Some Aspect of Implementing the Parallel DASWAM System

Kish Shen
Department of Computer Science
University of Manchester
Manchester M13 9PL
U.K.
kish@cs.man.ac.uk

1. Introduction

The DASWAM system is a proto-type implementation of the DDAS and-parallel execution scheme for Prolog [4, 3]. It was initially implemented as a pseudo-parallel system where the parallel execution was simulated, but this was subsequently converted to a real parallel system that has been ported to a wide range of shared-memory multiprocessors. The initial results from this system are very encouraging, and have been reported in [7]. The purpose of this paper is to serve as a companion paper to [7], documenting the implementation of the parallel DASWAM system itself. The pseudo-parallel DASWAM system has been documented extensively previously [4, 5, 8], but the parallel DASWAM system has not been described, except very briefly in [7].

This paper will assume that the reader has some familiarity with the pseudo-parallel DASWAM. A brief introduction will be given here, but readers are referred to [4, 8] for a more complete guide.

1.1. A Brief Overview of DDAS

DDAS (Dynamic Dependent And-parallel Scheme) is a parallel execution model for Prolog (or Prolog-like languages). It was designed to allow the exploitation of non-deterministic dependent and-parallelism. Details of this scheme is beyond the scope of this paper, but has been extensively documented elsewhere [4, 3, 6]. The main aim of the scheme is to exploit as much and-parallelism as possible, without incurring too high an overhead.

The key innovative idea behind DDAS is that of the *dynamic producer*. Binding conflicts to the same instance of a variable is resolved through a producer/consumer relationship for the and-goals, with the *leftmost* active goal sharing the variable being the producer for the variable. As the leftmost active goal changes during execution as goals complete their executions, so does the producer status: hence the dynamic nature of the producer.

The other important innovative idea behind DDAS is to distinguish *dependent variables* from other variables. Dependent variables are those variables which are shared by more than one and-goal, and can possibly be bound in more than one of these and-goals. Only these dependent variables have to obey the producer/consumer relationships; all other variables can be treated as in sequential

Prolog. Parallelism in DDAS is indicated via ECGEs, an annotations that indicates *where* parallelism is to be exploited – this is an extension (to exploiting DAP) of the CGE annotation of &-Prolog [2].

Finally, in order to maintain equivalence to Prolog, a backward execution scheme, which defines the actions to be taken when a failure occurs, has to be defined. This is more complicated than the simple backtracking in Prolog because actions may need to be co-ordinated between and-goals that are/were executed in parallel. For correctness, this scheme must maintain equivalence to sequential Prolog [4]. Due to the non-determinism, extra information has to be available to allow for the correct backward execution behaviour, which imposes extra overhead during the forward execution phase but may help to increase the cleverness (and complexity) of the backward execution scheme. Many schemes are possible; a relatively simple scheme which tries to minimise the overhead during normal (forward) execution and the complexity during the backward execution phase has been selected for the current DDAS.

1.2. The DASWAM System

The DASWAM system is an implementation of DDAS. The system consists of several workers which co-operate in parallel to execute the program code. Each worker executes a "chunk" of code (a *task*) more or less as a sequential Prolog engine would, except for the extra synchronisation due to parallelism. Parallelism arises when more than one worker is simultaneously executing code from different parts of the program's search-space. The system is an extension and enhancement of the independent and-parallelism RAP-WAM [2] system: the workers are WAMs [9] modified to deal with and-parallel execution under DDAS, and consists of the abstract machine, plus the various stacks associated with a WAM, and a couple of extra stacks/heaps needed for parallel execution (collectively referred to as a *stack set*), which are accessible by other workers. The execution state of a program is distributed across the various workers' stack sets, collectively forming a *distributed stack*. A worker picks up a task, starts executing it, and when the task either completes or is suspended, the worker can then pick up another task to execute. The memory used by each task is separated and managed via *markers*. Markers are of various types, including parcall markers, used for managing the ECGEs, and suspend and continuation markers for management of suspension and continuation of tasks.

The pseudo-parallel DASWAM simulates parallelism by interleaving the execution of the different workers' abstract machine instructions. Other than this, all other aspects of parallel execution are implemented in full. In the parallel DASWAM system, each worker is executed by a separate process/thread which shares at least some common address space with other workers.

2. Issues for Porting the Parallel DASWAM System

The main aim of both of the current proto-type DASWAM systems is to verify the DDAS execution model. Thus the main concern is for correct execution rather than performance. There has been little effort to try and achieve very high performance initially, and there has not yet been much effort in optimisation. Nevertheless, even in its current unoptimised state, the parallel DASWAM system is able to achieve generally very good performance both in terms of its basic speed and its speedups when compared to comparable and-parallel systems ([7]). Some early work on improving the execution has started, and the result are reported in section 4., but it is expected that much better performance would be achieved with further tuning, now that the basic concepts for DASWAM/DDAS have been verified.

One important consideration in implementing the parallel DASWAM system was to maintain its portability such that it can be compiled and ran on as wide a range of parallel systems and C compilers as possible. This allows DASWAM to be not tied to a particular computer system and allows it to be studied under different architectures. To allow this, the code was written as machine independently as possible, with the machine/system dependent code written as macros so that porting to a new system should involve only the rewriting of some of the macros.

The following issues are important for portability:

Word length To the uninitiated, a parallel or sequential Prolog abstract machine can have a surprisingly strong dependence on the machine's word-length. The main reason for this dependency is because such abstract machines use tagged pointers, and many implementations make non-portable assumptions about the relationship between the representation of pointers and basic data sizes: in 32 bit machines under C, the assumption made by many Prolog implementation is that `sizeof(int) == sizeof(pointer)`. This is almost universally correct for all 32 bit machines, but will break down for a different word length. An important example of this is the pre-3.0 versions of SICStus, which will not run correctly under 64 bit systems without a very significant rewrite of the code — for example, a probably partial porting of SICStus 2.1 to the KSR1, a 64 bit system, required several man-months of intensive effort by me with very significant help from the main author of SICStus, Mats Carlsson. This work was done just prior to the parallelising of the DASWAM system, so before the actual parallelisation was done, the pseudo-parallel DASWAM system was made to be as portable across machines with different word lengths as possible.

In the current implementation, word-length independence is achieved by using user defined types instead of C's builtin types. For example, a type of 'Int' is used to denote integers that have the same size as a pointer. All that needs to be supplied by the user is the word length, and all the appropriate types and tags needed are automatically generated during compilation. It should also be possible for the word length to be determined automatically, although this is not yet done. Currently, the top two and bottom two bits are used for tags: the bottom (least significant) bits are used because they are not significant in addressing words in a 32 bit word-length machine (assuming the smallest accessible unit is a byte). This reduces the number of most significant bits that needs to be used as tags (using these bits as tags reduces the total addressable memory). However, all accesses and conversion of the tags are done via macros, so it should be easy to change which bits are used for tagging. Thus, DASWAM should be portable to a machine with any word length, as long as four bits of the word can be used for tagging instead of forming part of the address.

One portability problem encountered with using the top two and bottom two bits of a word as tag is a problem specific to C: specifically, C's `switch` statement. The selector for a switch statement is defined to be of type `int` in C. With a 64 bit machine such as the KSR1, where the tag value is a 64 bit quantity, but type `int` is 32 bits, the tag needed to be converted to a 32 bit quantity by right-shifting the top 2 bits into the lower 32 bits before the switch statement can be executed. This is again done through macros, and means that the extra overhead of shifting (which is only paid for when tags are used in a switch statement) is paid only by machines with range of addressing space > range of `int`.

Processes DASWAM requires the system to provide some means to spawn processes which can execute in parallel with other processes with which it share some common address space. Each of these process is used to implement a worker. Different parallel systems have different means of providing this, and the interfaces for them are different. Again, this is hidden from the rest of the program through a standard user-defined interface which is then mapped to the system specific interface via macros. All the parallel systems that DASWAM was ported to provided the necessary functionality, either via some sort of threads package, or an equivalent package (e.g. the `m_fork` interface for the Sequent and SGI IRIX machines).

Another aspect that needs to be considered is how memory is shared by the processes. For many thread-based systems, memory are automatically shared between the different threads. In others, such as the Sequent's `m_fork` interface[1], memory are only shared if a special malloc is used, with global C variables not shared by the various processes. Thus, on the Sequent, each process gets their own copies of the global variables, which are not shared with the other processes: this is in contrast to the other systems, where the global variables are shared between the processes, so any changes to a global variable will (eventually) be seen by other processes. For DASWAM, most of the memory that needs to be shared are malloc'ed memory: the various stack areas and other shared structures. However, there are a few global variables that need to be shared. In order to allow the sharing of these variables on the Sequent, these variables are implemented by making these variables to be pointers to the type needed in allocated shared memory. This memory is allocated before the spawning of the process, so that the pointer value will be visible to all processes when they are spawned. In order to allow the code to be portable, allocation of memory is implemented via a macro, so that for cases like the Sequent, the special shared memory malloc can be used instead of the normal malloc. This avoids the problem of using non-portable features such as the special declarations for shared global variables for the Sequent Symmetry (which is not even portable to different versions of C on the Sequent; and as the Sequent version Andorra-I uses these declarations for shared global variables, it cannot be compiled with gcc, which do not recognise these declarations.[2]).

Thus, porting this aspect of the DASWAM system involves providing the macros to spawn off the workers and the allocation of shared memory. All that is required for each worker (i.e. a process/thread) is a way of obtaining a unique identifier. All other information that is needed by the worker is passed via the shared memory. Thus, very little assumptions need to be made about the actual interface, and porting is quite simple.

Locks Locks are required to synchronise accesses to certain parts of the DASWAM system. Like many other parallel Prolog systems, for parallelism to be exploited fully,locks are needed on actual memory locations, instead of only on parts of program code as in many other parallel applications. This is because locks are needed for dynamically created data-structures pointed to by pointers. Many parallel programming environments do not provide the full generality needed for this sort of lock, and special locking code had to be written for several parallel systems (such as Solaris-based systems). In order to maximise portability, the locking code are abstracted into macros. Porting to a new parallel system thus involves providing the code for these macros.

So far, locks are implemented in two basic ways on the DASWAM: one way is to use special hardware locks, as provided by KSR1. In this case, the machine has special extra memory bits (invisible to the user), which are used to implement locks, and these locks are then available to user. In the case of the KSR1, each subpage of memory has its own lock. For the other systems, no hardware locks are available, and locks are implemented via the standard method of atomic operations on memory locations. In these cases, a memory word has to be allocated for use with each lock, and this has to be added to existing data-structures. This is again done via macros on the definition of the data-structures, so that the same code can be used for the both types of locks, regardless of if a memory word is required for the lock or not – a word would be allocated in the case of the software locks, and nothing in the case of hardware locks. Note that a word is used in the case of software locks to maximise portability. Atomic

[1]Note that although the SGI IRIX also uses the m_fork interface, memory are automatically shared between processes.

[2]Personal communication with R. Yang, 1996.

operations on words are most likely to be available on all systems, and although in some cases (such as the Sequent), locks can be implemented via smaller units of memory, this will not be as portable. A memory word can always be used for locks that need less than a word, but not the other way round.

One problem with hardware locks is that they may lock a range of memory locations — a subpage in the case of the KSR1. If two memory locations that need to be locked by the same process at the same time fall on the same range, then deadlock can result. Fortunately, because deadlocks will only result from two attempted lockings by the *same* process, situations in the code in which this can occur are easily recognised, and extra code added in cases where they are needed, which perform run-time tests to check if the there will be attempted locking of the same memory range, and if so, the lock is acquired only once.

As an extra word has to be added for the normal software locks. The same macros can easily add extra words with each lock to allow for debugging purposes: these extra words can be used to store information that can then be used for debugging purposes (for example, which worker is holding the lock, etc.). They have proven to be useful when debugging problems associated with lockings.

Thus, porting of locks involves the provision of the macros for the locks. In some cases, the system either provides the needed locking functionality, or it provides functionality for atomic actions which can be used for locking. However, in many cases, the actual code for locking has to be written using an atomic machine instruction.

Thus, in summary, porting of the parallel DASWAM system involves determining the word length of the machine, provision of macros for locking and starting processes/threads which has the ability to share memory. The parallel system has to provide some form of shared memory, and either locks or some method of allowing atomic operation on memory. Also, the system assumes that all pointers are the same size, and that the bottom two bits can be used for tagging (although changing the tag to not use the bottom two bits should be quite easy).

3. Parallelising DASWAM

The pseudo-parallel system can be regarded as a parallel implementation where a worker has an implicit global lock on all resources when it is executing an abstract machine instruction and when it is performing backward execution on a stack section.[3]. In this context, the parallelisation process can be seen as simply introducing more specific locks that are more localised in their effect to allow for more parallelism.

Besides introducing locks, the parallelisation process introduced a few relatively minor modifications to the system: the pseudo-parallel system used a global structure to store woken tasks which have not yet been picked up by idling workers to work on. This was expected to be inefficient, so this single structure was split up so that each worker has its own "woken queue" to store woken tasks. Also, the pseudo parallel system repeatedly cycles through all the workers, and either executes an abstract machine instruction, perform backtracking, deal with interrupts ("signals" which has been sent from one worker to another, generally used to co-ordinate backward execution), or look for new

[3]Note that backward execution can affect many stack sections in different workers, so the backward execution implementation of DASWAM still has to deal with most of the complexity of a fully parallel one.

work if the worker is idle.[4] Obviously such a cycle does not exist for the parallel system, where each worker is spawn off as a separate process, so the code to deal with interrupts and looking for work have to be moved from the cycle to appropriate places in the program. For the looking for work code, it was moved into a procedure which is called whenever a worker becomes idle. This introduces no real change in the behaviour of the system. However, for the interrupt code, the interrupt is dealt with at each instruction cycle in the pseudo-parallel system, and this behaviour can be duplicated in the parallel system if the worker checks to see if its interrupt flag was set before it executes an instruction. However, it was felt that as interrupts are relatively rare, checking for this at each instruction is excessive, so the check at performed only before executing certain instructions: the `call`, `execute`, and the `allocate_parcall` instructions. This means that interrupts are effectively checked for at each procedure call. It is done for the `allocate_parcall` instructions because some of the "signals" (such as kill) would cause the worker to abandon its current task, and it is thus a good idea to avoid the expensive allocation of parallel work if the task is to be killed. In addition to performing the check before these instructions, it is also performed regularly when an idle worker is looking for work.

3.1. Implementing locks

As stated, the most significant changes in the parallel system is the addition of locks. Locks have been added to various data structures of the system to allow for parallelism when those structures are used. The main considerations for implementing the locks, and how they are added to the major data-structures of DASWAM, shall now be discussed.

Four main considerations were taken into account when implementing locks:

- Correctness: the locks must lead to correct behaviour.
- Amount of parallelism: locks are generally added to increase the amount of parallelism that can be correctly exploited.
- Implementation complexity: while increasing the amount of parallelism was important, it was also important that this does not lead to great complexity, as efficiency was not the major concern for the proto-type implementation. An example of this is that currently, the parcall marker has only one lock on it, although it is used to manage all the and-goals of the ECGE it represents. This means that actions such as updating the statuses of different sibling and-tasks are serialised, although there is no need for this if locks are added to individual and-goal slots of the parcall marker.
- Efficiency of the lock: while this is not a very important concern for the proto-type implementation, some small effort was nevertheless made to make the locks efficient by trying to reduce the number of times they are used, and the duration that they are held for.

The results presented in [7] suggest that the current DASWAM system has achieved quite effective and efficient lockings which allows most of the parallelism in a program to be effectively exploited. The actual implementation of some of these locks shall now be outlined.

Locks are placed on fixed structures for workers such as the interrupt queue, woken work queue, available work queue, and also allocated dynamically to such structures as markers, dependent cells. These shall be discussed in the following sections. However, a specialise lock is also needed for

[4]Actually a worker can also do nothing waiting on some locks: some locks were needed in the pseudo-parallel system to deal with actions that spans more than one abstract machine instruction. This are implemented in exactly the same way in the parallel implementation.

unbound variables which would be bound to a dependent cell. The locking of unbound variables presents some special problems, because a unbound variable do not normally need locking, unless it is to be bound to a dependent cell. It is thus very expensive to provide every unbound variable with an extra word for locking. Thus, a special type of lock is used for unbound variables: they are locked by turning them into a self-referencing pointer, thus any dereferencing of the variable when it is locked would simply spin.[5] No extra memory word is needed for this type of lock.

3.1.1. Locking on work queues

A worker can become idle when it finishes its current task, if its current task gets discarded because it was killed, or if the task becomes suspended. In these cases, the worker enters a routine and actively look for work until either it finds new work, or if the execution of the whole program completes.[6]

There are two types of work available in the DASWAM system: new tasks to be started, which are represented by available goals on the goal stack of the workers, and woken tasks which have become unsuspended, and are thus available for further execution. As already discussed, these were placed on a single global woken queue structure in the pseudo-parallel system, but this was replaced by individual woken work queues for each worker.

When a worker is looking for work, it cycles through the various woken work queues to look for available woken work, and if it did not find any, it then checks the various available work queue for work. At the end of each loop, it checks to see if it has been interrupted. Locking is of course needed to ensure that available work is correctly taken up by just one idle worker. In the current system, a worker is always actively seeking work when it becomes idle. It is expected that in the future, more sophisticated schedulers may allow a worker to become more passive if no work is available, such as releasing the physical processor for use by other jobs in the parallel machine, as has been done for Aurora.[7]

Because all idle workers are actively seeking work, it is important to make seeking work as cheap as possible, and certainly it should not interfere non-idle workers, and to a lesser extent, other idle workers. This is done by moving all the information necessary for the available work queue and goal stack to a separate area of memory that is self-contained. This means that some of the worker's register (such as the pointers to its goal stack and woken work queue) are moved from the worker data-structure. This should greatly lessen the chance of interference of other workers while seeking work. In addition, in order to reduce lockings, each goal stack keeps a count of the number of available goals in the stack. When a worker is seeking new work, it simply checks this number without locking, and only if it is non-zero will it attempt to lock the stack to see if work is available. Locking is not required because this initial check does not have to be absolutely accurate: precision is required only in the second stage when it actually tries to lock the stack. In addition, if the worker fails to acquire the lock, it simply gives up and carries on, instead of waiting for the lock to become free. This scheme should reduce the amount of memory traffic that would otherwise be caused by the continuous locking of the various workers' goal stack.

The woken work queue have been implemented in two different ways: in the original method,

---

[5] This was suggested by V. Santos Costa, and has been developed independently in several earlier and-parallel systems, the earliest of which seems to be Crammond's implementation of Parlog [1] (thanks to Johan Montelius for pointing this out).

[6] Actually, a different routine is called when a worker becomes idle because of suspension. This is partly because of special requirements for suspension, and also because this allows different scheduling decisions to be made: pushing a suspended marker onto the stack is expensive, and it may be desirable to delay this somewhat by delaying when a suspended worker start to look for new work.

[7] Personal communications with David Warren and Tony Beaumont, circa 1992.

woken work is added to and taken from the top of the queue (so actually it is a stack), and there is only one pointer per queue, which needs to be locked both when taking and adding work. In the second approach, work is added at the front of the queue and taken from the back in a circular buffer. The pointers to the front and back of the queue are kept in separate memory locations to minimise the possibility that they would interfere with each other. In this case, checking for and adding of work does not require locking (only the worker which "owns" the queue can add work to it). It was expected that this second method would be more efficient, but limited experiments on the Sequent Symmetry did not support this. This might be because the penalty for locking and extra bus traffic is low on the Sequent. It is unclear if the change in scheduling strategy — different woken goals will be picked up in the two methods — had any impact on the performance or not. These two methods have yet to be compared on more modern parallel processors where the importance of reducing bus traffic is more important.

3.1.2. Locks on dependent cells

Locks are used in a dependent cell to allow it to become bound, and also when a task becomes suspended on trying to bind the dependent variable in a consumer position. However, it is not locked when the producer/consumer status of a task is checked for, as locking is not needed for correctness. If the task is found to be a consumer, it may need to be suspended. However, as the dependent cell is not locked, and a finite amount of time will elapse before the suspension can take place, it is possible for the suspension to become incorrect: the dependent variable may become bound in the meantime, or the task could become the producer before it can be suspended. One way around this is to lock the dependent cell when the producer status is checked for, and if suspension is required, the lock would not be released until the suspension has occurred. This is not desirable, because it would introduce many unneeded lockings and, because in some cases the time interval between the status checking and suspension can be relatively long, the lock can be held for a long time. In addition, adding such locks would be quite complex, because the status checking and suspension are quite separate activities and do not always occur together, and indeed even if they do occur together temporally during the execution, they may be in very different parts of program textually, making the writing of the locking code more difficult.

Thus, when the lock for a dependent cell is acquired for suspension, it is checked to make sure that it is still unbound, and that it is still in a consumer position for the task. A full check for producer/consumer status can be relatively expensive, because it contains a potentially unbound cost component for finding which ancestral and-goal that the current task is under. However, this information is retained from the original status check and do not need to be performed again for the verification check during suspension, and so the cost for checking in these cases are quite low. Note that a side benefit of this approach is that it reduces the amount of actual suspensions. If the verification check fails, normal execution of the task can continue without suspension, thus avoiding the cost of a fairly expensive operation.

3.1.3. Locks on markers

A Marker is locked when some of its field needs to be updated, and also to synchronise certain interactions between different workers on the same stack section and/or (in the case of parcall markers) the same ECGE. In some cases, the same worker would need to lock more than one marker. Note that the initial allocation of a marker does not need to be locked, nor the linking in of the marker to the marker chains on the worker's stack.[8] In order to avoid possible deadlocks, the

[8]Linking the marker into the marker chain does not need locking because in DASWAM, the processing of signals, the only means of following the downward marker chain, is performed by the

convention of acquiring the lock for the older marker before the younger marker is adapted. In some cases, this may mean releasing the lock on a younger marker (which had been acquired earlier), acquiring the lock on the older marker, and then acquiring the lock on the younger marker again.

To reduce the amount of time a lock is held, not all interactions between different workers on a stack section is effected through locking the marker: flags may be used instead. For example, remote backtracking on a stack section may occur at the same time as when the same stack section is being roll-backed due to a killto/kill/redo signal. This is prevented through flags which are set when a marker receives a signal and when a worker is remotely backtracking through a particular stack section. If one flag is set, the other activity will not take place on the stack section.

4. Some optimisations

Some optimisations were applied to the DASWAM system after examining the execution of the parallel DASWAM system. The first two optimisations were only possible because results from the parallel DASWAM system can be compared to the pseudo-parallel DASWAM system. It is expected much more performance is possible with further tuning and optimisations, this include improvements to the existing code, and also improvements due to more sophisticated compilation techniques. For example, it is expected that much of the overhead for dependent variables can be eliminated when there is no possibility that the dependent binding can be unbound during inside backtracking: this sort of optimisation will only be possible with analysis of actual program behaviour.

4.1. Optimising cut

In the initial version of the parallel DASWAM system, on executing some parallel programs, notably the Boyer benchmark, the execution time was unusually slow and very low speedups were obtained. By comparing the actual speedups with that predicted by the simulator, it became obvious that it is due to some performance problem rather than an inherent lack of parallelism for the application. On closer examination of the execution, a performance problem with cut was discovered. The problem occurs in code such as `b:- (b & b), !`, where if `b` is highly recursive, so that many levels of ECGEs can be built up, with a large number of stack sections. When the cut is finally executed, then it has to “tunnel” through many stack sections, trying to determinise them by removing any choice-points. This is a very expensive operation when compared to the sequential cut in the situation, which simply updates one pointer. In addition, as the cut exist in every level of the ECGE, it is executed many times, and the same stack sections are repeatedly and wastefully checked. The repeated execution of a very expensive operation leads to the slow performance; and as each cut is performed by one worker, it essentially introduces extra sequential work, which results in the low speedups.

This problem was overcome by introducing a new flag to each slot of a parcall marker, which is set when a cut has been executed on it. Thus, when a second cut is executed, it does not need to “tunnel” into this particular and-goal. In effect, the entire ECGE is tunnelled through only once [performed in many parts by the various cuts], and while this is still much more expensive than the sequential cut, it is performed only once and thus becomes quite insignificant when compared to the overall execution time. This optimisation resulted in better performance, and increased the speedup. A comparison of the execution of Boyer before and after this optimisation (the speedups are generated from the best execution time in each case) is shown in figure 1. with the speedups normalised to the one worker unoptimised case being 1. The timings from which these speedups are

worker owning the stack: the same worker that will perform the link, so these two events cannot occur simultaneously and thus locking is not needed.

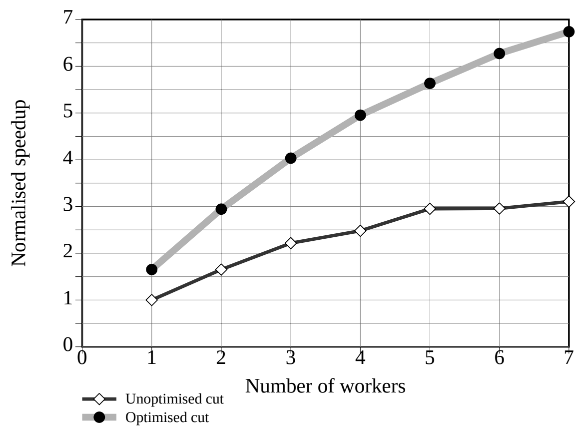

Figure 1: Comparison of optimised and unoptimised cut for Boyer

computed was performed some time ago, and unlike the results presented in [7] and the result in the next section, the timing measurement is from the start of the execution until the very end (the newer results measures time interval from the start until the return of the last solution), including the time to shutdown the whole DASWAM system, and also the final backtracking time, which can be (and is in the case of Boyer) expensive. Thus the optimised speedups of Boyer are not as good as reported in [7]. This final backtracking time is not important because it is after the return of the last solution, but it can be important if the program performs much backtracking before returning the last solution. There are certainly opportunities for optimising this backward execution, and work on this has already begun, as discussion in section 4.3..

4.2. Optimising starting of woken work

Another program that exhibited problems was the DAP version of kkqueens. Initially, the speedups obtained from this program were very low when compared to the predicted speedups from the pseudo-parallel system. On examining the program (by timing individual instruction execution times), it was discovered that a long, essentially sequential, tail of execution exists in the program. It was discovered that the problem is made worse because at the end of execution, a lot of suspended tasks are woken because they have become the producer for the variable they were suspended on. This is detected by the system as a worker finishes a task that was the producer for a dependent variable it did not bind, and finds that there are sibling and-tasks suspended on the dependent variable. The leftmost suspended task has now become the producer, so it has to be woken. Originally, these woken and-tasks are all pushed onto the worker's woken work queue, and the worker itself then becomes idle. This is somewhat inefficient, and an improvement was made whereby the worker, instead of idling, actually retains one of these woken task and starts working on it without putting it onto the woken work queue. This change improved the speedup for the DAP kkqueens benchmark, as shown in Figure 2, although it did not solve the more fundamental problem: the re-execution of the instruction the task was suspended on seem to take an order of magnitude longer to execute than the average instruction, as discussed in [7].

It is expected that further improvements to the performance of the type outlined above would be possible when the executions of programs are examined in more detail.

Figure 2: Comparison of old and new methods for dealing with woken leftmost task for KKqueens

4.3. New backward execution implementation

Currently, the backward execution code for DASWAM is being modified to make it more efficient and robust. The existing code is a straight adaptation of the pseudo-parallel code, with only a minimal amount of locks added, so that the execution is not very efficient. In addition, the code can generate excessive amounts of 'kill'/'redo' signals, which requires unnecessary processing and synchronisation. On examining this code, it was decided that the number of signals sent could be reduced, and the signals themselves simplified, and the locks made to be more specific. The changes will be briefly discussed in this section. The following discussion assumes some familiarity with the old code, and it is beyond the scope of this paper to describe this in detail. The reader is referred to [4] for details of the old code.

The new backward execution code differs from the old one only at the implementation level; the high-level view of the backward execution scheme stays the same. This new code arose from the complexity that the old code would lead to if the locks were made more specific to allow more parallelism during backward execution. The main differences from the old code are:

- Less signals are sent: in the old code, kill/redo/killto signals are sent regardless of if the particular stack section has been sent another signal already. It is possible for a particular stack section to receive multiple kill/redo/killto signals, e.g. a task could successively bind several dependent variables created in the same ECGE, and these variables' values can then be consumed by another task. When a failure occurs in the producer task, and these dependent bindings are undone, this can lead to successive sending of the 'redo' signal to the same consumer – these signals are to cause rollbacks on the same stack section, but the rollback needs to be performed only once. All these extra signals have to be processed, recognised as being redundant and no rollback performed (effectively, these signals needed to be filtered). This involves extra work and, in addition, extra synchronisation (delay of actions after the rollback) to be certain that these extra signals can be filtered out correctly. In the new implementation, the extra signals are filtered out before it is sent: when the marker associated with a particular stack section which is to be rollback is sent a signal (kill/redo/killto), a flag is set indicating which signals have been sent to the stack section. Thus, before a signal is sent, the associated marker is checked to see if a signal has already been sent, and if so, no new signal is sent.

- A stack section can be sent multiple signals, but these signals need not be the same, e.g. it can be sent a redo signal, and then a kill signal. In the old code, two signals are sent, with the code for performing kill/killto/redo being separate, and tests needed at the end of processing the signal to see if the stack section received a different signal (for example, if a stack section is sent a redo then a kill signal, that stack section should not be redone). In the new code, the signal is sent only once, but its status — redo/kill/killto — as indicated by the flag on the marker, may be changed subsequently. Note that the different types of signal have different 'priorities', so that some type of signals will always override others. The order is killto > kill > redo, i.e., a stack section that has been sent a redo signal (redo is sent when a dependent binding is undone, so that the speculative work done by a task has to be undone and the started started again) can be replaced by a kill signal (kill signal is sent when a task is to be abandoned entirely), and both can be replaced by a killto signal (sent when a whole region of computation has to be entirely abandoned because an ancestral ECGE has been sent kill/redo signals). Thus, a marker marked with 'redo' can be replaced by 'kill' and 'killto', and a marker marked with 'kill' can be replaced by 'killto'. Signals of the same or lower priority type will not need to be sent.[9] The code for the three signals are merged into one, with different actions taken at the end of a rollback, depending on the status of the marker; 'redo' means that the task will be restarted, 'kill' means that the task will not be restarted, and 'killto' means that the rollback would continue to a previous stack section. This simplifies the code, and should also reduce the signalling overheads.
- In the old code, one item of information is sent with kill and redo signals: the address of the marker that the signal is to be applied to. For killto, two items are sent: the marker's address, and the address of the marker that the rollback needs to continue back to (the target marker). However, this second item of information is not needed, and in addition can become inaccurate (for example, another killto can be sent with an "earlier" target). The target marker is not needed because a killto signal is always propagated as a result of the processing of another signal, so at the target marker must have been sent a kill/redo signal which are then propagated to other stack sections by killto signals. Thus, rollback can stop when a marker that is marked with either a kill or redo (instead of killto) is reached. This also simplifies the case of multiple signal, when a 'target' marker that caused killto be sent to its descendant stack sections is itself sent a killto. The old code had to do extra testing for such situations (including the possibility of multiple killtos), and all these complexities are simply avoided in the new code: the status of the target marker can be updated by subsequent killtos. This also means that the signal for killto has been simplified to sending only one item like the kill/redo signals (and indeed this is necessary for the same code to be used to process all three signals).

These changes should lead to better performance overall, but in addition, to code that is more robust as it actually simplifies the scheme that deals with signals: the complex race conditions in the old code should not arise. However, it does involve substantial modification of the existing code, and as for this writing (August 1996), this modification is still being carried out.

5. Conclusion

This paper briefly outlined some of the implementation issues found in the parallel DASWAM system, and how these issues are tackled currently. As shown in this paper, the current system is

[9]Notionally, a 'killto' that is to kill a greater region of computation has higher priority than one that kills a smaller region, but because the killto point is not sent with the signal (see next item), no new signals has to be sent even in this case.

a rather straight-forward modification of the pseudo-parallel system, and a very stable system that has reasonable performance was achieved in a short-time. The purpose of this paper is to document the major changed that were needed, and some of the issues involved, especially in the area of portability. In addition, some of the early efforts in optimisations are presented, and shown to be quite effective. It is expected than much more significant improvements will be possible with further optimisations.

References

[1] J. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Heriot-Watt University, 1988.

[2] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, The University of Texas At Austin, 1986.

[3] K. Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731. The MIT Press, 1992.

[4] K. Shen. *Studies of And/Or Parallelism in Prolog.* PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[5] K. Shen. Implementing Dynamic Dependent And-parallelism. In *Logic Programming: Proceedings of the Tenth International Conference*, pages 167–183. The MIT Press, 1993.

[6] K. Shen. Improving the Execution of the Dependent And-parallel Prolog DDAS. In C. Halatis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 438–452. Springer-Verlag, 1994. Published as Lecture Notes in Computer Science 817.

[7] K. Shen. Initial Results from the Parallel Implementation of DASWAM. Accepted at Joint International/Symposium of Logic Programming, 1996.

[8] K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 1996. In Press.

[9] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025, USA, 1983.