Parallel CLP on Heterogeneous Networks

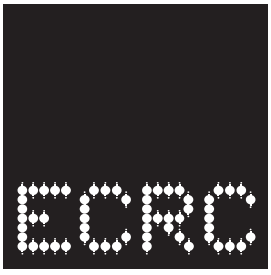
Shyam Mudambi
Joachim Schimpf

ECRC-94-17

technical report ECRC-94-17

Parallel CLP on Heterogeneous Networks

Shyam Mudambi
Joachim Schimpf



European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : Shyam Mudambi (mudambi@ecrc.de)
or
Joachim Schimpf (joachim@ecrc.de)

Abstract

The combination of Or-Parallelism and Constraint Logic Programming (CLP) has proven to be very effective in tackling large combinatorial problems in real-life applications. However, existing implementations have focused on shared-memory multiprocessors. In this paper, we investigate how we can efficiently implement Or-Parallel CLP languages on heterogeneous networks, where communication bandwidth is much lower and heterogeneity requires all communication to be in a machine-independent format. Since a recomputation-based system has the potential to solve these problems, we analyse the performance of a prototype using this approach. On a representative set of CLP programs we show that close to optimal speedups can be obtained on networks for programs generating large search spaces and that the overhead of recomputation is surprisingly low. We compare this approach with that of stack-copying and also discuss how side-effects can be dealt with during recomputation. The main conclusion of the paper is that *incremental* recomputation is a clean and efficient execution model for Or-Parallel CLP systems on heterogeneous networks.

Keywords: Parallel Logic Programming, CLP, Heterogeneous computing, Recomputation, Or-Parallelism.

1 Motivation

Networks of personal workstations are now ubiquitous and the idea of using such networks as high-performance compute servers has stimulated a great deal of interest. Often, these networks are heterogeneous, i.e. they link together machines that vary widely in terms of hardware and software. In this paper, we will investigate how we can efficiently implement Or-Parallel Constraint Logic Programming (CLP) languages on this type of hardware platform.

CLP is a generalization of logic programming where the basic operation of unification is replaced by constraint solving. CLP has been used to tackle a number of real-life combinatorial problems, where the basic paradigm is that of "constrain and generate". Since even after the constrain phase the remaining search space can be quite large, Or-Parallel CLP systems attempt to explore this remaining space in parallel. The combination of Or-Parallelism and CLP has first been suggested in [11]. Later the ElipSys system [12] which combines finite domain constraints with Or-parallelism has proven rather successful in practical applications.

However, existing implementations have focused on shared memory multiprocessors where communication is cheap and data structures can easily be shared. In a loosely coupled network of workstations, both assumptions are no longer true. First, any sharing of state between the processors has to be simulated by explicit communication, which is limited by the *bandwidth* of the interconnection network[1]. Second, a heterogeneous platform requires all communication to be done in a *machine independent* format, which adds conversion overhead and usually increases the amount of data to be transferred.

Our approach is to avoid all state sharing and instead rely on the *recomputation* of states[2, 10, 9]. This allows for both low bandwidth requirements and machine independency, thus solving the problems outlined above. We have implemented a prototype on a network of workstations based on the ECL^{i}PS^e CLP system developed at ECRC. The results of running a number of representative CLP programs on the prototype are very encouraging and indicate that such an approach is a clean and efficient way to implement Or-Parallel CLP systems on heterogeneous networks.

The rest of the paper is organized as follows: Section 2 introduces the Or-parallel task switching problem and describes how it is solved by the

[1] Recent work on optimizing a distributed version of the ElipSys system ([8]) has brought some progress in this respect. Nevertheless, the overheads associated with the use of virtual shared memory are still significant.

recomputation model and its competitors. Section 3 presents the prototype we have implemented and analyzes the benchmark results. Section 4 discusses some additional issues that will have to be tackled by a full implementation of such a programming environment, especially the problem of impure language features, and how to better exploit a network of multiprocessors. Section 5 concludes the paper.

2 The Task Switching Problem

An Or-Parallel system is one where alternatives of a nondeterministic program are explored in parallel. We can represent this as an Or-tree where the arcs represent deterministic computation and the nodes represent nondeterministic choices (figure 2.0.1).

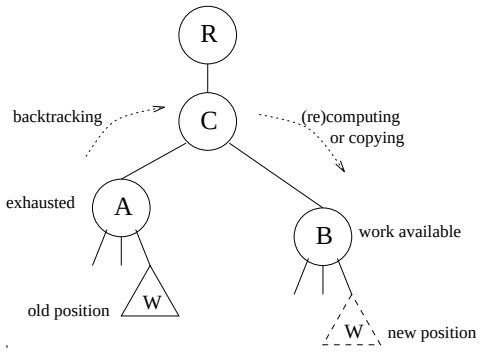


Figure 2.0.1: Moving a worker in the Or-tree

The processors, called *workers*, work on different parts of the tree in parallel. When a worker has explored a subtree (task) completely, it is assigned to another subtree. Since every node in the tree corresponds to a certain state of computation, taking a task from another node means the worker has to set up the corresponding state. Only then can the worker start exploring a subtree of this node. The main problem we investigate in this paper is how to efficiently achieve this state setup in a distributed environment.

In a WAM-based CLP system, the computation state is mainly represented by the (four) stacks[1] of the abstract machine. There are essentially three ways of setting up the computation state (i.e. the stacks): stack-sharing [7, 12], stack–copying[6] and recomputation [2, 9, 10].

The stack–sharing approach implies that all the stacks have to be potentially sharable and hence have to reside in (possibly virtually) shared memory. This direct sharing of a data structure is not viable in a heterogeneous setting, hence we will not discuss it further. Thus we are in particular interested in the comparison between the copying and the recomputation approaches, since in both these models each worker has its own copies of the stacks and so engine execution does not rely on shared data structures.

[1] For our purposes we can treat all stacks alike.

2.1 Stack-copying

In stack–copying, a state is set up by connecting to a worker that has a similar state and copying the stacks from there. The overhead consists in the actual transmission of the stack data, as well as in the fact that the sending worker has to interrupt its work while the copying is in progress. The amount of transmitted data is usually reduced by employing an *incremental* copying strategy. It consists of first identifying which old parts of the stacks the workers have already in common and then only copying the difference in the states of the two workers. Moreover, an appropriate scheduling strategy can also contribute to reduce the number of copying sessions by scheduling more than one node at a time. This approach has led to efficient Or-Parallel Prolog implementations on both UMA and NUMA shared memory machines [6].

Unfortunately, it is questionable whether a copying-based CLP system can be efficiently implemented on a network of workstations for the following reasons:

1. The copying overhead will be considerably higher due to much lower communication bandwidth and because CLP programs typically generate larger stacks than conventional Prolog programs.
2. Though each worker has its own copy of the WAM stacks, some global data (such as atom and functor tables) has to be maintained, which can have severe performance penalties in a distributed setting.
3. Lastly, in a heterogeneous setting, copying stacks is not straightforward, as they will have to be transmitted in some device-independent format.

2.2 Recomputation

Instead of copying a state of computation, it can as well be recomputed. This is trivial as long as things are deterministic[2]: One just starts with the same initial goal and does the computation again. If the program contains nondeterministic paths, then we must make sure we take the right one if we want to reach a certain state. This can be achieved by using *oracles*, i.e. by keeping track of nondeterministic choices during the original execution and by using this recorded information (the oracle) to guide the recomputation.

The idea of using oracles and recomputation for parallelism was first implemented in the DELPHI system [2]. Their results indicated that such an approach was well suited to exploiting Or-Parallelism in coarse-grain Prolog programs on networks of workstations. [10] also describes a recomputation based algorithm and its prototype implementation in Flat Concurrent Prolog. An example of a different use of the same kind of oracles can be found in [5].

[2] We ignore side effects for the moment.

A striking advantage of the recomputation model is its implementational simplicity:

- The workers can be completely separated and do not have to share any state.
- The workers communicate by exchanging simple data (mainly oracles, which are just sequences of integers).

2.3 Incremental Recomputation

The incrementality optimizations used in a stack-copying [6] or stack-sharing [7] systems to reduce the amount of data to be copied can also be used in a recomputation-based approach to reduce the amount of recomputation and communication. For example to move a worker from node A to node B (figure 2.0.1) a naive system would backtrack to the root node R and recompute the path from R to B. The incremental system will only backtrack to C and recompute from C to B. This is the same idea as in incremental stack copying. An incremental stack section corresponds to the incremental oracle which is needed to recompute it. Incremental recomputation has been successfully used in PARTHEO, a parallel theorem prover[9].

2.4 Comparison

The recomputation scheme has the potential to overcome both fundamental problems mentioned in the introduction: limited communication bandwidth and heterogeneity. Since oracles are a more compact representation of state than stacks, the size and volume of messages needed should be rather low. Secondly, since the workers do not share any state nor do they communicate internal representations of data structures (as is the case in stack copying), the recomputation scheme is suitable for heterogeneous networks.

Of course, recomputation can be computationally expensive for certain classes of programs. If we want to compare it with a stack copying scheme, we are interested in the difference between recomputing a piece of stack and copying it from another worker. Unfortunately, there is no fixed relation between the two. Reconstructing a piece of stack can be very expensive or quite cheap, depending on the program. A simple Prolog program can build up stacks at a rate of several Megabytes per second. On the other hand, a program may spend a lot of time while creating no (useful) stack data at all, e.g. computing a value of the Ackermann-function. Hence one of the issues we examined with our prototype were the incremental stack sizes generated by typical CLP programs.

The problem of global state and impure language features is often neglected when parallel declarative systems are discussed. With a stack copying approach, a solution to this problem would involve implementing some virtually shared data structures or a central manager for side effects. The recomputation idea, while holding out the promise to render this unnecessary, introduces new semantic problems that we discuss in section 4.2.

3 Results

The question we wanted to have answered was: Given similar scheduling strategies, would a recomputation-based CLP system be competitive with a copying-based system in a distributed setting? The main goal of implementing a prototype was therefore to gain a better understanding of the computational overhead of recomputation. Since the idea was to do it as quickly as possible, the prototype deals only with pure Prolog code and essentially returns all solutions to a particular query. The extensions necessary to deal with pruning and side effects are discussed later. First we describe the prototype implementation and then analyze the behaviour of six CLP programs and one plain Prolog program.

3.1 Prototype

Our results were obtained on a prototype which was implemented in three stages. First, the ECL^{i}PS^e WAM engine was extended to support recomputation, which is basically the ability to create and follow oracles. Secondly, the concept of parallel choicepoints was introduced, together with a flexible interface to control the parallel execution. This interface allows one to specify (via Prolog-coded event handlers) how the alternatives of a parallel choicepoint are executed. Lastly, on top of these features, a centralized scheduling scheme was developed in Prolog. The system was run on Sun IPC workstations connected via Ethernet.

3.1.1 Oracle Handling

The abstract machine has been extended for oracle handling. There are two modes of operation: in normal mode the machine executes as usual (figure 3.1.1) but builds up a record of which alternatives were taken at each choicepoint. In recomputation mode, the execution follows a given oracle (figure 3.1.2), i.e. the machine just deterministically takes the same alternatives that were successful while the oracle was recorded. This requires modifications of the WAM's choicepoint instructions (Try, Retry, Trust).

Oracles are implemented as a list on the global stack. The new ORC register points either to the end of the oracle list (during oracle recording) or to the current position in the oracle list (during oracle following). A small set of builtins is provided to manipulate oracles.

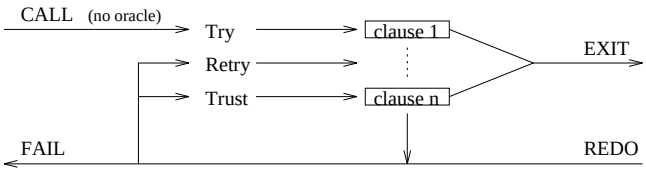


Figure 3.1.1: Backtracking in WAM

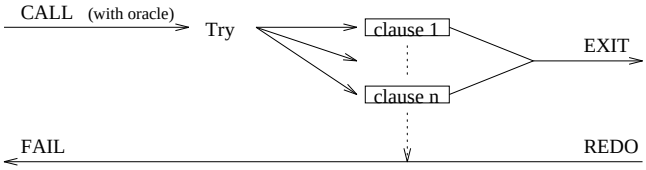


Figure 3.1.2: Oracle following in WAM

3.1.2 Scheduler Interface

To have a realistic prototype, it was necessary to introduce the distinction between sequential and parallel choicepoints. As in ElipSys[12], this is implemented by annotating a predicate as parallel, using the **parallel/1** declaration.

To make experiments with scheduling easy, we decided to provide a Prolog-level interface to the parallel choicepoints, rather than hard-wiring the scheduler interaction into the abstract machine. We have used the ECL^{i}PS^e event-mechanism to implement this, so that it is possible to control the parallel search with a Prolog-coded event handler. The following two events are associated with parallel choicepoints:

1. CREATING_PAR_CHP: This event is raised just before the parallel predicate is entered. The first argument to the event handler is the number of alternatives it is going to create. The handler now has the responsibility of managing these alternatives. Typically, some alternatives would be given to other workers, while the local worker takes one alternative itself. The handler forces the local worker to take a particular alternative by binding the second argument to a partial oracle. For example, in the following handler the local worker takes the first alternative.

```
create_par_chp(NumAlt, ContOracle) :-
    inform_scheduler(NumAlt),
    ContOracle = [1|_].
```

2. FAIL_TO_PAR_CHP: This event is raised when a worker fails to a parallel choicepoint. The first argument to the handler is the number of the failed alternative. The handler can either return a new oracle and force the worker to explore another alternative (by binding the second argument to the appropriate oracle) , or force it to fail across the parallel choicepoint (by not binding the second argument). Thus a simple handler would be:

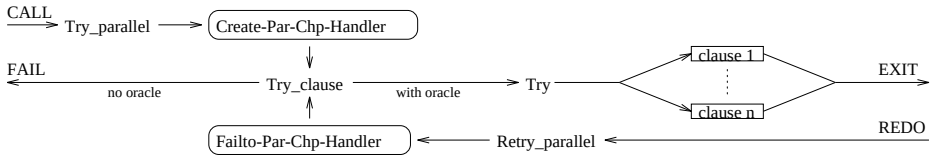


Figure 3.1.3: Parallel predicate implementation

```
fail_to_par_chp(FailedAlt, ContOracle) :-
     get_job_from_scheduler(ContOracle).
```

On the WAM level, this functionality has been implemented by prefixing the normal sequential choice instructions with the 3 new instructions Try_parallel, Retry_parallel and Try_clause (figure 3.1.3). The Try_parallel instruction creates a choicepoint, determines the number of alternatives, and raises the CREATING_PAR_CHP event. After the event handler returns, the Try_clause instruction is executed. If no oracle is given by the handler, it will cause the parallel predicate to fail. Otherwise it proceeds to the Try instruction which will just follow the oracle and execute a single clause deterministically. When the clause fails, the Retry_parallel instruction is executed, which raises the FAIL_TO_PAR_CHP event. After the event handler returns, Try_clause is executed as above.

3.1.3 The Scheduler

The Prolog implementation is made up of N ECL^{i}PS^e processes performing useful work (i.e. workers) and one ECL^{i}PS^e process which runs the central scheduler. As noted earlier, the scheduler code (in both the worker and central scheduler processes) is written in Prolog. The scheduler and the workers communicate with each other via sockets using normal Prolog reads and writeq's. The scheduler reads in a goal and returns all possible solutions to the goal. There is no special treatment of side effects, cut or commit. Thus the prototype can only be used to run "pure" programs or programs where the scope of the cut is limited to the "private" part of the stack (which is the section of the stack below the most recently created parallel choicepoint).

The worker processes read goals and oracles from the scheduler and run them. When a worker runs out of work, it informs the central scheduler. If the central scheduler has jobs, it sends a job (i.e. the corresponding oracle) to the idle worker. Otherwise, the scheduler process sends a request-work message to all busy workers. We have implemented a simple top-most scheduling strategy, where workers keep track of all parallel choicepoints seen so far and when requested for work, release the topmost (the one closest to the root) choicepoint to the scheduler. *Releasing* a choicepoint in this context means sending the scheduler the oracle (a list of integers) that leads up to the choicepoint. The workers poll for a request-work message each time either

parallel choicepoint handler is called (i.e. when a parallel choicepoint is created or failed back to).

We have implemented a partial incrementality optimization in the prototype, which eliminates repeatedly computing the initial deterministic path that leads to the first parallel choicepoint in the program. Thus when a worker runs out of work, it fails back to the first parallel choicepoint of the goal, rather than failing completely out of the goal, and starting all over again from the root.

3.2 Performance Analysis

We used seven programs to analyze the performance of the prototype.

1. N-queens - A naive pure Prolog n-queens program which has been used as a benchmark for most Or-Parallel Prolog systems[7, 6].
2. K-puzzle - A number puzzle which uses finite domain constraints.
3. Ncar1, Ncar2 - The car sequencing program first implemented in CHIP[3]. Ncar1 is the original program and ncar2 is the same program, except that the `atmost/3` predicate was recoded using `delay/2`, which had the effect of increasing the granularity of parallelism.
4. cbse1-14, cbse1-18 - The protein topology prediction benchmark from ICRF [1] with 14 and 18 strands.
5. qg2-8 - A finite algebra theorem proving benchmark[4].

3.2.1 Overall Benchmark Results

The sequential execution in our prototype is somewhat slower than in the original ECL^{i}PS^e system. There are two sources of overhead: oracle recording and the prototype's event mechanism for managing parallel choicepoints. Their effect is shown in table 3.2.1. The first column is the efficiency of the original ECL^{i}PS^e system, the second column shows the slowdown due to oracle recording only, and the last column shows the effect of both. Note that the latter is the speed of the actual parallel prototype operating sequentially with a single worker. The 9-queens benchmark turns out to be particularly sensitive to the overheads we have introduced. The reason is that the dominant operation in this benchmark is the creation of parallel choicepoints, and this operation is slowed down by an order of magnitude in the prototype due to the Prolog handler calls.

For an optimized implementation we would, however, expect efficiency close to the second column, since the event handler overhead would disappear with a tighter integration.

Goal	ECL^{i}PS^e	ECL^{i}PS^e with Oracles	Prototype
9-queens	1.0	0.88	0.42
k-puzzle	1.0	0.99	0.93
ncar1	1.0	0.97	0.85
ncar2	1.0	0.93	0.81
cbse1-14	1.0	0.94	0.69
cbse1-18	1.0	0.94	0.71
qg2-8	1.0	0.99	0.75

Table 3.2.1: Overhead of oracle recording and parallel choicepoint handling

The speedups obtained with up to 12 workers for each of the above programs is shown in table 3.2.2 below. We should note that though an extra process was used by the central scheduler for ease of programming, the cpu time used by this process was negligible. All the timings were performed on Sun IPC machines, though the memory configurations and work loads were not identical, hence there was slight variation in the sequential running times on the various machines. The parallel timings shown are the best of four runs and speedups were computed by comparing the best parallel run with the best sequential run. There was a big variance between runtimes (especially in the short runs) which was due to network load caused by other jobs. Since the average values reflect these variances which have nothing to do with our prototype, we chose to use the best runs for our performance analysis.

Goals	Workers				
	1	2	4	8	12
9-queens	69.7	35.0 (1.99)	17.9 (3.90)	9.3 (7.50)	6.6 (10.64)
k-puzzle	995.0	557.0 (1.79)	269.1 (3.70)	145.6 (6.83)	98.3 (10.12)
ncar1	22.7	12.7 (1.78)	7.1 (3.19)	4.9 (4.60)	4.2 (5.37)
ncar2	63.2	34.0 (1.86)	18.6 (3.41)	10.70 (5.91)	8.4 (7.57)
cbse1-14	45.6	24.0 (1.91)	13.7 (3.34)	8.9 (5.13)	7.6 (6.01)
cbse1-18	801.9	403.4 (1.99)	229.9 (3.49)	111.0 (7.22)	74.4 (10.79)
qg2-8	23244.5	11926.3 (1.95)	6247.2 (3.72)	3074.1 (7.56)	2165.9 (10.73)

Table 3.2.2: Elapsed times (in seconds) and speedups of the Benchmarks

As can be seen, the k-puzzle, 9-queens, cbse1-18 and qg2-8 speedups are quite good[1], whereas the ncar1, ncar2 and cbse1 speedups fall once the number of workers is above four. One of the main reasons for this drop off in speedups

[1]It should be noted that due to the variation in sequential runtimes on the machines and the fact that we chose to compute speedups using the best sequential times, perfect speedups were not possible.

for the shorter programs is the high startup time required for running jobs on a network. For example, it can take nearly a second before all the workers have tasks to work on. In a small job, a second is a significant portion of the runtime.

3.2.2 Parallel Execution Overheads

The main parallel execution overheads are due to recomputation and communication. A rough measure of the communication overhead can be obtained by examining the total number of oracles (jobs) sent out by the scheduler and their lengths. Table 3.2.3 provides these figures for the programs on the 12 worker runs. As we would expect the oracle lengths are quite small for the 9-queens query, implying that the communication overhead is quite low.

Goal	Jobs	Oracle Lengths	
		Min	Max
9-queens	229	1	20
k-puzzle	216	6	99
ncar1	135	1128	1374
ncar2	82	1146	1770
cbse1-14	302	329	525
cbse1-18	450	417	677
qg2-8	416	8678	8904

Table 3.2.3: Number of jobs and oracle lengths (12 workers)

However, the oracle lengths for the CLP program are much higher - over a thousand entries, in the case of the ncar and qg2-8 queries. If we assume that an oracle entry takes up one byte, the minimum amount of data that must be communicated for these programs is over 80 Kbytes.

We should note here that a naive implementation of oracles can lead to even larger lengths. For example, many CLP programs follow the general strategy of setting up the constraints and then exploring the constrained search space. Though setting up the constraints is usually deterministic, in practice the code creates choicepoints which are cut away later, but still fill up the oracle. A solution is not to record the oracle for a deterministic path but only record its length, since another worker recomputing this section does not need any more information. Thus an oracle would consist of alternative numbers, representing which alternative to take from the next choicepoint and lengths fields (if the path is deterministic). If a length field is seen, it is decremented each time a new choicepoint is created and once it is zero, the rest of the oracle is followed (as before). We have implemented this optimization in the prototype and preliminary results show that the oracle lengths are reduced by 10% to 90%. This is of course at the expense of some recomputation time (due to useless sequential backtracking).

Tables 3.2.4 and 3.2.5 show the percentage of elapsed time spent in various activities by all the workers for the cbse1-14 and k-puzzle queries respectively. We see that the idle times for the cbse1-14 query are quite high. The main reason for this is high network startup costs and the simple work release mechanism used in the prototype. The "other" row in these tables reflects overheads such as scheduling (reading in oracles and releasing work) and garbage collecting[2]. The scheduling overhead should decrease with tighter integration.

The most interesting statistic here is of course the percentage of time spent recomputing. This overhead is not very large when compared to the total elapsed time of the query (less than 10% for both queries), which is quite encouraging. One of the reasons why this overhead is so low, is the partial incrementality optimization, which eliminated repeatedly computing the initial deterministic prefix of a goal. Since this initial segment in CLP programs is used to set up the constraints, it can be quite time consuming. For example, without this optimization the time spent recomputing in the cbse1-14 program goes up from 8.5% to 25% for 12 workers. In addition, it also reduced the communication overhead significantly, since the oracles sent out by the scheduler could omit the initial deterministic prefix. The minimum oracle length column of table 3.2.3 shows the lengths of these prefixes.

Activity	Workers				
	1	2	4	8	12
Working	100.0	90.0	78.7	60.5	47.3
Idle	0.0	1.5	12.3	26.5	33.0
Recomputation	0.0	0.5	3.0	5.9	8.5
Other	0.0	8.0	6.0	7.1	11.2

Table 3.2.4: % Time spent in working and recomputing for cbse1-14

Activity	Workers				
	1	2	4	8	12
Working	100.0	88.9	92.0	85. 0	83.9
Idle	0.0	3.0	1.7	2.9	4.9
Recomputation	0.0	0.1	0.4	1.0	1.4
Other	0.0	8.0	5.9	11.1	9.8

Table 3.2.5: % Time spent in working and recomputing for k-puzzle

Table 3.2.6 gives the recomputation percentages of all the queries analysed in this paper. With the exception of the ncar1 program, all the other times are

[2]Because oracle following does not create any choicepoints, the incremental garbage collector ends up repeatedly scanning such stack sections. Hence the gc time in the parallel runs was higher than the sequential gc times.

below 10%. The relatively larger percentage of time spent recomputing in the ncar1 program is due to the presence of fine-grain parallelism deep in the search tree. A more complete incrementality optimization, where a worker is given work which requires the least amount of recomputation to reach relative to its current position should allow the system to better exploit such parallelism.

Workers	2	4	8	12
queen9	0.1	0.2	0.7	1.0
k-puzzle	0.1	0.4	1.0	1.4
ncar1	2.1	6.0	17.8	16.3
ncar2	1.4	4.1	8.9	9.2
cbse1-14	0.5	3.0	5.9	8.5
cbse1-18	0.0	0.3	1.2	1.7

Table 3.2.6: Recomputation overhead of all benchmarks

3.2.3 Oracle Lengths versus Stack sizes

As we stated earlier, one of the goals of this investigation was to compare the merits of copying the state versus recomputing it. We have already seen that the oracle lengths for CLP programs are quite large, thus increasing the communication overhead. In order to see how this overhead would compare with the stack copying approach, we computed the ratio of the incremental stack sizes to the incremental oracle lengths. The maximum and minimum ratios are shown below for the benchmarks. We computed the ratio in two modes - one with the garbage collector turned off and the other in which garbage collection was forced at every parallel choice point (just before the incremental stack changes were computed). The forced garbage collection figures gives us the best case scenario for stack copying and the no garbage collection gives us close to the worst-case[3]. The ratios again seem to be quite encouraging for the recomputation scheme - the lowest ratio is 40, which means that even in the best case one oracle entry corresponds to 40 bytes of stack.

Of course, using recomputation one has the additional overhead of recomputing, but we have shown this overhead to be quite small for these programs. The actual incremental stack size data for the programs show that the stacks can get quite large for the four CLP queries(see Table 3.2.7). For the forced gc runs, the largest incremental changes were seen between the root and the creation of the first parallel choicepoint. This is what we expected to

[3]The worst-case figure would actually be the sum of these two figures, since it could be the case that a worker X shares its state with another worker Y just before performing garbage collection. X then performs garbage collection immediately afterwards. Y then requests work from X again and this time has to copy the garbage collected state.

	Stack size (bytes)			Ratio to oracle length		
Goal	Average	Min	Max	Average	Min	Max
queen9 nogc	711	376	2164	100	71	376
queen9 gc	306	224	352	43	35	256
k-puzzle nogc	11321	384	35808	3046	128	13398
k-puzzle gc	6276	100	26092	1689	33	5218
ncar1 nogc	19554	448	135252	848	120	17228
ncar1 gc	5877	196	65400	255	49	4642
ncar2 nogc	19756	432	132408	295	93	18802
ncar2 gc	6612	100	65688	99	40	4626
cbse1-14 nogc	4114	880	66848	341	73	3323
cbse1-14 gc	1241	324	31676	103	30	984
cbse1-18 nogc	4778	880	97792	398	73	4192
cbse1-18 gc	1240	324	44284	103	30	1277
qg2-8 nogc	105776	8148	1119652	9178	129	71708
qg2-8 gc	30393	228	735388	2637	85	7538

Table 3.2.7: Incremental Stack sizes compared to oracle lengths

see as it reflects the space used to set up the initial constraints. However, when no garbage collection was performed, this was not always the case. As the figures indicate, there is a wide variation between the gc and no gc runs. We see that for the CLP programs between 1 to 30 K-Bytes of stack has to be copied per task (when garbage collection is turned on). Unfortunately, it is difficult to estimate the frequency of these task migrations, without simulating parallel execution.

4 Full System

For a scalable, real implementation of a recomputation system, a number of optimizations are necessary, as well as support for impure language constructs.

4.1 Optimizations

A centralized scheduler would become a bottleneck when running on a large number of processors. Thus a distributed scheduler should be used, consisting of sub-schedulers each of which is only responsible for a subset of the parallel choice points. When moving from a centralized scheme to a distributed one, the main difference will be that the message volume will increase, since a request-work message might have to traverse all the busy workers before work is found. This increase is, however, not specific to the recomputation model. In fact it will be seen on every distributed Or-Parallel system.

The prototype used a simple site-based topmost scheduling policy out of necessity, since it did not keep any representation of the search tree. The results show that a topmost strategy works sufficiently well for programs with large grain parallel jobs. Though simple, such a strategy does not allow one to easily exploit incrementality optimizations. In a real implementation, it would be preferable to use a tree-based scheduling strategy which will try to match idle workers with the closest possible task. “Closeness” would be measured in terms of the amount of recomputation necessary.

4.2 Impure Language Features

Impure language features cause problems for all parallel implementations of Prolog. We will distinguish three different classes: pruning operators (cut and commit), side effects affecting the state of the environment (e.g. file system) and side effects affecting the internal state of the Prolog system (e.g. assert, record)

4.2.1 Pruning Operators

The implementation of the pruning operators in the recomputation system is basically the same for as for other Or-Parallel models (the scheduler handles

pruning that affects parallel choicepoints). During recomputation, the cuts can just be ignored. We will therefore not consider this topic further.

4.2.2 Environment side effects

When, during recomputation, the system encounters a side effect predicate of the write-type, then execution of this predicate would duplicate the side effect (e.g. writing to a file) because it has already been done in the original computation. This can be relatively easily eliminated by suppressing this kind of side effects whenever a worker is in re-execution mode.

On the other hand, side-effects of the read-type need to be re-executed (because they have to return a result). But the environment state may not be the same as it was during the original execution. Non–critical examples are e.g. re-execution of compile-predicates, since the source files are not expected to change. But in general, read-type side effects may yield different results on re-execution, consider for instance a call to cputime/1 or a read/2 on a file that has been changed in the meantime. The solution is to record the results of such predicates together with the oracle during the original execution. On re-execution, the predicate is not executed, but the recorded result is used instead. This amounts to an extended notion of oracle: It predicts not only the correct branch of a nondeterminate choice, it also predicts the results of certain built-in calls. The cost of such a solution is a larger oracle.

4.2.3 Internal side effects

For internal side effects there are two sources of problems: the lack of shared state between distributed workers, and the repeated re-execution of side effect predicates during recomputation.

As long as internal side effects are not used to communicate between Or-branches, they pose no problems. They are just re-executed on recomputation and can be treated like pure logical code. The re-execution is in fact necessary because the internal state is not shared between workers and therefore has to be re-established.

The main problem is when such predicates are used to communicate between Or-branches (or in sequential terms: communicate across failures). There are three possible semantics for these predicates in an Or-Parallel setting and there is a gradual degradation of what one can do in terms of communication between Or-branches using this type of side-effects in the three different models:

1. sequential — the sequential Prolog semantics (imposing an order on the Or-branches)

2. parallel-shared — global data is (physically or virtually) shared, but no special precautions to preserve the sequential order of side effects

3. parallel-private — no data shared between Or-branches

We will not consider the first model further, since we believe that the ordering of Or-branches is an artifact from sequential implementations and should not be imposed on a parallel system. We note however, that it is default semantics implemented in [7, 6].

The parallel-shared model is implemented in ElipSys and behaves such that side effects executed in Or-Parallel branches are interleaved in some unspecified order which may be different in different runs.

In the parallel-private model without any shared state there is the additional effect that a side effect which is done in one Or-branch may never become visible in another Or-branch (when they are computed in parallel by different workers).

The simulation of the parallel-sharing semantics on a non-sharing recomputation model is quite expensive. It would be necessary to have a global manager to synchronize access to all dynamic predicates and recorded structures. This would be costly and remove the main attraction of a recomputation-based system, which is its simplicity. This problem is closely related to the issue of implementing a distributed dictionary in a copying-based scheme.

The other way is to make it impossible to use the internal side-effect primitives of sequential Prolog for Or-branch communication. The most pragmatic method would be to keep the current implementation and restrict their use to purely sequential parts of the search space.

To compensate for the lost feature of communicating information over Or-branches, we envisage a language construct that associates a non-logical object (a *bag*) with an Or-subtree. Inside this subtree, terms can be copied into this bag. The content is only retrieved once the subtree is exhausted. The content is the result of the subgoal and is also recorded in the extended oracle. On recomputation, the subtree is not searched again, but the recorded bag is taken from the oracle instead. An all-solutions predicate could then be written as:

```
findall(Goal, Bag) :-
    call_with_bag((Goal, bag_enter(Handle,Goal)), Handle, Bag).
```

The differences compared to the classical primitives are: The concept of a bag implies that the order of entries is unspecified, which maps naturally onto the parallel implementation. The handle-concept instead of global names simplifies

disposal of the object on failure and garbage collection. Linking the bag to an Or-subtree solves the recomputation problem.

5 Hybrid Model

A realistic scenario of a common computing environment in the near future is a network of workstations where some or all workstations are multiprocessors with a small number (2 – 8) of CPUs. Using a pure recomputation model on such a network would result in inefficient use of the multiprocessor workstations, since the basic tenet of recomputation is that it is cheaper to recompute than to communicate. As this will probably not hold on a shared-memory multiprocessor, a hybrid copying/recomputation scheme would be ideal in such a setting. The idea is illustrated in figure 5.0.1. The dashed line around the engine processes on the multiprocessor indicates that they share some state (such as global dictionaries).

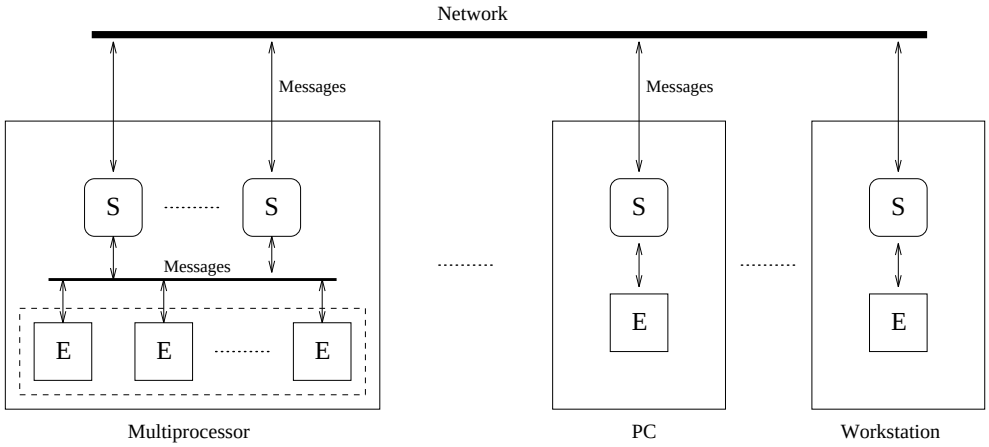


Figure 5.0.1: Hybrid Model

Such a model would use stack-copying when sharing work between processes on the same machine and use oracles across machines. The main issue in such a system would be the complexity of the scheduler, which would have to decide on the fly whether stack-copying or recomputation is the most appropriate work installation mechanism. The obvious advantage of such a system is that it would be able to exploit fine-grain parallelism within a cluster.

6 Conclusions

The main question investigated in this paper was whether an incremental recomputation-based scheme would be as efficient as a copying based scheme for implementing a CLP system in a distributed, heterogeneous setting.

Towards this end we implemented a recomputation-based prototype and analysed its performance on a set of representative CLP programs. The analysis revealed that the overhead of recomputation was between 1% and 18% per worker, which was surprisingly low, given that only partial incrementality optimizations were implemented in the prototype. As expected, this figure is higher than than the 3% to 9% overhead of copying reported for incremental stack-copying implemented on a shared-memory multiprocessor[6]. However, we also found that even the *incremental* stack sizes generated by the CLP queries were quite large and in fact even in the best case, a single oracle entry corresponded to 40 bytes of stack, while for some programs the average was well over a thousand bytes. Given these two results - reasonable recomputation overhead and the much lower bandwidth requirements of incremental recomputation, our conclusion is that such an approach will be as or more efficient than copying stacks in a distributed setting. In addition, since oracles are just sequences of integers, device independent communication is simple to implement (in contrast to stack-copying), making it easy to exploit heterogeneous networks.

Additional incrementality optimizations should decrease the recomputation overhead further and widen the class of problems that can be effectively parallelised using such an approach. In order to take advantage of cluster-based architectures (e.g. a network of multiprocessors) a hybrid model would be ideal. We are currently in the process of designing and implementing such a system.

Acknowledgements

Liang-Liang Li, Kees Schuerman and Alexander Herold's insightful comments on earlier versions of this paper were very helpful. Special thanks are due to Peter Kacsuk who initiated the idea of using a recomputation-based scheme for Parallel-ECL^{i}PS^e . This research was partially supported by the CEC under ESPRIT III project 6708 "APPLAUSE".

Bibliography

[1] D. Clark, C. Rawlings, J. Shirazi, Liang liang Li, Mike Reeve, Kees Schuermann, and Andre Veron. Solving large combinatorial problems in molecular biology using the ElipSys parallel constraint logic programming system. *Computer Journal*, 36(4), 1993.

[2] W.F. Clocksin. The DelPhi Multiprocessor Inference Machine. In *Proc. of the 4th U.K. Conf. on Logic Prog.*, pages 189–198, 1992.

[3] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in Constraint Logic Programming. Technical Report TR-LP-32, ECRC, 1988.

[4] M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proc. of IJCAI'93*, volume 1, pages 52–57, 1993.

[5] Pascal Van Hentenryck. Incremental constraint satisfaction in logic programming. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 189–202, Jerusalem, 1990. The MIT Press.

[6] Roland Karlsson. *A high performance OR-Parallel Prolog system.* SICS Dissertation Series 07, Kista, Sweden, 1992.

[7] Ewing Lusk, Ralph Butler, Terence Disz, Robert Olson, Ross Overbeek, Rick Stevens, D.H.D Warren, Alan Calderwood, Peter Szerdi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumil Hausman. The Aurora Or-Parallel Prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[8] Kees Schuerman and Liang-Liang Li. Tackling false sharing in a parallel logic programming system. In *Submitted to the International Workshop on Scalable Shared Memory Systems*, 1994.

[9] J. Schumann and R. Letz. PARTHEO : A high-performance parallel theorem prover. In *Proc. of CADE'90*, pages 40–56. MIT press, 1990.

[10] Ehud Shapiro. Or-Parallel Prolog in Flat Concurrent Prolog. *Journal of Logic Programming*, 6(3):243–267, 1989.

[11] P. Van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of chip within PEPSys. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 165–180, Lisbon, 1989. The MIT Press.

[12] A. Veron, K. Schuerman, M. Reeve, and L. Li. Why and how in the ElipSys Or-Parallel CLP system. In *Proc. of Parle '93*, pages 291–302. Springer-Verlag, 1993.