Realizing ConCert

Tom Murphy

Evan Chang

Margaret DeLap

Jason Liszka

February 19, 2002
February 26, 2002

ConCert

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

Our team develops a real framework (ConCert-v1) to:

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

Our Strategy

We use a two-pronged approach to the problem.

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

This Talk

- This Week
  - Cilk-NOW
  - Programmer's Interface
  - Low-level Interface
  - Node Discovery and Work Distribution
- Next Week
  - Leftovers
  - Application Design
    - Raytracer
    - Theorem Prover
  - Demos

Intended Applications

Characteristics of the network:

- Low communication (no shared memory)
- Trustless $\Rightarrow$ High Failure
- Very high parallelism
- Non-homogeneous network
- Potential for mobile code, run-time code generation

Appropriate sorts of applications:

- Prime search (GIMPS), alien search (SETI@home), etc.
- Game-tree search?

Cilk-NOW

Cilk-NOW is an implementation of Cilk version 2 (a parallel C variant) for networks of workstations.

- Distributed scheduler
- Work-stealing
- Failure recovery
- Process mobility
- Functional-like programming style

Some Cilk-2 code

```
thread Fib (cont int k, int n) {
  if (n < 2) {
    send_argument (k, n);
  } else {
    cont int x, y;
    spawn_next Sum(k, ?x, ?y);
    spawn Fib (x, n - 1);
    spawn Fib (y, n - 2);
  }
}

thread Sum (cont int k, int x, int y) {
  send argument(k, x + y);
}
```

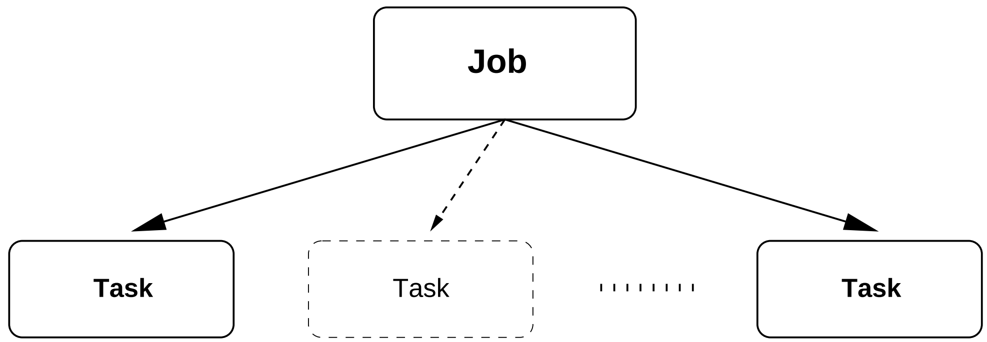
Cilk-NOW cont'd

Cilk's programming model is rudimentary, yet they were able to develop several significant applications.

- Protein folding
- A chess engine
- Fibonacci number calculator

We intend to provide a richer language in a similar execution environment, all in a trustless setting.

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 piece of closed code along with its arguments. The code should restartable.

Injecting a Task into the Network

```
type 'a taskId
exception InvalidTaskId

type ('a, 'b) task = ('b -> 'a) * 'b

val injectTask : bool -> ('a, 'b) task -> 'a taskId
val enableTask : 'a taskId -> 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 recvResult : 'a taskId -> '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.

Events

```
type 'a event

val recvResultEvt : 'a taskId -> 'a event

val sync       : 'a event -> 'a
val choose     : 'a event list -> 'a event
val select     : 'a event list -> 'a
val wrap       : ('a event * ('a -> 'b)) -> 'b event
val guard      : (unit -> 'a event) -> 'a event
val neverEvt   : 'a event
val alwaysEvt  : 'a -> 'a event
```

- Separate asking for the result from the actual operation of synchronizing on the result of some other task (like in CML).
- Non-trivial events can only be introduced by `recvResultEvt`.

Application Optimizations

```
val kill : 'a taskId -> unit
val exit : 'a -> 'b

datatype Status =
  Disabled
| Failed
| Finished
| Running
| Waiting

val status : 'a taskId -> Status
```

- `kill` is simply hint to the scheduler that the task is no longer needed.

Example: Merge Sort

```
(* Point at which we stop parallelizing subproblems *)
val PAR_CUTOFF = 5

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

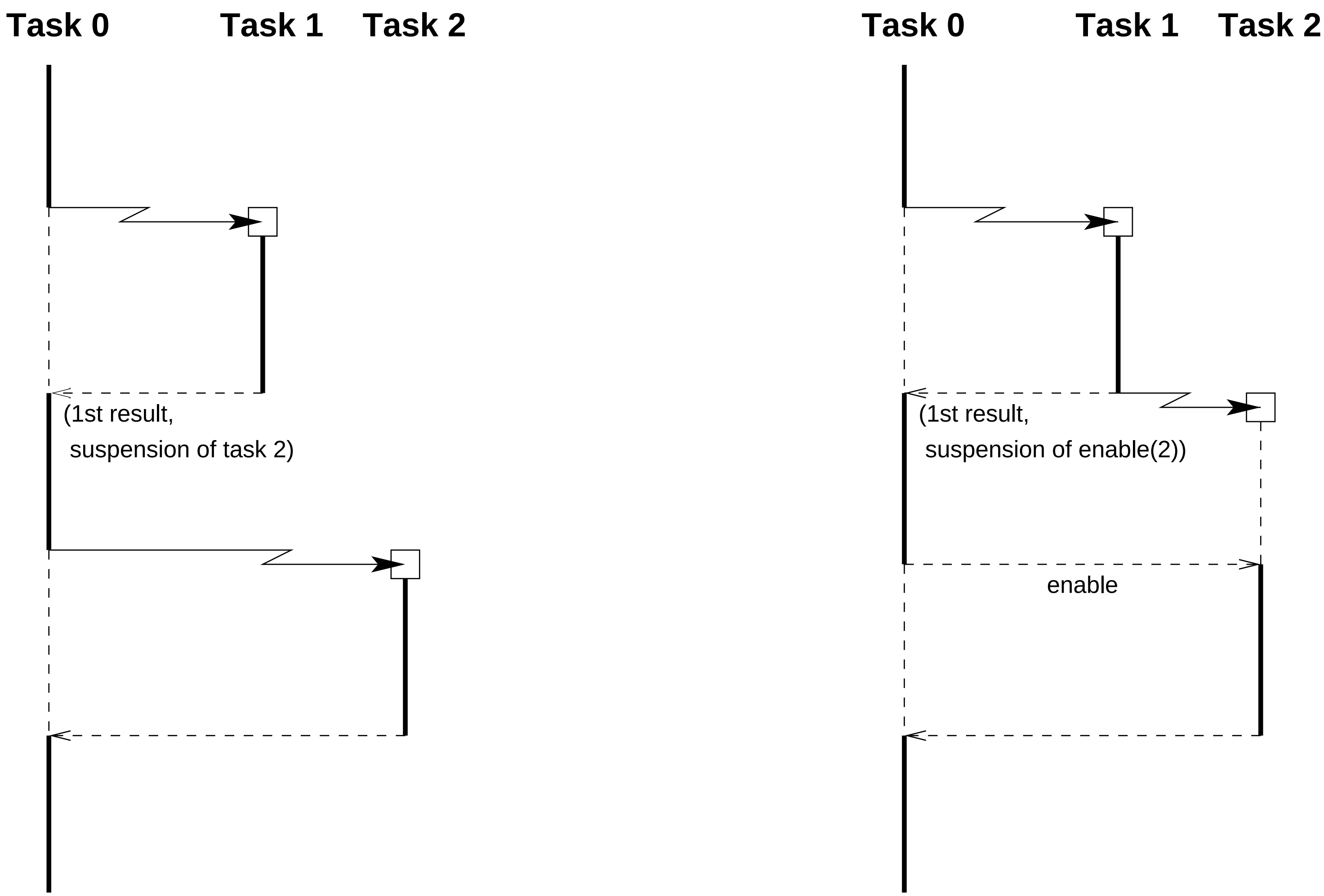
Example: Merge Sort (cont'd)

```
(* 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 *)
      ...

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

      val len = List.length l
      val (lt,rt) = partition (len div 2, l)
    in
      if (len <= cutoff) then
        merge (mergesort' (lt,cutoff), mergesort' (rt,cutoff))
      else
        ...
    end
```

Example: Merge Sort (cont'd)

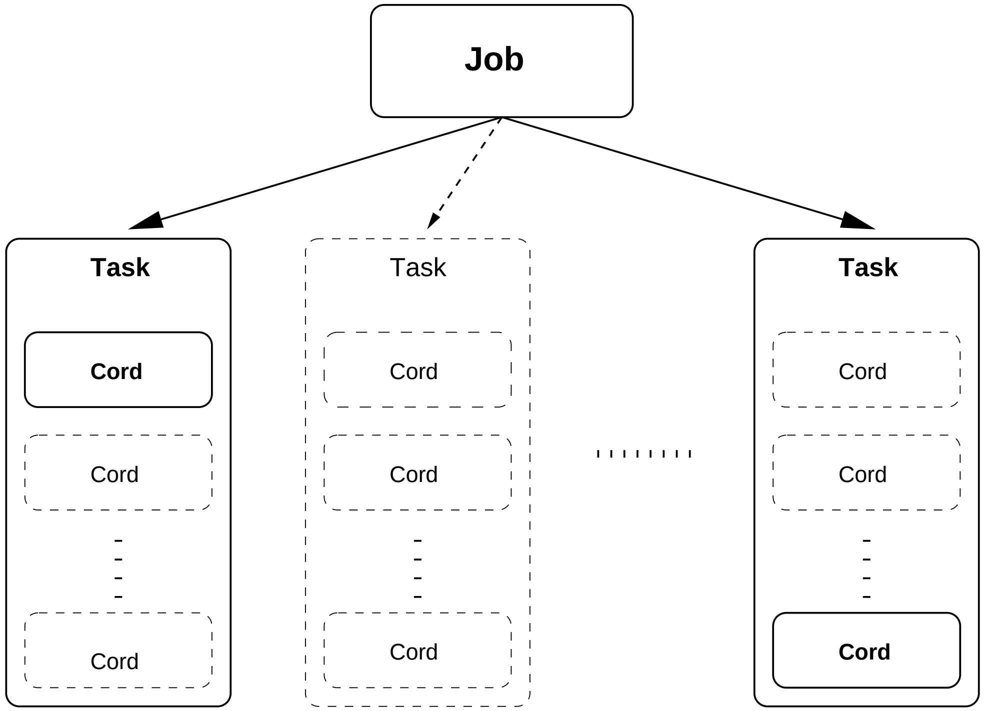
```
in
  if (len <= cutoff) then
    merge (mergesort' (lt,cutoff), mergesort' (rt,cutoff))
  else
    let
      open CCTasks

      (* Start sorting each partition *)
      val tid1 = injectTask true (mergesort', (lt, cutoff))
      val tid2 = injectTask true (mergesort', (rt, cutoff))

      (* Get the results of the two child tasks *)
      val (sortlt, sortrt) = select [
        wrap (recvResultEvt tid1,
                fn sortlt => (sortlt, recvResult tid2)),
        wrap (recvResultEvt tid2,
                fn sortrt => (recvResult tid1, sortrt))
      ]
    in
      merge (sortlt, sortrt)
    end
end
```

Modeling a stream of results



- Necessary for the theorem prover application.

Jobs, Tasks, and Cords



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

Structure of a Cord

A cord consists of:

- Code (we will want some way to cache code across cords)
- An environment
- A set of dependencies on the answers of other cords
- Safety policy, Certificate

Invariants

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

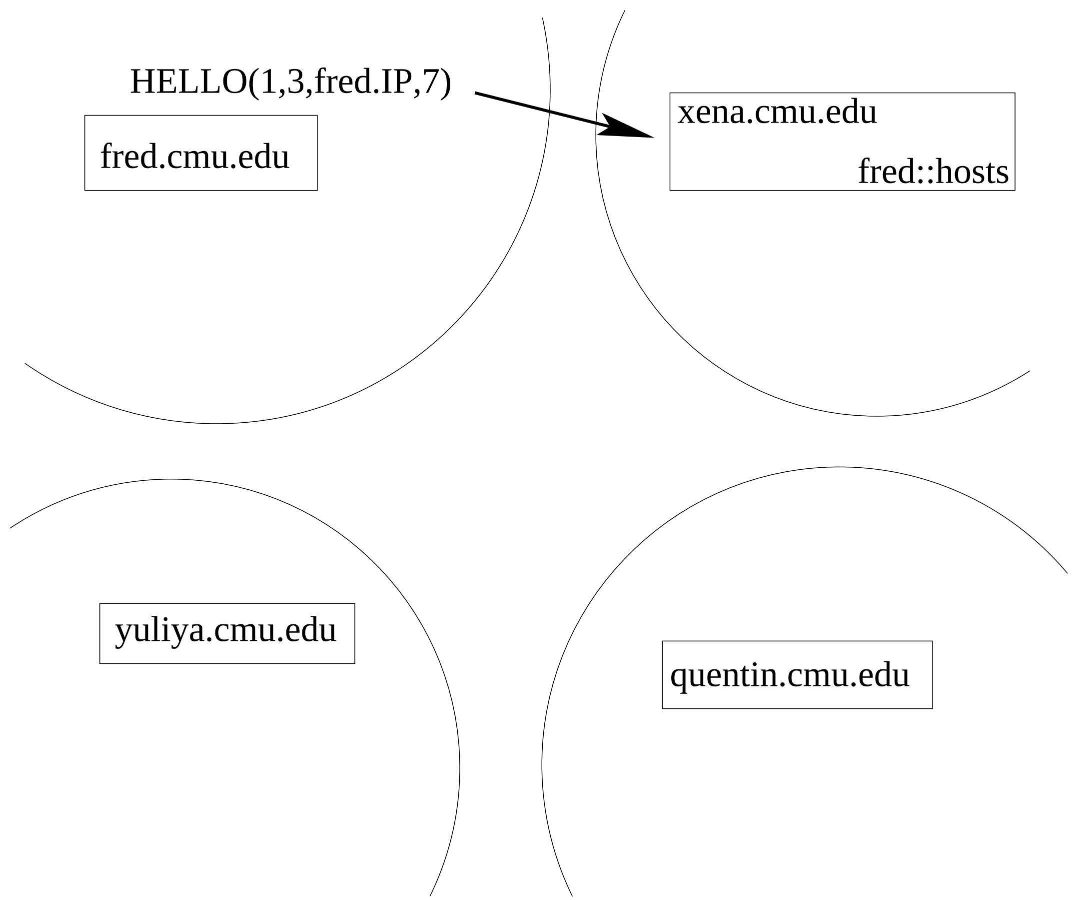
1. A cord is deterministic, or any possible result is “as good as” any other.
2. Cords do not communicate except through explicit dependencies.
3. Once its dependencies are filled, a cord is able to run to completion.

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

Distributing and Scheduling Cords

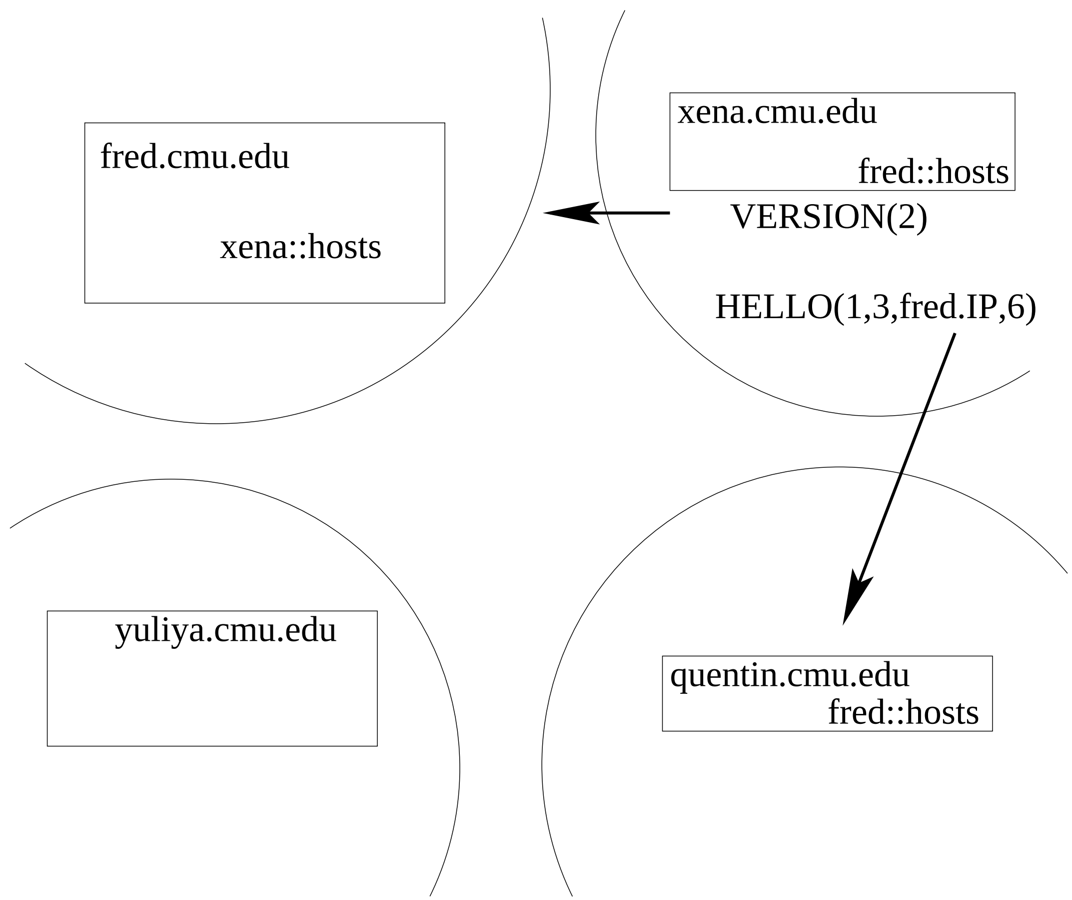
1. Peer-to-peer network (Gnutella)
2. Security Policies
3. Work-stealing
4. Failure tolerance

Finding Other Nodes: Example (1)



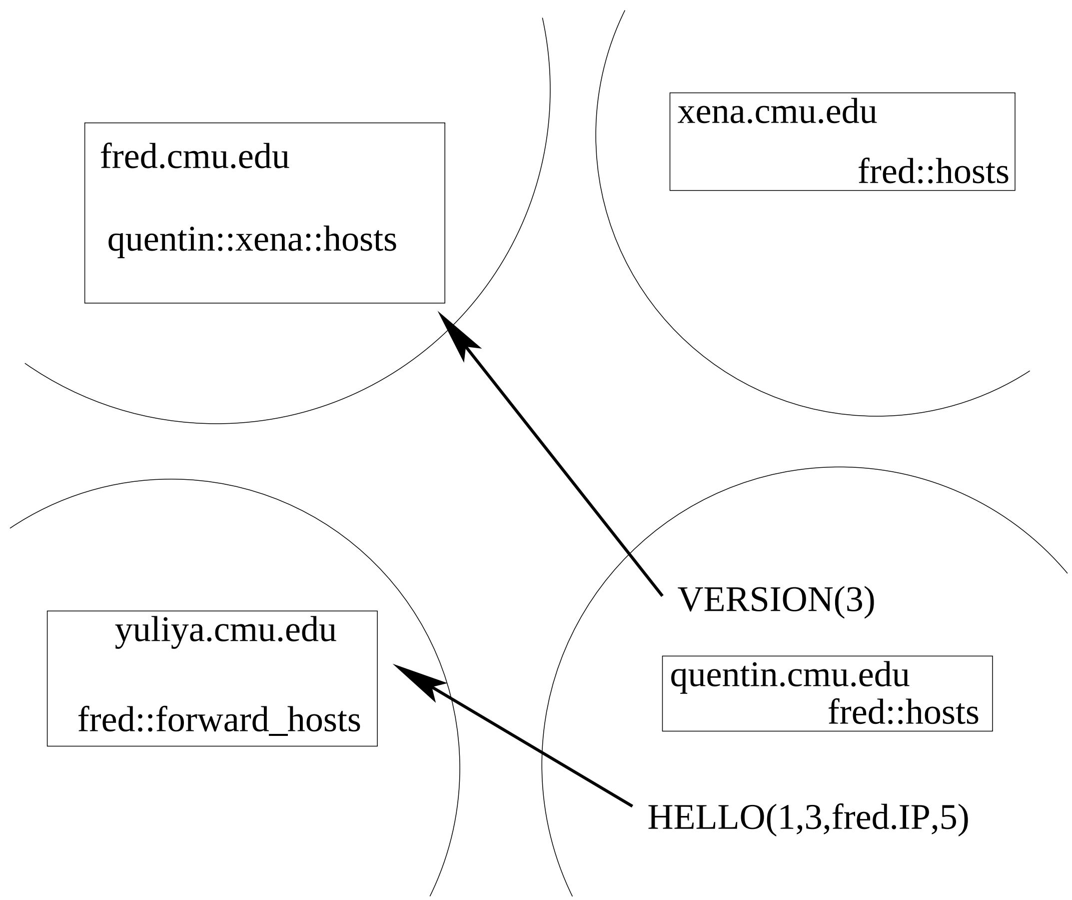
`fred` joins the network.

Finding Other Nodes: Example (2)



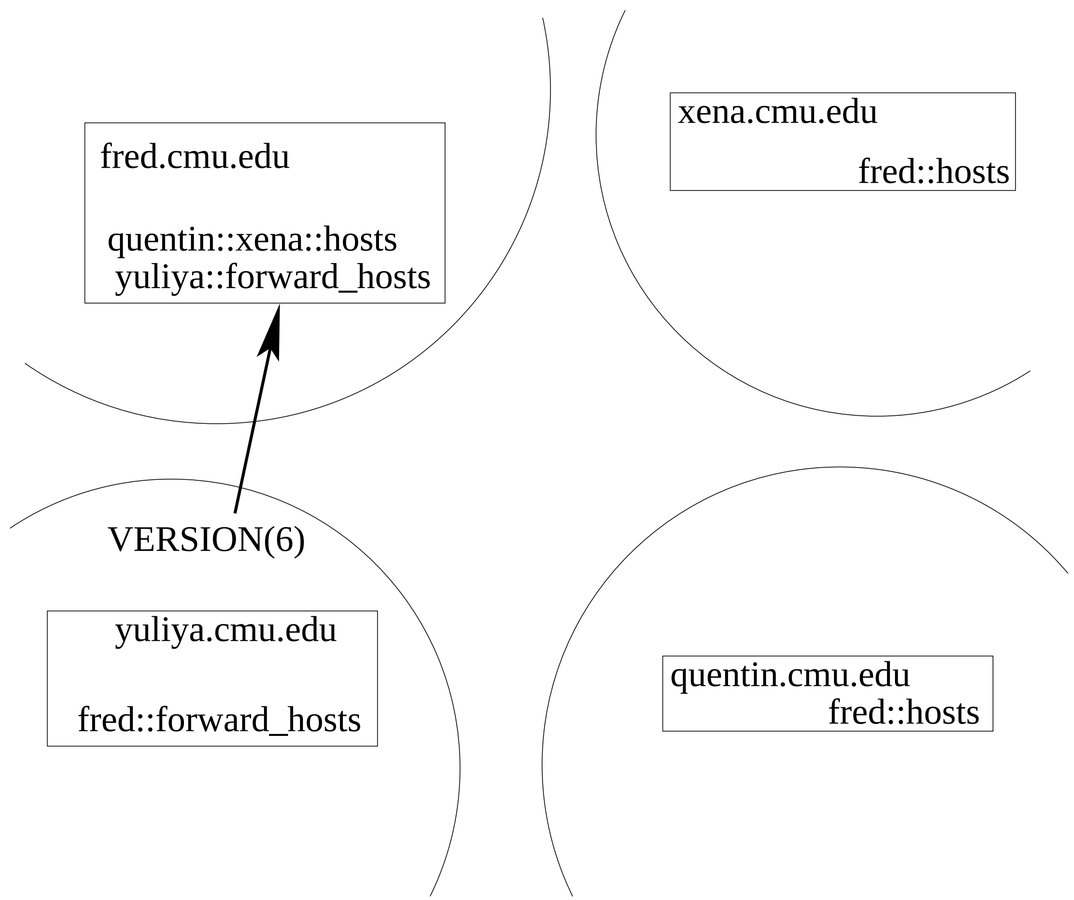
`HELLO` response and forwarding

Finding Other Nodes: Example (3)



Direct `VERSION` response

Finding Other Nodes: Example (4)



Version differences

Applications in Development

1. Distributed Raytracer
2. Parallel Theorem Prover for Linear Logic

Distributed Raytracer

- based on the ICFP'00 raytracer specification:
  `http://www.cs.cornell.edu/icfp/`
- in Popcorn, compiled to TAL
- manual "cordification"
- "one-level-deep" parallelism
- later, recursive raytracing parallelism

Parallel Theorem Prover for Linear Logic

- A subgoal-reduction based parallel theorem prover for intuitionistic linear logic
  - Advantages:
    - *focusing* strategy helps with branching breadth
    - able to check validity of results easily
    - few existing linear logic provers
  - Concerns:
    - how to balance the cost of communication
    - how to limit frivolous parallelism

Parallelism in Theorem Proving

- Independent Subproblems

$$\dfrac{\begin{array}{c}\vdots \\ \Gamma; \Delta \Longrightarrow A\end{array}}{\Gamma; \Delta \Longrightarrow A \oplus B}\oplus\mathrm{R}_1 \qquad \dfrac{\begin{array}{c}\vdots \\ \Gamma; \Delta \Longrightarrow B\end{array}}{\Gamma; \Delta \Longrightarrow A \oplus B}\oplus\mathrm{R}_2$$

- Non-Independent Subproblems

$$\dfrac{\begin{array}{c}\vdots \\ \Gamma; \Delta_1 \Longrightarrow A\end{array} \qquad \begin{array}{c}\vdots \\ \Gamma; \Delta_2 \Longrightarrow B\end{array}}{\Gamma; \Delta_1, \Delta_2 \Longrightarrow A \otimes B}\otimes\mathrm{R}$$

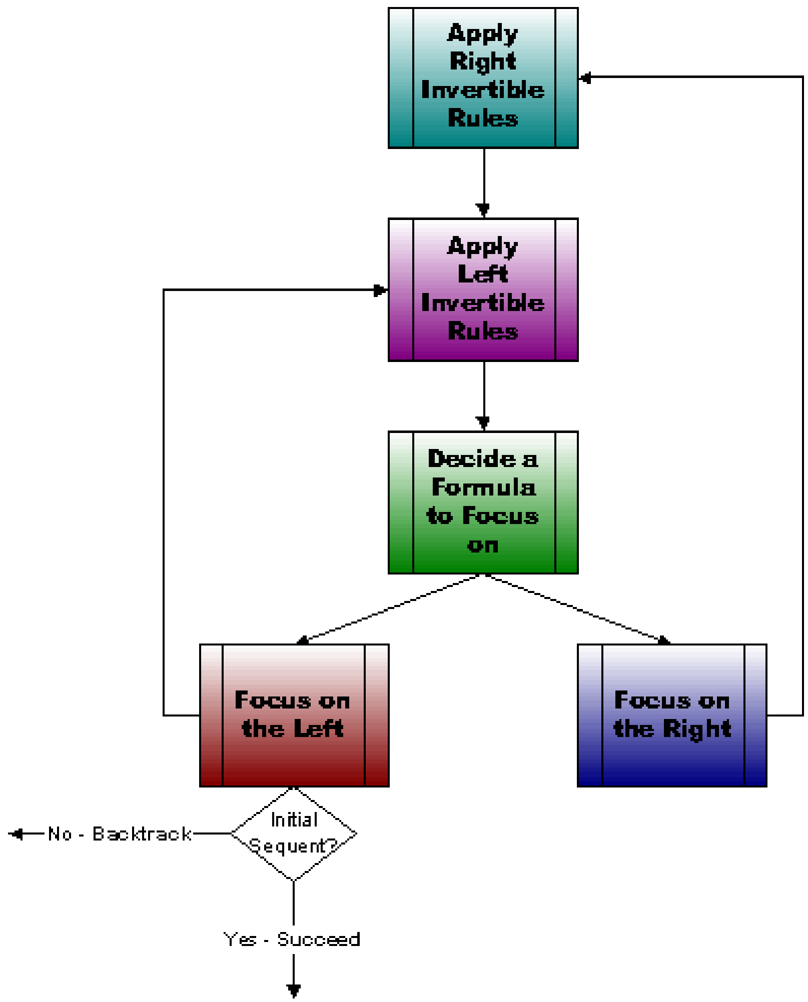
Core Algorithm

- Focusing Strategy [Andreoli '92][Pfenning '01]
  - first apply invertible rules eagerly
  - select a "focus" proposition and apply non-invertible rules until reach invertible connective or atomic formula
- *Resource-distribution via Boolean constraints* [Harland and Pym '01]

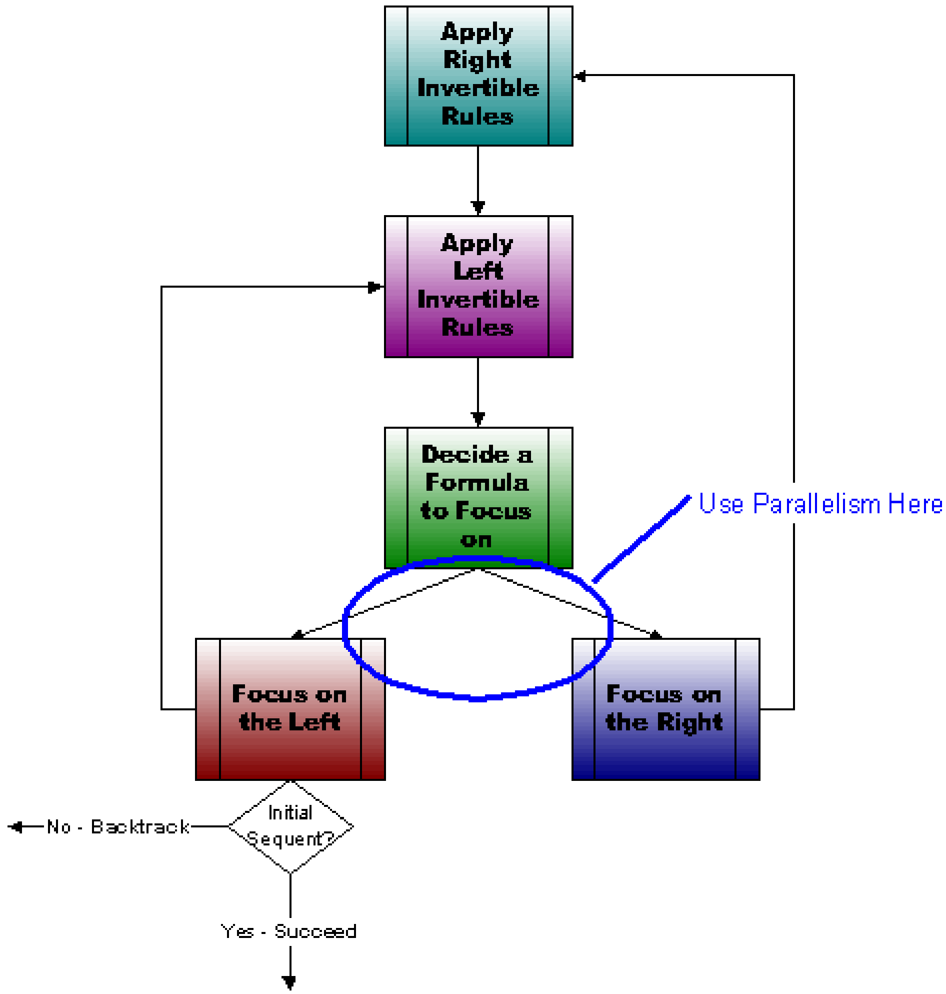
$$\dfrac{\begin{array}{c}\vdots \\ \Gamma; \Delta_1 \Longrightarrow A\end{array} \qquad \begin{array}{c}\vdots \\ \Gamma; \Delta_2 \Longrightarrow B\end{array}}{\Gamma; (\Delta_1, \Delta_2) \Longrightarrow A \otimes B} \otimes\mathrm{R}$$

  - represent constraints using OBDDs

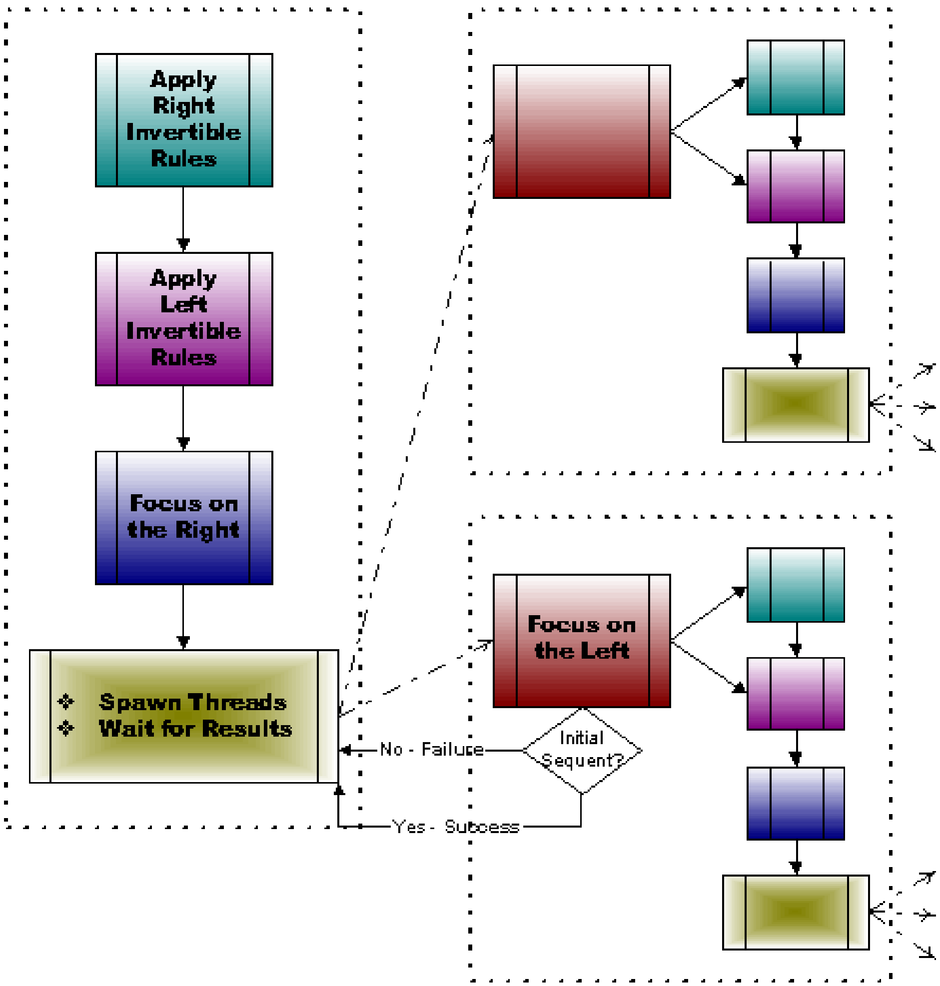
Focusing (Sequential)



Focusing (Sequential)



Focusing (Parallel)



Current Issues

1. Is it important for a grid computing language to support the automatic marshaling of data, or is this a task that can only be handled correctly by the application programmer?
2. What is the fastest path to a working implementation of our proposed language extensions?
3. Which of our invariants are actually necessary, or what others do we need? What classes of programs do we preclude with our invariants?
4. How scalable is our peer-to-peer network?
5. In a trustless network where anyone can spawn jobs, how do we prevent a naive programmer from making very inefficient use of everyone's resources, or a malicious user from swamping the network with worthless jobs?
6. *What other applications?*
7. How do we deal with incorrect or forged results?