Building an Interface Between Eden and Maple: A way of Parallelizing Computer Algebra Algorithms*

Rafael Martínez and Ricardo Peña

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
e-mail: rmartine@fdi.ucm.es, ricardo@sip.ucm.es

Abstract. Eden is a parallel functional language extending Haskell with processes. This paper describes the implementation of an interface between the Eden language and the Maple system. The aim of this effort is to parallelize Maple programs by using Eden as coordination language. The idea is to leave in Maple the computational intensive functions of the (sequential) algorithm and to use Eden skeletons to set up the parallel process topology in the available parallel machine. A Maple system is instantiated in each processor. Eden processes are responsible for invoking Maple functions with appropriate parameters and of getting back the results, as well as of performing all the data communication between processes.
The interface provides the following services: instantiating and terminating a Maple system in each processor, performing data conversion between Maple and Haskell objects, invoking Maple functions from Eden, and ensuring mutual exclusion in the access to Maple from different concurrent threads in the local processor.
A parallel version of Buchberger's algorithm to compute Gröbner bases is presented to illustrate the use of the interface.
Keywords: Functional parallel programming, skeletons, computer algebra algorithms, foreign language interfaces.

1 Introduction

Computer algebra algorithms, usually programmed in specialized systems such as Maple [8], are known to need much computer time. Many of these algorithms run for hours, days and even weeks.

Eden is a parallel functional language extending Haskell with processes [2]. It runs on most Unix-like platforms supporting the PVM (*Parallel Virtual Machine*) library [14]. It has been shown that building parallel algorithms in Eden is a rather easy task [7]. Moreover, Eden provides now a wide library of predefined *skeletons* fitting most of the typical parallel applications [6, 7]. Doing parallel programming with skeletons is as convenient as doing functional programming with higher-order functions.

* Work partially supported by the Spanish project TIC 2000-0738.

To try to bring together the advantages of both systems, Maple and Eden, is a worthwhile effort. On the one hand, computer algebra specialists might continue to use one of their favorite tools while getting the speedups of a parallel implementation. With an appropriate parallel machine, they could easily cut the running time of their algorithms. On the other hand, they would not have to deal with complex parallel libraries such as PVM and the like to explicitly program the parallel versions. In other words, they would not be forced to pay the effort of an explicit parallel implementation. Instead, if Eden provides the appropriate skeleton, only a few simple Haskell functions would have to be provided, as it will be shown in the example of Section 5.

The idea is simple: to leave in Maple the complex algebraic computations and to give to Eden the task of creating and communicating processes. The idea of having a coordination language on top of a computation language is by no means new (e.g., see [4, 9]) but this is the first time that the combination Eden-Maple has been tried. Then, what is needed is an interface through which Eden programs may invoke Maple functions. This paper explains in detail the implementation of such an interface and its use in the development of a complex hybrid Eden-Maple parallel algorithm.

The structure of the paper is as follows: In Section 2 we present previous work, an interface Haskell-Maple, on which we have based our own interface. Section 3 gives a quick introduction to Eden and explains the parallel structure of hybrid Eden-Maple algorithms. In Section 4, we give an account of the concurrency problems which may arise in every single processor and of how we have solved them. Section 5 presents the application example: computing the Gröbner basis of a set of polynomials. Once the problem is introduced, we develop a new Eden skeleton fitting the parallel structure of the algorithm. Then, the skeleton is instantiated with problem dependent functions, so solving in parallel the Gröbner basis problem. It is in some of these problem dependent functions where Maple functions are invoked by using the Eden-Maple interface. Finally, Section 6 draws some conclusions and future work.

2 The Sequential Interface

The starting point for our work has been an interface between GHC [11] and Maple written by Wolfgang Schreiner and Hans-Wolgang Loidl [12] in order to invoke Maple functions from Haskell. The versions used at that time were GHC-4.08.1 and Maple 5.1, both running under Unix-like operating systems. They tried also to build a parallel version to be able to interface *Glasgow Parallel Haskell* programs [15] to Maple but, to our knowledge, this version was never completed.

The main idea of the interface, which we have preserved in our version, is to run Maple 'as it is' in a separate Unix process. All the interface to Maple is done through the standard input and the standard output of this process, i.e. the Haskell process simulates a user terminal for the Maple process. Additionally, a small interpreter was written in the Maple side in such a way that, by using a simple protocol through the standard input/ouput, Haskell could ask the

interpreter to invoke any Maple function and to translate data objects from external format to Maple internal format and the other way around. The idea was to convert the initial data of a complex computer algebra algorithm to Maple internal format; then, to invoke Maple functions by delivering their parameters and recovering their results in internal format; and lastly, to convert the final results of the algorithm to external, character oriented, format. In this way, the total number of format conversions were minimized.

This separation of Haskell and Maple in different processes simplifies many problems concerning the memory management and the merging of both runtime systems, problems that would have arisen in an alternative approach consisting of running both systems as a single Unix process.

The sequential interface consists of a Haskell fragment and a C fragment. The C part is mainly devoted to invoke Posix services. They are needed to initiate the Maple process, to establish the pipe connections with it, and to send commands to or to receive results from this process. The C part also provides static memory to store Unix objects that must be preserved between calls. These are the file descriptors of the pipes, the process identity of the Maple process, and some buffers and global variables.

The Haskell part provides the interface functions to user programs and implements the communication protocol with the Maple process. It invokes the C functions by using the plain `_ccall_` facility of GHC. The main functions a Haskell programmer may use are the following ones:

```
mapleEval        :: String -> [MapleObject] -> MapleObject
mapleEvalN       :: String -> [MapleObject] -> [MapleObject]
string2MapleExpr :: String -> MapleObject
mapleExpr2String :: MapleObject -> String
```

The first two allows Haskell programs to invoke any Maple function just by giving the name and the list of its parameters as Maple objects. The second version must be used to invoke Maple functions returning more than one object. The other two provide conversion of Maple objects from external format to internal one and the other way around. In addition, the Haskell program must make an initial `_ccall_` to function `mapleInit`, and a final `_ccall_` to function `mapleTerm`.

A Maple object is returned by (must be sent to) Maple as a sequence of bytes. From the interface point of view, type `MapleObject` is defined as a Haskell `ByteArray`, a type now deprecated but available in GHC-4.08. Other deprecated Haskell types used by the interface were `Addr` and `MutableArray`. The type `MapleObject` is exported to user programs as an abstract type.

We acknowledge the big effort and the good ideas contained in this interface. But, from our point of view, this is only half of the way we have to go. Despite the fact that a hybrid Haskell-Maple algorithm is executed by two Unix processes, it is obvious that they will never be concurrently computing in the sense that, while the Maple process is involved in a call, the Haskell process must wait for the result before resuming it own computation, and while the Haskell process is computing, the Maple process must also wait for the next command to arrive.

From this perspective, it is not dangerous to call Unix primitives such as `sleep, fwrite, fgets` or `select` that completely block the calling process.

As we will see in the next sections, a hybrid Eden-Maple algorithm is more complex. In principle, several computers (in what follows we will refer to them as PEs, or *processing elements*) may participate in the computation and so a Maple process should be available in each computer needing to access Maple. Also, several concurrent Eden processes may be instantiated in the same PE. Some of them may need to access Maple and some other not. As we will explain, Eden processes, and Eden threads within a process, are *lightweight*, i.e. they do not correspond to Unix processes. All Eden threads running in the same PE are scheduled from inside a single Unix process. From this point of view, it is not admissible to block the entire Unix process just because a thread is accessing Maple. This will block the rest of the threads, what may lead the whole algorithm to a deadlock.

3 The Parallel Structure of Eden-Maple Algorithms

3.1 Eden

The parallel-functional language Eden extends the lazy functional language Haskell by constructs to explicitly define and communicate processes. The three main new concepts are *process abstractions*, *process instantiations* and the non-deterministic process abstraction `merge`.

A *process abstraction* expression of type `Process a b` defines the behaviour of a process having as input a formal parameter of type `a` and returning as output a result of type `b`. Process abstractions are created by the predefined function `process :: (a -> b) -> Process a b` which converts a function into a process. A process instantiation is achieved by using the predefined infix operator `(#) :: Process a b -> a -> b`. The main difference between a process and a function is that the former, when instantiated, is executed in parallel with the rest of the computation.

The evaluation of an expression `e1 # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* is responsible for evaluating and sending `e2` via an implicitly generated channel, while the new *child process* first evaluates the expression `e1` until a process abstraction `process (\x -> e)` is obtained and then the application `(\x -> e) e2`, returning the result via another implicitly generated channel. For input tuples, independent concurrent threads are created in the parent to evaluate each component. Also, if the output is a tuple, an independent thread is created in the child to evaluate each component. Once a process is running, only fully evaluated data objects (or lambda abstractions) are communicated. The only exceptions are lists, which are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access input which is not available yet, are temporarily suspended.

Lazy evaluation is changed to eager evaluation in two cases: Processes are eagerly instantiated, and instantiated processes produce their output even if it

is not demanded. These modifications aim at increasing the parallelism degree and at speeding up the distribution of the computation. The rest of the language is as lazy as Haskell.

Non-determinism is introduced in Eden by means of a predefined process abstraction `merge :: Process [[a]] [a]` which *fairly* interleaves a set of input lists, to produce a single non-deterministic list. Its implementation immediately copies to the output list any value appearing at any of the input lists. So, `merge` can profitably be used to quickly react to requests coming in an unpredictable order from a set of processes. This feature is essential in reactive systems and very useful in some deterministic parallel algorithms (see skeleton in Section 5.2).

A last feature are *dynamic reply channels*. The predefined function `new :: (ChanName a -> a -> b) -> b` creates a channel of type `a` whose name of type `ChanName a` can be sent to a remote process, and then executes an expression of type `b`. The creating process can receive data from the remote one by trying to evaluate the created channel. The remote one can send a value of type `a` through the channel by using its name and the predefined function `parfill :: ChanName a -> a -> c -> c`. This function creates a concurrent thread to send an expression of type `a` through the dynamic channel of type `ChanName a` and then executes an expression of type `c`. As an example, consider the following program where process `one` receives an integer from the remote process `two` by using the dynamic channel `c` of name `cn`:

```
one :: Process Int Int
one = process (\x -> new (\cn c -> let nothing = two # cn in x + c))

two :: Process (ChanName Int) ()
two = process (\cn -> parfill cn 3 ())
```

The Eden runtime system runs on top of the parallel library PVM [14]. This creates a parallel virtual machine by instantiating, previously to run time, a number of PVM processes. A PVM process corresponds to a Unix process and, ideally, only a PVM process should be instantiated in each PE (otherwise the virtual machine would be concurrent rather than parallel). The correspondence between Eden processes and PVM processes is as follows: all PVM processes of the parallel virtual machine contains a copy of the Eden code. Initially, only PVM process 0 has activity by executing the main Eden expression called `main`. The rest of the PVM processes are in a quiescent state. Each time a new Eden process is instantiated by the operator `#`, their threads are created in a different PVM process which abandons its quiescent state. When/if the number of instantiated Eden processes becomes bigger that the number of PVM processes, then one or more PVM processes will be overloaded with new Eden processes. The threads of all processes within a PVM process are managed by the Eden runtime system. We call this runtime system a *Dream* (*Distributed Eden Abstract Machine*). Summarizing, there is a bijective correspondence between PVM processes and Dreams, a many-to-one correspondence between Eden processes and Dreams, and a many-to-one correspondence between threads and Eden processes. All threads within a Dream share the same heap but each one has its own stack.

Eden has been successfully used to implement many parallel algorithms using machines with several dozens of processors. Good speedups have been obtained

and satisfactory comparisons with other parallel functional languages have been established [5]. A number of skeletons [3] have been defined in Eden to fit most of parallel algorithms [7]. Examples of useful skeletons are: *parallel map, parallel divide and conquer, parallel branch and bound, parallel iterate until, torus and ring topologies, etc.*. A distinguishing feature of Eden is that it gives programmers the possibility to define their own skeletons or to adapt the existing ones to their needs given that skeletons are just Eden programs.

3.2 Algorithms Eden-Maple

As it has been said in the introduction, the idea to parallelize computer algebra algorithms is to use Eden as the coordination language and Maple as the computing language. Eden would be responsible for instantiating processes, communicating them and controlling the global load balancing of the parallel algorithm. Ideally, this should be done in a problem independent way by using or creating a polymorphic higher-order skeleton.

Maple functions would be responsible for doing the intensive algebraic computations. All the problem specific aspects (data types, sequential algorithms, etc.) should be coded into simple Haskell functions calling Maple functions through the interface. These Haskell functions would be used to fit the parameters of the polymorphic skeleton. We believe that this separation of concerns leads to a low-effort parallelization of computer algebra algorithms. In Section 5 we illustrate the methodology with a typical computer algebra problem.

So, in the worst case, a Maple process is needed in each PE. It makes no sense to have several Maple processes in the same PE as only one of them would be computing at a given time. So, we establish a bijective correspondence between PVM processes needing Maple and Maple processes. We will call the Maple process the *companion* process of the corresponding Dream. Two unidirectional pipes will connect each Dream to its companion as in the sequential interface described in Section 2. Figure 1 shows the process topology.

The interface will create the companion process and establish the pipes the first time an Eden thread calls the initialization function, called now `mapleInitPE`. Accordingly, it will kill the companion process when the last thread using Maple calls the termination function, called now `mapleTermPE`.

In order to upgrade the interface to this new environment, the first important change has been to replace the intermediate C functions calling Unix by calls to the Posix package supported by GHC. The following functions are called:

```
forkProcess   :: IO (Maybe ProcessId)
createPipe    :: IO (Fd, Fd)
dupTo         :: Fd -> Fd -> IO ()
executeFile   :: FilePath->Bool->[String]->Maybe [(String,String)]-> IO ()
signalProcess :: Signal -> ProcessId -> IO ()
```

They are responsible for instantiating the companion, establishing the pipe connections, instructing the OS to initialize the companion with a Maple executable, and killing the companion when Maple is no longer needed. However, we refuse to use functions used by the sequential interface such as `fdRead, fdWrite` and

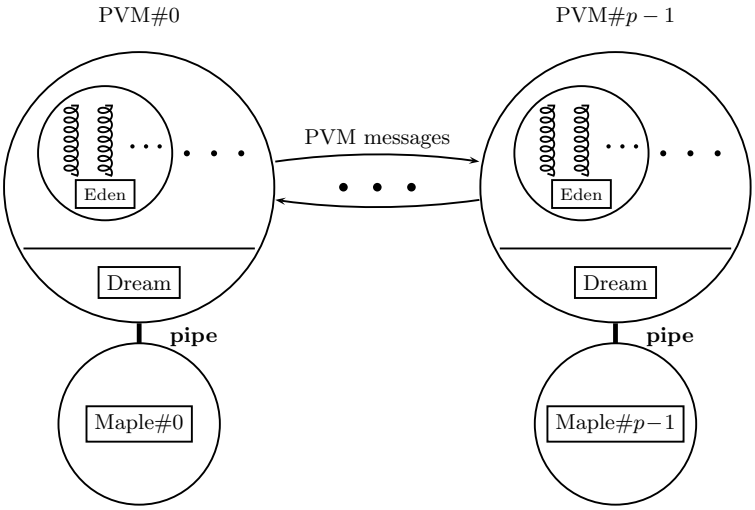

Fig. 1. Process topology of a hybrid Eden-Maple program

`fdClose` which block the calling Unix process. Instead, we lift Unix objects such as `Fd` (file descriptor of a pipe) to Haskell's world by using the conversion

```
fdToHandle :: Fd -> IO Handle
```

and then calling the standard Haskell I/O functions `hGetLine, hPutStrLn` and `hClose` which use a `Handle` as parameter. It must be said that the protocol with the Maple process is line-oriented, although a 'line' may as long a needed. These functions have been designed for a concurrent runtime system and only block the calling thread. Fortunately, the Eden runtime system is an extension of the Concurrent Haskell one [10] and these primitives still work with Eden.

For the same reason, we have replaced calls to Unix `sleep :: Int -> IO ()` by calls to Concurrent Haskell `threadDelay :: Int -> IO ()`.

Altogether, these changes have allowed us to eliminate most of the `_ccall_` and a big portion of the C part (the number of C pages has decreased from 11 to only 2 in the new interface), achieving a more compact and legible code.

4 Concurrency Issues within a Processor

The next group of changes deals with providing mutual exclusion in the access to Maple from different Eden threads. We have established the following protocol for an Eden process `p` to access Maple functionality:

```
p :: Process a b
p = process (\x -> unsafePerformIO $
            do mapleInitPE
               let y = ··· compute result from x using Maple ···
               mapleTermPE `demanding` rnf y
               return y)
```

The function `rnf` reduces result `y` to normal form. This one and the `demanding` function are defined in module `Strategies` developed by the creators of GpH (see [16] for details). They are needed to ensure that no calls to Maple are made after the call to `mapleTermPE`, as it would be the case if we allowed the result `y` to be lazily evaluated.

At runtime, there may be many Eden processes in the same PE following this protocol. So, the first problem to be solved is to initialize the companion Maple process only once, namely when the first call to `mapleInitPE` is received. The rest of the calls to `mapleInitPE` are just counted in order to know how many Eden threads are concurrently using Maple. When calls to `mapleTermPE` are received, the counter is decreased. Only when the counter becomes 0 (meaning that no more threads are using the interface) should the companion process be killed.

To this aim, the interface provides a global variable in C static memory counting the number of threads allowed to use the Maple interface. The same variable is also used to provide a primitive form of mutual exclusion during the transient state in which the first thread is creating the companion process but it is not still available. This may take a noticeable time during which the interface must not allow the remaining threads to proceed.

Once the interface is stable, the second problem is to make calls to Maple functions atomic, i.e. a critical region should be created from the call to the interface up to the returning of results to the user program. We remind the reader that Maple may need a noticeable time to reply to a call because Maple functions are assumed to perform the computation intensive parts of the algorithm.

To solve this problem we rely on Eden dynamic reply channels (see Section 3.1). For each Eden thread calling Maple when the interface is busy, the latter creates a reply channel by using Eden function `new` and blocks the thread on it. When the interface becomes idle again, the first waiting thread is awaked by sending a value through its corresponding channel using the Eden function `parfill`. These synchronizations create the desired critical region around each call. Reply channels must be stored somewhere between calls to the interface, so they are kept in C static memory. Conversions between Haskell objects and C data types are done through the facilities of GHC's `Ptr` library:

```
newStablePtr   :: a -> IO (StablePtr a)
deRefStablePtr :: StablePtr a -> IO a
freeStablePtr  :: StablePtr a -> IO ()
```

We have also developed an alternative mutual exclusion mechanism by using the Concurrent Haskell `MVar` [10]. A nonempty `MVar` is created by the interface at initialization time by using the function `newMVar`. Critical regions are created by executing `takeMvar` at the beginning of each Maple call, and `putMVar` at the end. This is transparent to the programmer: the `MVar` is created by the interface and stored in static memory between calls; calls to `takeMvar` and `putMVar` are done from inside the interface. Concurrent Haskell `MVar` works properly with Eden runtime system. The interface version with `MVar` could of course be used by Concurrent Haskell programs.

The part of the interface devoted to the proper shyncronization of the concurrent calls has been the more involved, and is responsible for half of the code.

```
function Buchberger (F = {f_1, ..., f_s}) return G
  G := F; P := {(f_i, f_j) | f_i, f_j ∈ F, i ≠ j};
  while P ≠ ∅ do
    (f, g) ← chooseAPair (P); P := P − {(f, g)}
    S(f, g) ⟶*(G) h such that h is reduced w.r.t. G
    if h ≠ 0 then
      P := P ∪ {(u, h) | u ∈ G};
      G := G ∪ {h}
    end if
  end while
  return G
end function
```

Fig. 2. Buchberger's sequential algorithm computing a Gröbner basis

5 Case Study: Gröbner Bases

5.1 The sequential algorithm

Gröbner bases computation is a very well-known algorithm for computer algebra researchers. Gröbner bases have plenty of applications in commutative algebra, geometry and systems theory. The problem can be explained in the following terms: Given a finite set of polynomials $F = \{f_1, \ldots, f_s\}$ in n indeterminates $x_1, \ldots, x_n$, a Gröbner basis is another finite set of polynomials $G = \{g_1, \ldots, g_t\}$ determining the same ideal and satisfying an additional canonical property.

The ideal I determined by a set S of polynomials, denoted $I = \langle S \rangle$ is the smallest set containing S and closed under polynomial addition and product:

$$\langle S \rangle \stackrel{\text{def}}{=} \{ \sum_{f_i \in S} u_i f_i \mid u_i \in P[x_1, \ldots, x_n] \}$$

being $P[x_1, \ldots, x_n]$ the set of all polynomials in n indeterminates.

Given an ideal I determined by a finite set F of polynomials, there exists an algorithm due to B. Buchberger [1] which computes a Gröbner basis G for I starting from F. It is shown in Figure 2. It makes intensive use of two elementary steps: computing to so called called S-polynomial of two polynomials f and g, denoted $S(f, g)$, and the *reduction* of a polynomial r to normal form h with respect to a set G of polynomials, denoted $r \stackrel{G}{\longrightarrow} * h$ If G is finite, being s its cardinality, the algorithm for computing $r \stackrel{G}{\longrightarrow} * h$ is quadratic. More precisely, it belongs to $O(ms)$ in the worst case, being m the length of the maximum strictly decreasing chain of power products that can be constructed starting with the leading term of f. This, in turn, is related to the degree of f and the number n of indeterminates. The cost of computing $S(f, g)$ is in $O(n)$.

It has been proved that the algorithm always terminates and that its cost is in $O(msp)$, where m, s are as before —now, they are considered to be worst case values for the polynomials and the cardinality of G—, and p is the number

of pairs in the final G. The value of p is *a priori* unknown and depends on the form of the initial polynomials in F. In the worst case, p can be exponential on the cardinality of F.

Maple systems usually provide a sub-library to compute Gröbner bases. But they also provide the elementary steps of the algorithm as individual functions. In particular, there exists a function called `spoly` computing the S-polynomial of two given polynomials, and a function called `normalf` computing the reduction of a polynomial to normal form with respect to a set of polynomials.

5.2 The stateful replicated workers skeleton

As we (as Eden programmers) are not interested in doing polynomial crunching in Haskell, the idea for the parallel version of Buchberger's algorithm is to leave in Maple the computation of $S(f,g) \xrightarrow{G} * h$ and to compute this reduction to normal form in parallel for different pairs (f,g). The order in which such pairs are chosen is not important for the correctness and the termination of the algorithm, so they can safely be done in parallel. The granularity of such computation is large enough to justify the communication of the polynomials f and g. So, the strategy chosen is to have a *manager* process communicating pairs (f,g) to a fixed set of *worker* processes, and getting back the results h of such reductions. If the result is 0, the manager just moves to the next pair. If it is different from 0, the manager computes additional pairs which are joined to the list of pending pairs.

In Eden we have developed several versions of a skeleton called *replicated workers* [6], which fits this idea of a manager process distributing work to a fixed number of workers. The main interesting property of the skeleton is that it achieves a very good load balancing as work is distributed to workers on demand: as soon as a worker finishes a task, it is fed with a new one, if there is one available. So, it may be the case that a worker solves a few big granularity tasks, while another one solves many small granularity tasks.

We have adapted one version of the replicated workers skeleton to fulfill the following new needs:

1. Worker processes must maintain an internal state. Notice that workers must reduce a pair (f,g) with respect to a polynomial set G which is changing along time.
2. There must be provisions to update from time to time workers' internal state.
3. The manager has also an internal state updatable as a consequence of a worker result.
4. The algorithm output depends on the final state reached to by the manager.

We call the resulting skeleton *stateful replicated workers*, abbreviated `strw`. This insistence in developing first a problem independent skeleton and then instantiating it with problem dependent parameters is just a separation of concerns strategy common to all areas of software development. In doing so, we concentrate first on the parallel nature of the problem, i.e. on the process topology and the load balancing issues of the algorithm and then, as a separate activity, on

```
strw  :: (Trans tsk, Trans act, Trans res, Trans wl) =>
      Int ->                                        -- no. of PE
      Int ->                                        -- buffer size
      (inp -> Int -> ([wl],[tsk],ml)) ->            -- split function
      (wl -> tsk -> [act] -> (res,wl)) ->           -- worker function
      (ml -> res -> Int -> ([[act]],[tsk],ml)) -> -- combine function
      (ml -> result) ->                             -- result function
      inp ->                                        -- skeleton input
      result                                        -- skeleton result

strw np prefetch split wf combine rf inp =  r
 where
  (iniwls,iniTasks,iniml)  = split inp np
  outss                    = [process (worker i wf) #  (wl,actsks) |
                              (i,wl,actsks)<-zip3 [0..np-1] iniwls actskss]
                              `using` spine
  unorderedResults         = merge # outss
  (moreReqs,results)       = unzip unorderedResults
  (moreTks,asss,moreGns,r) = manager np combine rf iniml results
  iniReqs                  = concat (replicate prefetch [0..np-1])
  iniGens                  = concat (replicate prefetch (replicate np 0))
  tasks                    = iniTasks ++ moreTks
  actskss                  = distribute np 0 (length iniTasks)
                              (replicate np 0) tasks asss
                              (zip iniReqs (repeat 0)++zip moreReqs [1..])
                              (iniGens ++ moreGns)
```

Fig. 3. The Eden skeleton `strw` of *Stateful Replicated Workers*

problem specific issues. There are also advantages in the testing phase: first we test the skeleton with a toy problem and, once the skeleton is properly working, we feed it with the (more complex) Gröbner basis computation problem.

Figure 3 shows the type and the implementation of `strw` in Eden. We begin by explaining the type. Eden type class `Trans` includes all types which can be transmitted in messages. Essentially, they are those for which a normal form can be computed. The first two parameters are the number of workers to be created by the skeleton and the size of the prefetch buffer. We usually create a worker per PE and leave the manager process to share a PE with one of the workers. The load on the manager is low and it does not justify a complete PE for it. The worker sharing its PE with the manager will do less work than the others, but this is not important as PE loads are dynamically balanced by the manager. The prefecth buffer size is the number of tasks initially assigned to each worker. We usually choose this parameter to be 2 so that, when a worker finishes the current task asking for a new one, and while this one arrives, it may work on the task in the buffer. As a consequence, workers idle times are minimized.

The next four parameters are the problem dependent functions delivered to the skeleton. The `split` function is initially called by the skeleton to get the

initial state of each worker, the initial set of tasks and the initial state of the manager. It receives as parameters the input data `inp` of the skeleton, and the number `np` of workers. The worker function `wf` is called by the skeleton to solve each individual task. It receives as parameters the current state of the worker, the task to be solved, and a list of pending updates. It delivers a result and an updated internal state. The manager is responsible for accumulating the pending updates for each individual worker. The updates are generated as a consequence of results received from other workers in the meantime while the worker is solving a task. The `combine` function receives the manager internal state, a worker's result and the number `np` of workers, and computes three results: a list of updates, one for each worker, a list of new tasks, and a new internal state for the manager. The internal list in the type `[[act]]` is used as a `Maybe` type: an empty list means no update; otherwise, the list contains a single update for the worker. The `combine` function is called by the skeleton each time a worker's result is received. Finally, the result function `rf` is called at the end of skeleton's execution to compute the output of the algorithm from the final internal state of the manager.

The implementation of `strw` presents the aspect of a set of mutually recursive definitions. These are needed to establish a circular communication topology between the manager and the workers. The communication channels are essentially lists: a worker receives a list of tuples, each one containing a list of pending updates and a task, while the manager receives a list of unordered results coming from the workers. The results are received in the temporal order in which they are produced. To achieve this goal, it is essential the application of the reactive process `merge` to the list of lists `outss` of outputs produced by the workers. Strategy `spine` is used to eagerly instantiate the worker processes.

Perhaps the most important auxiliary function of the skeleton is `distribute` whose details are shown in Figure 4. Its purposes are:

1. To detect when a worker has finished a task and to assign it a new one.
2. To compute the list of pending updates for each individual worker and to include it together with the new assigned task.
3. To detect the termination of the skeleton. To this aim, it controls the number `ngen` of tasks generated by the skeleton, the number `ndis` of tasks distributed to workers, and the number `nrec` of results received from workers. The termination condition is `ngen == ndis && ndis == nrec`.

5.3 The problem dependent functions

First, we instantiate the polymorphic types of the skeleton for our problem:

$$
\begin{array}{lcl}
\texttt{inp} & \stackrel{\text{def}}{=} & \{f_1, \ldots, f_s\} \subseteq P[x_1, \ldots, x_n] \\
\texttt{wl} & \stackrel{\text{def}}{=} & G \subseteq P[x_1, \ldots, x_n] \\
\texttt{tsk} & \stackrel{\text{def}}{=} & (f, g) \in P[x_1, \ldots, x_n] \times P[x_1, \ldots, x_n] \\
\texttt{ml} & \stackrel{\text{def}}{=} & G \subseteq P[x_1, \ldots, x_n] \\
\texttt{res} & \stackrel{\text{def}}{=} & [\,], [h] \in \mathit{List}\ (P[x_1, \ldots, x_n]) \\
\texttt{act} & \stackrel{\text{def}}{=} & h \in P[x_1, \ldots, x_n] \\
\texttt{result} & \stackrel{\text{def}}{=} & G \subseteq P[x_1, \ldots, x_n]
\end{array}
$$

```
distribute :: Int -> Int -> Int -> [Int] -> [tsk] -> [[[act]]] ->
              [(Int,Int)] -> [Int] -> [[([act],tsk)]]
distribute np ndis ngen cs ts asss ((i,nrec):is) (n:ns)
 |ngen' == ndis && ndis == nrec = replicate np []
 |ngen' == ndis && ndis >  nrec = distribute np ndis ngen' cs ts  asss is ns
 where ngen' = ngen + n
distribute np ndis ngen cs (t:ts) asss ((i,nrec):is) (n:ns) =
 insert i (as,t) (distribute np (ndis+1) (ngen+n) cs' ts asss' is ns)
 where ndif                    = nrec - cs !! i
       (ass1,ass2)             = splitAt ndif (asss !! i)
       as                      = concat ass1
       cs'                     = replace i nrec cs
       asss'                   = replace i ass2 asss
       replace 0     e ~(x:xs) = e : xs
       replace (n+1) e ~(x:xs) = x : replace n e xs
       insert 0 e ~(x:xs)      = (e:x) : xs
       insert (n+1) e ~(x:xs ) = x : insert n e xs
```

Fig. 4. Distribution function of skeleton `strw`

By looking at the sequential algorithm of Figure 2, it is straightforward to define the four problem dependent functions. We give them in a schematic way as the coded version would exhibit more (non essential) details:

$$
\begin{array}{ll}
\texttt{split}\ \ F\ \ \texttt{np} & \stackrel{\text{def}}{=} (\overbrace{[F,\ldots,F]}^{\texttt{np}}, [(f_i,f_j) \mid f_i,f_j \in F, i \neq j], F) \\
\texttt{wf}\ G\ (f,g)\ [h_1,\ldots,h_r] & \stackrel{\text{def}}{=} (\texttt{res}, G') \quad \text{where} \\
 & \quad G' = G \cup \{h_1,\ldots,h_r\} \\
 & \quad \texttt{res} = \begin{cases} [\,] & \text{, if } S(f,g) \xrightarrow{G'} * \, 0 \\ [h] & \text{, if } S(f,g) \xrightarrow{G'} * \, h \neq 0 \end{cases} \\
\texttt{combine}\ \ G\ \ \texttt{res}\ \ \texttt{np} & \stackrel{\text{def}}{=} (\overbrace{[\texttt{res},\ldots,\texttt{res}]}^{\texttt{np}},\ \texttt{tsks},\ G') \quad \text{where} \\
 & \quad (\texttt{tsks}, G') = \begin{cases} ([\,], G) & \text{, if } \texttt{res} = [\,] \\ ([(u,h) \mid u \in G], G \cup \{h\}) & \text{, if } \texttt{res} = [h] \end{cases} \\
\texttt{result}\ \ G & \stackrel{\text{def}}{=} G
\end{array}
$$

The worker function `wf` does its work by delegating to Maple the computation of $S(f,g) \xrightarrow{G'} * \, h$. To this aim, it calls through the interface the Maple functions `spoly` and `normalf`, respectively computing the S-polynomial of f and g and its reduction to normal form with respect to the updated set of polynomials G'.

In order to initialize and to finalize the interface, skeleton `strw` has been slightly modified. On the one hand, the predefined function `process` called in line 4 of `strw` definition in Figure 3, has been replaced by function `mapleProcess` defined as follows:

```
mapleProcess :: (Trans a, Trans b) => (a -> b) -> Process a b
mapleProcess f = process (\x -> unsafePerformIO $
```

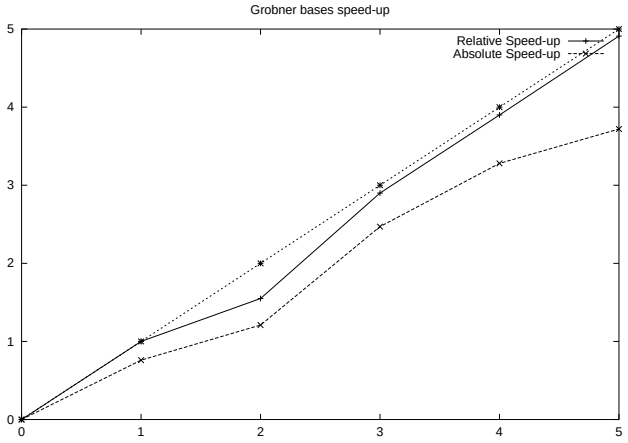

Fig. 5. Absolute and relative speedups for Grobner basis using `strw`

```
        do mapleInitPE
           let res = f x
           return res)
```

On the other hand, the first line of the definition of function `worker`, which represents the termination of the worker process, has been replaced by the following fragment:

```
worker i wf (local,[]) = unsafePerformIO $ do mapleTermPE
                                              return []
```

These modifications do not exactly match the process scheme at the beginning of Section 4. The reason has to do with preserving the laziness of the lists produced by the workers. Should we add to `mapleProcess` a line `mapleTermPE ‘demanding‘ rnf res` before `return res`, the entire skeleton would become blocked (in fact, we have 'seen' this deadlock while debugging the skeleton). The interface must be closed when worker's output list has been completely produced.

5.4 Performance results

We have run all the pieces together —the skeleton, the problem dependent functions and the Eden-Maple interface— in a home-made Beowulf cluster with five processors at 233 Mhz CPU and 64 Mb RAM, running *i386-mandrake7.2-linux*, Eden compiler *mec-5.02.3*, *Maple 7*, and *PVM 3.4*. The final Gröbner basis had 33 polynomials with a worst case of 4 indeterminates and a leading term of degree 8. A total of 528 tasks were generated and the absolute sequential time of the pure Maple algorithm was 212 sec. The speedups obtained can be seen in Figure 5. We have got an absolute speedup of 3.72 with 5 processors with respect to the pure Maple sequential version, and a rather good relative speedup of 4.91 with respect to the parallel version running on one processor. This one is already

32% slower than the reference Maple version, so better absolute speedups cannot be expected. This 32% overhead is due to having two Unix processes communicated by pipes, to the format conversions and to the Eden runtime system. All this machinery is absent in the Maple version.

So, we do not claim that parallelizing Maple using Eden is optimal compared to other approaches. In [13] a survey of many other strategies to parallelizing Maple programs can be found. All of them require some extra programming effort for splitting the work into parallel tasks, assigning tasks to processors or/and sending explicit messages between tasks. Perhaps [13] is one of the most implicit programming models we have found but, nevertheless, it requires new Maple primitives for launching remote tasks and for waiting for results. Our claim is that parallelizing with Eden is *easier* compared to more explicit approaches and, however that, *acceptable* speedups (as opposed to optimal ones) can be obtained.

The scalability of the replicated workers skeleton is good up to around 20 processors, depending on the concrete problem (see [7]). For a bigger number of processors the manager process becomes a bottleneck and the speedup decreases. We plan to do more complete masurements in the near future.

6 Conclusions

The first contribution of the paper is the engineering effort of developing the interface. In doing this work, we have put into action many different technologies. Firstly, we have achieved that two systems so distant from each other such as Maple and Eden may work together in a single problem. Also, while building the interface, we have made use of GHC libraries such as Posix, Ptr, Concurrent Haskell functions and Glasgow Parallel Haskell Strategies. Standard Haskell input/output with handles has also found its place in the interface. Finally, we have needed some knowledge of Unix functions, on how to interface Haskell to C, and on how to set up PVM.

With respect to the original sequential interface, the changes has been of three types: a) eliminating the old types and libraries in order to port the code to the new version of GHC; b) replacing primitives blocking the Unix process by primitives blocking only the current thread; and c) introducing several kinds of locks and synchronization primitives in order to cope with concurrent calls to the interface. As a whole, less than 10% of the original code remains.

The second contribution has been the parallelization of a non trivial computer algebra algorithm and showing that all the pieces, Eden, Maple and the interface, fit together. Now that the interface is running, we hope to reinforce our links with the computer algebra department of our university and to be able to parallelize some of their interesting algorithms. The proposed strategy is the cooperation between functional and computer algebra groups. Functional people would be responsible for understanding the sequential algorithm and for providing the appropriate Eden skeleton, while algebra people would implement the problem dependent functions of the skeleton and the Maple side of the algorithm.

Acknowledgments We appreciate again the work made by Wolfgang Schreiner and Hans-Wolgang Loidl in their sequential interface. We are also grateful to our

Marburg colleagues Rita Loogen, Jost Berthold and Nils Weskcamp for upgrading the Eden compiler to GHC 5.02 and for answering our numerous questions.

References

1. W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*. American Mathematical Society, 1994.
2. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report, Bericht 96-10. Revised version 1.998, Philipps-Universität Marburg, Germany, 1998.
3. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research monographs in parallel and distributed computing. Pitman, 1989.
4. J. Darlington, Y. K Guo, H. W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In *Proceedings of Europar*, volume 996, pages 55–69. Springer-Varlag, 1995.
5. W. Loidl H, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, A. J. Rebón Portillo, S. Priebe, and P. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.
6. U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. In *Constructive Methods for Parallel Programming (CMPP'2000)*. Nova Science, 2002.
7. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. *Patterns and Skeletons for Parallel and Distributed Computing*. F. Rabhi and S. Gorlatch (eds.), chapter *Parallelism Abstractions in Eden*, pages 95–128. Springer-Verlag, 2002.
8. R. Nicolaides and N. Walkington. *MAPLE, A Comprehensive Introduction*. Cambridge University Press, 1996.
9. S. Pelagatti. *Structured Development of Parallel Programs*. Taylor and Francis, 1998.
10. S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Symp. on Principles of Prog. Lang. POPL'96*. ACM Press, 1996.
11. S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.
12. W. Schreiner and H.-W. Loidl. GHC-Maple Interface, version 0.1. Available at http://www.risc.uni-linz.ac.at/software/ghc-maple/, 2000.
13. W. Schreiner, C. Mittermaier, and K. Bosa. Distributed Maple: Parallel Computer Algebra in Networked Environments. Preprint submitted to the Journal of Symbolic Computation, 2002.
14. V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, 1990.
15. P. W. Trinder, K. Hammond, J. S. Mattson Jr., and A. S. Partridge. GUM: a Portable Parallel Implementation of Haskell. In *ACM SIGPLAN PLDI, Philadelphia, USA*. ACM Press, May 1996.
16. P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.