Towards Programming in a Certified Grid Computing Framework

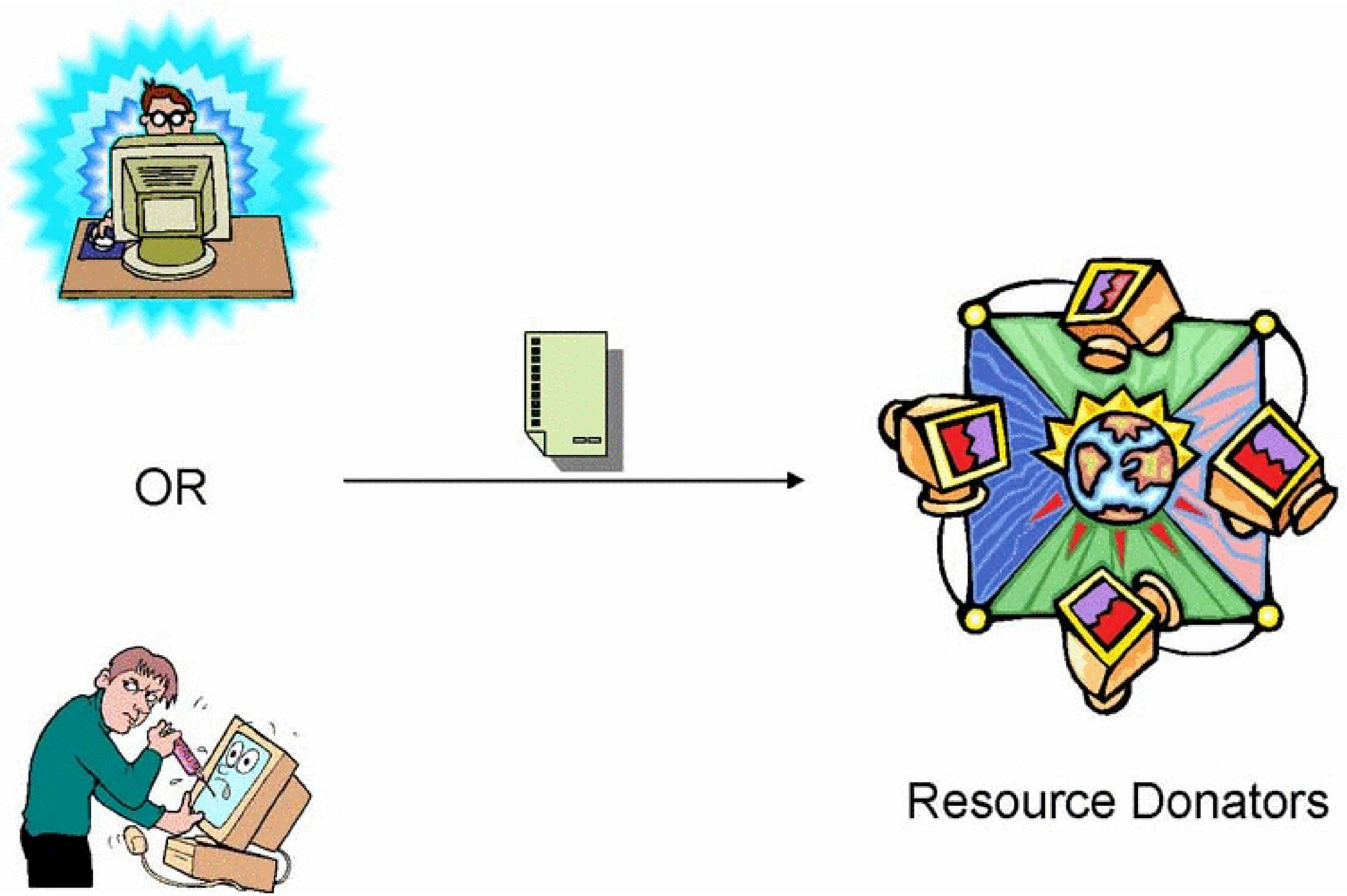
Bor-Yuh Evan Chang

Advisors: Professors Robert Harper and Frank Pfenning

Carnegie Mellon University

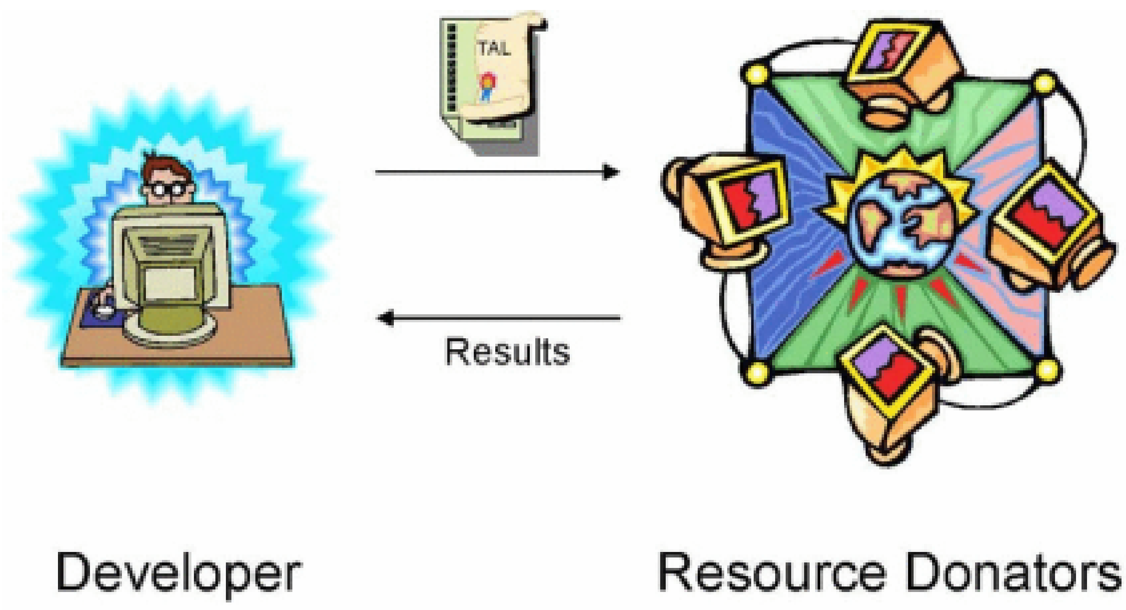
March 20, 2002

ConCert



The ConCert project seeks to develop programming language and type theoretic technology for Grid Computing in a trustless setting.

ConCert



Vision: Distributed-application developer utilization of donated resources is completely transparent to the donator, but the donator is confident the specified safety, security, and privacy policies will not be violated.

ConCert Framework

With Tom Murphy, Margaret DeLap, and Jason Liszka, we seek to develop a real framework to:

- Motivate theoretical work
- Provide a source of technical ideas and problems to solve
- Provide a testbed for implementation

Margaret and Jason: Low-level to discover implementation issues.

- Conductor
- Raytracer

Evan and Tom: High-level to discover programming issues.

- ML Interface, New Programming Language?
- Parallel Theorem Prover

My Contribution

Idea: The process of developing a substantial application using the ConCert framework will help us better understand the requirements on the framework and how to program in such an environment.

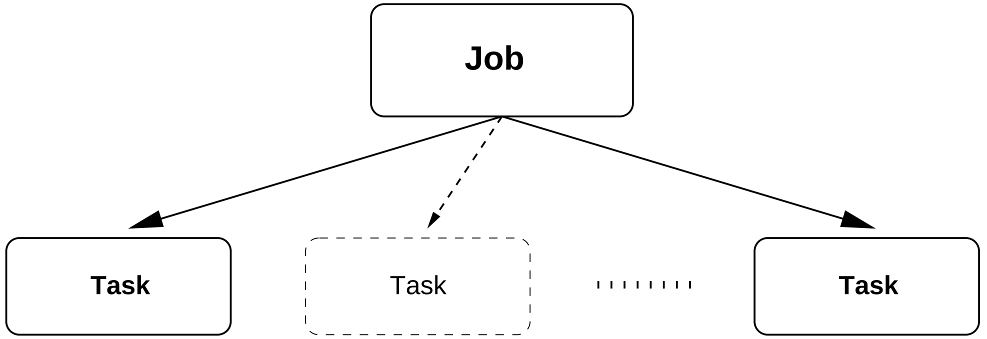
Goals

- drive the framework to a more robust and stable state
- better understand the requirements from a programmer's perspective

Last Semester: Develop a parallel theorem prover for linear logic.

This Semester: Bridge the gap between the implementation in CML and the low-level interface provided by Conductor.

ConCert Programming: Jobs and Tasks



Job: A whole-program that is injected into the network from the command-line.

Task: The unit of computation from the programmer's point of view. Consists of a piece of closed code along with its arguments. The code should restartable.

Injecting a Task into the Network

```
type 'a task

val injectTask : bool -> ('b -> 'a) * 'b -> 'a task
val enableTask : 'a task -> unit
```

- A task can optionally be injected into the network in a suspended state (i.e. *disabled*).
- If disabled, the task will not run until an explicit *enable* instruction is issued.

Retrieving Results

```
val sync : 'a task -> 'a
```

- Returning a result and asking for results from other tasks are the only form of communication between tasks.
- Blocks the calling task until the result can be obtained.
- Let t be the task that we seek the result from. Task t could be in four possible states:
  1. t has already completed execution successfully.
  2. t is currently executing.
  3. t has failed (or appears to have failed).
  4. t is currently disabled.

Results from Multiple Tasks

```
val syncall : 'a task list -> 'a list
val relax   : 'a task list -> 'a * 'a task list
```

- `syncall` blocks until results are obtained from all the given tasks.
- `relax` continues as soon as one result is available.

Example: Merge Sort

```
(* mergesort : int list * int -> int list *)
fun mergesort (nil, _) = nil
  | mergesort ([x], _) = [x]
  | mergesort (l, cutoff) =
    let
      (* partition : int * int list -> int list * int list * int list *)
      ...

      (* merge : int list * int list -> int list *)
      ...

      val len = List.length l
      val (lt,md,rt) = partition (len div 3, l)
    in
      if (len <= cutoff) then
        merge (mergesort (lt,cutoff), merge (mergesort (md,cutoff), mergesort (rt,cutoff)))
      else
        ...
    end
```

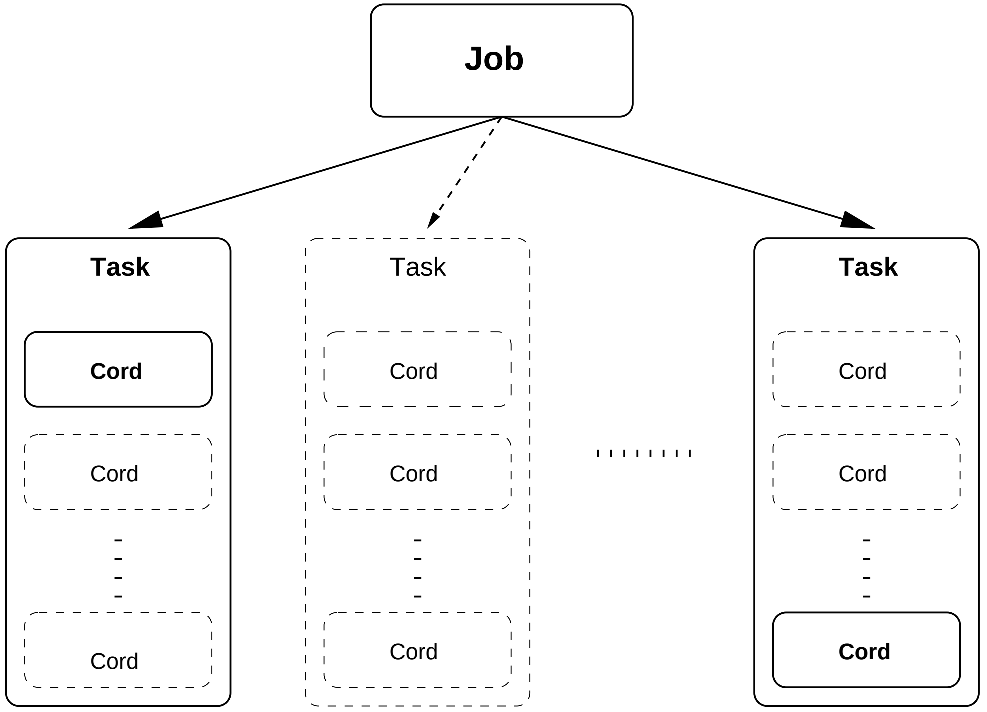
Example: Merge Sort (cont'd)

```
      if (len <= cutoff) then
        merge (mergesort (lt,cutoff), merge (mergesort (md,cutoff), mergesort (rt,cutoff)))
      else
        let
          open CCTasks

          (* Start sorting each partition *)
          val t1 = injectTask true (mergesort, (lt, cutoff))
          val t2 = injectTask true (mergesort, (md, cutoff))
          val t3 = injectTask true (mergesort, (rt, cutoff))

          (* Get the results of the three child tasks.  Start merging
             when receive 2 sorted lists. *)
          val (sort1, sort2) = let
                                 val (a, rest) = relax [ t1, t2, t3 ]
                                 val (b, [last]) = relax rest
                               in
                                 (merge (a,b), sync last)
                               end
        in
          merge (sort1, sort2)
        end
```

Jobs, Tasks, and Cords



Cord: The unit of computation scheduled by the ConCert framework (Conductor).

Invariants

To simplify implementation and allow for failure recovery and program mobility, we impose strong invariants on cords:

1. A cord is deterministic, or any possible result is "as good as" any other.
2. Cords do not communicate except through explicit dependencies.
3. Once its dependencies are filled, a cord is able to run to completion.

Are these invariants really necessary, and what sorts of applications do they preclude?

Next Steps

1. Make the theorem prover run on a simulator of the given interface.
2. Flush out as many issues as possible with regards to compiling the proposed interface.
3. Implement the interface (if possible).
4. Write everything up!