A Distributed Implementation of a Concurrent Logic Language

P. MERINO, R. RUBIO AND J.M. TROYA

Dept. de Lenguajes y Ciencias de la Computación. Universidad de Málaga, Apdo. 4114, 29080 Málaga, Spain (email: merino@tecma1.citma.unio.es, rijo@tecma1.citma.unio.es, troya@tecma1.citma.unio.es)

SUMMARY

Concurrent logic languages are accepted as very well suited to a wide range of applications and to exploit parallelism. Since concurrent logic oriented architectures are not available, it is usual to design an abstract machine and use current systems to emulate its behaviour. We define a distributed abstract machine for the execution of Flat Parlog programs, that includes protocols to maintain the global address space, and a dynamic load balancing method which is efficient compared to a programmer driven one. We use transputer networks as the target architecture to test the model. Experimental results demonstrate transputer systems to be an appropriate support for these kinds of languages, and the proposed abstract machine a satisfactory model for the distributed implementation of Flat Parlog.

KEY WORDS concurrent logic languages; abstract machine; transputer networks; load balancing

1 INTRODUCTION

Logic programming has become one of the most important programming paradigms in recent years. Great effort has been devoted to exploiting parallelism in logic programs. As in other kinds of languages, two different approaches have been taken to achieve this task. The first consists of extracting parallelism from sequential programs using special compilers. This method is applied to Prolog for exploiting the two principal kinds of parallelism: and- and or-parallelism[1]. The second approach is based on using programming languages designed to explicitly express concurrency, and compilers or runtime support to map concurrent processes onto the target parallel architecture. Within this approach, the most important languages are those known as concurrent logic languages, which are well suited to create any kind of parallel program[2]. These languages are characterized by guarded clauses, the commitment of choosing only one candidate clause for a goal (without backtracking) and a variable synchronization mechanism to control the execution of programs sufficiently to enable stream and parallelism to be exploited in a practical way. The most well known concurrent logic languages are Parlog[3], concurrent Prolog (CP)[4] and guarded Horn clauses (GHC)[5]. The main differences between them are in the form of declaring the mode of the arguments and the synchronization mechanism.

There are implementation difficulties associated with all the above mentioned languages[6]. Flat versions of these languages have been proposed to simplify them (flat

---

This work has been supported by the Spanish project CICYT TIC 340/90.

CCC 1070-454X/96/02017-19
©1990 by John Wiley & Sons, Ltd.

Received 22 September 1994
Revised 24 January 1996
5.1 Messages for data exchange

Figure 4 shows how the number of messages for data exchange increases as the number of nodes is higher. The explanation is quite simple: the greater the number of nodes, the more dispersion of data. Most of these messages are sent to read ground terms, during the test phase. Only a few of them represent remote unification of variables, due to lazy updating and local writing.

5.2 Dynamic load balancing

Figure 5 shows how our heuristic gives a good balance of processes in the sixteen node configuration. It is important to note the low number of processes that broadcast to other nodes in order to obtain the previous distribution (see Table 2).

<table>
<thead>
<tr>
<th>Table 2. Broadcast processes, execution time and speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast processes</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>Sequential</td>
</tr>
<tr>
<td>1 Node</td>
</tr>
<tr>
<td>2 Nodes</td>
</tr>
<tr>
<td>4 Nodes</td>
</tr>
<tr>
<td>9 Nodes</td>
</tr>
<tr>
<td>16 Nodes</td>
</tr>
</tbody>
</table>
5.3 Speedup

Table 2 summarizes speedups obtained in the parallel configurations. The first speedup column represents speedup with respect to a sequential version with no support for remote structures, and running with the same heap size and stack of arguments as other versions. The second one shows the comparison with the plus-overhead version configured for running in one transputer. In Figure 5, the plain line represents the second set of measurements.

5.4 Dynamic load balancing vs. programmer driven mapping

Dynamic load balancing allows the programmer to write parallel programs without paying attention to mapping the processes onto the actual architecture. Our algorithm is adequate to provide this service. To test its suitability, we have compared the method with a programmer driven mapping, using the call \( \Phi(\text{process, processor}) \) in the program to specify where to execute a process. The argument processor can be a node identifier, or the constants any, local, previous, next. The value any is used to prevent a node in the network from becoming idle, using the dynamic load balancing method.

The same mechanisms are used to move processes from one node to another in the implementation of the \( \Phi \) predicate as explained in Section 3.5.

To map the quicksort algorithm with \( \Phi \), the problem is solved by initially splitting the input list. Every piece of data can be sorted in a different node, using the \( \Phi \) to give information about the initial mapping.

```prolog
mode quicksort(?X,?Y,?Z).

quicksort(X,Unsorted,Sorted) :-
   N1 is X/2, N2 is N1/2, N3 is N1-N2,
   partition(N1,Unsorted,LessN1,MoreN1),
   partition(N2,LessN1,LessN2,MoreN2),
   partition(N3,MoreN1,LessN3,MoreN3),
   \( \Phi(\text{qsort}(\text{MoreN3},\text{Sort1},[]),1) \).
```
5.3 Speedup

Table 2 summarizes speedups obtained in the parallel configurations. The first speedup column represents speedup with respect to a sequential version with no support for remote structures, and running with the same heap size and stack of arguments as other versions. The second one shows the comparison with the plus-overhead version configured for running in one transputer. In Figure 6, the plain line represents the second set of measurements.

5.4 Dynamic load balancing vs. programmer driven mapping

Dynamic load balancing allows the programmer to write parallel programs without paying attention to mapping the processes onto the actual architecture. Our algorithm is adequate to provide this service. To test its suitability, we have compared the method with a programmer driven mapping, using the call @ (process, processor) in the program to specify where to execute a process. The argument processor can be a node identifier, or the constants any, local, previous, next. The value any is used to prevent a node in the network from becoming idle, using the dynamic load balancing method.

The same mechanisms are used to move processes from one node to another in the implementation of the @ predicate as explained in Section 3.5.

To map the quicksort algorithm with @, the problem is solved by initially splitting the input list. Every piece of data can be sorted in a different node, using the @ to give information about the initial mapping.

```prolog
node quicksort(?N,Unsorted,Sorted) :-
  N1 is N/2, N2 is N1/2, N3 is N1+N2,
  partition(N1,Unsorted,Less=N1,More=N1),
  partition(N2,Less=N1,Less=N2,More=N2),
  partition(N3,More=N1,Less=N3,More=N3),
  @(qsort(More=N3,Sort1,[]),1),
```
\(0(qsort(LessN3,Sort2,Sort1),2),\)
\(0(qsort(MoreN2,Sort3,Sort2),3),\)
\(0(qsort(LessN2,Sorted,Sort3),local).\)

\texttt{mode qsort(?,-,-).}
\texttt{qsort([MUX],Sorted,Rest) \leftarrow}
\texttt{\(0(partition(MUX,X,LessN,MoreN),local),\)
\(0(qsort(LessN,Sorted,[MUX],Sort1),any),\)
\(0(qsort(MoreN,Sorted,Rest),any).\)
\texttt{qsort([],X,X).}\)

Table 3 shows the execution statistics of the above program with different mappings. Every execution produces 105,761 processes in the system. We have also obtained the same results executing the same splitting by using the dynamic load balancing instead of the \(0\) predicate.

6 CONCLUSIONS

As far as we know, no other implementation of Parlog is available for transputer networks. We have presented a distributed implementation to exploit parallelism with flat Parlog, which is a more efficient variant of Parlog. The developed tool is composed of compiler, command interpreter, emulator, and \(V_0\) support. The TINY kernel has been used to route messages.

Some advantageous features of our model have been presented. The protocol for dis-
Table 3. Statistics using @

<table>
<thead>
<tr>
<th></th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Node</td>
<td>157.00</td>
<td>1</td>
</tr>
<tr>
<td>2 Nodes</td>
<td>83.48</td>
<td>1.88</td>
</tr>
<tr>
<td>4 Nodes</td>
<td>40.23</td>
<td>3.90</td>
</tr>
<tr>
<td>9 Nodes</td>
<td>23.72</td>
<td>6.51</td>
</tr>
<tr>
<td>16 Nodes</td>
<td>16.05</td>
<td>9.78</td>
</tr>
</tbody>
</table>

Distributed unification produces a low number of messages to unify remote terms, considering the high number of processes in the computation. The model for process migration supports a heuristic in order to obtain good load balancing in a transparent way.

Transputers have been proved suitable to implement these kinds of languages. Results show that even when sending 130,000 messages to execute 400,000 processes the system can still provide an acceptable speedup.

REFERENCES

[18] B. Rubio and J.M. Troya, 'A sequential implementation of flat Parlog', Departamento de


Parlog[7], FCP[8], FGHC[9]). Flat languages restrict guards to built-in predicates only to obtain more efficient implementations. Nowadays these languages are valuable tools to solve practical problems[10–13].

An abstract machine can be defined for a compiled implementation of these languages. All the proposed abstract machines are based on the WAM (Warren abstract machine), defined by Warren for Prolog[14]. Jim Cramond defined his JAM abstract machine for parallel implementations of Parlog on shared memory multiprocessors[15]. In [16] an abstract machine is described for a sequential implementation of Parlog on a single processor (SPM). For other languages of the same family the abstract machines proposed for FCP[17] and FGHC[9] stand out.

The main problem presented by the implementation of these languages on non-shared memory multiprocessors is global address space support. When implementation is based on keeping a logical variable in only one node and using remote references to it from other nodes, we need communications to read and unify terms. There is not a very great difference in the read operations among the different proposals for implementing these languages on distributed systems, but there is a big difference in the unification of remote terms.

In this paper we present a distributed abstract machine for the implementation of flat Parlog on non-shared memory multiprocessors. It is defined in terms of a sequential abstract machine for Parlog[18], which is an adaptation of the JAM. The execution model is a process oriented one based on an and-tree model of execution[19]. The abstract machine incorporates a distributed unification method based on local writing, if possible, and lazy updating of remote references, so that the number of messages for unification is minimized. It also possesses mechanisms to support process migration. In order to distribute the processes, we use a dynamic load balancing method.

Since real architectures matching abstract machines are not available, it is usual to construct an emulator to implement the languages on current single processor or parallel machines. Transputer networks are one of the systems employed to test the suitability of new computational models[20, 21].

Our goal is to benchmark a distributed implementation of Flat Parlog on transputer based multiprocessors. The current implementation has been developed with the ANSI-C toolset[22] on a Parsys SN1000 with sixteen T800 nodes[23]. Experimental results show transputer systems to be an appropriate support for this kind of implementation, and the proposed abstract machine as a satisfactory model for distributed implementation of the language.

The next subsections summarize some proposals for distributed implementations of concurrent logic languages and our goals are discussed. Section 2 of the paper reviews Flat Parlog as a language for logic programming and concurrent programming. The distributed abstract machine is defined in Section 3. Some details about the implementation can be found in Section 4, and comments on the results in Section 5. Finally, we present some conclusions about this work.

1.1 Related works

Implementations of concurrent logic languages on distributed machines are focused very strongly on specific kinds of applications (e.g. theorem proving, system programming, simulation, prototyping, etc.), and they have been developed with different specific requirements. This subsection reviews some of the better studied cases and the next one
summarizes the characteristics we have incorporated in our model.

The full metacall facility described in [3] is especially interesting for system programming for monitoring and controlling computations: it provides access to status information and a way to suspend, resume and abort particular tasks. Distributed implementations of Parlog in [24, 25] include kernel support for this mechanism, by providing a low level set of primitives used at the language level to complete the services. They also include algorithms to detect deadlock and termination in groups of tasks. A metacall oriented kernel to implement a flat version of GHC called KL1 (kernel language one) is described in [9]. It is based on metacall records to store links to descendant goal records, status information and the instruction pointer. Taylor’s FCP on Intel Hypercube has no low-level mechanisms to control computations completely [12]. In this case, only the programs to be controlled are transformed and executed in a metainterpreted fashion.

Every distributed kernel has support for a distributed unification algorithm in order to maintain a global address space for all processes. The complexity of this algorithm depends heavily on the needs imposed by the semantics of the language. FCP commitment is an atomic operation that involves a variable migration mechanism to make a set of variable unifications in the same processor. Other methods are based on remote unification or read operations with data exchange but no variable migration. Foster’s algorithms to detect termination and to detect deadlock in a set of tasks involve a high level of complexity due to the additional information to be stored, and the messages to confirm unification to the task which initiated the unification operation. Like the method described in [9], Foster’s binding rules are based on the processor identifier and memory address (to prevent circular references). Algorithms employed in KL1’s implementation do not involve acknowledgment messages to detect termination.

Another important issue to be discussed is how to control the process tree generated during the execution, that can be distributed when processes migrate from one processor to another. KL1 metacall allows descendant processes from the same metacall to be executed in different processors. If a goal is sent, a proxy record is linked to the metacall record to replace the process link. All descendant goals from the same metacall that run in the same processor are linked to a foster-parent record in order to reduce the communication with the proxy record. Foster’s and Taylor’s implementations take into account process migration by program transformation rather than by kernel support: a server process programmed in the language receives requests to create new processes.

Taylor’s work also includes load balancing by programming techniques and program transformation. Virtual machines and tools for dynamic process mapping and dynamic code mapping can be defined at the programming level. Foster considers that load balancing is not included in the implementation and must be programmed in the language using message passing. A discussion on load balancing in KL1 can be found in [26]; automatic balancing is only considered within a cluster of the PIM (parallel inference machine) and programmers must use manual goal scheduling among clusters by mean of pragmas.

The last problem we consider is memory garbage collection. Taylor uses a parallel method based on stopping all processors to copy useful cells in every memory. Ichiyoshi and another introduce the export table to obtain uncoupled local and global procedures [9]. In [27] the import table and an incremental inter-processor garbage collection method based on weighted reference counters are introduced.
1.2 Discussion

Our goal is to test transputer based systems for the transparent execution of Flat Parlog programs, focusing on data exchange and process migration. No special attention is paid to programs for system programming that must be controlled by means of metacall primitves. If necessary, programs could be automatically transformed as described in [25] or [12], neither are complex methods for global garbage collection considered which could influence the distributed management of the global address space[27]. If these characteristics are not taken into account, the low-level implementation is well suited to benchmark two of the most critical aspects of the distributed implementation of concurrent logic languages: distributed unification algorithms and the overhead to enable dynamic load balancing at kernel level.

We use remote references and broadcast note structures as described in [12]. Lazy updating and export tables used for KL1 are also incorporated into the flat Parlog execution so as to reduce the number of inter-processor messages and to allow local garbage collections without involving other nodes, respectively.

If metacall records are not used, granularity is too fine to maintain a low overhead using proxy and foster-parent records. This mechanism could also increase execution time due to the dynamic exchange of processes among nodes. Hence we use direct links from children to parent processes.

2 PROGRAMMING IN FLAT PARLOG

We refer to [2] for a detailed study of the family of concurrent logic languages. Here, flat Parlog is sufficiently reviewed as to enable understanding of the following sections.

From the point of view of logic programming, there are two main characteristics that make flat Parlog, and its family, a different logic language:

1. Only one execution path is explored and there is no backtracking if a reduction fails. Clauses are selected in a don't care non-deterministic way.
2. Clauses can have guards to set additional conditions that must be satisfied to choose them for reduction.

A flat Parlog program is a set of clauses with the form:

$$ H < -G_1, \ldots, G_m : B_1, \ldots, B_n \quad (m, n \geq 0). $$

H is the head of the clause. G1, … Gm represent the optional guard, which is only constructed with built-in predicates. B1, …, Bn are the body goals. A set of clauses with the same head can be read as a procedure. The whole procedure is preceded with a mode declaration to explicitly mark input and output arguments. A clause is only reduced if matching between the input arguments of the head and the corresponding arguments of the calling goal succeeds (input matching) and the guard also succeeds. If there are several candidate clauses (both the input matching and the guard are successful), only one is selected for reduction, which can be suspended temporarily waiting for some arguments to be ground. The arguments of the calling goal corresponding to input arguments cannot receive values during clause reduction. If both the input matching and the guard are successful, the clause is committed and the output arguments are unified with the corresponding arguments of the calling goal. This is then replaced with the body goals of the clause. If the unification of the
output arguments fails or any goal in the body fails, no other clause can be tried. Therefore the initial goal (user’s query) will conclude with failure. This is the reason why these languages are also called committed choice languages. The programmer must adequately construct the procedures for the selection of the correct clause every time.

From the point of view of concurrent programming, flat Parlog possesses mechanisms which describe the behaviour of a process, synchronization conditions, communication and the dynamic process creation. If we consider an and-tree execution model[19], a process corresponding to a flat Parlog goal and its behaviour is defined with the set of clauses having the same head. Each clause indicates a different process evolution depending on the input arguments and the guards, and the process progresses by selecting and reducing clauses. The user’s query determines the initial set of concurrent processes in the system.

Communications take place when a process consumes the values generated by another process using shared variables. This communication can be incremental, exploiting stream and parallelism. If a process must try a list of elements generated by another process, the former may consume the elements generated by the latter though the whole list has not been generated yet. For an easy and efficient implementation of this kind of parallelism it is necessary for the logical variable to have the property of single assignment, without backtracking[28]. For this reason these languages require a mechanism of synchronization to ensure that a shared variable is instantiated by the appropriate process. Synchronization is achieved by restricting the selection of a clause using arguments with input mode. The process can only progress if another process gives an appropriate value to a shared variable with input mode. New shared variables can be dynamically created to continue with communication. When all restrictions are satisfied and a clause is chosen for reduction, new variables are created, variables with output mode are unified and a new set of processes is spawned to replace the current process.

3 DISTRIBUTED ABSTRACT MACHINE

The abstract machine is composed of a set of flat Parlog nodes with the capability of executing flat Parlog processes and with protocols to exchange data and processes with other nodes. They also contain protocols to check deadlock and termination. An identifier is assigned to each node, which is used for communication among nodes, remote references and address comparison. There is a main node, with the lower identifier, which starts the computation and governs deadlock and termination conditions. A compiler generates the program with abstract instructions from the original flat Parlog program. The following description includes: the execution model, data areas and structures, instruction set, data exchange and process migration.

3.1 Execution model

This provides the basic control mechanism to execute flat Parlog programs on the abstract machine. The execution model is process oriented, based on an and-tree model [19], adapted for a flat version of Parlog and augmented with characteristics for distributed systems. A process is entirely executed in the same node, therefore it will only be sent to another node before starting its execution.

Figure 1 represents the four possible states for a process:
1. *Active*. The process waits to be selected for execution or to be sent to another node. Limiting migration for some processes is a heuristic in the load balancing strategy of the abstract machine.

2. *Execution*. The execution of a process consists of selecting and reducing the clauses from the set that composes its definition.


4. *Suspended on variables*. The process stays in this state until a variable or remote reference is bound.

Execution of a process consists of two phases: the *test phase*, in which it tries to find a candidate clause for the goal to commit to, and the *spawn phase*, involving the creation of new processes for the body goals of the selected clause.

In the test phase a process starts execution by performing input matching among its arguments and the ones in the head of the first clause in the procedure. If this succeeds, then the process executes the guard tests (if any) and if that also succeeds, the process commits the goal to this clause. If either input matching or one of the guard tests fails, the process tries the next clause.

If input matching or a guard test cannot continue because a variable in the calling goal is not sufficiently instantiated, then the variable is added to a suspension table (which is
empty at the start of the test phase), and the process tries the next clause. This algorithm is repeated until a successful clause is found or until all clauses have been tried.

If no candidate clause is found, but there are some variables in the suspension table, the process will suspend on these variables. When any one of these variables is instantiated (by another process), the process will be resumed and it will restart input matching from the first clause again. If there are no candidate clauses and the suspension table is empty (all the clauses have failed) then the process terminates with failure, and hence the computation fails.

In addition to parallel clause separators (',') flat Parlog can use sequential clause separators (';') for the alternative clauses of a procedure. A sequential clause separator specifies that the clauses after the separator should only be tried if previous clauses fail. If no candidate clause is found when the sequential clause separator is reached, but there are variables in the suspension table, the process suspends immediately. Alternatively, if all previous clauses have failed (the suspension table is empty), execution continues to try the following clauses. But if the process subsequently suspends on a variable, the execution will resume from the first clause after the sequential clause separator, since all clauses before it are known to have failed.

In the spawn phase, after the commitment of a clause, the process performs any output unification among the output arguments of the clause head and the corresponding arguments of the calling goal. If this fails, the process terminates with failure. When unification succeeds the following two cases are possible. If there are no body goals (unit clause) the process terminates with success. If the body contains goals it then executes the clause body creating child processes for the body goals.

Flat Parlog can also use sequential separators for goals (sequential conjunction separators '&&') in addition to parallel conjunction separators (','). If the body contains parallel conjunctions separated by sequential conjunction separators, e.g.

\[ H \leftarrow B_1, B_2 \& B_3, B_4 \& B_5, B_6, B_7. \]

then the process creates child processes for the goals in the first parallel conjunction (B1,B2) and then suspends on children. If they all terminate with success, this process resumes to execute the next conjunction. For the last parallel conjunction (B5,B6,B7) a form of tail recursive optimization can be employed: the process creates sibling processes for these goals, except for the last goal that is executed by itself. Its parent process inherits these new children and hence waits for these to terminate as well as the children it created itself. Note that in programs which do not use sequential conjunction separators, all processes are created as siblings of the root process that initiated the execution of the top level query.

### 3.2 Data areas and structures

Each node in the distributed abstract machine has a set of main data areas (Figure 2).

#### 3.2.1 Stack of processes

A process structure identifies a process and contains sufficient information for a process to be executed and to communicate its outcome to its parent when it terminates: parent pointer, instruction pointer, argument pointer, etc. A process structure can only reside in
one stack. Each node only executes processes from its local stack. Processes can migrate by sending their structures to another stack.

3.2.2 Stack of arguments

This basically contains values (or references to values) of the process arguments in the local stack of processes. It can also contain environments. A process needs an environment when the selected clause has variables that occur both before and after a sequential conjunction separator, i.e., permanent variables [19]. Arguments and environments are moved from one stack of arguments to another when the process migrates. The argument stack is a stack from the point of view of allocation. All new items (goal arguments and environments) are pushed onto the top of the stack. However, a stack has to be able to contain holes, so that items can be deallocated even though they may not be on the top of stack. Holes are created...
by marking the cells being deallocated as free cells. Whenever cells on top of the stack are
popped, a check is made to see if this has exposed a hole, in which case the hole is also
removed from the stack.

This means that allocating different sized blocks of cells for arguments or environments
is very cheap; if these blocks are deallocated in reverse order (i.e. exactly like a stack) then
memory is used in a very efficient way. In some cases, however, items are not deallocated
in reverse order and potentially large holes can be created. In this case a stack garbage
collection can be performed to reclaim space. This is a relatively easy and quick operation
because the cells of this area can only point to cells in the heap, and they can be only
pointed to by process structures. There are no intra-stack pointers. This is possible because
unbound variables are not created on argument stack cells, only references to their cells on
the heap are stored.

3.2.3 Heap

This contains data produced during the execution of the program. It can also contain remote
references to data in other heaps of the abstract machine. When the heap becomes full, a
general garbage collection is required to reclaim space. Unlike the stack garbage collection
this is a more complicated and slow operation since a given cell in the heap can potentially
have several pointers to it emanating from the argument stack, the export table, the proper
heap and some registers of the node. All these cells must be updated when the heap cell is
relocated.

3.2.4 Export table

A local variable is referenced from other heaps with a remote reference that must be mapped
onto an indirect pointer table to the heap. This localization operation is performed every
time a node receives a remote reference to its own heap. This mechanism allows each node
to reclaim useless local heap cells without affecting other nodes.

3.2.5 Execution queue

This only contains pointers to active processes in the local stack of processes.

3.2.6 Code area

The code generated by the compiler is broadcast to all nodes. The process structures
reference their current instructions.

In addition to these areas, each node has a set of internal registers: HP points to the top
of the heap, SP points to the top of the argument stack, PC is the program counter, etc.

3.3 Instruction set

The abstract machine code for a flat Parlog procedure consists of instructions for trying
alternative clauses followed by the code for each clause. The code for a clause can be
further divided into test phase and spawn phase instructions. We can divide the instruction
set as follows:
(a) For trying alternative clauses: \texttt{try, trust, suspend\_process}
   (i) \texttt{try} is a jump instruction to the beginning of the clause code.
   (ii) \texttt{trust} is similar to the \texttt{try} instruction, but for the last clause of the procedure.
   (iii) \texttt{suspend\_process} implements the sequential clause separator.

(b) Test phase: \texttt{wait, read}
   (i) \texttt{wait} instructions correspond to input arguments of the clause.
   (ii) \texttt{read} instructions try subterms of lists and structures on input arguments.

Both groups embody the possibility of reading remote terms.

(c) For selecting a clause: \texttt{Commit}.

(d) Spawn phase: \texttt{get, unify, proceed, fail, call, put, push,}
   (i) \texttt{get} instructions correspond to output arguments.
   (ii) \texttt{unify} instructions correspond to the subterms of lists and structures.

Both groups embody the unification algorithm described in subsection 3.4.

(iii) A process that executes the \texttt{proceed} instruction terminates successfully.
(iv) The \texttt{fail} instruction terminates the computation with failure
(v) \texttt{call} instructions create processes.
(vi) put and push instructions build process arguments.

(e) For creating and liberating environments: \texttt{allocate, deallocate}.
(f) For executing the built-in predicates in the guard: \texttt{builtin\_n, builtin\_o}.

3.4 Data exchange

To start the execution of a flat Parlog program the initial set of data from the user's query is stored in the the main node heap. Afterwards, processes migrate, arguments move from one node to another and new data structures are created in any node. So, data to unify terms can reside in several nodes.

If a logical variable resides only in one heap and remote references from other heaps are used, distributed unification algorithms are needed to preserve the semantics of the language. These algorithms define the protocol rules to read and write values into variable locations. We use a method to reduce the number of inter-node messages by using a lazy updating of cells containing remote references.

3.5 Reading data

Input matching and guard tests may require ground terms or references to override suspension. Messages \texttt{Read\_Ground} and \texttt{Read\_Reference} are required when remote references appear. On receiving these messages, a node executes the following algorithms to send data:
Read_Ground(caller, called, caller_address, called_address)
{
  heap_index = dereference(localize(called_address))
  case heap_index:
    unbound variable: create a Strict_Pending_Read
    remote reference: send Read_Ground(caller, new_called, caller_address, new_called_address)
    ground term: send Ground (caller, caller_address, heap_index)
  }

Read_Reference(caller, called, caller_address, called_address)
{
  heap_index = localize(called_address)
  new_heap_index = dereference(heap_index)
  case new_heap_index:
    unbound variable:
      if new_heap_index == heap_index
      create a NonStrict_Pending_Read
    else
      send Reference (caller, caller_address, new_heap_index),
      remote reference: send Reference (caller, caller_address, new_heap_index)
      ground term: send Ground (caller, caller_address, new_heap_index)
  }

The structures Pending_Read associated with a variable are examined when giving a
value to the variable. Reference and Ground messages are sent to the caller node to
resume processes waiting for values of remote terms.
A maximum depth is considered when passing structured terms in Ground messages.
The rest of the structure is replaced with a remote reference.

3.5.1 Writing data

The unification of terms containing remote references generates messages to continue
unification in other nodes. We attempt to minimize the number of these messages by
writing locally when possible and replacing remote references with their actual values in a
lazy way.

The Unify message carries the local term and the index in the remote export table to
continue the unification mechanism. The next algorithm shows the actions to be carried out
when receiving the Unify message.

Unify (caller, called, caller_term, called_address)
{
  heap_index = localize(called_address)
  term1 = restore(caller_term)
  term2 = dereference(heap_index)
  if not(general_unify(term1, term2) == Success)
    send Fail(main_node)
  }
Table 1 summarizes the actions in the general unify algorithm, also invoked when starting unification by program instructions. This algorithm may generate new Unify messages.

If general unify works with a remote reference, bindings are always made from the lower to higher node identifier to prevent circular references. Unify messages are only sent when a remote reference is overwritten.

3.6 Process migration

Load balancing among nodes is achieved by dynamically broadcasting processes from one node to another. A process migrates when its entry in the stack of processes and its arguments in the stack of arguments are moved to another node. Original copies of these structures are deleted. When passing arguments, only ground terms are really moved, and references to cells in the heap become remote references.

Our model is based on following assumptions:

1. Every node maintains an estimation of the number of active processes in other nodes. Initially, all estimations are considered to be zero.
2. Every message between nodes carries the number of active processes in the caller node. So, the receiver can update its table of estimations.
3. When a node is able to make a context switch, it consults the next active process. If the process can migrate then the best destination node is selected, using the estimation table, and the process is sent.
4. To select the best node, the estimation table is traversed in a ring, beginning from the next node until a node with a low load is found. If a low loaded node is not found the local node is selected as the best.
5. Processes arriving at a node are marked as non-migratable and added to the execution queue.
6. Built-in processes are always marked as non-migratable when created.
7. A process can only migrate before starting its execution.

4 IMPLEMENTATION

The parallel system has been designed around a main node that starts all flat Parlog computations. An emulator program written in C implements the functional character of each node of the distributed abstract machine. Each node is mapped onto one transputer. The emulator in the main node also runs the command interpreter and the compiler (Figure 3).

Every transputer runs a router to provide inter-node message passing. Every node in the system can perform input–output operations, (i.e. to execute input–output primitives or to print system errors) by means of a set of small client tasks that implements the Inmos Iserv Protocol[22]. These tasks take request packets from the flat Parlog emulator, send the packets to the root transputer using the router, await the replies, and give them to the emulator. The root transputer runs another small task to interface with the server running in the host. This mechanism has been proved very efficient.

The current version incorporates the TINY kernel to route messages[29]. We use synchronous messages for the Iserv Protocol. Flat Parlog emulators communicate via asynchronous sequential messages.
Table 1. Distributed unification

<table>
<thead>
<tr>
<th>X \ Y</th>
<th>Ground</th>
<th>Unbound variable</th>
<th>Remote reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unification</td>
<td>Y := X</td>
<td>Y := X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send UNIFY(X, Y)</td>
<td>send UNIFY(X, Y)</td>
</tr>
<tr>
<td>Ground</td>
<td>If X &lt; Y</td>
<td>X := Y</td>
<td>If X &lt; Y</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>Y := X</td>
<td>Y := Y</td>
</tr>
<tr>
<td>Unbound variable</td>
<td>X := Y</td>
<td>Y := X</td>
<td>send UNIFY(Y, X)</td>
</tr>
<tr>
<td></td>
<td>if Y &lt; X</td>
<td>Y := X</td>
<td>send UNIFY(Y, X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>X := Y</td>
<td>else</td>
</tr>
<tr>
<td>Remote reference</td>
<td>send UNIFY(Y, X)</td>
<td>X := Y</td>
<td>send UNIFY(Y, X)</td>
</tr>
<tr>
<td></td>
<td>send UNIFY(Y, X)</td>
<td>Y := X</td>
<td>send UNIFY(Y, X)</td>
</tr>
</tbody>
</table>

The whole system has been developed with the Inmos ANSI C tool set[22] on a Parsys SN1000 with sixteen T800 nodes[23]. Benchmarks were taken configuring every node with 800 Kbytes for the heap and 300 Kbytes for the stack of arguments.

5 EVALUATION

Normally a fixed small range of problems is used to benchmark new implementations of parallel logic languages, and the more suitable are not always chosen[30]. One example is the naïve reverse program, a simple solution to reverse a list of numbers. This problem, based on append processes to construct the reverse list, has been shown to be unsuited to our model: only small speedups can be obtained. The reason is discussed in [30]: the appends are serialized by synchronization on their input arguments and suspensions are frequent. Another example is the traditional solution of the n-queens problem to test Prolog systems. A stream and-parallelism oriented model cannot produce good speedups[15]. So, we decided to test different configurations with highly parallel programs such as the quicksort algorithm to sort a list of numbers.

The following piece of code partially shows how to program the quicksort algorithm in flat Prolog. A qsort process is spawned into three processes. The partition process makes two different lists: S contains the elements smaller than or equal to N, L contains the elements larger than N. Two sorters work with these lists as input data. The new sorters display the same behaviour as the parent. Synchronization is achieved by using the first argument with input mode. qsort processes can only progress when the partition process adds values to the shared variables S and L. Variables S, L and Sorted are shared variables for communication among processes. The second clause determines how to end the qsort process.

```prolog
mode quicksort(? , !).

quicksort(Unsorted, Sorted) <- qsort(Unsorted, Sorted, []).

mode qsort( ? , ! , ?).
```
qsrt([N|Tail], Sorted, Rest)←
partition (N, Tail, S, L),
qsrt(S, Sorted, [N|Sort1]),
qsrt(L, Sort1, Rest).
qsrt([], X, X).

To benchmark stream and parallelism using the quicksort algorithm, the executed query is \(\text{list}(1, N, L)\), \text{qsrt}(L,R)\) in every test. \(N\) denotes an integer. The list \(L\) with data to be sorted is produced by the list process, and has the form \([1, N, 2, N-1, 3, N-2, 4, N-3, \ldots]\). \(R\) denotes the sorted list \([1, 2, 3, 4, \ldots]\).

A very low execution time always restricts potential speedups, owing to processors with no work to do. A very large set of input data (e.g. increasing \(N\)) could produce undesirable garbage collections in small networks, but not in larger ones. Using the list described above, we obtain a high number of processes without excessively increasing the set of data to be stored.

Figures 4, 5, 6 and Table 2 show some results of the computation of the query with \(N = 900\) which generates 408,605 processes. The parallel version has been mapped onto five configurations: a single processor, two processors; and meshes of four, nine and sixteen processors, respectively.