Experiments with speculative parallelism in Parlog

Steve Gregory

Department of Computer Science
University of Bristol
Bristol BS8 1TR, England

steve@cs.bris.ac.uk

April 1993

Abstract

One way to execute a program faster on a parallel computer is to allocate some processors to speculative computation: work which may prove to be unnecessary. However, this technique, *speculative parallelism*, requires the programmer to have some control over the physical processors in the machine, and is therefore not usually possible in high-level concurrent languages.

This paper investigates the exploitation of speculative parallelism in concurrent logic programming languages. We propose a simple language extension that makes the technique possible, and illustrate its use in two applications: (1) the familiar branch-and-bound search algorithm, and (2) a new *dynamic buffer* protocol for interprocess communication. The new language feature has been implemented by extending the JAM Parallel Parlog system. The results of our experiments confirm that speculative parallelism can result in greater parallel speedups than is possible with other (divide and conquer) forms of parallelism alone.

Keywords: Concurrent logic programming, speculative parallelism, search algorithms.

1 Introduction

1.1 Speculative parallelism

Concurrent programs, like sequential programs, are normally written so as to minimize the amount of computation that they perform. They do this by synchronizing their activities so that each is executed only when all information that it needs is available. This strategy is effective on a uniprocessor machine, since less computation leads to a smaller execution time. On a parallel computer, however, the synchronization within the program may restrict its degree of parallelism, so that some processors remain unoccupied. In these circumstances, it is sometimes worth using the idle processors to execute some activities earlier than the optimal time, even though more computation may then be required. This technique — reducing execution time by possibly *increasing* the amount of computation — is known as *speculative parallelism*.

A popular application of speculative parallelism in artificial intelligence is the parallel traversal of search trees and game trees. For example, sequential best-first search algorithms repeatedly extend the single "best" branch of a search tree (according to the value of some heuristic function), which is the most likely to lead to a solution. To parallelize best-first search, given N processors, the best N branches must be extended concurrently. Since, in general, this means extending suboptimal branches, the chances are that more of the tree will need to be searched before a solution is found. Nevertheless, provided the increase in work is not too great, a good speedup can still be obtained.

1.2 Speculative parallelism in Parlog

To allow speculative parallelism to be exploited, a programming language must provide some control over the actions of processors: a program should behave differently according to the number of processors available. In other words, the language must provide some way to allocate processes to processors. Most work on speculative parallelism, e.g., (Leifker and Kanal, 1985; Kumar *et al.*, 1988), has been done using processes that are *statically* mapped to processors; for example, in a parallel best-first search, each processor executes a loop which repeatedly takes the best available node from the tree, expands it, and replaces the successor nodes in the tree.

Most high-level concurrent programming languages, including concurrent logic programming languages, deliberately hide the processors from the programmer: a program can dynamically create an arbitrary number of processes, which are automatically allocated to processors by a scheduler. This makes programming much easier, and increases portability between machines with different numbers of processors, but it does prevent the exploitation of speculative parallelism.

To make use of speculative parallelism in concurrent logic programming, some additional language feature is necessary. This should provide just enough control over the allocation of processes to processors without adversely affecting the normal dynamic scheduling of the language. Huntbach (1991a; 1991b) has proposed a *priority pragma* for this purpose: an integer priority, which defaults to infinity, is associated with each goal (process), and the scheduler executes goals with higher priority first. We have experimented with a simpler feature which, though less powerful, is sufficient to enable the exploitation of speculative parallelism. The new feature and its implementation are explained in the next section; Sections 3 and 4 describe some experiments in using it for various applications; while the final section gives some conclusions, including a comparison between our method and Huntbach's approach.

All references to Parlog in this paper apply equally to other concurrent logic programming languages (KL1, Strand, etc.).

2 Extensions to Parlog

We propose extending Parlog with a new built-in relation, `wait_idle/0`. `wait_idle` has no logical significance and can be ignored in the declarative reading of a program. Procedurally, a goal for `wait_idle` suspends until *at least one* processor is idle, and then succeeds; it never fails.[1]

In programs intended to exploit speculative parallelism, the speculative computation should be divided into "chunks", each preceded by a `wait_idle` goal. Each chunk will then only be executed when no mandatory computation is available. Once a chunk of speculative computation is begun, its execution will continue until the next call to `wait_idle`, even if new mandatory computation becomes available. Therefore, the chunks of speculative computation should not be too large. By adding low-priority processes to do speculative work in this way, a program can be guaranteed to perform no worse than, and possibly better than, the corresponding sequential program.

We have implemented `wait_idle/0` by modifying the JAM Parallel Parlog system, an implementation of the language for shared memory Unix multiprocessors (Crammond, 1988). The JAM system comprises a fixed number of Unix processes, which we shall call *workers* to distinguish them from Parlog processes. Each worker has its own *run queue*, a queue of executable Parlog processes. A worker repeatedly takes a process from the front of its queue, executes it, and adds any spawned body goals to the rear of its queue. If a worker's run queue becomes empty, it may take processes from the run queue of any other worker; if they all appear to be empty, the system checks for the possibility of deadlock. The algorithm that finds a process for a worker to execute is the *scheduler*.

Two main extensions were made to implement `wait_idle`:

1. A new global run queue (which we call the *low queue*) was introduced to contain low-priority executable processes. The low queue is initially empty, but any process that executes `wait_idle` is added to the queue.
2. The scheduler was modified to use the low queue if other (high priority) processes cannot be found. The modified scheduler repeatedly tries each of the following steps in the given order until a process is found or deadlock is detected. The fourth step (in italics) is new:
   (a) If timeslice, take a process from front of local run queue;
   (b) take a process from rear of local run queue;
   (c) take a process from some other worker's run queue;
   (d) take a process from low queue;
   (e) check for deadlock.

It would be possible to distribute the low queue among workers in the same way as the normal run queue, to avoid the need for some locking and to reduce contention. However, the increased complexity was not felt to be justified since the low queue is likely to be modified much less frequently than the normal one.

Note that, for `wait_idle` to operate correctly, we assume a one-to-one correspondence between worker processes and physical processors. In practice, the operating system of the host machine might dynamically schedule processes to processors, in which case our assumption is justified only if the machine is lightly loaded; the results below were obtained in such conditions.

---

[1] `wait_idle/0` may be compared with `deadlock/0`, proposed as a means of handling deadlock in Parlog (Broda and Gregory, 1984): a `deadlock` goal suspends until *all* processors are idle, and then succeeds. `wait_idle/0` and `deadlock/0` are equivalent on a uniprocessor machine.

3 Application: parallel A* search algorithm

3.1 The sequential A* algorithm

A* is a general best-first branch-and-bound search algorithm. The standard sequential version (Nilsson, 1980)[2] is shown in Program 1. It makes use of a list *Open*, which contains the generated leaves of the search tree (initially only the start node *S*) and searches until a solution node *N* is generated.

Open := [S];
while Open ≠ []
    remove N, the best node from Open;
    if solution(N) then return success(N);
    find successors of N; (*)
    insert successors in Open;
return fail.

Program 1: Sequential A* algorithm

The "best" node is defined as one which minimizes a function f, where $f(N)$ is defined as $g(N)+h(N)$. $g(N)$ is the actual cost of a path from the start node to N, while $h(N)$ estimates the cost of a path from N to a solution. Provided $h(N)$ is no greater than the actual cost from N to a solution, the algorithm is guaranteed to find an optimal solution, if one exists.

This algorithm contains potential parallelism within the step which expands a node (*). The extent of this parallelism depends on the problem, but is often limited. The only way to obtain a better parallel speedup is to modify the algorithm to introduce speculative parallelism.

3.2 The parallel A* algorithm

The parallel A* algorithm, described in (Leifker and Kanal, 1985; Quinn, 1987), introduces several *solver* processes executing essentially the same "remove–expand–replace" loop as the sequential algorithm. The main difference is that the termination criterion needs to be modified because (a) an empty Open list does not necessarily signal failure, and (b) the first solution found is not necessarily the best. The changes required are:

1. A record is kept of the best solution so far, which is updated whenever a solution is found.
2. The Open list is allowed to contain only nodes that are better than the best solution so far.
3. A solver trying to remove a node for expansion suspends if the Open list is empty.
4. The program terminates only if the Open list is empty *and* all solvers are waiting to remove nodes.

Alternative versions of the algorithm have been suggested (Kumar *et al*., 1988), which avoid the potential sequential bottleneck of the shared Open list by distributing it among the solver processes. For simplicity, we shall not consider the more elaborate algorithms in detail.

[2] The version of (Nilsson, 1980) actually used a search *graph* rather than a tree. Here we use the tree representation, partly for simplicity and partly because our application does not benefit significantly from the use of a graph.

3.3 The parallel A* algorithm in Parlog

The above algorithm can be implemented in Parlog by a number of solver processes, each executing the "remove–expand–replace" loop, communicating with a single *server* process, which manages the Open list and records the best solution so far, as shown in Figure 1. The solvers send messages to the server to add nodes to, and remove nodes from, the Open list. The server sends a node in reply to each `remove` message that it receives, sending an `end` reply instead when it detects termination.

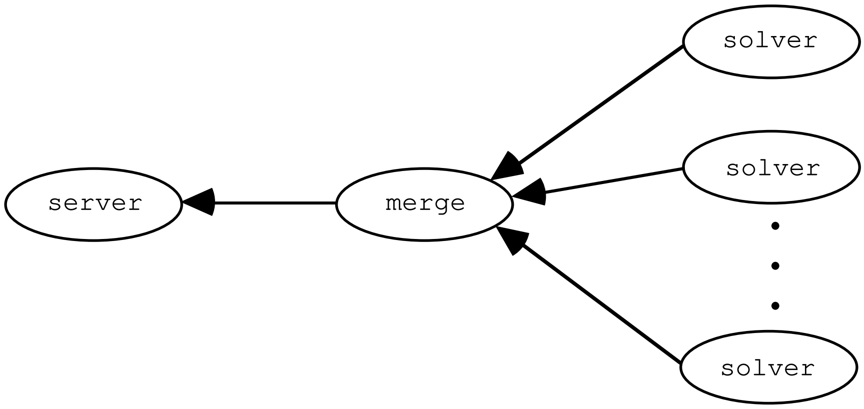

Figure 1: Parallel A* program structure

Program 2 shows a Parlog definition of a solver process, named `seq_solver`. Each tree node is represented by a term `o(N,F/G)`, where `F` and `G` are the values of f(`N`) and g(`N`), respectively. The other predicates that are used are `s(N,M)`, which gives a list `M` of the successors of node `N` with their costs, and `h(N,H)`, which gives the value of h(`N`).

```
mode seq_solver(^).
seq_solver([remove(Node)|Rest]) <-
    seq_solver1(Node,Rest).

mode seq_solver1(?,^).
seq_solver1(end,[]).
seq_solver1(o(N,_/G),Rest) <-                    % G is cost of N
    s(N,M),                                      % M is list of N's successors
    add_nodes(M,G,Rest,Rest1),
    seq_solver(Rest1).

mode add_nodes(?,?,^,?).
add_nodes([(M,C)|Nodes],G1,[add(o(M,F/G))|Rest],Rest1) <-
    h(M,H),                                      % H is heuristic estimate for M
    G is G1 + C,                                 % C is cost of arc from N to M
    F is G + H,
    add_nodes(Nodes,G1,Rest,Rest1).
add_nodes([],_,[done|Rest],Rest).
```

Program 2: Parlog program for A* solver process

The problem with this program is that it is not an effective way to exploit speculative parallelism: the programmer must determine in advance the number of `seq_solver` processes that exist. If there are too few (e.g., one, which corresponds to the sequential A* algorithm of Program 1), the parallelism will be insufficient; if there are too many, unnecessary work will be done expanding suboptimal nodes. Creating as many solvers as there are processors is not the answer, since processors may be more usefully employed in exploiting the “mandatory” parallelism in expanding nodes. The optimum number of nodes that should be expanded at any time is, in general, unpredictable and varies dynamically throughout the execution.

Consider instead an alternative definition of the solver process, named `par_solver`, in Program 3. This process looks similar, but it demands the next node to expand immediately after the previous node arrives, and thus expands all generated nodes in parallel. This program therefore actually performs a breadth-first search and fails to benefit from heuristics.

```
mode par_solver(^).
par_solver([remove(Node)|Rest]) <-
   par_solver1(Node,Rest).

mode par_solver1(?,^).
par_solver1(end,[]).
par_solver1(o(N,_/G),Rest) <-
   s(N,M),                                      % Expand node N…
   add_nodes(M,G,Rest1,[]),                  % …and add successors
   par_solver(Rest2),                  % Expand more nodes in parallel
   merge(Rest1,Rest2,Rest).
```

Program 3: Breadth-first solver process

The `par_solver` process of Program 3 is inadequate, but it is easily fixed to effectively exploit speculative parallelism. All that is needed is to make the process wait until processors are available before it demands the next node to expand. This can be done by calling `wait_idle` before sending each `remove` message. The final, “dynamic”, `par_solver` process is shown in Program 4. The program is run by the query `?- par_solver(Ms), server(Ms).`

```
mode par_solver(^).
par_solver(Rest) <-
   wait_idle &
   Rest = [remove(Node)|Rest1],
   par_solver1(Node,Rest1).
```

Program 4: Dynamic solver process

3.4 Results

In this section we report on a series of experiments to confirm that the parallel A* program that uses our new language extension behaves as expected and, in particular, shows real speedups due to speculative parallelism. All experiments were run on a lightly loaded 12-processor Sequent Symmetry multiprocessor.

The performance of the program depends on the nature of the problem it is used to solve. We have tested it on two problems: the 8/15-puzzle and the travelling salesman problem. In the 8/15-puzzle, it is a trivial matter to generate the successors of a node, so the execution time is dominated by the sequential processing of the Open list and the parallel speedup is limited. (Incidentally, for the same reason, the mandatory parallelism is also negligible.)

The travelling salesman problem (TSP) is a more sizeable application with greater scope to benefit from parallelism. We have implemented the algorithm of (Little *et al.*, 1963) in the A* framework, by defining the `s/2` and `h/2` procedures suitably. In this algorithm, each node of the search tree contains a cost matrix (of city-city distances) and a partial tour (initially empty). Every node has two successors: one which adds a promising link (a pair of cities) to the partial tour and another which eliminates the same link. The work involved in generating the successors is quite substantial, especially near the root of the tree. We have tested the program on randomly generated TSP instances with up to 25 cities. For example, Figure 2 shows the search tree (generated by sequential A*) for a 20-city instance of the TSP.

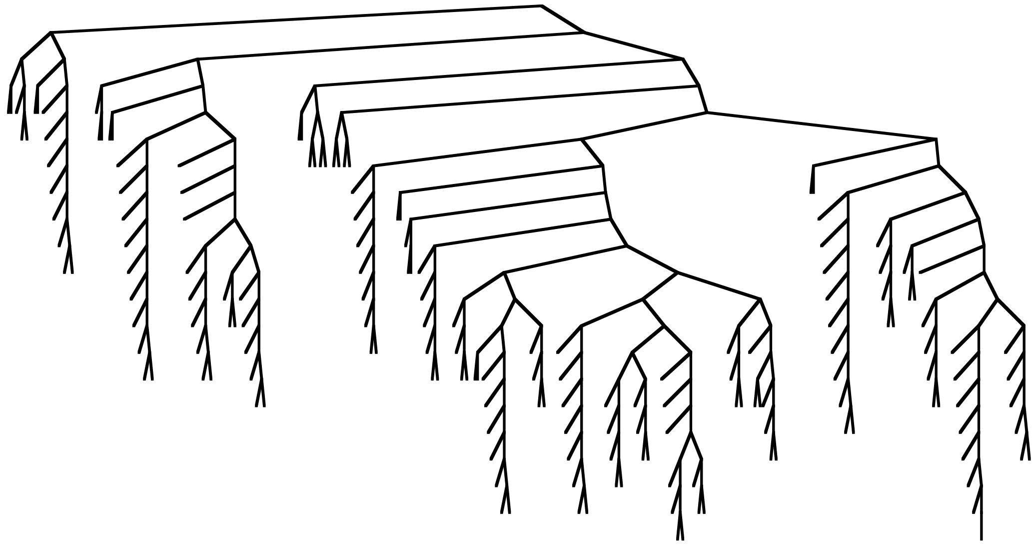

Figure 2: A* search tree for a 20-city instance of TSP

3.4.1 Overhead of program structure

For any parallel speedup to be worthwhile, a parallel program should not run on one processor significantly slower than a program tailored for sequential execution. To check the overhead of our parallel A* program, we measured the execution time on one processor of three versions of A* applied to a 20-city instance of the TSP:

Version 1: the sequential A* algorithm of Program 1 coded in Parlog.

Version 2: a concurrently structured Parlog program consisting of a `server` process and one `seq_solver` process (Program 2); this behaves like sequential A* on any number of processors. The overhead, compared with version 1, is due to the more complex parallel A* algorithm and the communication between solver and server processes.

Version 3: a concurrently structured Parlog program consisting of a `server` process and a `par_solver` process (Program 4); this is the dynamic parallel A* program. The overhead, compared with version 2, is due to the use of `wait_idle`.

The results, in Table 1, show that the difference in speed is negligible: less than 2% overall.

	Version 1	Version 2	Version 3
Execution time	160.96s	162.43s	163.84s

Table 1: Measuring overhead of parallel A* program

3.4.2 Speedup due to speculative parallelism

As noted above, there is potential mandatory parallelism in the expansion of a node, which is “transparently” exploited by Parlog. To enable us to focus on speculative parallelism, we first removed all mandatory parallelism by using sequential operators (‘`&`’ and ‘`;`’) throughout the node expansion code. Tests confirmed that the program implementing the sequential algorithm (Version 2 of Section 3.4.1) exhibited no noticeable speedup when run on multiple processors.

The program implementing the parallel algorithm (Version 3 of Section 3.4.1) was run on several randomly-generated instances of the TSP (10, 15, 20, and 25 cities) on different numbers of processors. Table 2 gives the execution times on one processor, while Figure 3 graphically shows the parallel speedups obtained.

	10 cities	15 cities	20 cities	25 cities
Execution time	4.40s	19.10s	162.43s	381.22s

Table 2: Sequential execution time for TSP

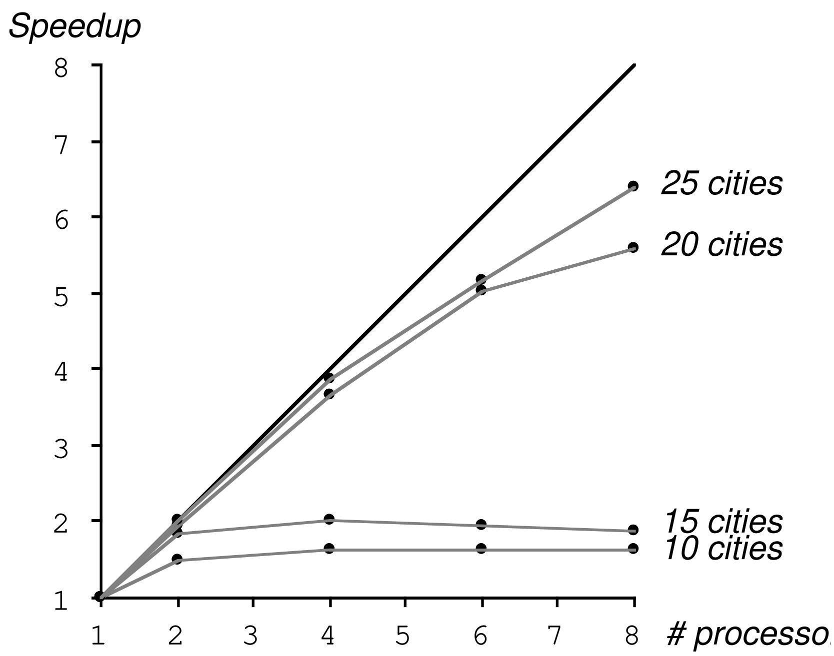

Figure 3: Speedups for TSP with speculative parallelism

Figure 3 shows good parallel speedups for the larger problems. However, there are two independent factors which conspire to limit the parallel speedup that can be achieved:

1. The number of branches in the search tree, after pruning, may be insufficient to usefully occupy many processors. Then the excess processors will resort to more speculative work, increasing the total amount of computation performed. Table 3 shows the number of nodes expanded for each of the problems run on different numbers of processors, while Figure 4 shows the difference between the search trees developed for the 10-city example run on 1 processor and on 8 processors.
2. There is some inherent sequentiality in the search. This has two causes:
   (a) Fewer nodes are available for expansion, and potential speedup therefore less, at the begin and end of the computation. For example, at the beginning, only the start node can be expanded, then only its successors are available, and so on.
   (b) The insertion of a node into the Open list (a priority queue) is a time-consuming, sequential, operation. Any solver process that becomes idle while a node is being inserted in the list will be delayed until the operation is complete.

Both of these effects are clearly visible from the gauge in Figure 5, which shows the number of nodes being expanded at any time during the solution of the 20-city TSP on 8 processors. (The program was instrumented to obtain the statistics for Figure 5, and so its performance differs slightly from the basic program used for speed measurements.)

Processors:	1	2	4	6	8
10 cities	17	18	24	29	33
15 cities	27	27	40	56	72
20 cities	181	181	174	170	171
25 cities	239	238	238	238	242

Table 3: Number of nodes expanded for TSP

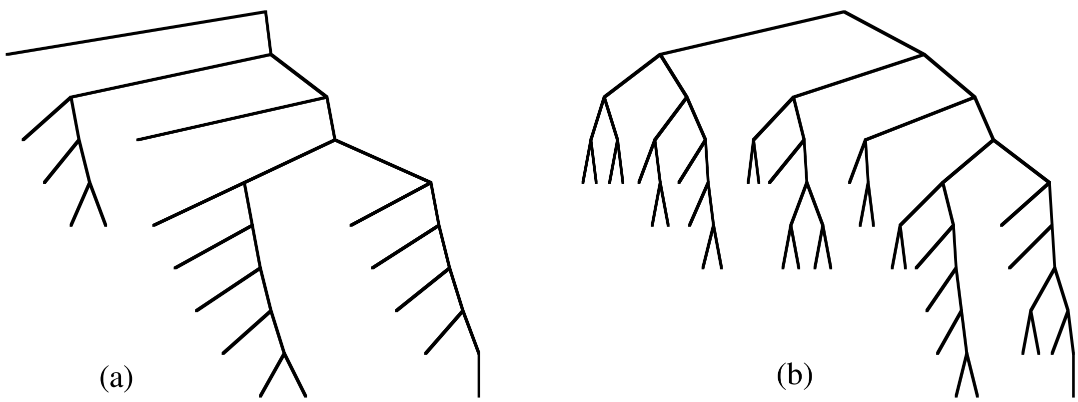

Figure 4: Search trees for 10-city TSP — (a): 1 processor; (b): 8 processors

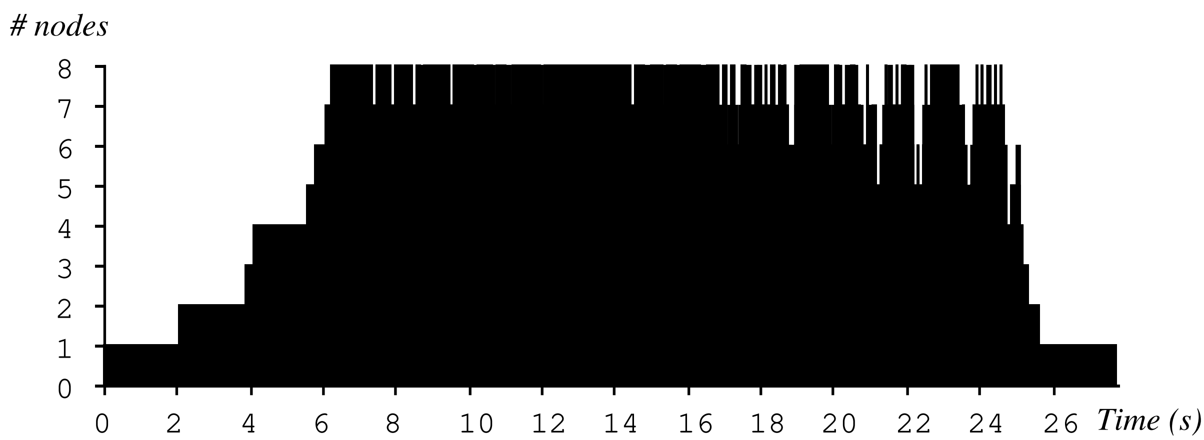

Figure 5: Number of nodes being expanded for 20-city TSP on 8 processors

3.4.3 Speculative vs. mandatory parallelism

The experiments reported above were done in the absence of mandatory parallelism, in order to focus on the effects of speculative parallelism. They show that our dynamic A* program, using the `wait_idle` primitive, automatically keeps all of the processors occupied with expanding nodes. The effect here is almost the same as statically creating a number of solver processes equal to the number of processors. However, the real *raison d'être* of our dynamic language extension is to effectively exploit speculative parallelism alongside the mandatory parallelism that is transparently exploited by Parlog: if some processors are busy executing mandatory computation, correspondingly less speculative work should be done.

The TSP algorithm is a good example to test the interaction of speculative and mandatory parallelism, since the work of expanding a node consists of several matrix operations which contain substantial parallelism. To check whether speculative parallelism is still worthwhile, we ran three versions of the A* program to solve the 20-city TSP on different numbers of processors:

1. *Speculative parallelism only*. This is the parallel A* program (Version 3 of Section 3.4.1) with mandatory parallelism removed from the node expansion code.
2. *Mandatory parallelism only*. The sequential A* program (Version 2 of Section 3.4.1) with full mandatory parallelism in the node expansion.
3. *Speculative and mandatory parallelism*. The parallel A* program (Version 3 of Section 3.4.1) with full mandatory parallelism in the node expansion.

Parallel speedups for the three versions are plotted in Figure 6. This clearly shows that, at least for this example, a combination of speculative and mandatory parallelism achieves a greater parallel speedup than either form of parallelism alone.

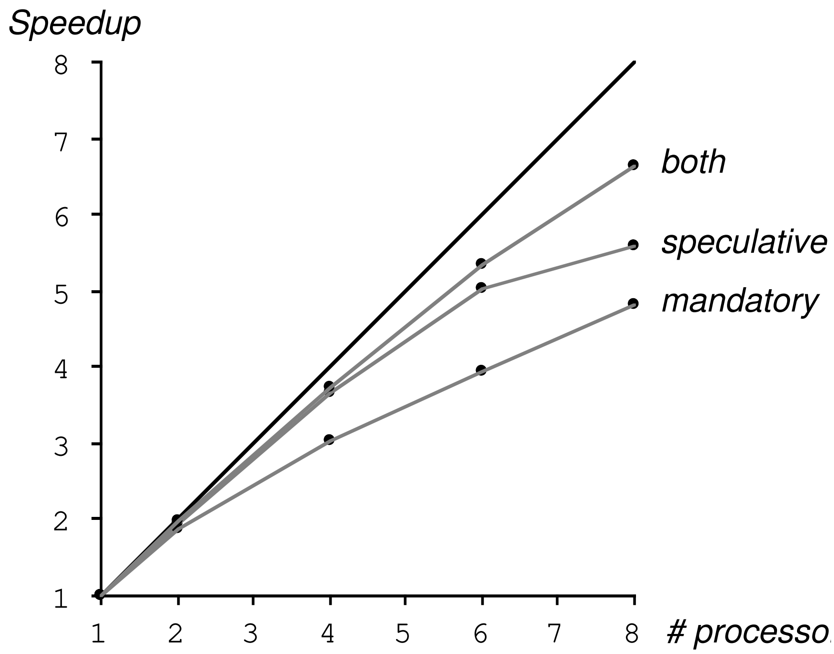

Figure 6: Comparing and combining speculative parallelism and mandatory parallelism — 20-city TSP

3.5 Fairness

Standard Parlog implementations are fair, in that every runnable process is guaranteed eventually to make progress. Using `wait_idle` destroys fairness by introducing two priority levels for processes: low-priority processes may be overtaken indefinitely by high-priority processes. For example, our parallel A* program would be starved by any other (mandatory) computation run in parallel. Fortunately, there is a simple solution which avoids this problem: we can add a sequential `seq_solver` process (Program 2) in parallel with the speculative search. The search will now make progress even if the program contains many other mandatory processes — in which case it will behave like sequential A* — while it will exploit speculative parallelism if spare processors are available:

```
?- seq_solver(Ms1), par_solver(Ms2), merge(Ms1,Ms2,Ms), server(Ms).
```

4 Application: dynamic buffer

Parallel search is an important application of speculative parallelism, but it is not the only one. In this section we present a new technique for interprocess communication in Parlog, the *dynamic buffer*, which allows the same principle to be applied to any pipeline-structured program.

4.1 Synchronous and asynchronous communication

Interprocess communication in Parlog is naturally asynchronous: where two processes communicate stream data via a shared variable, the producer may run ahead of the consumer. This has two potential drawbacks: (1) data that has been produced but not consumed has to be stored in memory; (2) effort may be wasted on producing data that is never required. Therefore, most practical programs use either *synchronous* communication, where the producer waits for an acknowledgement from the consumer before producing the next item, or *lazy* communication (Gregory, 1987), where the producer generates only what the consumer demands.

Synchronous communication may be programmed in Parlog using incomplete messages. Instead of sending a message *M*, the producer sends a pair (*M*,`Ack`) and then waits for `Ack` to be bound before continuing; the consumer binds `Ack` to `ok` when it consumes *M*. For example, Program 5 shows part of a synchronous program for the Eratosthenes (primes) sieve algorithm, illustrated in Figure 7. The `Stop` argument allows the consumer to terminate the (otherwise infinite) producer.

```
mode primes(^,?).
primes(Primes,Stop) <-
   integers(2,Ints,Stop),
   sift(Ints,Primes).

mode integers(?,^,?).
integers(N,[(N,Ack)|Ints],Stop) <-
   N1 is N+1,
   integers1(Ack,N1,Ints,Stop).

mode integers1(?,?,^,?).
integers1(ok,N,Ints,Stop) <- var(Stop) :
   integers(N,Ints,Stop).
integers1(_,_,[],stop).
```

Program 5: Synchronous primes sieve program

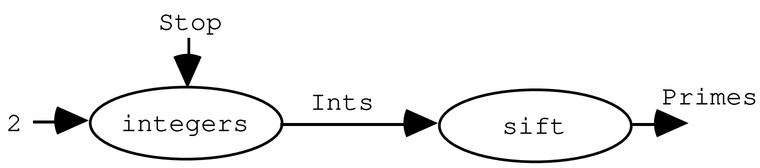

Figure 7: Synchronous primes sieve

Synchronous (or lazy) communication avoids the drawbacks noted above, but tends to restrict parallelism. For example, in the primes sieve program, each integer has to pass along the whole pipeline of processes that constitute `sift` before the next is generated, preventing them from working in parallel.

4.2 Bounded buffers

A standard compromise between synchronous and asynchronous communication is the bounded buffer: a fixed-size buffer is placed between producer and consumer, so that the producer can run ahead of the consumer by a fixed amount, until the buffer is full. Takeuchi and Furukawa (1985) first showed how a bounded buffer could be defined in a concurrent logic programming language, as a process interposed between two processes communicating synchronously as outlined above; see Figure 8. The `buffer` process here passes items that it receives on to its output; it acknowledges the first *Size* items that it receives and then waits for the *N*th item to be acknowledged by the consumer before it acknowledges the (*N*+*Size*)th item, and so on.

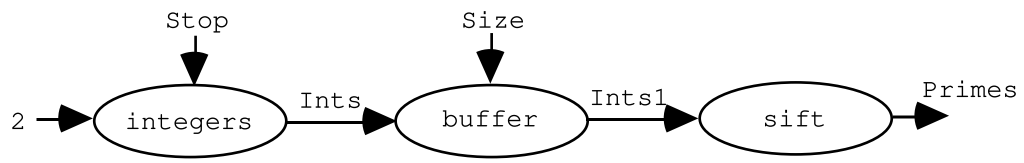

Figure 8: Use of bounded buffer process

Bounded buffers work effectively provided the size is chosen appropriately. However, if the buffer is too small or too large, the disadvantages of synchronous or asynchronous communication may reappear.

4.3 The dynamic buffer

Using our `wait_idle` primitive, we can define a process that acts like a bounded buffer which automatically adjusts its size according to the processing resources available. If all processors are busy, communication will be synchronous; if processors are idle, the producer will be allowed to run ahead. This `dyn_buffer` process (Program 6) is added to a synchronous producer/consumer program as shown in Figure 9: it monitors the communication stream and acknowledges the next item whenever idle processors are available.

```
mode dyn_buffer(?).
dyn_buffer([(_,Ack)|Ints]) <-
   wait_idle &
   Ack = ok &
   dyn_buffer(Ints).
dyn_buffer([]).
```

Program 6: Dynamic buffer process

Figure 10 shows how the buffer size varies dynamically during the execution of the primes sieve to compute the first 100 primes on 4 processors. (As with Figure 5, the collection of statistics slightly distorted the behaviour of the program.) Figure 11 compares the execution times of the primes sieve program with (a) synchronous communication, (b) a bounded buffer of size 200, and (c) a dynamic buffer. The program with synchronous communication gives no parallel speedup, but is faster on one processor than the one using a bounded buffer, because the latter performs unnecessary computation to keep the buffer full.

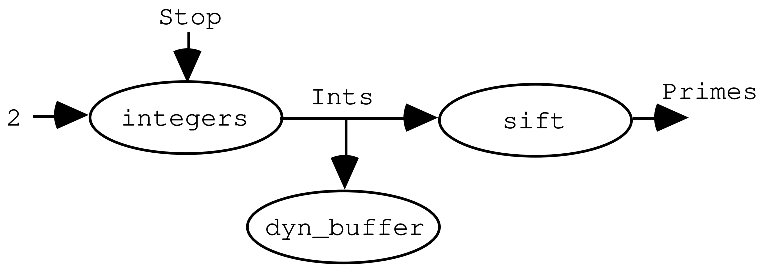

Figure 9: Use of dynamic buffer process

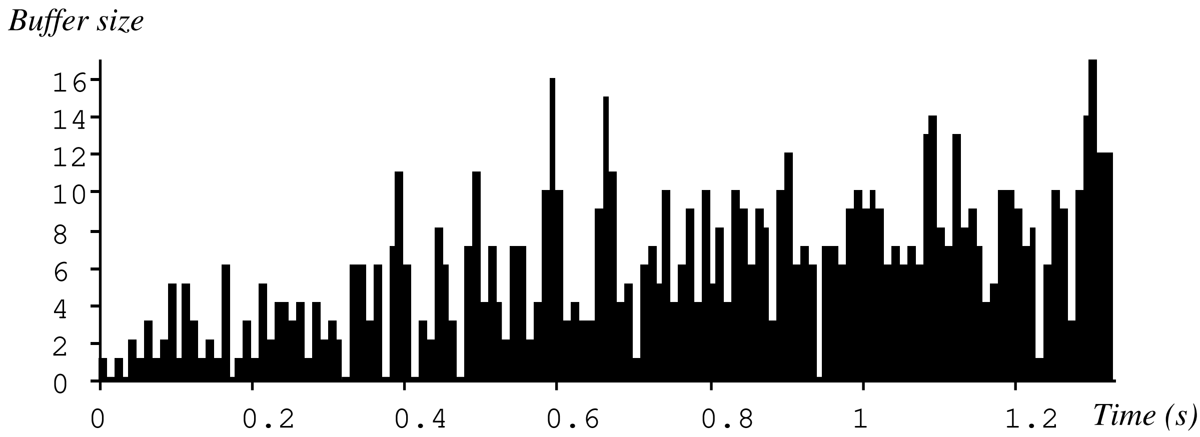

Figure 10: Buffer size for first 100 primes on 4 processors

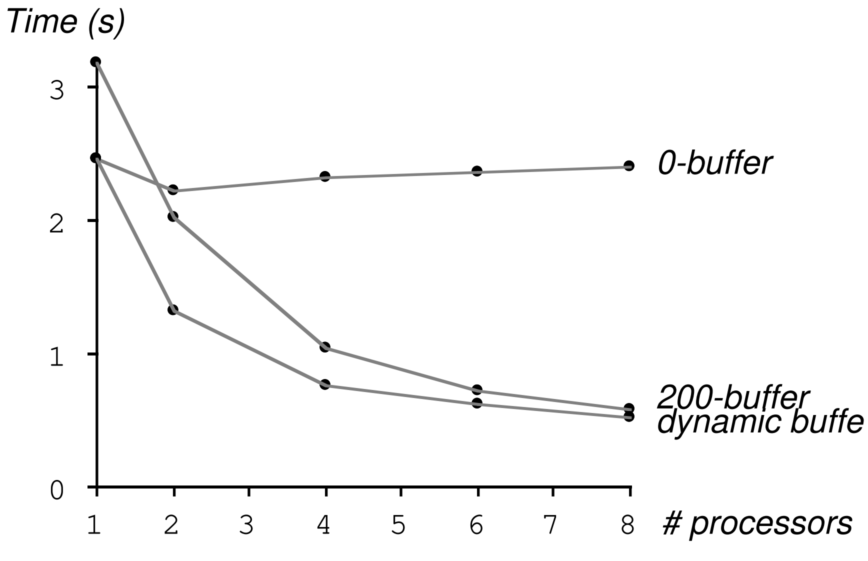

Figure 11: Execution time for first 100 primes — 0-buffer, 200-buffer, and dynamic buffer

The program with a dynamic buffer is faster than both of the others. It is as efficient as (in fact, equivalent to) synchronous communication on one processor, and consistently faster than the program that uses a bounded buffer, which is too large for the problem and wastes computation effort. Admittedly, the use of a dynamic buffer yields only a modest improvement in speed for the primes sieve example — reasonable performance could be obtained with a small bounded buffer — but for more realistic applications, where the producer performs substantial work to generate each item, the benefit should be greater.

The dynamic buffer process of Program 6 ensures that a producer process does not waste processor power by running ahead, but it does not avoid the risk of occupying memory with unconsumed data. However, it would be simple to modify the `dyn_buffer` process so that it imposes an upper limit on the number of items produced but not consumed.

5 Conclusion

5.1 Related research

The most closely related work is that of Huntbach (1991a; 1991b), who proposed the implementation of process priorities in Parlog, based on earlier work in functional programming (Burton, 1985). A *priority pragma*, `priority(P,Goal)`, associates an integer priority `P` with `Goal` (a Parlog goal), normal goals implicitly having infinite priority. The priority determines the order in which goals are reduced, but is not inherited by descendant goals.

Huntbach (1991b) has applied his priority pragma to the problem of best-first branch-and-bound search (equivalent to our A* algorithm). The special feature of his program is that there is no explicit Open list data structure. Instead, the search tree is represented by a tree of processes, and each leaf node is represented by a *goal* whose priority equals the heuristic value of the node; this goal will actually expand the node if and when it attains the highest priority. In effect, the run queue of the system provides the priority queue needed to represent the Open list. The priority pragma has been simulated but not been incorporated in a real implementation, and so no performance results are reported by Huntbach.

It is easy to see that Huntbach's priority pragma is strictly more powerful than our `wait_idle` feature: `wait_idle` is equivalent to `priority(`k`,true)` for any finite integer k. The principal advantage of the more general feature is:

- The priority pragma offers a more direct implementation of applications involving many priority levels, e.g., A*. In contrast, a "meta-level", interpretive, approach is required using `wait_idle`. The direct approach can be more elegant and/or efficient.

Potential disadvantages of the direct approach include:

- It can be more difficult to program. For example, in Huntbach's A* program, the processes expanding the leaf nodes need to communicate with a central server process to examine and update the best solution so far, a global variable; this communication is quite complex.
- A computation comprising finite-priority processes can be starved by other, mandatory, processes running in parallel. We can see no simple solution to this problem using the direct approach, unlike the interpretive approach, in which a mandatory process can be added to guarantee fairness (Section 3.5).
- When discussing the implementation of Parlog with the priority pragma, Huntbach (1991b) leaves open the possibility of distributing the run queue across processors in a parallel machine, with some load balancing mechanism giving a reasonable distribution of high-priority goals between processors. For the A* program, this corresponds to distributing the Open list among the processors. However, (Kumar *et al.*, 1988) demonstrate that whether the Open list is centralized or distributed, and the way in which nodes are reallocated among processors, has a profound effect on performance. These decisions should therefore be made by the programmer, which can only be done with a data structure representation of the Open list.

It is worth noting that the idea of associating priorities with processes in concurrent logic programming languages has been suggested elsewhere, usually for the purpose of systems programming, e.g., (Foster, 1990; Ueda and Chikayama, 1990).

An established use of speculative parallelism elsewhere in logic programming is in the or-parallel implementation of Prolog. Or-parallel Prolog systems traverse in parallel a search tree which corresponds exactly to the Prolog program: one branch for each clause. If a program uses cuts, some sequencing is imposed on the tree search; for example, a clause containing a cut must normally be executed as far as the cut before subsequent clauses can be tried. Some or-parallel Prolog systems incorporate schedulers that treat the later clauses as speculative computation to be executed when idle processors are available (Beaumont and Warren, 1993). The difference from our work is that the scheduling is done by the implementation, using one of several strategies designed to extract the greatest possible parallelism from a Prolog program; there is no intention to allow the Prolog programmer to control the speculative parallelism.

Szeredi (1992) describes a language and system for parallel branch-and-bound search, which allows either depth-first or best-first search and can handle both search trees and game trees. The language is Prolog augmented by some special-purpose language primitives, and the implementation is an extension of the Aurora or-parallel Prolog system. The main issues in the implementation are the communication of search bounds between branches of the tree, the use of bounds to prune branches, and the provision of best-first scheduling. The system applies a specified, fixed, number (*N*) of processors to the traversal of the tree — which reflects the structure of the Prolog program — to extend the best (or leftmost) *N* branches at a time. It is not clear from the paper (Szeredi, 1992) whether the tree can include mandatory branches as well as (speculative) branches that will be pruned, and, if so, how the system combines mandatory and speculative work. (This latter issue is the main subject of our work and that of Huntbach.)

5.2 Summary and future work

The main purpose of this paper was to investigate the exploitation of speculative parallelism in concurrent logic programming languages, an area which has received relatively little attention. We have proposed and implemented a simple language extension that makes the technique possible, and have shown some potential applications of it. The results of our experiments confirm that speculative parallelism can result in greater parallel speedups than is possible with other (divide and conquer) forms of parallelism.

Similar extensions could be made to other concurrent languages which schedule processes dynamically, e.g., Ada, and similar results could be expected. However, a distinguishing feature of concurrent logic programming languages is that they "transparently" exploit all available mandatory parallelism without any effort from the programmer. Even so, we have found that even better speedups can be obtained by making use of speculative parallelism as well.

One area for future research is to implement similar facilities in other logic programming languages such as Pandora (Bahgat and Gregory, 1989; Bahgat, 1993). Another is to investigate other applications which can benefit from speculative parallelism. An interesting application would be the parallel execution of finite-domain constraint logic programming (Gregory and Yang, 1992). The two main techniques suggested by van Hentenryck (1989) to solve constraints are *forward checking*, which is cheap to implement, and *lookahead*, which is more expensive but potentially more effective at reducing the search space. We could get the best of both worlds by using *both* methods in parallel, with lookahead performed at a lower priority.

Acknowledgements

Thanks are due to Azlinah Mohamed, who completed an initial implementation of the travelling salesman problem as part of her Master's project in 1992. We are also grateful to David Warren for the use of his Sequent Symmetry multiprocessor.

References

Bahgat, R. 1993. *Pandora: non-deterministic parallel logic programming*. Singapore: World Scientific Publishing Co.

Bahgat, R. and Gregory, S. 1989. Pandora: non-deterministic parallel logic programming. In *Proceedings of the 6th International Logic Programming Conference* (Lisbon, June), G. Levi and M. Martelli (Eds.), Cambridge, Mass.: MIT Press, pp. 471–486.

Beaumont, A. and Warren, D.H.D. 1993. Scheduling speculative work in or-parallel Prolog systems. In *Proceedings of the 10th International Logic Programming Conference* (Budapest, June), Cambridge, Mass.: MIT Press.

Broda, K. and Gregory, S. 1984. PARLOG for discrete event simulation. In *Proceedings of the 2nd International Logic Programming Conference* (Uppsala, July), S.-A. Tarnlund (Ed.), Uppsala: Uppsala University Press, pp. 301–312.

Burton, F.W. 1985. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers C-34*, 12, pp. 1190–1193.

Crammond, J.A. 1988. *Implementation of committed choice logic languages on shared memory multiprocessors*. PhD thesis, Heriot-Watt University.

Foster, I.T. 1990. *Systems programming in parallel logic languages*. Prentice Hall.

Gregory, S. 1987. *Parallel logic programming in PARLOG*. Reading, Mass.: Addison-Wesley.

Gregory, S. and Yang, R. 1992. Parallel constraint solving in Andorra-I. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo, June), H. Tanaka (Ed.), Tokyo: Ohmsha Ltd., pp. 843–850.

Huntbach, M.M. 1991a. Speculative computation and priorities in concurrent logic languages. In *Proceedings of ALPUK Conference* (Edinburgh).

Huntbach, M.M. 1991b. Parallel branch-and-bound search in Parlog. *International Journal of Parallel Programming 20*, 4, pp. 299–314.

Kumar, V., Ramesh, K., and Rao, V.N. 1988. Parallel best-first search of state-space graphs: a summary of results. In *Proceedings of AAAI*, pp. 122–127.

Leifker, D.B. and Kanal, L.N. 1985. A hybrid SSS*/alpha-beta algorithm for parallel search of game trees. In *Proceedings of IJCAI*, pp. 1044–1046.

Little, J.D.C., Murty, K.G., Sweeney, D.W., and Karel, C. 1963. An algorithm for the traveling salesman problem. *Operations Research 11*, pp. 972–989.

Nilsson, N.J. 1980. *Principles of Artificial Intelligence*. Springer Verlag.

Quinn, M.J. 1987. *Designing efficient algorithms for parallel computers*. McGraw Hill.

Szeredi, P. 1992. Exploiting or-parallelism in optimisation problems. In *Proceedings of the Joint International Conference and Symposium on Logic Programming* (Washington, DC, November), K. Apt (Ed.), Cambridge, Mass.: MIT Press, pp. 703–716.

Takeuchi, A. and Furukawa, K. 1985. Bounded buffer communication in Concurrent Prolog. *New Generation Computing 3*, 2, pp. 145–155.

Ueda, K. and Chikayama, T. 1990. Design of the kernel language for the Parallel Inference Machine. *Computer Journal 33*, 6, pp. 494–500.

van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. Cambridge, Mass.: MIT Press.