Or-Parallel Prolog on a Distributed Memory Architecture

Fernando Silva Paul Watson

Technical Report Series: DCC-97-6

Departamento de Ciência de Computadores – Faculdade de Ciências
&
Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto
Rua do Campo Alegre, 823 4150 Porto, Portugal
Tel: +351+2+6001672 – Fax: +351+2+6003654
http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Or-Parallel Prolog on a Distributed Memory Architecture

Fernando Silva

Departamento Ciência de Computadores
& LIACC, Universidade do Porto
R. Campo Alegre, 823
4150 Porto, Portugal
(fds@ncc.up.pt)

Paul Watson

Department of Computing Science
University of Newcastle upon-tyne
Claremont Tower
Newcastle NE1 7RU, U.K.
(Paul.Watson@newcastle.ac.uk)

Abstract

Whilst there has been much work over the last ten years on the parallel implementation of logic programming, the vast majority of it has been focused on shared memory multiprocessor systems. In this paper, however, we focus on the exploitation of the other major type of parallel architecture - distributed memory systems.

We present the design and study the performance of the Dorpp or-parallel Prolog system on the EDS parallel machine. This is basically a distributed memory machine except for one interesting and novel feature - access to non-local memory is supported, and copied data is cached locally, though there is no built-in memory coherence scheme. Dorpp uses a novel distributed shared memory model for implicitly exploiting or-parallelism in Prolog programs. It attempts to exploit locality and reduce communication overheads through its scheduling mechanisms and by caching accesses to remote shared data. The specific problems of memory coherency for or-parallel Prolog are discussed and novel solutions are proposed.

We present detailed performance analysis of the behaviour of Dorpp on a parallel simulator of the EDS machine. This analysis shows that the design is efficient and highlights some general results which are relevant to all implementations of or-parallel prolog on distributed memory systems.

Keywords: Or-Parallel Prolog, Distributed Memory, Memory Coherency, Scheduling.

1 Introduction

Whilst there has been much work over the last ten years on the parallel implementation of logic programming, the vast majority of this work has focused on shared memory multiprocessor systems in which a set of CPUs share a memory over a common bus. This work has produced some encouraging results [2, 13, 4, 14, 8], but has some limitations in scalability inherent to the parallel architecture being used.

In this paper we focus on the exploitation of the other major type of parallel architecture - distributed memory systems (which are sometimes known as Massively Parallel Processors - MPP) in which a set of nodes consisting of both CPU(s) and local memory are interconnected by a high speed network. This architecture is important because it is extremely scaleable. When a node is added to the system a full set of extra computational resources (both CPU(s) and memory) is added. Therefore, provided that the network interconnecting the nodes is scaleable, systems with large numbers of nodes can be constructed. This contrasts with shared memory multiprocessors in which the memory and bus bandwidth is fixed, and so as extra CPUs are added, the bandwidth available to each CPU decreases.

Whilst distributed memory systems have greater scaleability, they have the disadvantage of a lack of uniformity of memory access. Typically, each parallel node has its own local memory which cannot be directly accessed by other nodes. Therefore, in most of these systems the only way for nodes to communicate is by passing messages. This severely complicates the task of parallelising logic programming systems because of their reliance on access to a global environment holding the values of variables.

In this paper we describe the design, and analyse the behaviour of, or-parallel programming on the EDS parallel system [19]. This is basically a distributed memory machine except for one interesting and novel feature - access to non-local memory is supported, and copied data is locally cached, but there is no built-in memory coherence scheme. If coherence is required then it must be provided by the application writer, though the operating system provides "hooks" that allow software coherence schemes to be added in a straightforward manner. Mindful of the run-time costs incurred by implementing a full coherence scheme in software, we investigated an alternative idea. The requirements for coherence in or-parallel Prolog were analysed, and it was discovered that it is possible to produce a correct implementation without incurring the run-time overhead of a full coherence scheme. Instead we identified a much cheaper scheme which was sufficient to ensure correct behaviour.

We believe that the work described in this paper makes the following contributions:

1. It analyses the design choices to be made when an or-parallel Prolog model is mapped onto a distributed memory parallel system, especially one which does not provide coherent virtual shared memory.
2. It pinpoints for the first time, the cases where coherency is required in distributed memory implementations of or-parallel Prolog and describes techniques to meet these requirements.
3. It describes the novel design of a scheduler for balancing the computation across the nodes of the parallel system.
4. It presents a detailed performance analysis of the behaviour of the system which shows that the design is efficient. This analysis highlights some general results which are relevant to all implementations of or-parallel prolog on distributed memory systems.

The rest of the paper is structured as follows. Section 2 describes the architecture of the EDS parallel system on which the project was focused. We concentrate on those aspects of the system which are important when attempting to map an or-parallel logic programming model onto it. In Section 3 and 4 we describe the design of the or-parallel system emphasizing the coherence mechanisms and scheduling. A key aspect of our work has been the use of a detailed simulator of the EDS system to perform detailed analyses of the behaviour of the logic programming system described in the paper. Using this work, section 5 presents a detailed performance evaluation of the system for a set of logic programs. Finally, in section 6 we draw some conclusions.

2 The EDS Parallel Machine

The EDS (European Declarative System) parallel machine [26] was designed and built by a consortium of companies and Universities working on the EU ESPRIT funded EDS project. It has a largely conventional distributed memory parallel architecture, except for some novel features which are exploited by the Dorpp logic programming implementation described in the later sections of this paper. In this section we describe the EDS system, concentrating on those novel features.

2.1 Hardware Architecture

Figure 1 shows the EDS machine architecture [24]. A set of Processing Elements (PEs) are interconnected via a fast internal Delta Network. Each PE contains two SPARC CPUs, a 64MB main

Figure 1: The EDS Machine Architecture.

memory, and a Delta Network connection. One of the CPUs is responsible only for the sending and receiving of messages to/from other nodes. This frees up the other processor to concentrate on application processing. The effect of this split of functionality is to reduce the message passing latency seen by applications, and increase throughput.

The Delta Network is built from 8x8 crossbar switches. The maximum number of nodes interconnected in the EDS machine is 64, and 2 layers of switches are required to achieve this. Each node can send and receive messages at a rate of 20MB/s in each direction simultaneously.

Connection to the outside world is provided by Host Elements which connect to both the Delta Network and external LANs.

The Diagnostic Element is used for the debugging and initialisation of the system. As well as the standard Delta Network connection, it also has a low bandwidth communication channel to every other node of the system, which can be used if the Delta Network is inoperable.

IO Elements contain SCSI attached disks for persistent data storage.

2.2 Software Architecture

Each Procesing Element (PE) runs an instance of an operating system - EMEX- which is designed for the efficient support of parallel applications. It was built by adapting the Chorus Systemes microkernel. In this section we will describe the three aspects of EMEX which are of most importance to the design of a parallel logic programming implementation.

2.2.1 Message Passing

A light weight communications stack is provided to allow message passing between nodes at high performance and low latency. The stack exploits the way in which the PE hardware has one CPU dedicated to message passing, by ensuring that almost all the work required to send or recieve a message is scheduled on that CPU.

2.2.2 Process Model

EMEX provides light weight threads as the basic unit of sequential computation. Threads are grouped into Teams which reside on the same PE and share a common virtual address space. Teams are then grouped into Tasks that can span more than one node, while still sharing a virtual address space.

2.2.3 Memory Model

The mechanisms that EMEX uses to share a virtual address space across a set of PEs is a novel compromise between the standard "share nothing" distributed memory architecture, and a full, coherent virtual shared memory system such as is offered by SCI hardware [20]. EMEX provides a basic, hardware supported system for copying pages between nodes when a remote address is accessed, but it is up to the application builder to provide an appropriate software cache coherency mechanism which meets their requirements. An area of virtual address space shared between Teams residing on different PEs is logically partitioned up among the PEs. Each Team has its own portion of the Virtual Address (VA) space which it "owns" and is allowed to read and write. A team is also allowed to read a part of the VA space which it does not own. When this occurs, the requested part of the VA space is copied from the owning PE and placed in a local cache so that future accesses to it will not need a further remote access (of course, at some stage the cache may become full and pages will have to be rejected- when this occurs it may be necessary to re-fetch the data).

Implementation of this mechanism is achieved using, as much as possible, the facilities of the standard SPARC MMU in order to maximise the performance. The VA space is split into standard 4KB pages which are managed by the MMU. The protection on each page is set to allow read and write access to locally "owned" pages, but the other pages in the address space initially have their protection fields set to disallow both reading and writing. If an attempt is made to access one of these pages, then a page fault is generated by the MMU. This calls an EMEX software handler to process the access. If the access was to write to the page then this is an application error. However, if the access was to read the page then a remote copying mechanism is triggered to fetch a copy of the page from its owning PE. When the page returns, it is stored in a local cache and the page table information is modified so that any future access will be directed to that cached page.

The latency for a remote page copy is 410μs. In order to optimise performance, two techniques are used. Firstly, the thread which required the remote copy is suspended and another thread run. Secondly, the EDS machine hardware provides support for sector copying. When the first access is made to a remote page, rather than copy the whole 4KB page, only a 128 Byte sector containing the required data is copied. This reduces the latency from 410μs to 37μs. On future access to the cached page, the hardware checks to see if the required data is in a sector which has been copied. If not then the required sector is copied and cached. The advantage of this scheme is that it keeps the granularity of page table and page allocation at the 4KB page level so that it is infrequent and efficient, whilst allowing 128 Byte sector copying to reduce the latency of remote accesses.

The application writer can build on this basic hardware supported scheme to provide whichever cache coherency regime is needed for the application. This is done by providing software handlers which are called when accesses are made to memory. For example writing to a local page which has been copied could call a handler to invalidate the remote copies. As these handlers are implemented in software, the cost of implementing such a scheme would be relatively high when compared to hardware solutions. Therefore for the or-parallel Prolog system described in this paper, we decided to investigate whether it would be possible to implement a specialised coherency scheme which would be cheaper than a general, fully coherent scheme. To do this we analysed the coherency requirements of a set of possible designs, and chose the one which lead to the cheapest solution. In the next section we describe this design, analyse its coherency requirements and show how they can be met by a relatively low cost software coherency scheme. This analysis is of more general interest than just for the particular design we chose because it highlights the access and update patterns which occur when Prolog's shared environment is mapped onto a distributed memory parallel machine.

3 Design of the OR-Parallel System

In this section we describe the key issues related to mapping or-parallel Prolog onto a distributed memory architecture, the EDS parallel machine, and the design choices we made. These key issues are: the representation of the environment, maintaining cache coherency and scheduling.

3.1 Representing the Environment

We analysed a range of possible environment representation schemes. The key issue for or-parallel Prolog is how to efficiently represent the multiple bindings for the same variable produced by the parallel execution of different branches of the search tree.

In conventional serial Prolog, alternative solutions to the same goal are found one at a time through backtracking. Backtracking restores the computation to an earlier state, including resetting some variables to unbound again, before the computation proceeds to try an alternative solution to the goal. Or-parallel Prolog is different - separate branches of the search tree are explored in parallel and a variable may be bound to different values in the different branches.

Different methods for handling multiple bindings of the same variable have been proposed. They have different cost ratios for operations such as variable access, task creation or task switching. On a distributed memory parallel machine such as the EDS system, they also raise different coherency issues and costs. We now examine the main methods devised to solve this problem, and consider their efficiency on a distributed memory system.

3.1.1 Binding Arrays

The SRI model [25] proposes using binding arrays to store all bindings made to shared variables. In this model, when a new variable is created, it is given a variable number uniquely identifying its entry in the binding array. The numbering of variables in the binding array is maintained by a counter that is incremented with every new variable being created. This counter is saved at every parallel choice-point so that whenever a processor attempts to execute an alternative branch it can get a copy of the counter and continue its own numbering of the variables it creates. A variable binding is conditional if its variable number is smaller than the counter stored in the current choice-point. Conditional bindings are stored in the binding array and in a global binding tree. This binding tree (corresponding to the trail in the WAM) is used to de-install bindings from the binding array during backtracking and to install new ones in case of a task switch.

The main advantage of this model is that, as bindings of all variables are accessible in constant-time, it is relatively simple to extend the sequential WAM to incorporate it (an example is the Aurora [12] system for shared memory machines). Furthermore, since each processor has its own binding arrays in its local store, the model is highly distributed in nature. The WAM stacks are generalized to cactus stacks mirroring the shape of the computation tree, and are shared by all workers. Nevertheless, high locality of reference can be expected as most of the variable accesses occur within the current frame which is held in local memory [22]. The model, however, induces high costs for task switching as it is necessary to de-install and reinstall bindings in the binding array.

3.1.2 Environment copying

Muse [2] is the most prominent system based on environment coying - a mechanism derived from the multi-sequential model [1]. In this model the workers do not share the environment state, that is, the WAM stacks. Instead, each worker maintains a separate environment almost as in sequential Prolog on which the bindings it makes are independently recorded, hence solving the multiple bindings problem. When a worker becomes idle as it has no work, it searches for a busy worker and copies its environment up to a certain or-node with an untried alternative branch that can be explored in parallel. One disadvantage of this method is the large amount of copying it may involve. Muse, however, significantly reduces the copying overheads by using an incremental environment copying mechanism. Nevertheless, in order to efficiently schedule the parallel work [3], Muse requires shared

Figure 2: A distributed stack corresponding to the search tree of some program.

memory to keep information about available parallel work equally accessible to all workers. It is through shared memory that workers synchronize to avoid re-visiting the branches already searched. Although a distributed scheduler could also be devised, we believe it would incur in significant communication overheads just to guarantee that different workers do not explore the same branch.

3.1.3 The Dorpp Binding Arrays Scheme

The Dorpp binding model design is based on binding arrays, as used in the SRI model [25]. The main reasons for adopting binding arrays are: it is a naturally distributed structure - each processor has its own binding arrays in its local memory; it induces high locality of reference since most variable accesses occur within the current frame which will be held in the local memory [22]; and it has proved to be efficient for shared memory machines [13]. Furthermore, scheduling can be totally distributed, hence avoiding any large shared data structures which could be a major source of overhead in a distributed memory architecture.

In Dorpp, a set of workers, one per PE, executes a Prolog program by traversing an or-search tree. Each worker executes Prolog sequentially in a depth-first left-to-right order with backtracking and performs scheduling duties when necessary. The WAM's local, global and trail stacks, are now generalised to *distributed stacks* mirroring the shape of the computation tree, and are physically distributed among the memories of the PEs, although a global logical structure is implicitly preserved. Each worker physically owns part of each stack, together with a private binding array used to keep a record of conditional bindings. Figure 2 illustrates a distributed stack corresponding to the search tree of some program. The figure shows the points were the shared variables were created and the conditional bindings made to those variables by the various workers.

When a worker starts a new task, that is when it starts executing an alternative branch of an or-node of the computation tree, it inherits an environment from the parent worker. This inherited environment, which corresponds to parts of stacks owned by other workers, has the property of being read-only. If a worker needs to modify the shared environment, for example to bind a shared variable

Figure 3: Shared environments in a computation tree.

it does so through its local binding array. Therefore, each worker always writes only to its local store and grows its own stacks (which are linked to other workers' stacks forming a logical stack).

The computation tree is not only physically distributed, it is also implicitly divided into shared and private parts, with corresponding shared and private or-nodes. A node is created as private and it becomes shared when a remote worker starts working on one of its branches. Nodes with untried alternatives are termed *live-nodes*, and those with no untried alternatives *dead-nodes*. A path from the root node to a certain node, N, below in the tree, is said to be *shared*, if and only if there are two or more workers working below node N, on alternative branches. Obviously, a shared environment corresponds to a shared path. Figure 3 shows the shared parts of a computation tree.

The computation tree can, simplistically, be seen as a tree of or-nodes. To this abstract view three types of root nodes have been added: *global-root*, *local-root*, and *temporary-root*. A global-root is defined by the worker that starts executing the initial query, the *root-worker*, and marks the root of the computation tree. A local-root is defined by each of the remote workers to indicate the root of their sub-tree. A temporary-root is defined whenever a worker suspends its current task and switches to a new temporary task; it marks the root of the sub-tree corresponding to the temporary task. A task may be suspended by a worker whenever it backtracks to a dead-node with child workers still active below that node. Figure 3 illustrates these three types of root nodes. When a worker backtracks to a root node, it indicates that the current task has just terminated. Root nodes are also used to save the state of suspended tasks so that they can be re-activated later.

Contrary to other shared environment schemes, such as the one adopted by Aurora, workers can only access shared nodes lying on their shared path. However, shared nodes may be stored on remote memories, therefore allowing a worker to "walk" throughout its shared path is not an appealing idea on a distributed memory machine where remote accesses are always costly. In Dorpp, a worker accesses the shared environment mainly when it has to dereference a non-local shared variable for the first time.

3.2 Efficiency considerations

Variable access and binding is a constant-time operation for shared environment models based on the Binding Arrays method [9], therefore it is a constant-time operation within the proposed model. Variable dereferencing is not a constant-time operation and therefore deserves special attention on any or-parallel model design.

Variable dereferencing, in a shared environment, often implies following a chain of pointers, possibly

Figure 4: Contents of the Binding Arrays of a worker before and after a task-switch: a) worker w2 has completed its task at node n1; and, b) w2 is starting a new task at node n2.

throughout the parent's frames. This chain of pointers might lead to various remote memories in a distributed memory machine. If, every time a worker tries to dereference a shared variable, it has to make various remote store accesses, then this operation would be a major source of overhead in our model. However, simulation studies indicate that when dereferencing shared variables, less than 1% of them require access to 2 or more ancestor frames, about 1/3 access one ancestor frame, and the majority, 2/3, access just the current frame which is always local [22]. This indicates that a very high percentage of the variable accesses within the dereferencing operation are localised. Furthermore, in Dorpp, whenever a worker has to dereference a non-local shared variable, only the first access to that variable incurs the cost of the remote copying of data. Thereafter, the variable is cached locally, and in the process other potentially shared variables in the same sector will also have been localised, reducing even further the number of non-local accesses. It is reasonable therefore to expect that the variable dereferencing operation will be fast, efficient and, in most cases, purely a local operation.

Task creation (or environment creation) relates to the creation of new nodes in the search tree and allocation of space for variables. This is a constant-time operation (as in the SRI model) since space for new variables is automatically allocated in the binding array when a variable is created even though its binding may later turn out to be unconditional.

A third operation which has to be done efficiently is task-switching (often called task-migration). A worker is said to switch tasks when it leaves its current task at a certain node and moves to another task at another node in the search tree. This occurs when a worker terminates its current task or when it backtracks to a dead node which has active workers below it. In both situations there is no local work left, therefore the worker has to search for a new task and if it succeeds in finding one then it performs a task-switch. The cost of task switching is the time taken by the worker to change its current environment to that at the new node.

In Dorpp, a task-switch involves copying all the conditional bindings corresponding to the path from the new node up to the root of the search tree from the BA of the worker owning the new node, and copying all WAM registers stored in that node.

When a worker switches from a task at node `n1` to a task at node `n2` it copies all the conditional bindings up to `n2` from the BA of the worker owning `n2` (uses the trail to restore the ones bound after `n2` was created) as well as the WAM registers stored in `n2` (figure 4 illustrates the conditional bindings copied by worker, `w2`, when it moved from `n1` to `n2`). It shares the environment (that is, the WAM stacks) defined by the path from `n2` up to the root of the search tree. The computation state (WAM registers and conditional bindings) for the new task is determined locally, that is at the processor where `n2` is physically located, and then copied eagerly by the worker switching tasks, `w2`, to its store.

3.3 Achieving Cache Coherency

The sector copying mechanism and the caching of remote sectors of data contribute to achieve high locality of reference in Dorpp's binding scheme. There is, however, one major problem arising from the caching of remote data and that is *memory coherency*. For example, suppose that a worker twice accesses the same remote sector, and suppose that in between those accesses the sector is overwritten by its owner. In a simple, but incorrect implementation this would cause problems. At the second access, the copying mechanism detects that the sector is already in the local cache, as result of the first access, and accesses it locally. This results in a memory coherency problem since the sector has been modified at its origin and so the cached copy is out of date.

The EDS operating system allows for user-controlled-coherency mechanisms to be added in as part of the language specific run time system [10]. Since general coherence mechanisms are bound to introduce extra complexity and some run time overhead in any parallel system, Dorpp has a specialised scheme designed specifically for or-parallel Prolog. We identified that by adopting appropriate scheduling measures there were only two cases where coherency problems could occur, and designed a specific solution to each one. These two cases are now discussed in detail.

First coherency issue

Figure 5 illustrates the first memory coherency issue. Consider a sector, `s1`, containing a choice point, `cp1`, and part of the environment before the choice point. It might well happen that a remote worker, `w2`, accesses a location within `s1`, while a local worker, `w1`, is still growing its local stack within the same sector (figure 5 (a)). If at a later stage, `w2`, now working on a different task corresponding to a branch of `cp2`, accesses a location within the same sector, but after choice point `cp1`, and the sector `s1` is found init independently records `w2`'s cache then the value returned can be inconsistent with that of the master copy in `w1`'s memory (figure 5 (b)).

A possible solution would be to impose on remote workers a delayed read-access to sectors containing choice points and still being write-accessed by the local worker. This is not, however, an easy solution to implement. Instead, a simpler solution was adopted. It consists of adjusting the top-of-local-stack pointer to point to the beginning of the next sector whenever a choice point is created in the local stack. This gap left in a stack-sector after each choice point creation, ensures that when the sector is remotely accessed, any copy being sent across will not contain any locations which might be later modified by the worker owning the sector.

This measure by itself is not sufficient to avoid all memory coherency problems. It still requires a scheduling strategy which makes it impossible for remote copied sectors from a shared path between two workers to be modified by the worker physically owning the sector. In Dorpp this is ensured by the scheduling mechanism in which a worker always grabs work from the topmost node in another worker's sub-tree which still has some alternatives left. Furthermore, the scheduling prevents a worker from backtracking past a node below which there are still active workers, hence preserving the integrity of the environment being shared with these workers.

Figure 5: First memory coherency issue: a)a sector is being copied into the cache of worker `w2` after it tried to access a remote location for the first time; b) worker `w2` is accessing a remote location within a sector that has been copied earlier into its cache and is now out of date.

Second coherency issue

There is another situation which, unless care is taken, could lead to a memory coherency problem. It occurs whenever a worker is backtracking through a previously shared address space and reaches a node with no alternatives left but still with active workers below it. In this situation, the worker voluntarily suspends its current task, switches to a new task and starts re-using the same previously shared address space. Now suppose that a location in a sector within the shared space had been accessed, and therefore cached, by a remote worker. If this worker, now working on a different task, tries to access a location within the same sector, the sector is found in its cache and, therefore, it is accessed locally. However, since the sector has been re-written at its origin, it is now out of date.

Figure 6 illustrates the above situation. Suppose that worker `w2` is working on a branch of choice point `cp2` and accesses a remote location within sector `s2`. The sector `s2` is copied by the remote sector copy mechanism into the cache of `w2` (figure 6 (a)). Consider now that `w2` terminates its current task and that `w1` backtracks up to `cp1`. Since `cp1` is a dead-node with still active workers below, `w1` suspends its current task and switches to a new one corresponding to a branch of `cp3` from worker `w3`. When computing the new task, `w1` starts re-using the address space below `cp1` that was previously shared by `w1` and `w2`. If at a later stage, `w2`, now working on a different task from `w1` (node `cp4`), tries to access a location within the same sector `s2`, the sector is found in its cache and therefore the value returned can be inconsistent with the master copy in the memory of `w1`.

A solution for this problem can be achieved by invalidating all remote copies of sectors corresponding to previously shared space that is about to be re-used. A worker backtracking from a previously shared node to a shared dead-node with active workers below, has to suspend its current task, look for another task, and switch to the new task when found. This worker would trigger the invalidation mechanism when initialising the new task, that is, before it starts executing it. This is the solution mechanism adopted for Dorpp. We do not expect it to be activated very frequently and since it is confined to just one previously shared chunk of a workers' sub-tree, it should not impose any major run time overhead on our scheme.

Figure 6: A memory coherency fault resulting from the re-use of memory space: a) a sector is being copied into the cache of worker `w2` as a result of a remote memory access; b) worker `w2` is accessing a remote address within the space being re-used. Since the sector containing that address has been cached earlier, the local value is accessed and therefore it may be now out of date.

4 Distributed Scheduling

Although distributed shared memory systems support a shared memory programming model, a different programming philosophy should be adopted for scheduling. Sharing large global data strutures among potentially a large number of processes should be avoided so as to reduce communication overheads. Compilers and execution systems should attempt to exploit locality to improve performance.

Within this framework, Dorpp employs a distributed and receiver-initiated scheduling strategy. One scheduler per worker makes independent and localised decisions related to work-sharing without incurring in extra interprocessor communications. Work-sharing decisions are receiver-initiated in that a worker chooses to search for busy workers from which work may be transferred only when it becomes idle after completing its current task.

Each worker has a local work-queue, and it adds to the back of queue entries to every node (or parallel choice-point) it creates during execution. Remote workers looking for work take the node entry from the front of the queue as it corresponds to the topmost node in the sub-tree of the local worker. The local worker consumes work from the back of the queue whenever it tries to get more local work. This strategy, called dispatching on the topmost [5], is designed to induce large grain tasks and so reduce task-switching and communication overheads. The fact that each worker executes, where possible, its locally produced work contributes to better locality.

Whenever a worker runs out of local work, it searches for more work from work-queues of other workers. To avoid the obvious inefficiencies of a blind search, each worker has a work-load bitmap that indicates which other workers have shareable work (i.e. high work-load) in their work-queues. Furthermore, two threshold values, a *lower threshold* and an *upper threshold*, are associated to each work-queue. The idea behind this scheme is that initially the workers build up a reserve of local work until the upper threshold is reached. At this point the worker informs the other workers that its work-load is high, so allowing for idle workers to steal work from its work-queue. When the amount of work in the work-queue falls to the lower threshold, the worker informs the other workers that its work-load is low hence stopping them from stealing work from its work-queue.

A Dorpp worker can be found in any of three states. It is in the *active* stage when it is executing a task. It is in the *idle* stage when in transition from the active to the scheduling state, when suspended waiting on a remote store access, or when pre-empted by a higher priority instance being executed in the same PE; and it is in the *scheduling* stage when looking for work. A worker in the active state traverses the computation tree in a depth-first left-to-right manner with normal backtracking. When an active worker fails to get local work through normal backtracking, it becomes idle. This failure to get local work happens when it completes its current task or when it backtracks to a shared dead-node with child workers working below it. Whenever a worker becomes idle as a result of either situation, it moves to the scheduling state and starts looking for work.

The work-search strategy followed by a worker in the scheduling state is determined within the backtracking algorithm. When a worker backtracks to a root-node, in which case it has completely exhausted the task it was working on, then it directs the work search to the parent worker from which it got the newly exhausted task. If it fails to get work from the parent, then any other worker known to have shareable work will be targeted, unless the root-node was a temporary-root in which case a previously suspended task will be reactivated. If, however, the worker backtracks to a node with no parallel work left, but still with child workers below it, then the worker directs the work search to one of the child workers. If it fails to get work from any child, then any other worker with shareable work will be targeted.

This work-search strategy is designed so that a worker tries to get a new task from the same worker (the parent) that gave it the newly terminated task. This has two advantages: firstly, it helps to reduce task creation/switching costs, since it may well be the case that the new task is found off the same node as the previous task, in which case no conditional bindings need to be copied. Secondly, there will be a maximum overlapping environment between the previous task and the new task, which helps to increase locality of reference.

5 Performance Evaluation

The evaluation of Dorpp was performed by running it through a parallel simulator for the EDS machine [16]. A simulator has some advantages over the real machine as it provides more detailed information from the execution of programs.

5.1 Benchmark programs

A number of benchmark Prolog programs commonly used by other researchers to assess their parallel Prolog systems [17, 21, 3, 14] are also used here to assess Dorpp. Simulating and analysing a program is extremely time consuming. It was necessary to focus on a relatively small number of programs which were chosen because of their differing characteristics - particularly the amount of parallelism they contain. All benchmarks find all the solutions for the problem. Multiple solutions are computed through "automatic backtracking on failure" after a solution has been found. The benchmark programs are:

Chat this program represents a database search type application. The program uses part of the database from Warren and Pereira's Chat-80 natural language query system. It finds all the possible solutions to the query "Which Western-European country borders a country in Eastern-Europe?".

Atlas this program is also related to the Chat-80 system. It finds the countries in the database which have approximately the same population densities.

Houses this is the "who owns the zebra" puzzle, using constraint directed search [21].

Map this solves the problem of colouring a map with four colours, such that no two neighbours have the same colour.

Queens8 this is a generate and test solution for the problem of placing 8 queens on a chess-board such that no two queens attack each other.

Cubes4, Cubes5 these programs, from [23], solve the N-cubes (also called Instant Insanity) problem, with $N = 4$ and $N = 5$. It consists of stacking N colored cubes in a column, so that no color appears twice within any given side of the column.

Some of these benchmark programs were evaluated through simulation by Kish Shen [17, 18, 21]. He found that even under the assumption of no overheads `Chat` and `Houses` have a low level of or-parallelism. `Atlas` and `Map` have a medium amount of parallelism and `Cubes5, Cubes4 and Queens8` have higher amounts of parallelism.

5.2 Timings and Speedups

To put the performance results in perspective we have compared Dorpp's performance, as measured by the EDS parallel simulator configured with a single PE, with the performance of SICStus Prolog version 0.7 on a SPARC-480 server, that is nominally has the same speed as the EDS processor, with the same benchmark programs. It shows that SICStus is between 2 and 4.5 times faster than Dorpp. A number of factors contribute to this difference: Dorpp includes the overheads of parallelism; SICStus uses shallow backtracking optimisation, a technique not yet implemented in Dorpp and which was shown by Carlsson [6] to significantly improve performance; poor optimisation by the ACK C-compiler used by the simulator; SICStus is already a mature, high speed and commercial implementation of Prolog, which is not the case of Dorpp. The overheads of parallelism are analysed in detail later in this section. However there is no reason why, if time and effort were available, the other optimisations found in SICStus could not be included in Dorpp.

The performance of Dorpp with multiple workers is presented in table 1. It presents the execution times in milliseconds, for the benchmark programs, with speedups relative to the 1 worker case given in parentheses. The benchmarks are divided in three groups according to the speedup shown: group H (high speedup), group M (medium speedup) and group L (low speedup).

		Dorpp - Workers (msecs)				
Group	Programs	1	2	4	8	16
H	Cubes5	9874.8	4938.9 **(2.00)**	2495.7 **(3.97)**	1288.0 **(7.67)**	700.6 **(14.09)**
H	Cubes4	2478.8	1245.6 **(1.99)**	636.9 **(3.89)**	350.5 **(7.07)**	209.3 **(11.84)**
H	Queens8	5678.0	2855.6 **(1.99)**	1454.6 **(3.90)**	768.1 **(7.40)**	494.8 **(11.48)**
M	Atlas	127.3	67.2 **(1.89)**	36.8 **(3.46)**	21.1 **(6.03)**	15.5 **(8.21)**
M	Map	779.2	395.8 **(1.97)**	230.4 **(3.38)**	140.0 **(5.57)**	112.0 **(6.96)**
L	Chat	91.1	51.7 **(1.76)**	33.8 **(2.70)**	23.0 **(3.96)**	23.4 **(3.89)**
L	Houses	186.0	100.1 **(1.86)**	64.8 **(2.87)**	58.9 **(3.16)**	51.1 **(3.64)**

Table 1: Execution times and speedups for the benchmarks.

The results show the ability of Dorpp to exploit or-parallelism, giving effective speedups over execution on one PE. The quality of the speedups achieved depends significantly on the amount of parallelism in the program being executed. The programs in group H have rather large search spaces, and are therefore amenable to the execution of coarse-grained tasks (see discussion on granularity, section 5.4).

This group shows good speedups up to 16 workers. A striking result here is the increase in speedup for `Cubes5` relatively to `Cubes4`, which is mainly due to the increase in granularity. `Atlas` and `Map` are in an intermediate performance group, group M, which shows relatively good speedups up to 8 workers. `Chat` and `Houses`, group L, show poor results with speedups levelling off very quickly.

The quality of the speedups shown by these programs is also highly dependent on the Dorpp model which determines task sizes, communication overheads, locality of reference and scheduling efficiency. We shall now analyse the impact of each of these factors on the benchmarks.

5.3 Locality of Reference

On the EDS machine the store is physically distributed, hence the issue of locality of reference is particularly important in order to avoid major overheads due to remote memory accesses. Our scheduling scheme was designed to take into account the existence of the EDS machine remote sector copying mechanism, by ensuring that workers search for new tasks that are closely related to their previous tasks. Note that with the remote copying mechanism, sectors that are remotely copied are cached locally therefore localising any further 'non-local' accesses to addresses within the copied sector.

Group	Programs		Local(%)	Hit(%)	Miss(%)
H	Cubes5	w=8	85.3	14.7	0.0
		w=16	84.2	15.6	0.1
	Cubes4	w=8	85.2	14.7	0.1
		w=16	84.2	15.6	0.2
	Queens8	w=8	84.2	15.7	0.1
		w=16	82.8	17.0	0.2
M	Atlas	w=8	76.1	23.7	0.2
		w=16	74.9	24.7	0.4
	Map	w=8	87.9	11.8	0.3
		w=16	87.5	12.0	0.5
L	Chat	w=8	74.3	25.2	0.5
		w=16	76.9	21.8	1.3
	Houses	w=8	86.8	12.7	0.5
		w=16	87.5	10.9	1.6

Table 2: Locality of reference for the benchmark programs.

Table 2 provides a breakdown on the type of memory references that where made during program execution. The column **Local** gives the percentage of references to local memory. The column **Miss** represents the percentage of remote references which resulted in a remote sector being copied. The remaining references will be made to locally cached copies of remote sectors thus referred to as **Hit**(s). The table shows very good results for locality of reference, the question now is why such good results?

- The data structures that each program uses are sequentially built on one of the stacks by just one worker. This makes the layout of the data structure, in terms of sectors, optimal. Moreover, whenever any remote worker needs to access the data structure it just has to copy the data once. That is, it has to copy only a minimal number of sectors.

  In an attempt to further investigate Dorpp's store copying design, considerable efforts were made to find programs which displayed less locality. It was expected that programs of this type would build distributed data structures in parallel before they were copied to other processors. However, the authors could not find such programs which at the same time had a reasonable amount of or-parallelism. The main reason is that the process of building data structures in parallel and having them copied around by most processors is exhibited by and-parallel rather than or-parallel programs, such is the case with matrix-multiplication, tree-sort and search, travelling salesman, quick-sort that are mostly divide-and-conquer and branch-and-bound algorithms.

- Once the remote data is copied it is extensively reused.
- The Dorpp binding model (based on the SRI model) induces high locality as the shared-environment is read only and just a very small proportion of the shared variables are non-local.
- The EDS machine has virtual memory but no discs to page out physical memory. Hence 64 MB is the limit for local and remote cache memory. Pages are allocated on demand for either remote caching or local memory. So, there is no fixed limit on remote cache memory size - though 64 MB is the upper bound. It is impossible to fill the cache and to experience a deterioration in performance due to the need to remove cached pages to free up memory because programs which used this amount of store would take months to simulate. To some extent this is similar to the ALLCACHE memory system of the KSR-1 machine [15] in which all available memory is cache.

Cache Size (KB)	Remote Sector-Copies	Pages Removed	Local%	Hit%	Miss%
no_limit	2469	0	84.2	15.6	0.2
256	3799	617	84.2	15.6	0.2
128	7125	1428	84.3	15.3	0.4

Table 3: Locality of reference for `Cubes4` with a cache limit of 256 and 128 KB.

In order to further investigate the effect of cache sizes on locality in Dorpp, experiments were made with a limit on each cache. For example, for the `Cubes4` with 16 workers no significant deterioration in performance was noticed when the cache size was limited as low as 256 and 128 kilobytes (a very small cache for a virtual memory system) and a FIFO strategy was used to remove pages from the cache. Table 3 summarizes the values observed for locality of reference.

The **Miss** results reported in table 2 are good up to 16 workers for the programs in groups H and M. However for those in group L the miss-rate value is rather high which indicates there will be major communication overheads. While miss-rate values of 2-3% may be considered good in a shared memory machine, it is not the case in a machine were a miss induces a rather high copy latency (about $52\mu s$). The results also suggest that on this machine it is not worth localising all the environment before unification, as done by Conery [7] on his closed environments binding model.

5.4 Scheduling and Parallel Overheads

On a distributed shared memory system a number of parameters influentiate the efficiency of the parallel execution of applications. Among these are the granularity of tasks - which should be high enough to compensate the overhead of executing tasks remotely - and the number and size of network messages - which should be kept as low as possible to reduce related execution overheads. Table 4 summarizes data for those parameters for three benchmark programs with different worker configurations. These benchmarks were chosen from each speedup group (high, medium and low speedup) as detailed earlier.

These results show that the decrease in the granularity of tasks is correlated with the increase in the overheads due to communication. The likely explanation is that workers run out of work more frequently and therefore not only have to search for work more often but also have to broadcast more work-load messages. More messages however contribute to further reduce the average granularity.

Granularity of Tasks

Programs		CALLS per TASK	BACK-TRACKS per TASK	NODES per TASK	TASKS per WORKER	SIMUL. Average Granularity
Queens8	w = 4	1225.0	620.8	260.2	30.8	71221
	w = 8	458.0	232.1	97.3	41.1	18226
	w = 16	103.7	52.5	22.0	90.5	2690
Atlas	w = 4	19.3	17.2	0.7	9.5	4643
	w = 8	14.5	12.8	0.5	6.4	2338
	w = 16	8.4	7.3	0.3	5.6	1207
Chat	w = 4	46.6	27.3	13.2	12.0	1159
	w = 8	36.7	21.7	10.3	7.6	537
	w = 16	19.7	11.8	5.5	7.1	357

Network Messages

Programs		COPY	(% Total Msgs) BROAD-CAST	WORK-SEARCH	TOTAL Msgs	Msgs/ PE/Sec
Queens8	w = 4	69.55	14.80	15.65	1764	303
	w = 8	60.31	26.35	13.34	6403	1042
	w = 16	45.62	41.99	12.39	38913	4915
Atlas	w = 4	48.83	31.95	19.22	385	2613
	w = 8	41.26	43.23	15.51	761	4504
	w = 16	41.15	43.21	15.64	1701	6858
Chat	w = 4	33.21	53.45	13.33	825	6088
	w = 8	25.46	61.27	13.26	1885	10227
	w = 16	21.31	67.37	11.32	4965	13266

Table 4: Parallel overheads.

This behaviour is more noticeable with programs in group `L`, such as `Chat`, hence explaining the poor speedups attained when running it with increasing numbers of workers.

We found that in order for programs to achieve good speedups the average granularity should be greater than 100 instructions. The results also suggest some inefficiency in the work-load propagation scheme which induces very high number of broadcast-messages. This is also due to the eagerness of workers in broadcasting work-load changes. Experiments have been made to control the rate of work-load propagation and determine its impact in the performance of the benchmark programs - the details are given in a later section.

We have further investigated the relationship between the performance of these parameters and the activities of the Dorpp's scheduler. For programs with lower-grain parallelism (such as `Chat`) the overheads due to scheduling increase steeply with the number of workers. The two main sources for this were the activities related to searching for work and messages broadcasting the work-load of workers. Workers run out of work more frequently if the tasks they execute are small, hence, increasing the number of work-load-messages (other workers need to know that there is no more work on this worker's work-queue) and increasing the overheads due to work-search. These activities are clearly related with increase in network messages and thread suspensions, as well as the decrease in granularity as was shown in table 4.

5.5 Progress of parallel computation in Time

So far we have analysed a number of performance parameters to gain a better understanding of how Dorpp scales up with the increase in the number of processing elements (that is workers).

In this section, we analyse the progress in time of parallel evaluations, for the `Queens8` benchmark with 8 and 16 workers, in order to gain further understanding of the behavior of Dorpp. The progress of a Dorpp execution is observed by monitoring, within the simulator, at regular time intervals the parameters directly contributing to communication overheads: number of Instances run, number of messages transmitted and received, and the number of remote sector copies (graphs (a) and (c)) in figure 7). The progress of workers' activities during execution was also traced by recording the times when each worker changed activity. In graphs (b) and (d) in figure 7, the progress of each worker's activity is represented by a line which varies between four levels each representing one of the activities traced. The activities traced are: searching for work (level 1), broadcasting low-work-load messages (level 2), actively working (level 3) and broadcasting high-work-load messages (level 4). They are depicted in the graphs by four sequences of points (belonging to four horizontal lines) with the sequence for level 1 at the top and the sequence for level 4 at the bottom. An activity line alternating quickly between different levels indicates high overheads of scheduling due to the broadcast of work-load messages and the search for more work. On the other hand, a straight line at level 3 represents the period during which a worker is actively working.

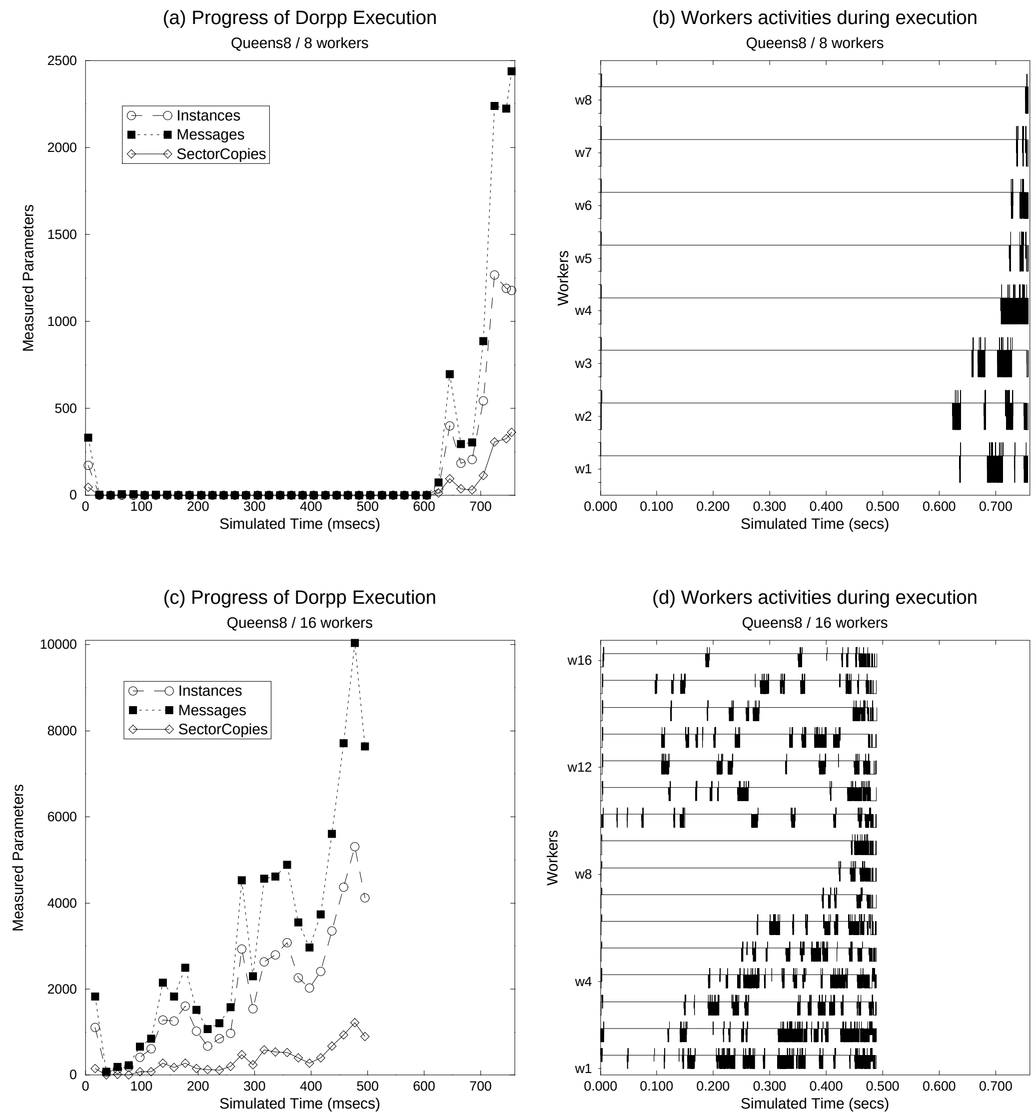

Figure 7: Progress of Dorpp and workers activities during execution for the `Queens8` program for 8 and 16 workers.

For the Queens8 program with 8 workers (graphs (a) and (c)), there is a small peak in the number of messages and instances at the beginning of the computation. This correspond to an initial worker activity (almost unnoticeable) that occurs the first time remote workers search for work (at the very start, only worker w1 has work to do). After the initial period, the overhead activity of Dorpp goes down which corresponds to a high amount of useful work being performed. Approaching the end of computation, fewer and smaller tasks are available to keep the workers busy. Therefore more scheduling and work-load broadcasting is done by the workers. Because this period is short, overall the performance is good. With 16 workers (graphs (c) and (d)), the overheads have increased sharply because there are more workers competing for the same amount of work. Figure (d) now shows that the granularity has decreased immensely with workers having to search more frequently for work during execution.

We have also collected similar results and graphs for the Atlas and Chat benchmarks with 8 workers and 16 workers [19]. The overheads increase when going from 8 to 16 workers are more noticeable on these programs as one would expect.

5.6 Impact of Threshold Parameters

As described earlier, the Dorpp's scheduling scheme associates two threshold values, a *lower threshold*, L, and an *upper threshold*, U, with the work-queue of each worker. The idea behind this scheme is that, initially, the workers build up a reserve of local work until the upper threshold is reached. At this point the worker informs the other workers that its work load is high, hence allowing for idle workers to steal work from its work-queue. When the amount of work in the work-queue falls to the lower threshold, the worker informs the other workers that its work load is low hence stopping them from stealing work from its work-queue. The selection of the correct values for the threshold values is crucial to performance. For example, if U is set too high, the work is exposed much more slowly and we may have a situation in which there is enough parallel work but still having workers idle waiting for the work to be published. On the other hand, if U is too small, the parallel work is exposed too early which may lead to an increase in network traffic and the system's eagerness to search for work. The threshold L can be set to zero to correspond to the situation in which workers only search for work when they become idle, that is when there is no work left in their work-queue.

The initial version of the system, the version for which the results presented so far in this section relate, used L=0 and U=1 as threshold values. These values represent the number of choice-point entries in the respective work-queue. However, counting the number of choice-points in the work-queue is only a crude measure of the amount of parallel work available. An alternative estimate is to count the number of parallel branches in the work-queue (used, for example, in Muse [11]). This is likely to be a better estimate because it considers the amount of work within the choice-point, except for U=1 in which case it is equivalent to the choice-point counting with U=1.

Table 5 shows the impact of the threshold parameters $(L, U) = \{(0,1),(0,3),(0,5)\}$ using choice-point counting and branch counting, on the performance of the benchmark programs for 16 workers. In general the results show an overall improvement for upper threshold values greater than one. The performance of benchmark programs with poor speedups, such as Chat and Houses, and also for Map, improved notably, especially Chat which doubled its speedup when parameters (0,5) and branch counting were used. These gains in performance are a consequence of restricting the exposure of parallel work which in turn imposes some control on the eagerness of the workers in searching for work and in broadcasting work-load messages. By allowing a worker to build a reserve of local work before broadcasting its work-load, a slowdown on the propagation of parallel work is achieved. This turns out to be a good thing for these programs as they have plenty of parallel tasks which are rather small in size. This contrasts with what happened when the parameters (0,1) were used. Here, the parallel work was exposed too early, tempting eager workers (commanded by a "work-stealing" strategy) to take parallel tasks. The granularity has increased enormously just by adjusting the thresholds. For example, the granularity for Chat and Houses increased by a factor of 5 and 4, respectively, while at the same time the network traffic was halved.

Programs		(Lower,Upper) thresholds CP-count (0,1)	(0,3)	(0,5)	Branch-count (0,1)	(0,3)	(0,5)
`Cubes5`	Speedups	14.09	14.18	14.72	14.03	14.20	14.07
	Messages/PE/sec	2165	1884	1737	2216	2023	2156
	Sim-Granularity	6057	7726	8018	6025	6177	6749
`Cubes4`	Speedups	11.84	12.79	13.65	12.04	12.95	13.16
	Messages/PE/sec	5007	3421	2300	4931	4117	3842
	Sim-Granularity	2362	4350	7455	2455	2763	3762
`Queens8`	Speedups	11.48	12.21	12.66	11.73	12.05	12.13
	Messages/PE/sec	4915	3721	3177	4320	3876	3838
	Sim-Granularity	2690	4464	5551	3275	4226	4594
`Atlas`	Speedups	8.21	1.00	1.00	8.38	8.43	8.43
	Messages/PE/sec	6858	*	*	7305	7310	7310
	Sim-Granularity	1207	*	*	1208	1205	1205
`Map`	Speedups	6.96	7.56	8.20	7.14	9.05	8.03
	Messages/PE/sec	11489	6841	6715	7646	6349	6682
	Sim-Granularity	538	1788	2021	1230	2048	1943
`Chat`	Speedups	3.89	5.02	5.72	3.52	5.35	7.06
	Messages/PE/sec	13266	9360	4742	14777	10212	6528
	Sim-Granularity	357	516	1718	281	535	1294
`Houses`	Speedups	3.64	5.37	6.04	3.75	5.52	5.67
	Messages/PE/sec	13590	8342	7109	12350	8853	6299
	Sim-Granularity	302	1098	1529	456	1079	1660

Table 5: Performance effects of threshold parameters with 16 workers.

For the other programs there are some improvements (although not as significant) which may be because they have large enough granularity of parallelism. The results also indicate that different programs may require different sets of lower and upper threshold values to achieve optimal parallel performance, and extensive experimentation is needed for tuning the threshold levels. A promising area of research might be to automatically adjust the thresholds depending on characteristics of the execution, for example, a worker could measure its granularity and use this to tune the thresholds.

Comparing the two measures used for work-load in the work-queue, choice-point and branch counting, branch counting does slightly better on most of the benchmarks. For benchmarks such as `Atlas` with a rather fat and shallow or-tree (a very small number of nodes, but many alternative branches per node), choice-point counting is inappropriate as it allows for a situation in which there are idle workers while at the same time potentially parallel tasks are not exposed. The `Atlas` program with choice-point counting and upper-threshold values of 3 or 5, was executed sequentially by just one worker. The worker never reached a situation of broadcasting high-work-load to the other workers so that they would try to steal work. On the other hand, with branch counting, the speedups were better and improved slightly with those threshold values.

6 Conclusions and further work

One underlying thesis of this work is that implicit or-parallelism can be exploited just as efficiently in distributed memory (with globally addressable memory) as in shared memory systems. To substantiate this belief, we have designed, implemented the Dorpp or-parallel Prolog system and evaluated it on a parallel simulator for the EDS machine. The main results of this work are:

- Effective parallel speedups can be obtained, provided there is plenty of medium to large-grain parallelism. These speedups are comparable to speedups reported for other systems running on shared memory machines. Under lower-grain parallelism, the speedups obtained are not so good and scheduler overheads become important.

- The Dorpp store model is very successful in restricting remote accesses. A very high locality of reference was achieved for this set of benchmarks.
- The novel memory coherency mechanism specifically designed for Dorpp successfully ensures cache coherency with minimal overhead - less than 0.5% of the the execution time for most of the programs.
- Dorpp's scheduler is successful in achieving good speedups and low overheads for the medium to highly parallel benchmark programs.
- The Dorpp execution model is successful in keeping communication overheads to a level where they are not significant, except at very low granularities.
- Granularity is critical, partly because of high thread switching overheads. Performance could be much better on a machine where these were lower.
- Branch counting rather than choice-point counting is a slightly better measure for the work load in the work-queue for most of the benchmarks.
- The use of upper-threshold work-queue length values greater than one improves parallel performance significantly for programs with low-grain parallelism. For example, the speedup for the **Chat** program, with an upper-threshold of 5, was doubled while at the same time granularity increased 3 times and the messages traffic decreased 4 times.

There are some optimizations that should be considered in order to further improve Dorpp's performance. One important optimization relates to the Prolog engine which should incorporate the shallow backtracking optimization as this helps to delay the creation of choice-points, therefore optimizing shallow failure and reducing the cost of backtracking. This would certainly reduce the gap between the serial performance of Dorpp and that of other Prolog compilers.

More experiments with scheduling should also be considered in order to tune the system and further reduce the scheduling overheads. In particular, experiments to improve the strategy of propagating the work load to other workers are needed, for example by incorporating the concept of processor neighborhoods. At the level of hardware architecture, the current scheme would benefit from having the second processor on each EDS PE carrying out load balancing, and therefore relieving the main processor of some of its duties.

Acknowledgements

The authors would like to acknowledge and thank for the contribution and support that John Sargeant and Nigel Paver gave to this work by writing and maintaining the EDS parallel simulator. They are also grateful to Vítor Santos Costa for his comments on this paper. This work was carried out while the authors were at Manchester University. They would like to thank the following institutions, Universidade do Porto, JNICT (Portugal) and EEC-ESPRIT for their financial support.

References

[1] K. Ali. Or-Parallel Execution of Prolog on a Multi-Sequential Machine. *International Journal of Parallel Processing*, 15(5):339–387, Oct 1987.

[2] K. Ali and R. Karlsson. The Muse or-parallel Prolog model and its performance. In *The 1990 North American Conference on Logic Programming*, pages 757–776. The MIT Press, October 1990.

[3] K. Ali and R. Karlsson. Schedulling or-parallelism in Muse. In *8th International Conference in Logic Programming*. The MIT Press, June 1991.

[4] K. Ali, R. Karlsson, and S. Mudambi. Performance of Muse on the BBN Butterfly TC2000. In *8th International Conference in Logic Programming*. The MIT Press, June 1991.

[5] A. Calderwood and P. Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *The 6th International Conference on Logic Programming*, pages 419–435. The MIT Press, June 1989.

[6] M. Carlsson. On the efficiency of optimising shallow backtracking in compiled prolog. In *The 6th International Conference on Logic Programming*, pages 3–16. The MIT Press, June 1989.

[7] J. S. Conery. Binding environments for parallel logic programs in non-shared memory multiprocessors. In IEEE, editor, *The 1987 Symposium on Logic Programming*, 1987.

[8] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: a Parallel Prolog system that transparently exploits both And- and Or-Parallelism. In *Principles and Practice of Parallel Programming*, pages 83–93, April 1991.

[9] G. Gupta and B. Jayaraman. On criteria for or-parallel execution of logic programs. In *The 7th International Conference on Logic Programming*. The MIT Press, 1990.

[10] P. Istavrinos and L. Borrmann. A process and memory model for a distributed-memory machine. In *CONPAR'90*, volume 457 of *Lecture Notes in Computer Science*, pages 479–488. Springer-Verlag, September 1990.

[11] R. Karlsson. *A High Performance OR-parallel Prolog System*. PhD thesis, SICS, Sweden, March 1992.

[12] E. Lusk, D. H. D. Warren, S. Haridi, et al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, Tokyo, Japan, November 1988.

[13] E. Lusk, D. H. D. Warren, S. Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[14] S. Mudambi. Performances of Aurora on NUMA Machines. In *The 8th International Conference on Logic Programming*, pages 793–806. The MIT Press, June 1991.

[15] J. Rothnie. Kendall Square Research Introduction to the KSR1. In *DAGS/PC'92*, June 1992.

[16] J. Sargeant. EDS Parallel Machine Simulator: version 2. EDS Research Document EDS.UD.3I.M016, Manchester University, November 1991.

[17] K. Shen. An investigation of the argonne model of or-parallel prolog. Master's thesis, Department of Computer Science, University of Manchester, 1986.

[18] K. Shen. *Studies of AND/OR Parallelism in Prolog*. PhD thesis, Cambridge University, UK, 1992.

[19] Fernando M. A. Silva. *An Implementation of Or-Parallel Prolog on a Distributed Shared Memory Architecture*. PhD thesis, Department of Computer Science, University of Manchester, September 1993.

[20] IEEE/ANSI standard 1596. Scalable Coherent Interface (SCI), 1992.

[21] P. Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *The North American Conference on Logic Programming*, pages 713–732. The MIT Press, October 1989.

[22] E. Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, 1987.

[23] E. Tick. *Parallel Logic Programming*. The MIT Press, 1991.

[24] M. Ward, P. Townsend, and G. Watzlawik. EDS hardware architecture. In *CONPAR'90*, volume 457 of *Lecture Notes in Computer Science*, pages 816–827. Springer-Verlag, September 1990.

[25] D. H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In IEEE, editor, *The 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[26] P. Watson and P. Townsend. The EDS Parallel Relational Database System. In *The Prisma Workshop on Database Systems, Holland, LNCS 503*. Springer-Verlag, 1991.