Design of A Distributed Scheduler for A Parallel CLP System

Liang-Liang Li
European Computer-Industry Research Centre
Arabellastrasse 17, 81925 Munich, Germany
Email: lll@ecrc.de

Abstract: ECL^{i}PS^e is an advanced constraint logic programming system. It is being extended to exploit Or-parallelism. Multiple sequential ECL^{i}PS^e engines are employed to jointly execute an application. The parallel ECL^{i}PS^e scheduler *coordinates* the concurrent activities of individual engines to comply with the language semantics, and *distributes* Or-parallel job loads among the engines for high performance. Its design has been tailored to having parallel ECL^{i}PS^e run efficiently on a wide spectrum of computer systems, from multiprocessors with shared memory to a set of networked such systems. The scheduler framework is a distributed scheduler tree which profiles the dynamic execution status and parallel load distribution of an application. The scheduling functions are cooperatively achieved via individual tree components which communicate with each other through message passing. Parallel backtrack and cut operations are associated to the pruning and transplant actions of the scheduler tree, and job-hunting is carried out via messages flowing around within the scheduler tree. Other novelties of the design include a clean interface between the scheduler and an engine, minimal modifications imposed on sequential engines for adapting to Or-parallelism, and in particular, a basic mechanism, *reincarnation*, to facilitate an engine to migrate nicely within a parallel application.

Keywords: Logic programming, Or-parallelism, scheduler, message passing

1 Introduction

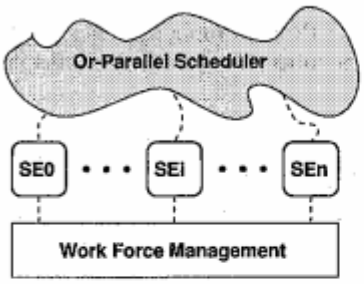

Figure 1: Parallel ECL^{i}PS^e System

Real-life combinatorial problems make up an important area of computer applications, and have typically huge search spaces. Using constraint logic programming (CLP) techniques can drastically reduce the search spaces *a priori*. Or-parallelism is able to concurrently search nondeterministic paths via available computer resources. Integrating the two into one system has proven profitable [18, 14]. ECL^{i}PS^e is an advanced CLP systemECL^{i}PS^e [11, 12] providing several constraint solvers (e.g. finite domains, rationals and generalized propagations). Its augmentation to support Or-parallelism is being undertaken [16]. Parallel ECL^{i}PS^e has a wide spectrum of targeted platforms, from multiprocessor systems with shared memory to distributed memory platforms (say a set of networked workstations).

Parallel ECL^{i}PS^e consists of three major components, as illustrated in figure 1, and an underlying layer which provides platform-independent parallel programming facilities, e.g. shared memory and message passing libraries.

The *Worker Management* maintains a process(or) pool, and supports dynamic addition and reduction of the working forces; and the *SEs* are a set of sequential ECL^{i}PS^e engines. The component *Scheduler*, the topic of this paper, has two main functions. One is to coordinate the concurrent activities of multiple engines to comply with the language semantics. More concretely, the cut operations and backtrack mechanisms advocated in the language, for instance, will necessarily involve interactions among multiple engines in a concurrent setting. The second function is to schedule parallel jobs among working forces available, so that the applications can be more efficiently executed with multiple engines running concurrently.

We present in this paper the framework design of a distributed scheduler for parallel ECL^{i}PS^e . By *distributed* we mean that the scheduler will not rely on the availability of shared memory, in contrast to those in some existing systems [15, 2], as we want to tailor the design for the diversity of the targeted running environments of ECL^{i}PS^e . Instead the scheduler is built exclusively on the concept of message passing: it is composed of numerous small independent agents which cooperate with each other only via messages. Therefore we are able to decompose the typically complicated scheduling functions into simple message handlers. Another design criterion is that the scheduler should be cleanly separated from the engines, and the two should interact with each other only through well-defined interface services. In particular, the adaptation modifications imposed on sequential engines should be kept minimal.

Our design is centered around the buildup and reshape of a distributed scheduler tree which profiles the dynamic execution status and parallel load distribution of a CLP application. The leaves and nodes comprising the scheduler tree act as active scheduler agents. This paper proceeds as follows. Section 2 will first brief the sequential ECL^{i}PS^e engine concept. Then it presents the scheduler tree, its components, and their functions and interactions. In particular, we examine how the scheduler interfaces with engines. Section 3 presents how the scheduler tree expands by taking over and dispatching parallel loads. A special mechanism (leaf-reincarnation) is introduced in section 4. This mechanism facilitates flexible engine migrations within a parallel application. With respects to the scheduler tree, it enables clean and timely subtree withering. As two concrete examples of making use of this mechanism, we examine in section 6 and section 5 how the major logic programming control mechanisms, backtrack and cut operations are handled in this proposed framework. Section 7 then details some other practical considerations, including anatomies of scheduler tree and message traffic, garbage collection issues, subtree state transitions, and a listing of services interfacing the scheduler and the engines. Section 8 deals with the support to serialise Prolog operations with side-effects (e.g. sequential predicates, Prolog cut, I/O, etc.). The job-scheduling is an important part of the scheduler design. Section 9 examines how this task can be carried out through message traffic among the distributed tree components. We will show that the framework provides an appropriate arena to achieve versatile job-scheduling strategies. Section 10 concludes the paper with the current status and planned work for parallel ECL^{i}PS^e .

In the sequel, we will treat ECL^{i}PS^e as a conventional Prolog system. We can do this because its CLP part is orthogonal to that of Or-parallelism. Familiarity with logic programming [13] in general and some Or-parallel systems [15, 2, 18] in particular is assumed. Another important topic for Or-parallel systems is job installation, which involves setting up a proper state for an engine so that it can receive and execute an alternative scheduled remotely. There have been mainly three schemes available, i.e. stack sharing (plus binding array) [20, 15, 18], stack copying [2], and recomputation (plus oracle copying) [9, 16]. Both of the latter two have been used for the parallel ECL^{i}PS^e to cope with potential platforms with hybrid (shared and/or distributed) memory architectures. This paper does not cover the installation details. The scheduler framework presented here is however indifferent to which of the two schemes is employed.

2 Scheduler Tree and Engines

2.1 Logic Programs and Or-Parallelism

A logic program comprises a set of predicate definitions and a query, and a predicate can have alternative definition clauses [13]. To resolve a query, first select a subgoal (a literal) out of the query, find a unifiable clause from the predicate definitions, unify them, and generate a new query (resolvent). This process repeats until an empty resolvent is obtained. To describe this process, upside-down *search trees* have been popularly used as a

picturesque formalism. A logic predicate with alternative definition clauses corresponds to a tree node with multiple branches. A path of a search tree is a link of adjacent branches starting from the root downwards. It is the tip-end of a path that tells if the path is with or without a solution (variable bindings along the path). The execution of a logic program can thus be regarded as a process of traversing the search tree for solutions. Below listed is a simple logic program which is to find descendents of *astrid*, given the definitions of predicates *ancestor/2*, *parent/2*. Its corresponding search tree is depicted in figure 2.

```
?- ancestor(astrid,D).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
parent(astrid,bruce).
parent(astrid,bob).
parent(bob,carmen).
parent(bob,chris).
parent(cindy,dan).
```

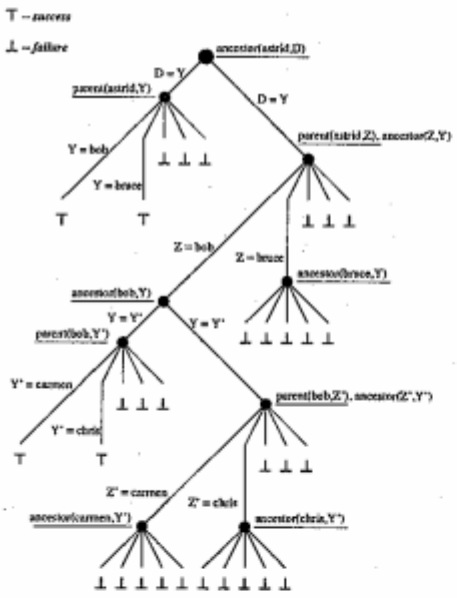

Figure 2: Logic Program and Search Tree

Prolog employs a depth-first strategy to sequentially traverse the search tree, supported with a *backtrack* mechanism. As a *defacto* standard Prolog machine, WAM (Warren's Abstract Machine) [19] introduces an *ad hoc* control stack to administer the search tree. What recarded in the stack are actually the tree nodes (or *choice points* in the WAM' terms) encountered along a path in question. A *choice point* maintains the remaining alternative paths and the necessary state to try the alternatives on backtracking. An ECL^{i}PS^e engine is an extended WAM. Shown in figure 3 is a simplistic picture of an ECL^{i}PS^e engine. The vertical line specifies the path currently being searched, and the circles embedded in the line are choice points.

Figure 3: ECL^{i}PS^e engine: A Simplistic View

The Or-parallelism stems from the fact that some choice points, called *parallel choice points*, contain alternatives which can be tried concurrently so that one or more solutions can be found more quickly (Whether a choice point is created parallel relies on the properties of programs executed. We assume some sort of parallel annotation is either user supplied or generated). The parallel ECL^{i}PS^e approach is to use multiple engines to concurrently exploit as many parallel alternatives as possible. At any time an individual engine searches in a separate part (i.e. a subtree of the search tree) exactly as a Prolog engine (i.e. using a depth-first plus backtrack strategy). It interacts with a coordinating component of the system when it finishes the subtree, when it is requested by other engines to give up its execution, or when other engines want to share its subtree. We have called this coordinating component of the system *scheduler*.

2.2 Scheduler Structure

As a coordinator on the concurrent activities of multiple engines, the scheduler must have a clear view about the current status of the search tree of an application: which parts have been, are, and will be traversed. In particular, it needs to administer parallel alternatives for job-scheduling. Therefore, the scheduler data structure should be a neat profile of the search tree and load

distribution.

We call this profiling structure a *scheduler tree*. It consists of *nodes* and *branches* (Trees, nodes, branches, paths and etc. hereafter refer to the scheduler tree, unless otherwise explicitly stated). Figure 4 shows a scheduler tree, together with several engines. The small circles on engine bars are the parallel choice points which have been made known (or *published*) to the scheduler, while other choice points, sequential or non-published parallel ones, are not shown as they are not relevant to the scheduler.

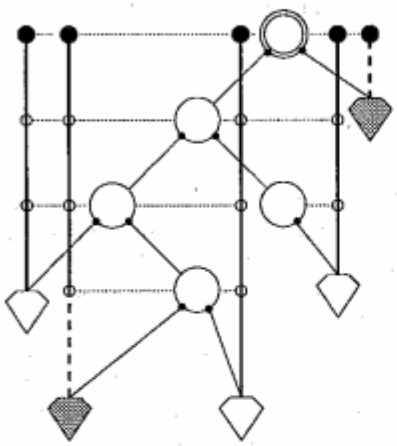

Figure 4: Scheduler Tree and Multiple Engines

2.3 Nodes, Leaves, and Engines

A node of the scheduler tree corresponds to a published parallel choice point. There can exist multiple instances of a published parallel choice point. It happens when, e.g. two paths followed by two engines share a common section starting from the search tree root, and conceptually every engine has a whole copy of the path of its own. Multiple instances of a published choice point are associated to the same node by maintaining a handle on to the node, shown as dotted horizontal connections in the figure (In the oracle based installation scheme, the node association is directly to the corresponding oracle entry). Note, the multiple instances are conceptually symmetric. The engine which initially creates the choice point is not distinct from others holding an instance (or a copy) of the choice point. Two nodes associated to two adjacent published choice points are said to have a parent-child relation, and are linked with a branch. The node associated to the older of the two choice points is the parent, the other the child.

Tip nodes, termed as *leaves*, of the scheduler tree are distinct from other nodes. Leaves serve as the interface between the scheduler and the engines. In fact, an engine and a leaf are the dual faces of a single entity (say a UNIX process [3] or a MACH thread [17]). From the scheduler's viewpoint, a leaf denotes an engine, and is the real working force to sprout the scheduler tree (detailed in section 3) or to reshape the tree. On the other hand, an engine resorts to its denoting leaf for any scheduler services. More concretely, let's call the youngest published choice point of an engine *border choice point* or simply the border, which associates to the parent node of the leaf. As has been said, an engine behaves as a normal Prolog engine. But it does so only in an area delimited by its border choice point. The rest, from the border up to the root, is shadowed by the scheduler. To manipulate the shadowed area, for sake of e.g. backtracking and cut operations, the engine has to resort to the scheduler for instructions. Section 5 and 6 will elaborate these topics. The concrete interactions between a leaf and its engine are through a handful of function calls (section 7.6).

2.4 Job-Scheduling

A node maintains a list of untried alternatives on behalf of the associated choice point. When an alternative is scheduled to an engine, a branch is created to connect the node and the leaf denoting the engine.

A leaf can be alive or *lodged* (gray colored in figure 4), denoting an active engine or an idle engine, respectively. We further call the parent of a leaf the leaf's *lodging node*, and call the path from the parent up to the root the leaf's *lodging path*. A leaf, either alive or lodged, denotes, with respect to job-scheduling, an engine state which is needed to execute any alternative scheduled from the ancestor nodes on its lodging path. A node holds only published alternatives which can be as simple as integer-valued indices, and it does not hold any engine state. This observation is crucial, as it influences fundamentally the backtrack mechanism and the job-scheduler design (to be elaborated in section 9).

2.5 Active Nodes & Message Passing

The scheduler tree are to be concurrently expanded and reshaped by multiple engines (or leaves). In order that the interactions among multiple engines are clean and regulated, we decide to use *message passing* as the basic and sole vehicle underlying these interactions. As a consequence, nodes can be implemented as *active objects* with their own private data, in much the same way as leaves (engines). Through message passing, nodes and leaves jointly fulfill the scheduler tasks.

In practice, the nodes can be independent computing entities, say operating system processes or threads, as well as (a)synchronous event-handlers of the related engine/leaf entities.

2.5.1 Pros & Cons

Below listed are some other motivations to use conceptually distributed tree representation and message passing.

- Avoid any potential abuse of shared memory (if the shared memory ever exists), while optimization based on locality of nodes and leaves can be easily added;
- Achieve a very clean scheduler design. As a matter of fact, the scheduler functions are divided and isolated into simpler message handlers of nodes and leaves.

There are disadvantages, of course. Job-scheduling, for example, will suffer from some typical problems encountered in a distributed decision-making system, e.g. *uncertainty* due to out-of-date information, *incompleteness* due to limited information, and *instability* due to mismatch of the parallel applications and the underlying hardware. A significant research effort will have to be devoted to such topics as how related information should be distributed, updated, and knowledgeably used in concurrent scheduling.

2.5.2 Message Flows

Message traffic exists mostly between parent and child nodes. A leaf can also initiate messages directly to any of its ancestors, as well as to its parent because the handles on to these ancestors are available in the published choice points of its engine. Messages flow via branches. A node can have many branches, each leading a subtree. We call a branch and its subtree via which a message flows to the node the *receiving branch* and *receiving subtree*.

2.6 Glossary

Nodes and leaves are the scheduler components. Related concepts are branches, children, parents and ancestors; lodging nodes and paths; receiving branches and receiving subtrees; and border choice points. A further discussion of the scheduler tree is staged in section 7.

3 Publish Job

Parallel choice points are created by individual engines. Before their alternatives can be scheduled to remote engines, they need to be first transferred to the scheduler discourse. It is also termed as *publishing* from the viewpoint of the engines in question. Publishing is choice point based. An engine publishes, actively or on request from its leaf, one or more of its parallel choice points. Figure 5 shows an engine has just published two consecutive parallel choice points And figure 4 shows the tree structure before the publish happens. Note, for simplicity, only the engine in question is shown, others are not. This is also the case in the rest of the paper.

Publishing proceeds as follows. The engine selects a parallel choice point (which should be younger than the current border choice point), grabs all of its remaining alternatives, asks the scheduler to create a node to administer them, and gets back a handle on to the node. The choice point just published keeps the handle, and becomes the new border.

In response, the leaf, instead of creating a new node, turns itself into the node, and creates a new leaf with the node as its parent. Thus this operation is kept completely local to the leaf, so that the previous parent needs not concern itself with its children's evolution (say, from a tip into a subtree). Note that initially the scheduler consists of only a root node, with all leaves except one lodged to it. The engine of that solely alive leaf starts the execution of applications. The scheduler tree then expands owing to job-publishing and job-distributing.

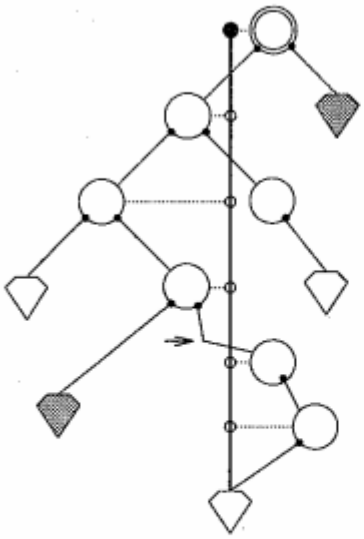

Figure 5: Publish Parallel Alternatives

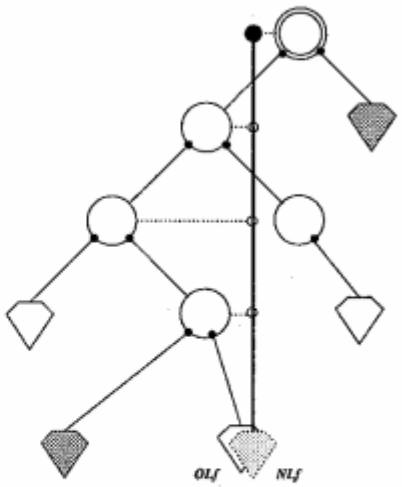

Figure 6: Before an Engine Migrates

4 Migrate Engine

When an engine finishes (or fails) its current job, it needs to migrate within the parallel application to get another job. For its leaf, it means a *transplant* within the scheduler tree, from its old parent to a new one. We now study the leaf transplant as a more general mechanism than as an operation only for job-scheduling.

A typical leaf transplant consists roughly of disconnection from the leaf's current parent, and connection with its new parent. To avoid the leaf messing up its two conflicting roles, we propose the following scheme. The leaf in question creates a *skeleton* for a new leaf, associates it to the engine, and initiates to the new parent a connection request for the skeleton, and sets itself *dying*. When the skeleton receives a connection reply, it becomes a real leaf. It also decouples it from the old leaf. In the meantime, the dying leaf, with the engine decoupled, passively handles any pending messages dedicated to it via the old connection, and eventually become withered. By *passively* we mean that the leaf simply absorbs all the messages which do not expect any reply, and bounces back reply-requesting messages. The essential point of this scheme is that as soon as the leaf (and its engine) decides to transplant, it shrugs off its duty (e.g. job-installations) and concentrates on adapting itself to the new role within the scheduler tree. Figure 6 illustrates an engine is about to migrate. The engine is detached from the old leaf (*OLf*), and associated to a leaf skeleton (*NLf*) which is waiting for a connection reply from the new parent. Section 5 and 6 will further illustrate engine migrations with two concrete examples.

We call this scheme *reincarnation*, as the old leaf actually gives away its own life to a new leaf. It is well in compliance to the concept that engines are immortal with respects to the scheduler, and there should be as many alive (or lodged) leaves as engines [1].

5 Handle Backtrack

An engine handles backtrack as in a sequential system as long as it does not cross over its border choice point. We focus therefore on the border crossing backtracks. A seemingly simple approach is that the engine in question stops, its leaf sends a terminating message to its parent and the couple enter into a job-searching session (see section 9). However, there is a constraint.

[1] Engine mortality is dealt with at the worker management layer. To interface with flexible and dynamic worker management is also an important part of the scheduler design, but not covered in this paper

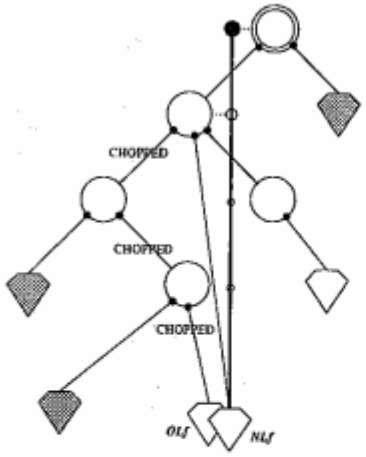

Figure 7: Backtrack: Resume Seq. Choice Points

5.1 Resume Sequential Choice Points

Sequential choice points shadowed behind the border need to be resumed by some engine. A simplified solution is the exhausted node entrust the last backtracking leaf to take it over. The procedure is as follows. A backtrack message is initiated by the leaf to the parent, and forwarded to the ancestors. This message flow stops at an ancestor (or the parent) node if the node is not exhausted or it has more than one alive branch. In this way, a maximal and single-threaded exhaustion subtree can be found. The ancestor then replies to the leaf to let its engine be responsible for all sequential choice points younger than the published one associated to the replying node (which actually becomes the leaf's new parent). And in the meantime, it chops the exhaustion subtree. On receiving this reply, the leaf will ask the engine to undo the published choice points (up to the one associated to the replying node) to become exhausted sequential ones, and resume backtrack.

Recall the reincarnation-based leaf transplant mechanism introduced in section 4. Therefore a new leaf *NLf* is created to wait for the reply, while the old leaf *OLf*, detached from the engine, still maintains the old connection. As a consequence, the exhausted subtree is replaced with a short-cut one. Figure 7 illustrates this outcome. Of course, if the old parent node is not exhausted, it can immediately reply to the NLf with an alternative, which will be elaborated in the job-scheduling section (section 9). In either of the cases, the receiving subtree is chopped.

6 Handle Parallel Cut

Cut is an important operator commonly seen in logic programming languages. It is to give users a way to interfere with the runtime search process by pruning some part of the search tree. The parallel ECL^{i}PS^e is a so-called *cavalier commit* which is both *symmetric* and *exclusive* (see section 8 for sequential cut handling). For example, assume a goal *a/0* defined as below is executed in a concurrent setting.

```
:- parallel a/0.
 a :- g, !, write(black).
 a :- h, !, write(white).
```

Then either *black* or *white* is printed, but not both, when both threads of execution can manage to reach to the point to cut.

6.1 Operational Background

We assume that a cut *!* is compiled into an instruction *'cut_to ChP'*, where the parameter ChP holds a pointer to a choice point in the control stack. If the ChP is not older than the border choice point, this instruction simply pops all the choice points younger than ChP. The ChP becomes then the current choice point (at the top of the control stack).

The execution of a parallel *'cut_from ChP'* crossing the border is however a bit more complicated, because it involves requesting some related engines to give up their work. We first define which engines are to be effected. We introduce a concept of *destination choice point*, *DChP* in short, for a cut instruction.

1. ChP is the DChP if ChP is published; otherwise
2. The first published choice point older than ChP is the DChP.

And correspondingly the destination node, *DNd* in short, is defined as the node associated to the DChP, and the destination subtree *DST* is the one led from the DNd and embracing the leaf initiating the cut. The cut instruction in a concurrent setting requires that the whole

destination subtree except the path leading to the cutting leaf be pruned. That is, all of the alive leaves except one within the DST are to give up their work.

6.2 Prune Cut Destination Subtree

Our approach is as follows. The engine executing the cut first gets the DNd handle and resorts to the leaf which initiates a cut request to the DNd. On receiving the message, the DNd checks if the destination subtree DST has already been pruned. If so, this means that this cut-requesting leaf loses in a cut competition to other leaves. The DNd offers then a lodge to the leaf, and replies with a cut-failure message to the leaf. If not, the DNd creates an alive branch connecting the leaf, and replies with a cut-ok message to the leaf, and chops the DST. On receiving a reply, the leaf let the engine adjust the engine state accordingly. If the reply is negative, the leaf starts job-search. If positive, it lets the engine conclude the initial cut and continue. Figure 8 shows the case of a successful cut. Recall again the leaf transplant mechanism. The whole destination subtree is simply chopped and replaced with a short-cut one.

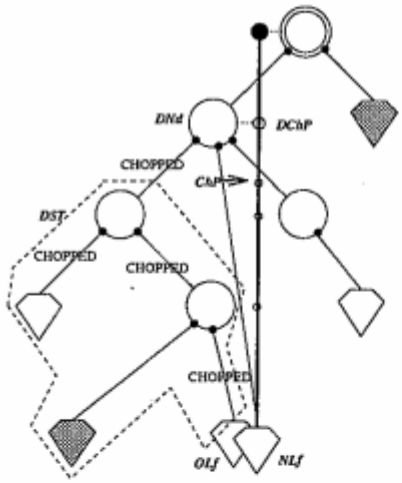

Figure 8: A Successful Cut-Request

We consider this cut approach innovative as it elegantly solves all of the typical cut problems encountered in a concurrent setting, i.e. cut competitions among sibling clauses and with nested destinations, and pruning operations with cautious steer clear of the winner path. Furthermore, the destination node provides an ideal start-point for the cut-failed leaf to search for job.

7 Revisit Scheduler Tree

Having described some of the major scheduler functions within the proposed scheduler framework, we are in a better position to examine more technical issues relating to the manipulation of the scheduler tree and some other practical topics, including garbage collection, state transitions, and engine vs. scheduler services.

7.1 Anatomy of Scheduler Tree

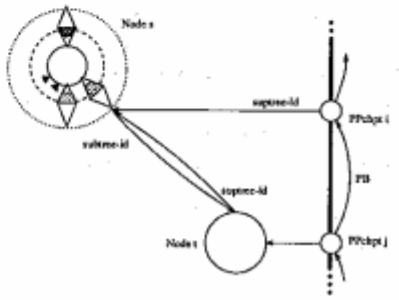

Figure 9: A Micro View of the Scheduler Tree

First we take a closer look at the scheduler tree structure, shown in figure 9. A node maintains parallel alternatives (shown as black triangles). For every scheduled alternative, there exists a branch, shown as a diamond, leading the subtree evolved. It is this diamond, instead of the node itself, which connects to a child of the node. And through this diamond, the child also refers back to the parent. Furthermore, the handle held by a published choice point is also to a diamond of the associated node. The existing branches of a node are internally chained together for sake of bookkeeping.

As a consequence, a branch (diamond) identifies a *subtree*. Tree components, nodes/leaves, refer to each other by means of subtree identifiers, and parallel choicepoints are associated to subtrees too. Recall in section 6.2 we need to figure out the destination subtree for a cut operation. This task becomes trivial as the subtree handle is available. Subtree is a relative concept. For example, a subtree is actually a super tree for

nodes/leaves within the subtree. Nevertheless we stick to using subtree as a general term.

7.2 Unique Subtree Identifier

Subtree identifiers (handles) are used as addresses for message passing. Therefore they have to be exclusively unique in the whole range of the distributed scheduler tree. For an easy solution, we introduce a concept of *site*. And a unique subtree identifier will consist of a site id, a local node id and a local branch id within the site.

7.2.1 Site

A site is a virtual address space, which corresponds to an operating system process. The site concept is not explicitly visible to the scheduler message passing but it nevertheless underlies the tree distribution. Thus site-based locality of the scheduler components presents an important source for message passing optimizations. The site concept has been known elsewhere as *worker* [15, 2, 18]. The term worker sounds more active. Indeed a worker here would denote an object collection which includes an engine, the leaves that have been associated to the engine, and the nodes that have evolved from these leaves.

7.3 Up and Down Message Flows

There are only two kinds of scheduling message flows: *DOWN*, from a parent to one of its children; and *UP*, from a node to its parent and from a leaf to any of its ancestors. As any messages are directed to a specific branch, we can have a perfect analogy here. A node, acting independently, is no more than a message handler. It has multiple ports for receiving and sending messages. Each of the node's subtrees has a separate port. There is also a port, for the communication to its parent. That is it.

7.3.1 A Regularity Assumption

In a distributed system, it is not intended to predict when a message must arrive at its destination. Our only assumption on the underling message passing system is the so-called *regularity*.

> Messages between two specific nodes arrive in the same order as they have been sent.

7.4 A Note on Garbage Collection

As for any dynamically expanded data structures, one essential issue for the scheduler tree is garbage collection. The following simple policies are proposed to enforce a disciplined message flow for a safe garbage collection.

> A chopping message is the last message flowing down via a branch; a chopped acknowledgement is the last message flowing up via a branch.

Therefore a chopped node can be collected only when all of its branches have been collected and it has sent a chopped acknowledgement to its parent; and a chopped branch can be collected when its chopping is acknowledged. Another consequence of the policies is that if a leaf exists, so do all of its ancestors (including the parent) and are properly connected, i.e. it is safe for a leaf to initiate a message to any of its ancestors.

7.5 State Transitions

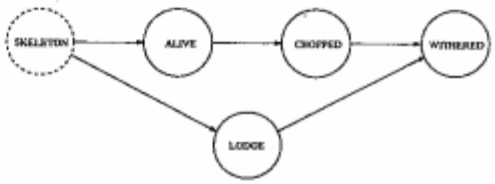

Figure 10: Branch State Transitions

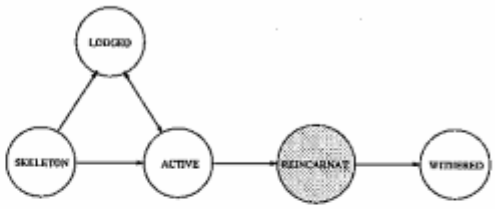

Figure 11: Leaf State Transitions

The states of a node are represented collectively through those of its branches. Figure 10 and figure 11 illustrate how a branch and a leaf spend their life, from being born to becoming withered, i.e. collected. The state transitions are straightforward, except that the state *REINCARNATION* needs a bit more explanation. As we have

presented in section 4, a leaf does not die, but simply give away its life to a new leaf. Therefore, a transition of *ALIVE* $\longrightarrow$ *REINCARNATION* of a leaf always coexists with the transition of *SKELETON* $\longrightarrow$ *ALIVE* or *SKELETON* $\longrightarrow$ *LODGE* of a newly introduced leaf.

7.6 Scheduler vs. Engine Services

Leaves and engines interact with each other through function calls corresponding to a handful of well-defined services (listed here to give the flavor).

```
sch_left_most     (leaf)
sch_load_report   (leaf)
sch_publish_one   (leaf,alts,&parent)
sch_backtrack     (leaf)
sch_cut           (leaf,parent)

eng_left_most     (engine)
eng_publish       (engine)
eng_backtrack_ok  (engine,parent,alt)
eng_undo_publish  (engine,parent)
eng_cut_ok        (engine,parent)
eng_install_state(engine,parent,coma,leaf)
```

With engine services a leaf is able to get its engine's state updated accordingly whenever its own state changes, owing to receiving scheduler messages. Note the usage of scheduler tree handles. Typically a leaf will ask its engine to manipulate the engine stacks delimited by a choice point holding a specific handle. For example, *eng_backtrack_ok(engine,parent,alt)* is to let the engine backtrack to a choice point holding the handle (*parent*), and set itself ready to execute the *alternative*.

8 Support of Prolog Operations

In order that existing applications developed on sequential systems can be easily made run on parallel ECL^{i}PS^e, support is needed to serialise certain Prolog operations with side-effect, e.g. some I/O operations, database updates, and sequential cut. In a sequential system, their execution order is fixed (by the depth-first search strategy). To achieve the similar effect in ECL^{i}PS^e with multiple active engines, a concept of *left-most path* (LM path in short) is supported within the scheduler tree. A LM path profiles a path in the search space where any of these serial operations can be executed without being delayed. Lets call the nodes (and the leaf) on the LM path the LM nodes (and the LM leaf). On the engine side, a suspension mechanism is introduced. On executing such a serial operation, an engine consults its leaf which checks if it is on the LM path. If so the engine goes ahead with the serial operation. Otherwise it suspends its execution and waits. On becoming a LM one, the leaf tells the engine to resume its execution.

The left-most path concept is achieved in the parallel ECL^{i}PS^e by maintaining a LM flag for nodes/leaves, and we let LM set nodes act as watchdogs for any LM state updates. A LM set node needs to take actions only on two occasions, i.e. when connecting an alive leaf, and when its solely LM set branch becomes non-alive (due to e.g. backtrack). Its currently left-most child branch is then instructed to become LM set when necessary. Note only on the second occasion, an explicit LM setting message is needed, flowing down along the left-most path of the node in question. On the first occasion, the LM instruction can be combined with a new connection acknowledgement message. It is to minimise the LM state update overhead.

Sequential cut. Although the LM condition for such a cut is sufficient, it is more than necessary. In fact, the cut can go ahead if the path in question is left-most within the cut destination subtree. Furthermore, the branches to the right of the cutting path (including the remaining alternatives) can also be pruned at once if the cutting path is left-most within any subtrees of the destination subtree. Therefore an *ad hoc* message for the sequential cut is initiated to exploit early prune opportunities. This message is from the leaf to the parent, and will be forwarded, up to the cut destination ancestor, if the receiving branch is left-most under the receiving node. On its way, the branches to the right of the receiving branch are chopped. The forwarding flow suspends when the receiving branch is not left-most, and resumes when it becomes left-most. Further optimizations with respects to the cut and other serial operations are possible through sophisticated program analysis, in order to minimise suspensions.

9 Search for Job

Job-scheduling consists of two sub-tasks, finding a job and installing the job. When an alternative is found for an engine, the engine can not start execution on the new alternative until its stacks have been properly installed

for the alternative. We first examine the installation topic and then examine the basic mechanisms for job-searching.

As a node maintains only alternatives, but not the necessary state for the execution of the alternative, it therefore has to entrust a leaf within its subtrees to carry out the state installation which will happen between two engines of two leaves in question (i.e. the state supplier and receiver, a topic out of the scope of this paper). Afterwards the supplier leaf acknowledges to the entrusting node. If the node is still rich with alternative, one alternative is indeed scheduled to the receiver leaf whose engine has been properly installed. The leaf can let the engine start execution. If the node becomes exhausted, it provides a new lodge for the receiver leaf, and lets the leaf continue its job-search.

Two additional notes. Looking for a leaf to entrust for state installation can be speeded up by traversal of only intra-site subtrees, if we make sure that there is a local leaf (see the following section); incremental installation can be enabled if a common ancestor node is known, which should be the outcome of the job-search procedure (see also the following).

9.1 Job-Search Prologue

We divide job-searching into two separate phases: a prologue and a more general searching procedure. In the prologue, the job-search range is the lodging path of the requesting leaf, i.e. search starts from the lodging node up to the root (the search can be further delimited among the adjacent and local ancestors). The consequence is that a leaf will stay within the subtree evolved from the leaf itself as long as there remain alternatives. It is straightforward to see the advantages to have this prologue. First, scheduling an alternative from the lodging path of a leaf does not incur any state installation overhead, as the needed state is available in the engine in question. The engine needs simply to backtrack to some associated choice point and executes the scheduled alternative. Secondly, a node offering an alternative to a job-requesting leaf can always find a local leaf to entrust for the installation task if the installation is indeed needed.

Note this job-search prologue is almost obligatory, imposed by the fact that the scheduler nodes do not hold a complete description about their parallel alternatives (i.e. missing states). Therefore engines under the nodes in question can not freely leave. A similar approach can also be found in Muse [1].

9.2 Traverse the Tree

Next phase of job-search has a more wide search space, i.e. the whole scheduler tree. It is tree traversal based, as illustrated in figure 12. The search follows the lodging path as a kind of controlling axis, starting from the lodging node of the leaf. When all of the subtrees led by an ancestor node have been traversed in a depth-first fashion, it steps up along the lodging path to the parent of the node in question, until the root is reached. (for the sake of incremental installation, the position that the search has reached along the lodging path is maintained during the traversal, as that will serve as the needed common ancestor for incremental installation).

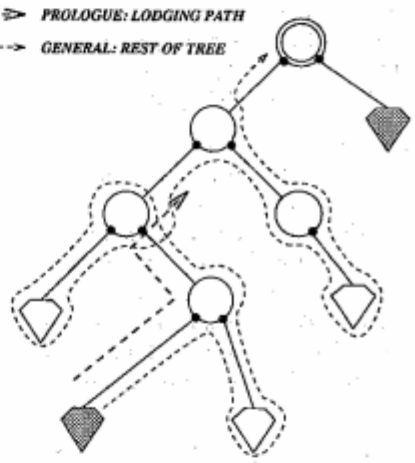

Figure 12: Search for Job – Two phrases

9.2.1 Search Only Rich Subtrees

In order to improve the traversal efficiency, we add to each branch a rich/poor flag. A subtree is set poor if it rejects a job-request, and job-search messages flow only into supposedly rich subtrees. When an engine accumulates enough local load, it will inform the leaf of its load. In case of being poor, the leaf turns itself into rich, and forwards the report up to the first non-poor ancestor, i.e. all of its poor ancestors are set rich.

9.2.2 Divide Job-Search into Sessions

What happens if a job-search finishes empty handed after the whole tree is traversed? We call it a *session*. A successful job-search may consist of multiple sessions. Clearly a new session should not be started if the load status of the scheduler tree remains the same, i.e. poor, as otherwise the system would suffer from constant disturbance. The ideal would be that a job-search suspends after a session is over, and woken up on the job availability. A straightforward solution is to let the root maintain a list of suspended job-search leaves, which will be woken up when a richness report flows to the root. To further reduce the search range, we introduce a concept of *job-search tree* to reduce job-search and richness report activities. A job-search tree is the biggest subtree of the scheduler tree, whose root node is with multiple alive branches or of alternatives to offer, and all of its ancestors (up to the scheduler root) have become exhausted. Obviously the initial job-search tree is the scheduler tree, and is dynamically updated to become smaller. On receiving a richness report, the job-search tree root tells the job-hunting leaves to restart the search, and determines a new root for the job-search tree (through downward message, of course).

9.3 Discussions

Job-search behaviours can basically be tuned by varying the search policies, and the publish policies. The publish policies are at the engines' disposal as they have better knowledge about the local load they hold. The decisions that when and what to publish and how much to publish at a time are essential to shape the scheduler tree. The scheduler side maintains the published alternatives, and are responsible for various search policies. For example, assume a node receives a job request. The node can either fulfill the request by first looking into its own alternatives, and then, if an alternative is not available, forward the request to one of its subtrees, or vice versa (i.e. the subtrees first searched, and the node itself second). When there are more than one subtree for a node, then one has to decide in which order they are traversed. Obviously knowledge about the participating sites (processors) and their physical relationship becomes important here. Traversing first the subtrees located on better connected sites can result in higher search speed. Applications involving serialised Prolog operations would favor a left-first policy. This issue becomes more complicated when there exists speculative parallelism [5]. For example, the efficiency of applications based on Branch & Bound search can be very sensitive to the variations of the job-search policies.

How to tune the job-search behaviours for high speed-up is a very important topic. Different fixes of the above-mentioned open decisions can result in various kinds of job-schedulers. We are currently busy adding instrumentation and experimenting various policy combinations, in order to evaluate within this framework some existing job-scheduling strategies [1, 4, 5, 6, 7, 8] and strategies specific to applications.

10 Conclusions and Future Work

We have presented a distributed scheduler framework designed for the Or-parallel ECL^{i}PS^e system. The framework is centered around a distributedly constructed scheduler tree, which is completely separate from the engine stacks. This design has two essential advantages. One is its clean and simple interface with the sequential engines and the modification imposed on a sequential engine is kept minimal, i.e. published choice points with scheduler handles. The other novelty is the decomposition of typically complicated scheduling functions among numerous independent agents which communicate with each other solely via messages. We have illustrated how major scheduling functions, including support for Prolog serial operations, can be nicely adapted to this distributed framework. In particular, a reincarnation approach has been introduced to achieve elegant engine migrations within a parallel application. We have also illustrated how job-scheduling can be carried out within this distributed framework. Important topics, such as state installation, multi-session job-search, dynamic load information distribution, etc., have been examined.

Current Status and Future Work. Individual components of parallel ECL^{i}PS^e have been implemented and integrated into a running parallel system where nodes are emulated as asynchronous event handlers. Preliminary tests and benchmarking are under way. We are adding instrumentations into the scheduler tree to enable flexible experiments with various job-search policies. Further research on the scheduler will focus on more sophisticated load information distribution schemes for the

scheduler tree, so that the job-search activities can be more knowledgeably guided. In addition to the locality optimization of the underlying message passing, attention will be given to various optimizations at the scheduler level, in order to reduce message traffic. One optimization currently under investigation is to add a *short-cut* into scheduler message traffic among local nodes/leaves (i.e. intra-site tree components). According to our preliminary statistics, as much as two third of the message traffic is intra-site, two third of which can be safely short-cut, i.e. a message sending operation is replaced with a message handling operation whenever it is safe to do so. We expect this optimization, combined with the underlying level ones, will basically eliminate the artificial message-passing operations on the occasion when either shared memory is available and can be efficiently used.

Acknowledgement

The author's experience with the ElipSys kernel design and development has been the major source of the basic design ideas presented here. The author would like to thank the following people for the many discussions he had with them and for their encouragement and support: Alexander Herold, Shyam Mudambi, Jacques Noyé, Mike Reeve, Joachim Schimpf, Kees Schuerman. Many Thanks to Steven Prestwich, Kees Schuerman for reading a draft of this paper and suggesting improvements, and to some anonymous referees for comments and suggestions on an ealier draft of this paper.

The parallel ECL^{i}PS^e project is partly funded by the Commission of the European Communities through Esprit Project 6708 (APPLAUSE).

References

[1] K. A. M. Ali and R. Karlsson. Scheduling Or-Parallelism in Muse. In *Proceedings ICLP'91*, The MIT Press, June 1991.

[2] K. A. M. Ali and R. Karlsson. The Muse Approach to Or-Parallel Prolog. Research report, SICS, May 1990.

[3] M. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[4] A. Beaumont, S. M. Raman, P. Szeredi, and D. H. D. Warren. Flexible Scheduling of Or-parallelism in Aurora: The Bristol Scheduler. In *PARLE'91*, Springer Verlag, June 1991.

[5] T. Beaumont and D. H. D. Warren. Scheduling Speculative Work in Or-parallel Prolog Systems. In *Proceedings ICLP'93*, The MIT Press, June 1993.

[6] P. Brand. Wavefront scheduling. 1988. Internal Report, Gigalips Project.

[7] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling Or-parallelism: an Argonne perspective. In *Proceedings ICLP'88*, The MIT Press, June 1988.

[8] A. Calderwood and P. Szeredi. Scheduling Or-parallelism in Aurora - The Manchester Scheduler. In *Proceedings ICLP'89*, The MIT Press, June 1989.

[9] W. F. Clocksin. The DelPhi Multiprocessor Inference Machine. In *Proceedings of the 4th U.K. Conference on Logic Programming*, pages 189–198, 1992.

[10] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings FGCS'88*, Tokyo, Japan, December 1988.

[11] ECL^{i}PS^e 3.4 User Manual. ECRC, January, 1994.

[12] ECL^{i}PS^e 3.4 Extensions User Manual. ECRC-94-9, 1994.

[13] R. A. Kowalski. *Logic for Problem Solving.* Elsevier North Holland, New York, N.Y, 1979.

[14] L. L. Li, M. Reeve, K. Schuerman, A. Véron, J. Bellone, C. Pradelles, Z. Palaskas, T. Stamatopoulos, D. Clark, S. Doursenot, C. Rawlings, J. Shirazi, and G. Sardu. APPLAUSE: Applications using the ElipSys parallel CLP system. Poster Abstract in *Proceedings ICLP'93*, The MIT Press, June 1993.

[15] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwodd, P. Szeridi, S. Haridi, P. Brand, M. Carlson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *Proceedings FGCS'88*, Tokyo, Nov-Dec 1988.

[16] S. Mudambi and J. Schimpf. Parallel CLP on Heterogeneous Networks. In *Proceedings ICLP'94*, The MIT Press, June 1994.

[17] A. Tevanian Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* PhD thesis, Carnegie Mellon University, 1987.

[18] A. Véron, K. Schuerman, M. Reeve, and L. L. Li. Why and How in the ElipSys OR-parallel CLP system. In *Proceedings of PARLE'93*, June 1993.

[19] David D.H. Warren. An Abstract Prolog Instruction Set. Technical Note TN-309, SRI, October 1983.

[20] D. S. Warren. Efficient Prolog memory management for flexible control strategies. *New Generation Computing*, 4:361–369, 1984.