Message-Oriented Parallel Implementation of Moded Flat GHC

Kazunori Ueda
NEC C&C Systems Research Laboratories
1-1, Miyazaki 4-chome, Miyamae-ku, Kawasaki 213, Japan
ueda@csl.cl.nec.co.jp

Masao Morita
Mitsubishi Research Institute
3-6, Otemachi 2-chome, Chiyoda-ku, Tokyo 100, Japan

Abstract

We proposed in [Ueda and Morita 1990] a new, *message-oriented* implementation technique for Moded Flat GHC that compiled unification for data transfer into message passing. The technique was based on constraint-based program analysis, and significantly improved the performance of programs that used goals and streams to implement reconfigurable data structures. In this paper we discuss how the technique can be parallelized. We focus on a method for shared-memory multiprocessors, called the *shared-goal method*, though a different method could be used for distributed-memory multiprocessors. Unlike other parallel implementations of concurrent logic languages which we call *process-oriented*, the unit of parallel execution is not an individual goal but a chain of message sends caused successively by an initial message send. Parallelism comes from the existence of different chains of message sends that can be executed independently or in a pipelined manner. Mutual exclusion based on busy waiting and on message buffering controls access to individual, shared goals. Typical goals allow *last-send optimization*, the message-oriented counterpart of last-call optimization. We have built an experimental implementation on Sequent Symmetry. In spite of the simple scheduling currently adopted, preliminary evaluation shows good parallel speedup and good absolute performance for concurrent operations on binary process trees.

1. Introduction

Concurrent processes can be used both for programming computation and for programming storage. The latter aspect can be exploited in concurrent logic programming to program reconfigurable data structures using the following analogy,

records	$\longleftrightarrow$	body goals
pointers	$\longleftrightarrow$	streams (implemented by lists)

where a (concurrent) process is said to be *implemented* by a multiset of goals.

An advantage of using processes for this purpose is that it allows implementations to exploit parallelism between operations on the storage. For instance, a search operation on a binary search tree (Program 1), given as a message in the interface stream, can enter the tree soon after the previous operation has passed the root of the tree. Programmers do not have to worry about mutual exclusion, which is taken care of by the implementation.

```
nt([],               _, _, L,R) :- true | L=[], R=[].
nt([search(K,V)|Cs],K, V1,L,R) :- true | V=V1, nt(Cs,K,V1,L,R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K<K1 | L=[search(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K>K1 | R=[search(K,V)|R1], nt(Cs,K1,V1,L,R1).
nt([update(K,V)|Cs],K, _, L,R) :- true | nt(Cs,K,V,L,R).
nt([update(K,V)|Cs],K1,V1,L,R) :- K<K1 | L=[update(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([update(K,V)|Cs],K1,V1,L,R) :- K>K1 | R=[update(K,V)|R1], nt(Cs,K1,V1,L,R1).

t([]              ) :- true | true.
t([search(_,V)|Cs]) :- true | V=undefined, t(Cs).
t([update(K,V)|Cs]) :- true | nt(Cs,K,V,L,R), t(L), t(R).
```

Program 1. A GHC program defining binary search trees as processes

This suggests that the programming of reconfigurable data structures can be an important application of concurrent logic languages. (The verbosity of Program 1 is a separate issue which is outside the scope of this paper.)

Processes used as storage are almost always suspending, but should respond quickly when messages are sent. However, most implementations of concurrent logic languages have not been tuned for processes with this characteristic. In our earlier paper [Ueda and Morita 1990], we proposed *message-oriented* scheduling of goals for sequential implementation, which optimizes goals that suspend and resume frequently. Although our primary goal was to optimize storage-intensive (or more generally, demand-driven) programs, the proposed technique worked quite well also for computation-intensive programs that did not use one-to-many communication. However, how to utilize the technique in parallel implementation was yet to be studied.

Parallelization of message-oriented scheduling can be quite different from parallelization of ordinary, *process-oriented* scheduling. An obvious way of parallelizing process-oriented scheduling is to execute different goals on different processors. In message-oriented scheduling, the basic idea should be to execute different message sends on different processors, but many problems must be solved as to the mapping of computation to processors, mutual exclusion, and so on. This paper reports the initial study on the subject.

The rest of the paper is organized as follows: Section 2 reviews Moded Flat GHC, the subset of GHC we are going to implement. Section 3 reviews message-oriented scheduling for sequential implementation. Section 4 discusses how to parallelize message-oriented scheduling. Of the two possible methods suggested, Section 5 focuses on the shared-goal method suitable for shared-memory multiprocessors and discusses design issues in more detail. Section 6 shows the result of preliminary performance evaluation. The readers are assumed to be familiar with concurrent logic languages [Shapiro 1989].

2. Moded Flat GHC and Constraint-Based Program Analysis

Moded Flat GHC [Ueda and Morita 1990] is a subset of GHC that introduces a *mode system* for the compile-time global analysis of dataflow caused by unification. A unification body goal of the form $t_1 = t_2$ can cause bidirectional dataflow in general, but mode analysis tries to guarantee that at least one of t_1 and t_2 is an uninstantiated variable and hence the goal does not fail except due to occur check.

Our experience with GHC and KL1 [Ueda and Chikayama 1990] has shown that the full functionality of bidirectional unification is seldom used and that programs using it can be rewritten rather easily, if not automatically, to programs using unification as assignment.

These languages are indeed used as general-purpose concurrent languages, which means that it is very important to optimize basic operations such as unification and to obtain machine code close to that obtained from procedural languages.

For global compile-time analysis to be practical, it is highly desirable that individual program modules can be analyzed separately in such a way that the results can be merged later. The mode system of Moded Flat GHC is thus constraint-based; the mode of a whole program can be determined by accumulating the mode constraints obtained separately from the syntactic analysis of each program clause. Another advantage of the constraint-based system is that it allows programmers to *declare* some of the mode constraints, in which case the analysis works as mode checking as well as mode inference.

The modularity of the analysis was achieved by the rather strong assumption of the mode system: whether the function symbol at some position (possibly deep in a data structure) of a goal g is determined by g or by other goals running concurrently is determined solely by that position specified by a *path*, which is defined as follows. Let *Pred* be the set of predicate symbols and *Fun* the set of function symbols. For each $p \in \mathit{Pred}$ with the arity n_p, let N_p be the set $\{1, 2, \ldots, n_p\}$. N_f is defined similarly for each $f \in \mathit{Fun}$. Now the sets of *paths* P_t (for terms) and P_a (for atoms) are defined using disjoint union as:

$$P_t = \Big(\sum_{f \in \mathit{Fun}} N_f\Big)^*, \quad P_a = \Big(\sum_{p \in \mathit{Pred}} N_p\Big) \times P_t.$$

An element of P_a can be written as a string $\langle p, i\rangle\langle f_1, j_1\rangle \ldots \langle f_n, j_n\rangle$, that is, it records the predicate and the function symbols on the path as well as the argument positions selected. A mode is a function from P_a to the set $\{\mathit{in}, \mathit{out}\}$, which means that it assigns either *in* or *out* to every possible position of every possible instance of every possible goal. Whether some position is *in* or *out* can depend on the predicate and function symbols on the path down to that position. The function can be partial, because the mode values of many uninteresting positions that will not be realized can be left undefined.

Mode analysis checks if every variable generated in the course of execution will have exactly one *out* occurrence (occurrence at an *out* position) that can determine its top-level value, by accumulating constraints between the mode values of different paths.

Constraint-based analysis can be applied to analyzing other properties of programs as well. For instance, if we can assume that streams and non-stream data structures do not occur at the same position of different goals, we can try to classify all the positions into

(1) those whose top-level values are limited to the list constructors (*cons* and *nil*) and
(2) those whose top-level values are limited to symbols other than the list constructors,

which is the simplest kind of type inference. Other applications include the static identification of 'single-reference' positions, namely positions whose values are not read by more than one goal and hence can be discarded or destructively updated after use. This could replace the MRB (multiple-reference bit) scheme [Chikayama and Kimura 1987], a runtime scheme adopted in current KL1 implementations for the same purpose.

3. Message-Oriented Sequential Implementation

In a process-oriented sequential implementation of concurrent logic languages, goals ready for execution are put in a queue (or a stack or a deque, depending on the scheduling). Once a goal is taken from the queue, it is reduced as many times as possible, using last-call optimization, until it suspends or it is swapped out. A suspended goal is hooked on

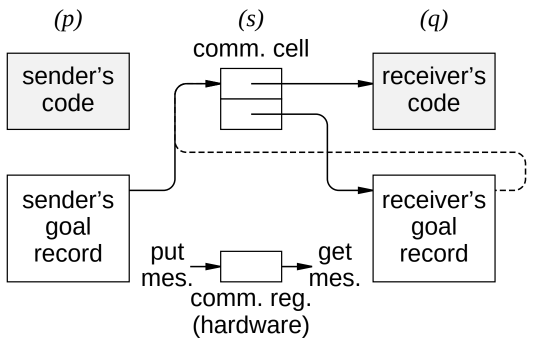

Fig. 1. Immediate message send

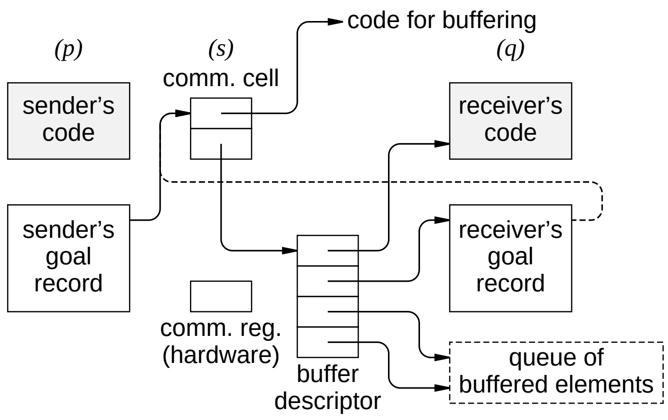

Fig. 2. Buffered message send

the uninstantiated variable(s) that caused suspension, and when one of the variables is instantiated, it is put back into the queue.

Message-oriented implementation has much in common with process-oriented implementation, but differs in the treatment of stream communication: It compiles the generation of stream elements into procedure calls to the consumer of the stream. A stream is an unbounded buffer of messages in principle, but message-oriented implementation tries to reduce the overhead of buffering and unbuffering by transferring control and messages simultaneously to the consumer whenever possible. To this end, it tries to schedule goals so that whenever the producer of a stream sends a message, the consumer is suspending on the stream and is ready to handle the message. Of course, this is not always possible because we can write a program in which a stream *must* act as a buffer; messages are buffered when the consumer is not ready to handle incoming messages.

Process-oriented implementation tries to achieve good performance by reducing the frequency of costly goal switching and taking advantage of last-call optimization. Message-oriented implementation tries to reduce the cost of each goal switching operation and the cost of data transfer between goals, in order to improve the performance of programs in which the response to messages is particularly important.

Suppose two goals, p and q, are connected by a stream s and p is going to send a message to q, which is suspending on s. We assume that a suspended goal will resume its execution from the instruction following the one that caused suspension, not from the first instruction of the predicate. Message-oriented implementation represents s as a two-field *communication cell* that points to (1) the instruction in q's code from which the processing of q is to be resumed and (2) q's goal record containing its arguments (Fig. 1). We call the first field the *code field* and the second the *environment field.*

To send a message m, the goal p (i) loads m in a hardware register called the *communication register*, (ii) makes the goal record pointed to by the communication cell of s the 'current' one, and (iii) calls the code pointed to by the communication cell of s. The goal q gets m from the communication register and may send other messages in its turn. Control returns to p when all the message sends caused directly or indirectly by m have been processed. However, if m is the last message which p can send out immediately (i.e., without waiting for further incoming messages), control need not return to p but can go directly to the goal that has outstanding message sends. This is called *last-send optimization*, which we shall consider in Section 5.4 in more detail.

We have observed in GHC/KL1 programming that the dominant form of interprocess communication is one-to-one stream communication. It therefore deserves special treatment, even though other forms of communication such as broadcasting and multicasting become a little more expensive. One-to-many communication is done either by the repeated sending of messages or by using non-stream data structures.

Techniques mentioned in Section 2 are used to analyze which positions of a predicate and which variables in a program are used for streams and to distinguish between the sender and the receiver(s) of messages.

When a stream must buffer messages, the communication cell representing the stream points to the code for buffering and the descriptor of a buffer. The old entries of the communication cell are saved in the descriptor (Fig. 2). In general, a stream must buffer incoming messages when the receiver goal is not ready to handle them. The following are the possible reasons [Ueda and Morita 1990]:

(1) The receiver is waiting for a message from other input streams.
(2) The receiver is suspending on non-stream data, possibly the contents of messages.
(3) The sender of a message may run ahead of the receiver.
(4) When the receiver r belongs to a circular process structure, a message m sent by r may possibly arrive at r itself or may cause another message to be sent back to r. However, unless m has been sent by last-send optimization, r is not ready to receive it.

The receiver examines the buffer when the reason for the buffering disappears, and handles messages in it, if any.

Process-oriented implementation often caches the whole or a part of a goal record in hardware registers, but this should not be done in message-oriented implementation because process switching takes place frequently and access locality cannot be expected of goal records.

4. Parallelization

How can we exploit parallelism in message-oriented implementation? Two quite different methods can be considered:

Distributed-goal method. Different processors take charge of different goals, and each processor handles messages sent to the goals it is taking charge of. Consider a binary search tree represented using goals and streams (Fig. 3) and suppose three processors take charge of the three different portions of the tree. Each processor performs message-oriented processing within its own portion, while message transfer between portions is compiled into inter-processor communication.

Shared-goal method. All processors share all the goals. There is a global, output-restricted deque [Knuth 1973] of outstanding work to be done in parallel, from which an idle processor gets a new job. The job is usually to execute a non-unification body goal or to send a message, the latter being the result of compiling a unification body goal involving streams. The message send will usually cause the reduction of a suspended goal. If the reduction generates another unification goal that has been compiled into a message send, it can be performed by the same processor. Thus a chain of message sends is formed, and different chains of message sends can be performed in parallel as long as they do not interfere with each other. In the binary tree example, different processors will take care of different operations sent to the root. A tree operation may cause subsequent message sends inside

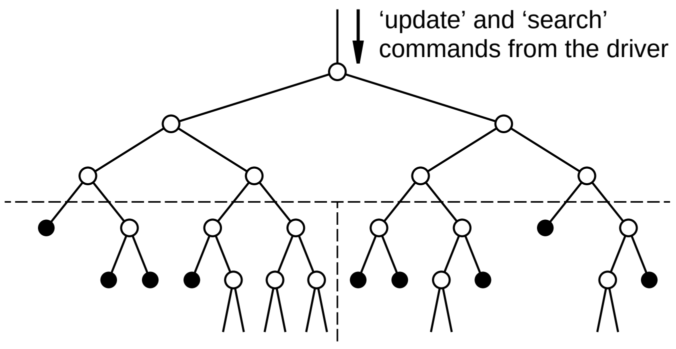

Fig. 3. Binary search tree as a process

the tree, but they should be performed by the same processor because there is no parallelism within each tree operation.

Unlike the shared-goal method, the distributed-goal method can be applied to distributed-memory multiprocessors as well as shared-memory ones to improve the throughput of message handling. On shared-memory multiprocessors, however, the shared-goal method is more advantageous in terms of latency (i.e., responses to messages), because (1) it performs no inter-processor communication within a chain of message sends and (2) good load balancing can be attained easily. The shared-goal method requires a locking protocol for goals as will be discussed in Section 5.1, but it enables more tightly-coupled parallel processing that covers a wider range of applications. Because of its greater technical interest, the rest of the paper is focused on the shared-goal method.

5. Shared-Goal Implementation

In this section, we discuss important technicalities in implementing the shared-goal method. We explain the method and the intermediate code mainly by examples; the details of our implementation is the subject of another paper.

5.1 Locking of Goals

Consider a goal `p(Xs,Ys)` defined by the following single clause:

```
p([A|Xs1],Ys) :- true | Ys=[A|Ys1], p(Xs1,Ys1).
```

In the shared-goal method, different messages in the input stream `Xs` may be handled by different processors that share the goal `p(Xs,Ys)`. Any processor sending a message must therefore try to lock the goal record (placed in the shared memory) of the receiver first and obtain the grant of exclusive access to it. The receiver must remain locked until it sends a message through `Ys` and restores the dormant state. The standard protocol of message sending is the same as that of sequential implementation (Section 3), except that the sender must lock the goal record pointed to by the environment field of the communication cell before making that goal record the 'current' one. We currently use one byte of the first word of a goal record as a lock, and atomically modify it using an xchg (exchange) instruction for locking.

The locking operation is important in the following respect as well: In message-oriented implementation, the order of the elements in a stream is not represented spatially as a list structure but as the chronological order of message sends. The locking protocol must therefore make sure that when two messages, α and β, are sent in this order to `p(Xs,Ys)`, they are sent to the receiver of `Ys` in the same order. This is guaranteed by locking the receiver of `Ys` *before* `p(Xs,Ys)` is unlocked.

5.2 Busy Wait vs. Suspension

How should a processor trying to send a message wait until the receiver goal is unlocked? The two extreme possibilities are (1) to spin (busy-wait) until unlocked and (2) to give up (suspend) the sending immediately and do some other work, leaving a notice to the receiver that it has a message to receive. We must take the following observations into account here:

(a) The time each reduction takes, namely the time required for a resumed goal to restore the dormant state, is usually short (some tens of CISC instructions, say), though it can be quite long sometimes.
(b) As explained in Section 5.1, a processor may lock more than one goal temporarily upon reduction. This means that busy wait may cause deadlock when goals and streams form a circular structure.

Because busy wait incurs much smaller overhead than suspension, Observation (a) suggests that the processor should spin for a period of time within which most goals can perform one reduction. However, it should suspend finally because of (b).

Upon suspension, a buffer is prepared as in Fig. 2, and the unsent message is put in it. Subsequent messages go to the buffer until the receiver has processed all the messages in the buffer and has removed the buffer. As is evident from Fig. 2, no overhead is incurred to check if the message is going to the buffer or to the receiver. The sender simply follows the standard protocol and locks the record pointed to by the communication cell, which happens to be a buffer descriptor in this case.

The receiver could notice the existence of outstanding messages by checking its input streams upon each reduction, but it incurs overhead to programs which do not require buffering. So we have chosen to avoid this overhead by letting the *sender* spawn and schedule a special routine, called the *retransmitter* of the messages, when it creates a buffer. The retransmitter is executed asynchronously with the receiver. When executed, the retransmitter tests if the receiver has been unlocked, in which case it sends the first message in the buffer and re-schedules itself.

For the shared resources other than goals, such as logic variables and the global deque, mutual exclusion should be achieved by busy wait, because access to them takes a short period of time. On the other hand, synchronization on the values of non-stream variables due to the semantics of GHC should be implemented using suspension as usual.

5.3 Scheduling

Shared-goal implementation exploits parallelism between different chains of message sends that do not interfere with each other. In the case of a binary search tree (Fig. 3), different operations on the tree can be processed in a pipelined manner as long as there is no dependence between the operations. This condition does not hold when, for instance, the key of a search operation depends on the result of the previous search operation. When such dependence exists, parallel execution can even lower the performance because of synchronization overhead.

Another example for which parallelism does not help is a demand-driven generator of prime numbers which is made up of processes, one for each prime, for filtering out the multiples of those primes. The topmost goal that receives a new demand from outside filters out the multiples of the prime computed in response to the last demand. However, until the last demand has almost been processed, the topmost goal doesn't know what prime's multiples should be filtered out, and hence will be blocked.

These considerations suggest that in order to avoid ineffective parallelism, it is most realistic to let programmers specify which chains of message sends should be done in parallel with others. The simple method we are using currently is to have (1) a global deque for the work to be executed in parallel by idle processors and (2) one local stack for each processor for the work to be executed sequentially by the current processor. Each processor obtains a job from the global deque when its local stack is empty. We use a global deque rather than a global stack because, if the retransmitter of a buffer fails to send a message, it must go to the tail of the deque so it may not be retried soon.

Each job in a stack/deque is uniformly represented as a pair $\langle \mathit{code}, \mathit{env} \rangle$, where *code* is the job's entry/resumption point and *env* is its environment. The job is usually to start the execution of a goal or to resume the execution of a clause body. In these cases, *env* points to the goal record on which *code* should work. When the job is to retransmit buffered messages, *env* points to the communication cell pointing to the buffer.

When a clause body has several message sends to be executed in parallel, they will not put in the deque separately. Instead, the current processor executing the clause body performs the first send (and any sends caused by that send), putting the rest of the work to the deque after the first send succeeds in locking the receiver. Then an idle processor will get the rest of the work and perform the second message send (and any sends caused by that send), putting the rest of the rest back to the deque. This procedure is to guarantee the order of messages sent through a single stream by different processors. Suppose two messages, α and β, are sent by a goal like `Xs=[`α`,`β`|Xs1]`. Then we have to make sure that the processor trying to send β will not lock the receiver of `Xs` before the processor trying to send α has done so.

5.4 Reduction

This section describes what a typical goal should do during one reduction, where by 'typical' we mean goals that can be reduced by receiving one message. As an example, consider the distributor of messages defined as follows,

```
p([A|Xs],Ys,Zs) :- true | Ys=[A|Ys1], Zs=[A|Zs1], p(Xs,Ys1,Zs1).
```

where we assume `A` is known, by program analysis or declaration, to be a non-stream datum. Otherwise a somewhat more complex procedure is necessary, because the three occurrences of `A` will be used for one-to-two communication. The intermediate code for above program is:

```
entry(p/3)
  rcv_value(A1)
  get_cr(A4)
  send_call(A2)
  put_cr(A4)
  send_call(A3) }
  execute       }or send_jmp(A3).
```

The A*i*'s are entries of the goal record of the goal being executed, which contain the arguments of the goal and temporary variables. Other programs may use X*i*'s, which are (possibly virtual) general registers local to each processor, and GA*i*'s, which are the arguments of a new goal being created. The label `entry(p/3)` indicates the initial entry point of the predicate `p` with three arguments.

The instruction `rcv_value(A1)` waits for a message from the input stream on the first argument. If messages are already buffered, it takes the first one and puts it on the communication register. A retransmitter of the buffer is put on the deque if more messages exist; otherwise the buffer is made to disappear (Section 5.7). If no messages are buffered, which is expected to be most probable, `rcv_value` unlocks the goal record, and suspends until a message arrives. In either case, the instruction records the resumption address, namely the address of the next instruction, in the communication cell. When the communication cell points to a buffer, the resumption address is recorded in the buffer descriptor instead. *The goal is usually suspending at this instruction.*

The instruction `get_cr(A4)` saves into the goal record the message in the communication register, which the previous `rcv_value(A1)` has received. Then `send_call(A2)` sends the message in the communication register through the second stream. The instruction `send_call(A2)` tries to lock the receiver of the second stream and if successful, transfers control to the receiver. If the receiver is busy for a certain period of time or it isn't busy but is not ready to handle the message, the message is buffered. The instruction `send_call` does not unlock the current goal record. When control eventually returns, `put_cr(A4)` restores the communication register and `send_call(A3)` sends the next message.

When control returns again, `execute` performs the recursive call by going back to the entry point of the predicate `p`. Then the `rcv_value(A1)` instruction will either find no buffered messages or find some. In the former case, `rcv_value(A1)` obviously suspends. In the latter case, a retransmitter of the buffer must have been scheduled, and so `rcv_value(A1)` can suspend until the retransmitter sends a message. Moreover, the resumption address of the `rcv_value(A1)` instruction has been recorded by its previous execution. Thus in either case, `execute` effectively does nothing but unlock the current goal. This is why last-send optimization can replace the last two instructions by a single instruction, `send_jmp(A3)`.

The instruction `send_jmp(A3)` locks the receiver of the third stream, unlocks the current goal, and transfers control to the receiver without stacking the return address. Last-send optimization enables the current goal to receive the next message earlier and allows the pipelined processing of message sends. Note that with last-send optimization, the `rcv_value(A1)` instruction will be executed only once when the goal starts execution. The instructions executed for each incoming message are those from `get_cr(A4)` through `send_jmp(A3)`.

The above instruction sequence performs the two message sends sequentially. However, a variant of `send_call` called `send_fork` stacks the return address on the global deque instead of the local stack, allowing the continuation to be processed in parallel. Note that `send_fork` leaves the continuation to another processor rather than the message send itself for the reason explained in Section 5.3.

The reduction of a goal may in general involve the spawning and the termination of goals and the explicit control of message buffering; they are described in Section 5.5 and 5.6, respectively. Finally, we note that although process-oriented scheduling and message-oriented scheduling differ in the flow of control, they are quite compatible in the sense that an implementation can use both in running a single program. Our experimental implemen-

```
The program:  (1) nreverse([H|T],O) :- true | append(O1,[H],O), nreverse(T,O1).
              (2) nreverse([],   O) :- true | O=[].
              (3) append([I|J],K,L) :- true | L=[I|M], append(J,K,M).
              (4) append([],   K,L) :- true | K=L.

entry(nreverse/2)
  rcv_value(A1)             receive a message from the 1st arg
                            (the program is usually waiting for incoming messages here)
  check_not_eos(101)        if the message is 'eos', collect the current comm. cell and goto 101
  get_cr(X3)                save the message H in the comm. reg. to the reg. of the current PE
  commit                    Clause 1 is selected (no operation)
  put_cc(X4)                create a comm. cell with an empty buffer
  push_value(X3)            put the message H into the buffer
  push_eos                  put 'eos' into the buffer
  g_setup(append/3,3)       create a goal record for 3 args and record the name
  put_value(A2,GA3)         set the 3rd arg of append to O
  put_value(X4,GA2)         set the 2nd arg of append to [H] (comm. cell with the buffer
                            created above)
  put_com_variable(A2,GA1)  create a locked variable O1 and set the 2nd arg of nreverse and the
                            1st arg of append to the pointer to O1,
                            assuming that append will turn O1 into a comm. cell soon
  g_call                    execute append until it suspends
  return                    unlock the current goal and do the job on the local stack top
label(101)
  commit                    Clause 2 is selected (no operation)
  send_call(A2)             send 'eos' in the comm. reg. to the receiver of O
  proceed                   deallocate the goal record and return

entry(append/3)
  deref(A3)                 dereference the 3rd arg L
  rcv_value(A1)             receive a message from the 1st arg.
  check_not_eos(102)        if the message is 'eos', collect the current comm. cell and goto 102
  commit                    Clause 3 is selected (no operation)
  sendn_jmp(A3)             send the received message to the receiver of L, where
                            'n' means that the instruction assumes that L has been dereferenced
label(102)
  commit                    Clause 4 is selected (no operation)
  send_unify_jmp(A2,A3)     make sure that messages sent through K are
                            forwarded to the receiver of L, and return
```

Fig. 4. Intermediate code for naïve reverse

tation has actually been made by modifying a process-oriented implementation.

5.5 An Example

Here we give the intermediate code of a naïve reverse program (Fig. 4). In order for the code to be self-explanatory, some comments are appropriate here.

Suppose the messages m_1, ..., m_n are sent to the goal `nreverse(In,Out)` through `In`, followed by the *eos* (end-of-stream) message indicating that the stream is closed. The `nreverse` goal generates one suspended `append` goal for each m_i, creating the structure in Fig. 5. The ith `append` has as its second argument a buffer with two messages, m_i and

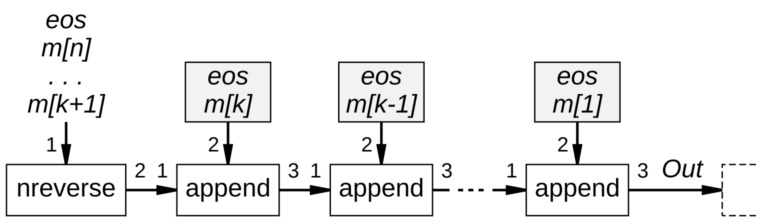

Fig. 5. Process structure being created by `nreverse([`m_1`,...,`m_n`],Out)`

eos. The final *eos* message to `nreverse` causes the second clause to forward the *eos* to the most recent `append` goal holding m_n. The `append` holding m_n, in response, lets different processors (if available) send the two buffered messages, m_n and *eos*, to the `append` holding m_{n-1}. The message m_n is transferred all the way to the `append` holding m_1 and appears in `Out`. The following *eos* causes the next `append` goal to send m_{n-1} and another *eos*.

The performance of `nreverse` hinges on how fast each `append` goal can transfer messages. For each incoming message, an `append` goal checks if the message is not *eos* and then transfers both the message and control to the receiver of the output stream. The message remains on the communication register and need not be loaded or stored.

The `send_unify_jmp(`r_1`,`r_2`)` instruction is used for the unification of two streams. Arrangements are made so that next time a message is sent through r_1, the sender is made to point directly to the communication cell of r_2. Note that the `send_unify_jmp` instruction itself has no access to the pointer in the sender's goal record and hence cannot change it. If the stream r_1 has a buffer, which is the case with `nreverse`, the above redirection is made to happen after all the contents of the buffer are sent to the receiver of r_2.

It is worth noting that the multiway merging of streams can transfer messages as efficiently as `append`.

5.6 Buffering

As discussed in Section 5.2, the producer of a stream s creates a buffer due to timeout when the receiver is locked for a long time. However, this is a rather unusual situation; a buffer is usually created by s's *receiver* when it remains unready to handle incoming messages after it has unlocked itself. Here we re-examine the four reasons for buffering in Section 3:

(1) *Selective message receiving.* This happens, for instance, in a program that merges two sorted streams of integers into a single sorted stream:

```
omerge([A|X1],[B|Y1],Z) :- A< B | Z=[A|Z1], omerge(X1,[B|Y1],Z1).
omerge([A|X1],[B|Y1],Z) :- A>=B | Z=[B|Z1], omerge([A|X1],Y1,Z1).
```

Two numbers, one from each input stream, are necessary for a reduction. Suppose the first number `A` arrives through the first stream. Then the goal `omerge` checks if the second stream has a buffered value. Since it doesn't, the goal cannot be reduced. So it records `A` in the goal record and changes the first stream to a buffer, because it has to wait for another number `B` to come through the second stream. Suppose `B`$(>$ `A`$)$ arrives and the first clause is selected. Then the second stream should become a buffer and `B` will be put back. The first stream, being now a buffer, is checked and a retransmitter is stacked if it contains an element; otherwise the buffer is made to disappear. Finally `A` is sent to the receiver of the third stream. The above procedure is admittedly complex, but this program is indeed one of the hardest ones to execute in a message-oriented manner. A simpler example of selective message receiving appears in the `append` program in Section 5.5; its second input stream buffers messages until the non-recursive clause is selected.

(2) *Suspension on non-stream data.* The most likely case is suspension on the content of a message which happens when, for instance, the first argument of an `update` message to a binary search tree is uninstantiated. When a goal receives from a stream s a message that is not sufficiently instantiated for reduction, it changes s to a buffer and puts the message back in it. A retransmitter is hooked on the uninstantiated variable(s) that caused suspension, which will be invoked when any of them is instantiated.

(3) *The sender of a stream running ahead of the receiver.* It is not always possible to guarantee that the sender of a stream does not send a message before the receiver commences execution, though the scheduling policy tries to avoid such a situation. The simplest solution to this problem is to initialize each stream to an empty buffer. However, creating and collecting a buffer incurs certain overhead, while a buffer created for the above reason will receive no messages in most cases. So the current scheme defers the creation of a real buffer until a message is sent. Moreover, when the message is guaranteed to be received soon, the `put_com_variable` instruction (Fig. 4) is generated and lets the sender busy-wait until the receiver executes `rcv_value`.

(4) *Circular process structure.* When the receiver sends more than one message in response to an incoming message, sequential implementation must buffer subsequent incoming messages until the last message is sent out. In parallel implementation, the same effect is automatically achieved by the lock of the goal record, and hence the explicit control of buffering is not necessary.

The retransmission of a buffer created due to the reason (1) or (3) is explicitly controlled by the receiver. When a buffer is created due to the reason (2) or by the sender of a stream, a retransmitter of the buffer is scheduled asynchronously with the receiver.

5.7 Mutual Exclusion of Communication Cells

A communication cell representing a stream may be updated both by the sender and the receiver of the stream. For instance, the sender may create a buffer and connect it to the cell when the receiver is locked for a certain period of time. The receiver may create or remove a buffer for the cell when buffering becomes necessary or unnecessary, may set or update the code field of the cell by the `rcv_value` instruction, may execute `send_unify_jmp` and connect the stream to another, and may move or delete the goal record of its own.

This of course calls for some method of mutual exclusion for communication cells. The simplest solution would be to lock a communication cell whenever updating or reading it, but locking both a goal record and a communication cell for each message send would be too costly. It is highly desirable that an ordinary message send, which reads but does not update a communication cell, need not lock the communication cell.

However, without locking upon reading, the following sequence can happen and inconsistency arises:

(1) the sender follows the pointer in the environment field of the communication cell,
(2) the receiver starts and completes the updating of the communication cell under an appropriate locking protocol, and then

(3) the sender locks the (wrong) record r (the goal record for the receiver or a buffer for the communication cell) obtained in Step (1) and calls the code pointed to by the code field of the updated communication cell.

This can be avoided by not letting the receiver update the environment field of the communication cell. The receiver instead stores inside the record r the pointer p to the right record. The receiver accordingly sets the code field of the communication cell to the pointer to a code sequence that notifies the sender of the existence of the pointer p, which will be called by the sender in Step (3).

The sender can now access the right record pointed to by p via the wrong record r, but it is usually desirable that p is finally written into the environment field of the communication cell so that subsequent access to the right record may be direct. This update of the communication cell must be done before the sender is unlocked and the control is completely transferred to the receiver. For this purpose, we take advantage of the fact that the 1-byte lock of a record can take states other than 'locked' and 'unlocked'. When the lock of a record has one of these other states, a special routine corresponding to that state runs before the goal record of the sender is unlocked. This feature is being used for updating the environment field of a communication cell safely.

The principle behind the the above scheme is that the code field of a communication cell is updated only by the receiver goal and the environment field only by the sender goal. To obey this principle, the sender should not update the code field when it creates a buffer due to timeout. This is again achieved by setting the 1-byte lock of the buffer to a special state, which causes the code for buffering to be invoked in subsequent message sends.

For a buffer created by a receiver, the above scheme implies that the buffer is first pointed to from the receiver's goal record and then the sender moves the pointer to the communication cell before it inserts the first element. In our actual implementation, however, we chose not to move the pointer from the goal record but to put elements in the buffer indirectly via the goal record. If we move the pointer, the environment field of the communication cell must be rewritten again (by the sender, according to the principle) when the buffer disappears. The overhead of these move operations will not justify the elimination of extra cost incurred by the indirect access to the buffer. The management of a buffer is much easier when it is pointed to from the receiver's goal record.

6. An Experimental System and Its Performance

We have finished the initial version of the abstract machine instruction set for the shared-goal method. An experimental runtime system for performance evaluation has been developed on Sequent Symmetry, a shared-memory parallel computer with 20MHz 80386's. The system is written in an assembly language and C. The abstract machine instructions are expanded into native codes automatically by a loader. A compiler from Moded Flat GHC to the intermediate code is yet to be developed.

The current system employs a simple scheme of parallel execution as described in Section 5.3. When the system runs with more than one processor, one of them acts as a master processor and the others as slaves. They act in the same manner while the global deque is non-empty. When the master fails to obtain a new job from the deque, it tries to detect termination and exceptions such as stack overflow. The current system does not care about perpetually suspended goals; they are treated just like garbage cells in Lisp. A slight overhead of counting the number of goals in the system will be necessary to detect perpetually suspended goals [Inamura and Onishi 1990] and/or to feature the *shoen*

Table 1. Performance Evaluation (in seconds)

		binary process tree (5000 operations)		naïve reverse (1000 elements)
Language	Processing	(search)	(update)	
GHC	1 PE (no locking)	1.25	1.83	2.23 (225 kRPS)*
	1 PE	1.38	2.10	3.27 (154 kRPS)
	2 PEs	0.78	1.15	2.43 (207 kRPS)
	3 PEs	0.55	0.81	1.71 (294 kRPS)
	4 PEs	0.44	0.63	1.33 (377 kRPS)
	5 PEs	0.36	0.53	1.10 (456 kRPS)
	6 PEs	0.33	0.46	0.96 (523 kRPS)
	7 PEs	0.33	0.39	0.85 (591 kRPS)
	8 PEs	0.33	0.36	0.77 (652 kRPS)
C (recursion)	`cc -O`	0.71	0.72	
C (iteration)	`cc -O`	0.32	0.35	

(* kilo Reductions Per Second)

construct of KL1 [Ueda and Chikayama 1990], but it should scarcely affect the result of performance evaluation described below.

Locking of shared resources, namely logic variables, goal records, communication cells, the global deque, etc., is done using the xchg (exchange) instruction as usual.

Using Program 1, we measured the processing time of the following:

(1) 5000 `update` operations with random keys, given to an empty binary tree, and

(2) 5000 `search` operations with the same sequence of keys, given to the tree with 4777 nodes created by (1).

The number of processors was changed from 1 to 8. For the one-processor case, a version without locking/unlocking operations was tested as well. The numbers include the execution time of the driver that sends messages to the tree. The result was compared with two versions of sequential C programs using records and pointers, one using recursion and the other using iteration. The performance of `nreverse` (Fig. 4) was measured as well. The results are shown in Table 1.

The results show good (if not ideal) parallel speedup, though for `search` operations on a binary tree, the performance is finally bounded by the sequential nature of the driver and the root node. Access contention on the global deque can be another cause of overhead. Note, however, that the two examples are indeed harder to execute in parallel than running independent processes in parallel, because different chains of message sends share goals. Note also that the binary tree with 4777 nodes is not very deep.

The binary tree program run with 4 processors outperformed the optimized recursive C program. The iterative C program was more than twice as fast as the recursive one and was comparable to the GHC program run with 8 processors. The comparison, however, would have been more preferable to parallel GHC if a larger tree had been used.

The overhead of locking/unlocking was about 30% in `nreverse` and about 10% in the binary tree program. Since `nreverse` is one of the fastest programs in terms of the kRPS value, we can conclude that the overhead of locking/unlocking is reasonably small on average even if we lock such small entities as individual goals.

As for space efficiency, the essential difference between our implementation and C implementations is that GHC goal records have pointers to input streams while C records do not consume memory by being pointed to. The difference comes from the expressive power of streams; unlike pointers, streams can be unified together and can buffer messages implicitly.

One may suspect that message-oriented implementation suffers from poor locality in general. This is true for *data* locality, because a single message chain can visit many goals. However, streams in process-oriented implementation cannot enjoy very good locality either, because a tail-recursive goal can generate a long list of messages. Both process-oriented and message-oriented implementations enjoy good *instruction* locality for the binary tree program and `nreverse`.

Comparison of performance between a message-oriented implementation and a process-oriented implementation was reported in [Ueda and Morita 1990] for the one-processor case.

7. Conclusions and Future Work

The main contribution of this paper is that message-oriented implementation of Moded Flat GHC was shown to benefit from small-grain, tightly-coupled parallelism on shared-memory multiprocessors. Furthermore, the result of preliminary evaluation shows that the absolute performance is good enough to be compared with that of procedural programs.

These results suggest that the programming of reconfigurable storage structures that allow concurrent access can be a realistic application of Moded Flat GHC. Programmers need not worry about mutual exclusion necessitated by parallelization, because it is achieved automatically at the implementation level. In procedural languages, parallelization may well require major rewriting of programs. To our knowledge, how to deal with reconfigurable storage structures efficiently in non-procedural languages without side effects has not been studied in depth.

We have not yet fully studied language constructs and their implementation for more minute control over parallel execution. The current scheme for the control of parallelism is a simple extension to the sequential system; it worked well for the benchmark programs used, but will not be powerful enough to be able to tune the performance of large programs. We need a notion of priority that should be somewhat different from the priority construct in KL1 designed for process-oriented parallel execution. The notion of fairness may have to be reconsidered also. KL1 provides the *shoen* (manor) construct as well, which is the unit of execution control, exception handling and resource consumption control. How to adapt the *shoen* construct to message-oriented implementation is another research topic.

Acknowledgments

This work was done as part of the Fifth Generation Computer Project while the first author was with Institute for New Generation Computer Technology.

References

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura, Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 276–293.

[Inamura and Onishi 1990] Y. Inamura and S. Onishi, A Detection Algorithm of Perpetual Suspension in KL1. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 18–30.

[Knuth 1973] D. E. Knuth, *The Art of Computer Programming, Vol. 1 (2nd ed.).* Addison-Wesley, Reading, MA, 1973.

[Shapiro 1989] Shapiro, E., The Family of Concurrent Logic Programming Languages. *Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

[Ueda and Morita 1990] K. Ueda and M. Morita, A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version to appear in *New Generation Computing*.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (Dec., 1990), pp. 494–500.