# Towards a Functional Library for Fault-Tolerant Grid Computing*

Bor-Yuh Evan Chang, Margaret DeLap, Jason Liszka, Tom Murphy VII,
Karl Crary, Robert Harper, and Frank Pfenning

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
{bechang,mid,jliszka,tom7,crary,rwh,fp}@cs.cmu.edu

**Abstract.** To make development of grid applications less arduous, a natural, powerful, and convenient programming interface is required. First, we propose an expressive grid programming language which we hope will provide such an interface. Then we show how to map programs in this language onto a low-level, more compact architecture that can more easily provide the fault tolerance and inexpensive scheduling suitable for grid computing. Finally, we discuss programming techniques for taking advantage of the underlying architecture, as well as issues to be resolved in future work.

## 1 Introduction

While numerous grid frameworks have been proposed to exploit idle resources on the global network, we seek a natural and effective abstraction for programming the often unreliable grid. In an accompanying paper, we describe the ConCert grid software, which uses certified code to provide safety, security, and privacy assurances in a trustless setting [CCD+02]. We demonstrate the ability to implement applications with simple parallel models, such as a ray tracer, on this network. In such highly parallel applications, the program on the grid does not need to interface directly with the grid framework and, in essence, is not aware of where it is running. Unfortunately, this method is ill-suited for applications that do not have such trivial branching structure and cumbersome even for programs that do. In this paper, we discuss mapping expressive high-level language ideas to the succinct but robust low-level interface that ConCert provides.

## 2 Programming Model

We seek to give the developer of grid applications access to the parallelism from *within* his application in a convenient and simple way. Drawing on Multilisp *futures* [Hal85] and a similar construct in Cilk-NOW [BL97], our primitive notion

* The ConCert Project is supported by the National Science Foundation under grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination".

on the grid is a *task*, a piece of code that runs to produce a result. Grid programs are composed of a number of *tasks* executing in parallel.

Below, we give a simplified interface for creating and managing tasks as a Standard ML [MTHM97] signature. The actual interface is slightly more sophisticated to allow for better control or for optimization. Implementing this interface for the ConCert software would require special compiler support, which has not been completed. However, we have written a simulator and a significant grid application, namely a parallel theorem prover [Cha02]. A value of type $\rho$ `task` is a handle for a computation running on the grid that returns an answer of type $\rho$. For example, an `int task` is a task that returns values of type `int`. To put a new task onto the network, `inject` is used by passing it a function to run and yields a handle to a new task on the grid.

```
signature CCTASKS =
sig
  type ρ task      (* an ρ task is a computation yielding a result of type ρ *)
  val inject  : (unit -> ρ) -> ρ task
  val sync    : ρ task -> ρ
  val syncall : ρ task list -> ρ list
  val relax   : ρ task list -> ρ * ρ task list
  val forget  : ρ task -> unit
end
```

The only form of explicit communication between tasks is returning and receiving results. This is vital to ensuring the restartable nature of tasks, which is important for implementing failure recovery. The most basic method to ask for a result is to use `sync`, which blocks the calling task until the desired result can be acquired from the grid. The `syncall` construct waits for a list of tasks but is more efficient than successive calls to `sync`. Lastly, we desire a means to continue execution as soon as one result is ready from a set of tasks. The `relax` construct completes as soon as one result is available, returning that result along with the remaining tasks.

To give the developer some control over the proliferation of tasks, calling the `forget` function provides a hint to the ConCert software that the given task is no longer needed. Since the underlying architecture must deal with failures, we consider this a special case of failure in which the task dies deliberately.

**Example: Merge Sort.** In figure 1, we give a (hypothetical) example of a simple merge sort implementation that recursively divides into three subproblems using the interface described above. Lines 4–14 contain straightforward implementations of partitioning a list into three and merging two lists. In lines 16–18, tasks for the subproblems are injected into the network. Then, in lines 22–24, we wait for the results. Notice that we use the `relax` primitive to begin merging as soon as we have two sorted lists.

**Mapping to ConCert.** The work on the programming model as described above has been coordinated with progress on the ConCert grid software. In ConCert, programs are broken up into fragments called *cords* to ease scheduling and to

```
fun mergesort []  () = []
  | mergesort [x] () = [x]
  | mergesort l   () =
    let (* partition3 - split a list into 3 equal parts *)
        ...
        (* merge      - merge two sorted lists *)
        ...
        val (lt,md,rt) = partition3 l
        val t1 = inject (mergesort lt)
        val t2 = inject (mergesort md)
        val t3 = inject (mergesort rt)

        (* Get the results of the three child tasks.  Start
           merging when two sorted lists have been received. *)
        val (a, rest) = relax [ t1, t2, t3 ]
        val (b, [last]) = relax rest
        val (sort1, sort2) = (merge (a,b), sync last)
    in
        merge (sort1, sort2)
    end
```

**Fig. 1.** Programming Example: Merge Sort

enable failure recovery. Cords take arguments (as dependencies on other cords) and return a result to the caller. Cords adhere to the following invariants: (1) cords do not block once they begin execution; (2) execution of cords is deterministic; (3) cords only rely on the results of other cords and not any secondary effects. The first invariant simplifies scheduling because we need not suspend a running cord. The remaining invariants make it possible to recover from failure by restarting any cord.

Notice that tasks do not satisfy the first invariant because they may block `sync`ing on other tasks; however, tasks can be compiled to cords by a process similar to a continuation passing style transformation [FSDF93]. Specifically, a call to `sync` *id* terminates the current cord and spawns a new cord with a dependency for the result of *id*. ConCert supports both *and*- and *or*-style dependencies; `syncall` is compiled in a similar way with *and*, `relax` with *or*. By compiling to cords from high-level primitives, we retain the ability to carry out failure recovery and simple scheduling while increasing the expressiveness of the language.

### 2.1 Advanced Techniques

We have shown one way to use the simple cord primitives to provide more advanced functionality (a blocking `sync` function). Now, we look at ways to leverage the restartable nature of cords to implement failure recovery techniques similar to *checkpointing* and *message logging* [Joh89].

**Checkpointing.** In our model of failure recovery, we can only recover a failed task from the beginning of a cord. For long computations that need not otherwise spawn cords, this means that we have very coarse-grained recovery. Fortunately, our interface gives the ability to implement checkpointing easily. Rather than have a task return an `int` (for instance), we can have it return an `intcheckpoint` (figure 2).

```
datatype intcheckpoint =
    Done of int
  | More of intcheckpoint task
```

```
fun loop(n, r) =
  if n = 1000000 then Done(r)
  else let fun rest () = loop(n + 1, f(r))
       in
         if (n mod 5000) = 0
         then More(inject(rest)) else rest()
       end
```

**Fig. 2.** Checkpointing Example

A value of type `intcheckpoint` is either the tag `Done`, with the integer result of the task, or `More` with the identifier of another task that returns an `intcheckpoint`. A program expecting a result from a checkpointing task just loops, doing a `sync` on any `More` result and succeeding when a `Done` is received. Now, at any moment during the evaluation of the task, it can simply spawn a new task (thus saving its state) and return that task's identifier instead. The `loop` code iterates the function `f` one million times, checkpointing every five thousand calls.

**Modeling Communication.** Our definition of cords prohibits communication between them except in the form of dependencies and results. However, we are again able to use the cord primitive to implement more sophisticated communication. For example, suppose we wanted to implement a task $S$ that sends a sequence of integers that it computes to another task $R$. To accomplish this with cords, we can make the result of the cord implementing $S$ be a pair: the first integer and the identifier of a cord that computes the rest of the sequence. Task $R$ simply needs to `sync` on $S$ (as described in section 2) for the first integer and also receives a cord identifier for the remainder.

```
datatype intstream =
    Empty
  | Cons of int * intstream task
```

In fact, we can also implement certain kinds of synchronous two-way communication. Suppose we have a client task $C$ and a server task $S$. The client sends string commands to the server, which responds with integers.

We can model this similarly to the previous example, except that the server passes back a *function* that returns a task rather than a task identifier. We can think of this function as *blocking* waiting for the client's command. The example in figure 3 consists of a server that returns, in response to a string, the length of the longest string it has received so far.

The `startserver` function takes an integer `m` (its state, indicating the largest string it has seen) and a string command, and returns the id of a `server_result task`. This task returns the response according to the command as well as the new server function. Though the client runs the `server` function by sending it a command, the actual work (in this case, execution of the `max` function) occurs in another cord, potentially on another computer in the grid.

This style of programming is quite flexible but can be a bit awkward. However, it would seem possible to implement more natural `send`- and `recv`-style primitives that are mapped down to this idiom by a compiler. Again, the benefit

```
datatype server_result =
     Res of int * (string -> server_result task)

fun startserver m str =
  let fun compute () =
      let val new_m = max(m, size str)
      in Res(new_m, startserver new_m)
      end
  in
      inject compute
  end

  val x = client (start_server 0)

fun client svr0 =
    let val Res(max1, svr1) =
              sync(svr0 "hello")
        val Res(max2, svr2) =
              sync(svr1 "aviator")
    in
        max2
    end
```

**Fig. 3.** Communication Example

of using this style is ease of failure recovery. Since code is restartable after every message, we have the effects of message logging and never need to deal with restarting a piece of code "in the middle" of a protocol.

## 3 Additional Concerns

Apart from the implementation work that needs to be done, some points will require further investigation. Among these are marshaling and garbage collection. First, since tasks may spawn other tasks, there is the possibility that cords will have to carry successor or child cords with them when they are shipped over the network to cycle donor machines. Therefore, we need a convenient way to encode and decode cords—and possibly even modules—rather than whole precompiled programs. It is possible that a new type system might help us to do this by allowing for more control over data layout. In addition, cords will need to contain only *closed* code when they are shipped, so we may need to perform at least some automatic closure conversion.

Second, the programming model we have presented never permanently destroys tasks. A task may fail or be terminated through `forget`, but it can be restarted by anyone who desires its result. This, of course, requires that the task's code be stored somewhere on the network. It would clearly be a problem if more and more tasks were created over time without removing some tasks. Thus it will be necessary to devise some system for garbage-collecting old or unused tasks.

## 4 Conclusion

At a low level, our framework uses basic units of work that may be combined using dependencies. The purpose of these *cords* is to make scheduling and failure recovery practical. To make *programming for* our architecture manageable, we wish to map expressive high-level constructs onto the robust low-level model. We intend to perform this mapping automatically through compilation, so as to allow programming for the grid in a less cumbersome style. Such a system would hide scheduling and failure tolerance concerns from grid application developers.

## References


[BL97] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, 1997.

[CCD+02] Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in ConCert. Technical Report CMU-CS-02-152, Carnegie Mellon University, June 2002. Submitted to GRID 2002.

[Cha02] Bor-Yuh Evan Chang. Iktara in ConCert: Realizing a certified grid computing framework from a programmer's perspective. Technical Report CMU-CS-02-150, Carnegie Mellon University, 2002. Undergraduate honors thesis.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 28(6), pages 237–247, Albuquerque, New Mexico, June 1993.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Joh89] David B. Johnson. Distributed system fault tolerance using message logging and checkpointing. Technical Report COMP TR89-101, Rice University, December 1989.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.