the > found in the first restrict has been converted into a *cm_i_gt* (or CM Integer Greater Than). This function will be performed on the CM and will be applied to all facts in the database at once. The function *cm_conv_si_pi* converts serial integers, 10 in this example, into parallel integers. The 32 appended to various functions is the bit length of the operand. Finally, all references to fact variables is converted into *cm_get_var*. This routine will use current databases to acquire a value for computation. The CLIPS

expression evaluator has to be extended to allow PPS to evaluate parallel expression trees.

3.4 PPS Expression Evaluator

The expression evaluator is extended by adding new data types and by creating new parallel functions. Two data types have been added to the standard CLIPS set. One type is used for parallel databases and the other for parallel variables. The parallel database is used by restrict, join and set_context to identify which database is being manipulated. The parallel variable type points to an address in CM memory. All the parallel arithmetic and boolean functions return this type.

PPS creates a separate set of virtual processors for each class of parallel facts. Each field of a fact is stored in separate parallel variables inside the virtual processors. The system also creates two auxiliary variables. The first is an in-use flag which determines active facts. The second is a timestamp which holds the time this fact was created or last modified (see section 2.3). When facts enter parallel working memory, a free virtual processor is selected and its data fields are initialized. The in-use flag is set and the timestamp is initialized to that of the current rule. When facts are retracted, the in-use variable is cleared.

A PPS database also creates a set of virtual processors. Each virtual processor contains a set of indexes, an in-use flag, and a recent-fact flag. Instead of managing the fact data directly in a database, indexes are used to point to the processors containing the actual facts. When joins merge two databases into a new one, each active processor of the new database has one index for each source database. The in-use flag is true in all processors that contain active database information. The recent-flag is true for database entries that contain facts whose timestamp designates this fact as a recent fact and therefore requiring rule evaluation.

A separate function is used to evaluate each parallel instruction. Each arithmetic and boolean function acquires its operands and invokes the appropriate CM instruction to perform its function. For example, the *cm_i_gt* function in PPS acquires its operands and calls the *CM_s_gt_1L* instruction on the Connection Machine.

The database instructions manipulate the database data structures. The *db* function creates a new database. It examines the database's source facts and initializes a set of virtual processors with the appropriate information. The *cm_get_var* function uses the information in the CM database description and returns the value from this field in the fact. The restrict and join commands directly manipulate the database data structures. Restrict modifies the in-use flag and join creates a new database whose indexes point to the source facts of the original databases.

This section has outlined the major modification made to CLIPS in the process of creating PPS. These modifica-

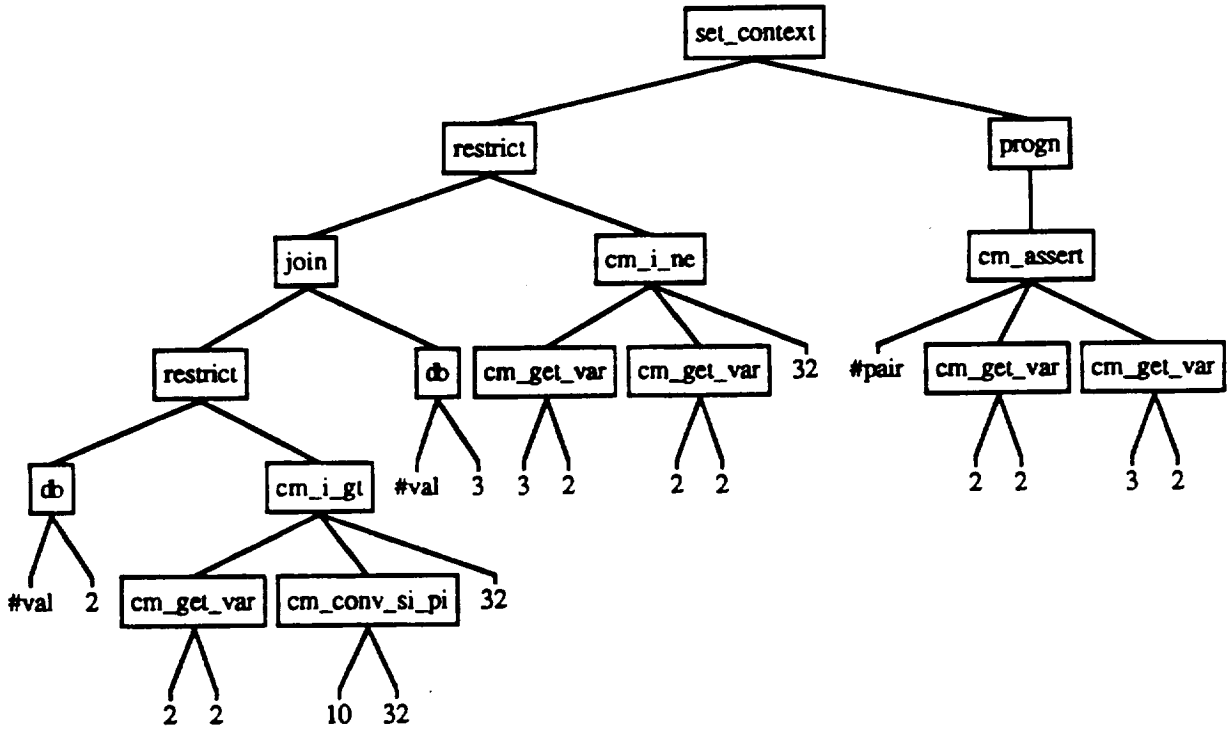

Figure 4. The final parallel expression

tions have focused on converting parallel patterns into database queries and giving the expression evaluator the ability to evaluate expressions on the CM.

4 Performance results

This section will describe the procedure used to test the performance of PPS. Two different tests will be used to compare the PPS results to those of CLIPS.

Both CLIPS and PPS were run on the same Sun-4. In addition to the Sun-4, a CM-2A with 8K processors was used for the PPS benchmarks. Runtimes were measured using the Sun-4 system clock (with *ftime*) and they represent the wall clock runtime. We chose to perform the benchmarks without disabling the normal background processing performed by the Sun-4. This processing occasionally caused small hiccups in the data.

A special command has been added to CLIPS and PPS to perform a benchmark. This command performs a series of runs differing only in the number of facts processed. Runs begin by marking the start time and then entering the correct number of facts into working memory. They are of the form (x *index*) where *index* is from 1 to the number of facts being tested. The production system is started and allowed to run to completion. Finally, a stop time is recorded and the total runtime presented to the user.

The first benchmark examines the ability of a production system to perform simple pattern matching with field values being restricted. The rule used for the benchmark was:

```
(defrule test-rule-1
  (x ?i&:(evenp ?i))
  =>
  (assert (y ?i)))
```

This rule examines working memory for any fact of the form (x ?i) where ?i is constrained to be an even value. When such a fact is found, a new fact (y ?i) is asserted into the working memory. Given the initial facts (x 1) (x 2) (x 3) (x 4), this rule will assert (y 2) and (y 4).

Figure 5 displays the results of PPS as the number of input facts runs between 2048 and one million. The steps seen in this graph are a result of the number of virtual processors required to process the input facts. Since the number of virtual processors is constrained to be a power of two, steps form in the data each time the number of facts forces the CM to go to the next higher size of virtual processors. Once some virtualization level is reached, the runtimes are independent of the number of facts. Since each fact being matched is processed by a separate virtual processor, and there are no interactions between facts (in this benchmark), then the runtime for one fact is the same as that for many. This is very encouraging. To process one million facts, the 8K processor CM-2A took 4.55 seconds. If, however, a CM with 16K processors were available, it would be processing one million facts at the next lower step, or in 2.26 seconds. If a 64K machine were available, the runtime for one million facts would be .57 seconds.

The benchmark run for CLIPS was between 128 and 8192 input facts. As the number of facts increase, CLIPS total runtime increases mostly linearly. Figure 6 shows the results of CLIPS against those of PPS (originally seen in Figure 5). Compared to the PPS, CLIPS increases its runtime very rapidly.

The second benchmark examines how well a production system can perform matches that require more than one fact. The benchmark used the following rule:

```
(defrule test-rule-2
  (x ?i) (x ?j)
  =>
  (assert (y ?i ?j)))
```

This rule will create a (y ?i ?j) with all combinations of indexes found in the (x ?) facts. Given the initial facts (x 1) (x 2), this rule will assert (y 1 1) (y 1 2) (y 2 1) (y 2 2). The number of facts asserted into the working memory is the square of the input facts.

The result from this benchmark can be seen in Figure 7. Both the results from PPS and CLIPS are displayed together. CLIPS was run between 8 and 160 facts and PPS between 16 and 560 facts. At their maximum, CLIPS will assert 25.6K facts (from 160 input facts) and PPS will assert 313.6K facts (from 560 input facts). The runtimes of CLIPS nearly form a parabola and clearly show that for CLIPS the work increases quadratically to the number of input facts. The PPS results show a slow growth in runtime as the number of facts increase.

A real application using PPS is currently under development. When this application is finished it will be possible to get a better understanding of how PPS scales with a mixture of rules. Since the two benchmarks tested represent the major operations performed in production systems, we are optimistic the results will be good.

5 Future work

Most of the restrictions discussed in Section 2.2 were solely for the purpose of making the development of PPS simpler. These restrictions were created primarily to avoid dynamic memory allocation on the CM. With some modification on how parallel fields are managed, variable length facts and strings could be supported. The need to specify field datatypes could also be eliminated.

Section 2.3 discussed the round robin scheduling of parallel rules and mentions that better approaches are possible.

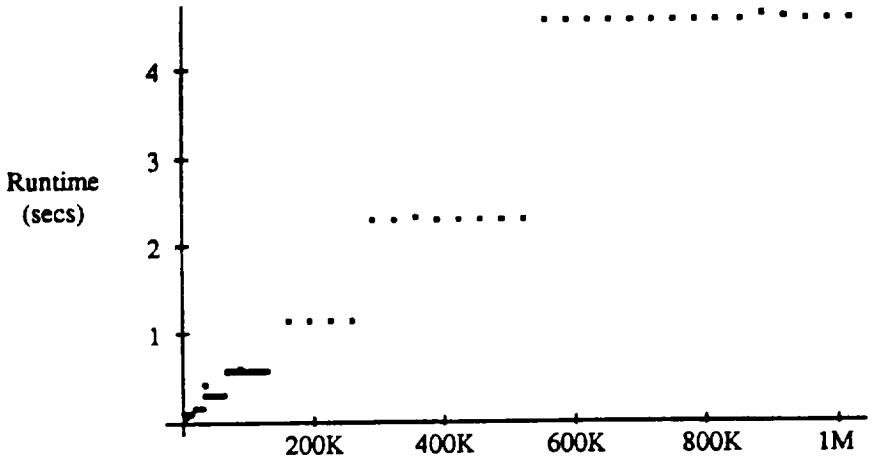

Figure 5. PPS runtimes for the first benchmark as the number of facts processed goes to one million.

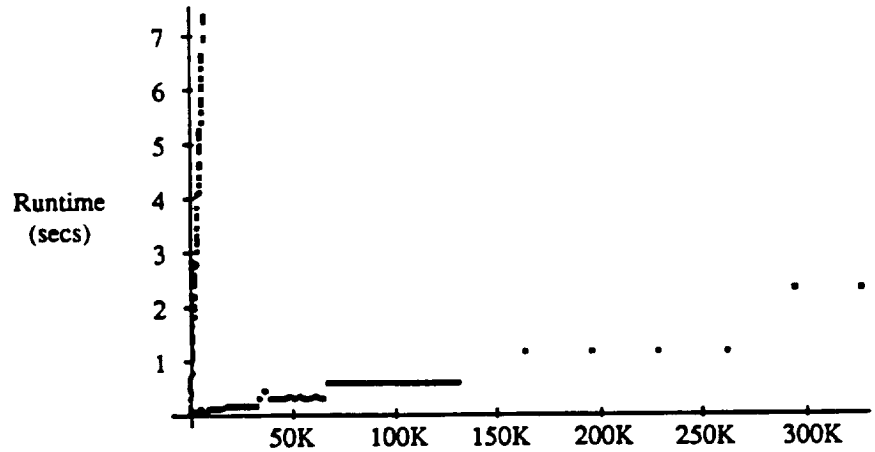

Figure 6. CLIPS versus PPS for the first benchmark. Graph has been rescaled from Figure 5.

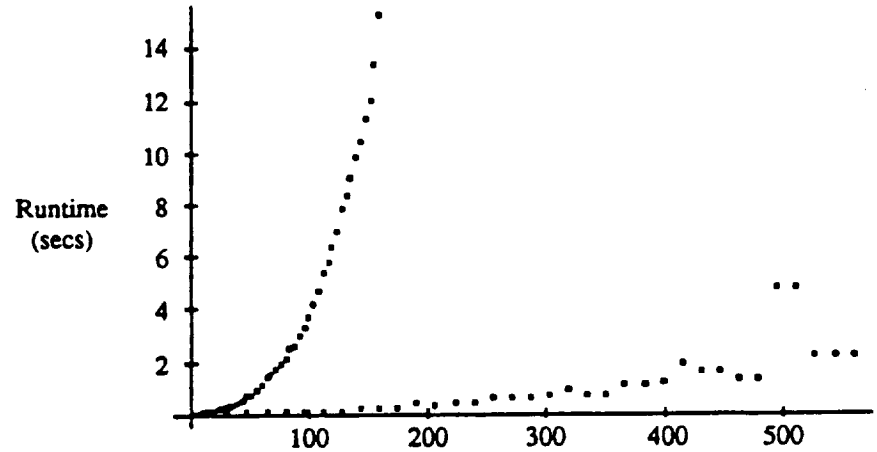

Figure 7. CLIPS versus PPS for the second benchmark.

One approach would be to modify the Rete algorithm used in CLIPS. While it would not be wise to have the Rete algorithm identify and record all partial matches in parallel memory, specialized parallel tests could be added to the Rete tree to determine if at least one parallel fact matches each of the clauses found in a parallel rule pattern. By knowing at least one fact matches each clause of the rule pattern, it has a higher probability of performing useful work when executed. This should increase the overall efficiency of the system.

Another improvement would be to group multiple rule queries together. Subqueries used by more that one parallel rule would only have to be performed once and their answers could be used by all. This is very much like the discrimination network used in Rete which also performs common pattern tests and shares results. It is even be possible to merge this approach with the modification to the Rete algorithm discussed above to seriously limit the unnecessary database queries performed by PPS.

Before attempting any of these modifications, we wish to gain a better understanding of the current performance of PPS. In this way, we will be better able to understand the impact made by changes.

6 Summary and Conclusions

We have described a method, based on parallel database queries and parallel rule evaluations, that allow production systems to process large numbers of facts. Using this technique, CLIPS was modified to create PPS. This new system has a high degree of compatibility with its parent while allowing the user to build applications impossible to process on CLIPS.

Data has been presented demonstrating PPS's ability to perform well in fact rich situations. Particularly encouraging is how well PPS scales as the number of facts increase. Many runtimes are a function of the virtualization level of the CM and are independent of the number of facts being processed. In these situations, the runtime can be controlled by the number of physical processors supplied (which controls how many virtual processors will be emulated on each physical processor).

We are now applying PPS to a low-to-mid level image understanding problem. Since this task generates hundreds of thousands of facts, we believe that PPS is well matched to the problem. Based on the results of this project, we hope to apply PPS to other areas of interest. Some of these interest areas are simulation and modeling, package/vehicle scheduling, and intelligent databases.

References

[1] Forgy, C.L., Rete: A fast algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19, 1982, pp 17-37.

[2] Connection Machine Model CM-2 Technical Summary, Version 5.1 May 1989, Thinking Machines Corporation, Cambridge, Massachusetts.

[3] CLIPS Architecture Manual, COSMIC.