Fast dynamic multiway merge using destructive operations

Ehud Shapiro

The Weizmann Institute of Science

26-11-84

Abstract

A method for implementing multiway dynamic stream merge which achieves constant delay and bounded waiting is described.

The method can be implemented almost entirely in Concurrent Prolog, with the addition of destructive assignment. This is in contrast with the method of Ueda and Chikayama [3], which achieves similar performance but should be provided by the underlying implementation language, and the two-three tree merge of Shapiro and Mierowski [1], which is implemented in pure Concurrent Prolog, but achieves only a logarithmic delay.

The implementation described uses Takeuchi's "short-circuit" programming technique [2] for the detection of global termination.

1. Introduction

Access to a shared resource is best implemented in Concurrent Prolog by message passing via a shared stream.
Allowing several processes access to a shared requires the merging of their corresponding streams. Efficient stream mergers were studied by Shapiro and Mierowsky [1], and by Ueda and Chikayama [3].

One criterion for evaluating stream-mergers is according to their @i(delay), which is the number of primitive operations required for each message to pass through the stream merger. Another criterion is @i(fairness). A stream-merger is said to guarantee @i(k-bounded waiting) if a message that arrives in one o its input streams will be overtaken by at most k meesages, arriving later then it, in the merger's output stream. A stream merger is @i(fair) if it guarantees k-bounded waiting, for some k>0.

Shapiro and Mierowsky [1] show how to achieve logarithmic delay and linear bounded waiting multiway dynamic merge, using the concept of two-three trees. Ueda and Chikayama [3] present an implementation technique that achieves constant delay and linear bounded waiting. In principle, the algorithm they use can be generated by an optimizing compiler, applied to a naive Concurrent Prolog merge program, and hence its external behavior can be specified by a Concurrent Prolog program. They propose, however, that this code will be provided as a system predicate.

In this paper we propose a different algorithm that also achieves constant delay. Due to the simplicity of the algorithm, the constant is much smaller. The algorithm guarantees n-bounded waiting for merging n streams if a breadth-first scheduler is used, and (k*n)-bounded waiting using a k-bounded depth first scheduler.

An added benefit of the algorithm is that, in contrast to the algorithm of Ueda and Chikayama, it can be implemented almost entirely in Concurrent Prolog, as shown below, with the addition of destructive-assignment primitives.

Another difference between the two algorithms is that Ueda and Chikayama's algorithm requires the awakened merge process to know which variable woke it up, whereas the algorithm proposed here achieves this effect wihout the need of this mechanism. This can be useful in implementations of Flat Concurrent Prolog [1] which, in general, do not require this information.

2. The Algorithm

Even though the algorithm uses destructive operations, it seems that the easiest way to specify it is using Concurrent Prolog, (actually Flat Concurrent Prolog) souped up with destructive assignment primitives. Since destructive assignment is not part of the logic program computation model, the resulting program can be understood only through the operational semantics of Concurrent Prolog, in cunjunction with the way destructive assignment can be added into it.

We do not, as yet, propose to add destructive assignment to Concurren Prolog. The one example shown in this paper that uses it (which should actually be a system provided predicate) is not quite enough of a reason. However, when considering what is the easiest way to implement a multiway dynamic merge system predicate that embodies this algorithm, adding destructive assignment to Concurent Prolog and using the pseudo-code described below seems to be one of the easier approaches.

2.1. A first approximation

The key idea of the algorithm is to use a multiplte-assignment variable as a shared, updatable, pointer to the tail of the mergerd output stream. The algorithm operates as follows. The stream merger is invoked with a (lazy) stream of streams to be merged and an output stream. It first initializes a multiple-assignment variable '!Out' to refer to the output stream. Then, for every input stream Xs it spawns a 'destructive_copy' process, with a reference to Xs and to the multiple-assignmnet variable !Out. For every new input strem element X the 'destructive_copy' process operates as follows: it allocates a new list cell [X|Ys], unifies it with the variable referenced by !Out, and modifies the value of !Out to be a reference to Ys. This last modification is the only destructive operation in the algorithm.

A definition of the algorithm in pseudo Concurrent Prolog is shown below. By convention, the form !X is used for multiple-assignment variables. The implementation uses the following two operations on such variables:

*(!X)=T :- unify the term referenced by !X with T.

!X:=T :- assign !X a reference to term T.

Pseudo-code using these operations is surrounded by double-quotes.

```
merge(InStreams,OutStream) :-
    % Initialize the shared multiple-assignmnet variable
    "!Out:=OutStream" |
    merge1(InStreams?,!Out).
```

A more elegant, distributed, way to detect global temination that does not use multiple-assignmnet shared variables can be achieved using the short-circuit, a Concurrent Prolog programming technique developed by A. Takeuchi [2]. The best way to understand the technique is via an analogy.

Consider a blind slave-driver who wants to detect whether his slaves have finished their jobs. He chains all the slaves in a row through their feet, using iron chains, so that each slave's foot is connected to one end of a chain, and each chain is connected to the left foot of one slave and to the right foot of another, except for two chains, which are connected to one foot only each. The slave-driver keeps the two free ends of these chains. He attaches one to power, and the other, via a light-bulb, to ground.

The rule is that every slave combines his legs so that the two chains touch each other as he finishes his work, but not before that. If every slave obeys the rule (and no-one's body conducts electricity), then the light-bulb will turn on exactly when every slave finishes working. The remaning question, how does the blind slave-driver detects that the light is on, is left as an exercise to the reader.

An analogoues algorithm can be used in Concurrent Prolog: subordinate processes represent slaves, and shared variables represent chains. The driver process that creates the slave processes chains them using shared variables. When a process terminates it unifies its two 'chain' variables. The driver instantiates one end of the chain (the 'power'), to a constant, and keeps the other end to itself. When it finishes spawning processes, it examines the other end of the chain (the 'ground') in read-only mode, and, if everyone plays by the rules, that variable will become instantiated as soon as all the subordinate processes terminate.

The technique is used in the enahanceement to Program 1, shown below. The addition in functionality compared to Program 1 is that upon the detection of global termination, the merge process closes the output stream, by unifying *(!Out) with nil.

```
merge(InStreams,OutStream) :-
    "!Out:=OutStream" |
    merge1(InStreams,!Out,done).

merge1([stream(Xs)|InStreams],!Out,Left) :-
    destructive_copy(Xs,!Out,Left,Right),
    merge1(InStreams?,!Out,Right).
merge1([],Chain) :-
    close_outstream(Chain?,!Out).


close_outstream(done,!Out) :-
    "*(!Out)=[]" | true.

destructive_copy([X|Xs],!Out,Left,Right) :-
    "*(!Out)=[X|Ys], !Out:=Ys"  |
    destructive_copy(Xs?,!Out,Left,Right).
destructive_copy([],!Out,Chain,Chain).
```

Program 2: Closing the output stream upon global termination

```
merge1([stream(Xs)|InStreams],!Out) :-
    destructive_copy(Xs?,!Out),
    merge1(InStreams?,!Out).
merge1([],_).

destructive_copy([X|Xs],!Out) :-
    % allocate a list-cell, unify its car with X,
    % unify the variable referenced by !Out with the list-cell,
    % and destructively update !Out to be a reference to the cell's cdr.
    "*(!Out)=[X|Ys], !Out:=Ys"  |
    destructive_copy(Xs?,!Out).
destructive_copy([],_).
```

Program 1: Multiway dynamic merge using destructive operations

The algorithm requires n+1 processes for merging n streams. It requires one process reduction, and the allocation of one list cell, per one stream element merged.

If breadth-first scheduling is used, then each ready destructive_copy process will copy one element at a time, and no process will copy two elements before another ready process has copied one, so linear bounded waiting is guaranteed. If k-bounded depth-first scheduling is used, where every (iterative) process is allowed at most k reductions before it is suspended, then each destructive_copy process can copy at most a run of k elements at a time, and a (k*n)-bounded waiting is guaranteed, when n is the number of active destructive_copy processes.

The behavior of the algorithm in case the output stream is connected to a bounded-buffer [4] is slightly more intricate. If breadth-first scheduling is used, it may be the case that all n destructive_copy processes are suspended on the variable referenced by !Out. When this variable is bound to a list cell, all are woken up, but only the first will succeed in copying an element to the output stream, and all the others will suspend again.
If non-busy waiting is used, then to guarantee bounded-waiting, the suspension/wakeup mechanism should preserve the relative order of processes in the active queue and the suspension queues.

If bounded depth-first scheduling is used, the situation is a bit more complicated, but not much better. In other words, when bounded-buffer is used, the algorithm may exhibit linear delay. The algorithm of Ueda and Chikayam suffers from a similar problem. The two-three tree merge algorithm, on the other hand, can be adapted to bounded-buffers in a way that preserves its logarithmic delay.

Another problem with the algorithm just described is that it does not close the output stream upon termination of all the merged input streams. The second version of the algorithm solves this problem.

2.2. Detecting global termination using the short-circuit technique

One way to detect global termination in the algorithm described is to have an additional multiple-assignment global variable, '!Active', to maintain the number of active processes.
This variables is initialized to 1 by the merge process.
When a destructive_copy process terminates it checks whether its is the last such active process (!Active=0),
if so, unifies the variable referred to by '!Out' with nil. Similarly for the merge process.

3. Discussion

Efficienct multiway dynamic merge is essential to realize efficient applications in Concurrent Prolog, especially systems-type applications. A simple method for realizing it have been shown. The method is not readily specifiable by pure Concurrent Prolog, ias previous proposal

/*
Mention two choices:
    Shared destructive variable in the heap or in the process descriptor.
Mention why cannot hide side-effects (with a stream merge).
*/

References

[1] C. Mierowsky
Flat Concurrent Prolog: Design, Implementation, Applications.
M.Sc. Thesis, Weizmann Institute of Science, Technical Report
CSXX-84, November, 1984.

[2] E. Shapiro and C. Mierowsky
Fair, biased, and self-balancing merge operators: their specification
and implementation in Concurrent Prolog.
Journal of New Generation Computing, 2(3), 1984.

[3] K. Ueda and T. Chikayama
Efficient stream/array processing in logic programming languages
Proc. of FGCS'84, pp.317-326, ICOT, 1984.

[4] A. Takeuchi
How to solve it in Concurrent Prolog
Unpublished memorandum, 1983.

[5] A. Takeuchi and K. Furukawa
Interprocess communication in Concurrent Prolog
Proceedings of the 1983 Logic Programming Workshop,
Portugal, University Nova de Lisboa, 1983.

Curriculum Vitae

Ehud Y. Shapiro

Ehud Y. Shapiro received B.A. degree in mathematics and philosophy from Tel Aviv University in 1979, and PhD degree in computer science from Yale University in 1982.

His PhD thesis, "Algorithmic Program Debugging", was selected as an ACM Distinguished Dissertationl. Since 1982 he has been associated with the Department of Applied Mathmatics at the Weizmann Institute of Science.

Ehud Shapiro is the designer of the programming language Concurrent Prolog. His current research interests include logic programming and parallel processing.

He is a member of the editorial or advisory boards of Computer Compacts, The Journal of Logic Programming, and the Journal of New Generation Computing, and is the Program Chairman of the Third International Logic Programming Symposium.