Programming in JoCaml (Tool Demonstration)

Louis Mandel[1] and Luc Maranget[2]

[1] Lri, Univ Paris-Sud 11, cnrs, Orsay F-91405
Inria Futurs, Orsay F-91893
[2] Inria Paris - Rocquencourt, Le Chesnay F-78153
{Louis.Mandel,Luc.Maranget}@inria.fr

Abstract. JoCaml is a language for concurrent and distributed programming. The language is an extension of Objective Caml with concurrent features inspired by the join-calculus.
We here present the recent release of JoCaml, motivate our fundamental design choices, compare the new release with previous ones, and give a taste of JoCaml by means of a few examples.

1 Introduction

JoCaml is a language for programming concurrent and distributed systems. It is based on ML for the computational part, and on the join-calculus for the concurrent part.

The join-calculus is a name passing calculus. The purpose of such calculi is to describe concurrent and distributed systems. Programming such systems is a different, although related, issue, since a good model offers suitable abstractions that help programmers.

Our language, JoCaml, is an extension of Objective Caml (OCaml), a popular dialect of ML. By choosing to extend an existing language, and not to design one of our own, we first intend to minimize our work. We also intend to benefit from functional programming, from pre-existing code base, and from a population of programmers open to innovation.

Up to three new keywords, JoCaml is a conservative extension of OCaml: OCaml programs retain their type and behavior. But we understand compatibility in a stronger sense: JoCaml provides a concurrent extension of ML that strictly adheres to the spirit of functional programming. Channel definitions and synchronization behaviors are programmed concisely, by the high-level join-definition concept, and declaratively, by the introduction of ML pattern matching of messages in channel definitions. Moreover, channels are typed polymorphically, as functions are in ML, types being inferred. Channels are first class-values that, amongst other things, can be passed as arguments to functions, sent as messages on channels, and occur as members of modules. This, with the polymorphic typing of channels, is our way to code re-use for concurrent components.

JoCaml web site is `http://jocaml.inria.fr/`. The site offers a source release (dating June 2007), links to articles, and a 70 pages tutorial and reference

expression	::=	*ocaml-expression*	
	|	def $x_1(p_1)$ & ... & $x_n(p_n)$ = *process*	
		...	join-definition
		or $x_k(p'_k)$ & ... & $x_m(p'_m)$ = *process*	
		in *expression*	
	|	spawn *process*	process execution
process	::=	$x(expression)$	message sending
	|	reply *expression* to x	reply to synchronous channel
	|	*process* & *process*	parallel composition
	|	*expression* ; *process*	sequential composition
	|	let ... in *process*	local value definition
	|	def ... in *process*	local channel definition

Fig. 1. JoCaml syntax

manual. We have programmed a few applications in the language ourselves. Amongst those, a distributed ray tracer is the most mature. The ray tracer is available on the web site and its source code amounts to about 7000 lines.

2 The new JoCaml

The new JoCaml system is a re-implementation from scratch of the previous prototype. It focuses on compatibility with OCaml. Any OCaml source code is a valid JoCaml source code and JoCaml can also call external OCaml libraries that do not need to be re-compiled.

Briefly, we proceed by altering the OCaml compiler from parsing phase to first intermediate code generation, and by enriching the thread library of OCaml with specific support. Compiler alteration is justified by specific typing and pattern matching compilation, which both need to be perform inside the compiler. Compiler alteration is limited in the sense that we change or add a few thousand lines in the compiler original source files, add a few source files, and retain the OCaml formats for binary files.

Our focus over compatibility and limited alteration of OCaml, made us abandon the mobility features of the join-calculus. Nevertheless, there are useful distributed programs that can be written without code mobility.

Moreover, the new JoCaml extends the synchronization mechanism of the join-calculus with pattern matching. It allows to define synchronization not only on the presence of a message on a channel, but also on the value of the message.

3 A join-definition

JoCaml adds the new syntactical category of *processes* to OCaml syntax (Fig. 1). In contrast to expressions processes yield no result and execute asynchronously. Additionally, JoCaml slightly extends OCaml expressions. The `spawn` *proc* construct introduces processes in expressions: *proc* is executed asynchronously and spawn returns immediately.

The join-definition is the distinctive feature of the join-calculus: it defines several channels and their reception behavior at the same time. In JoCaml, join-definitions are introduced by def and can occur both in processes and expressions. We illustrate join-definitions by the example of a concurrent buffer based on the two-lists implementation of functional FIFO queues.

```
type 'a buffer = { put: 'a -> unit; get: unit -> 'a }

let create_buffer () =
  def state(xs,ys) & put(x) = state(x::xs,ys) & reply () to put
   or state(xs,y::ys) & get() = state(xs,ys) & reply y to get
   or state(_::_ as xs,[]) & get() =
     state([], List.rev xs) & reply get() to get
  in
  spawn state([],[]) ;
  {put=put; get=get;}
```

Our buffers are records, a pure OCaml concept, the novelty resides in the join-definition (def... in above). Three channels are defined: state, put and get. Channel state is *asynchronous*. Message sending on an asynchronous channel is an elementary process, as illustrated by spawn state([],[]) above, for instance. By contrast, put and get are synchronous channels. Message sending on a synchronous channel yields a result, and thus is an expression. In fact, to the sender, synchronous channels behave as functions and have functional types.

The behavior of the buffer is expressed by three *reaction rules* that compete (or) for consuming messages. A reaction rule consists in a *join-pattern* and in a *guarded process* (separated by =). The semantics is as follows: when there are messages pending on all the channels in the join-pattern and they match the patterns present as formal arguments, then the guarded process may be fired. The guarded process is executed asynchronously, but may transmit return values to the callers of synchronous channels (reply/to).

The idea of the buffer is to store the FIFO queue (implemented by a pair of lists) as a message on the channel state. By the organization of join-patterns, which all include state, and the fact that there is at most one message on this channel, exclusive access to the internal state of the buffer is granted to the callers of synchronous put and get.

The first join-pattern state(xs,ys) & put(x) is satisfied whenever there are messages on both state and put. The behavior of the guarded process is to perform two actions in parallel (& in processes): (1) send a new message on state where the value x is added to the list xs and (2) return the value () to the caller of put.

The second join-pattern state(xs,y::ys) & get() is satisfied when there are messages on both state and put *and* that the message on state matches the pattern (xs,y::ys). That is, the message is a pair whose second component is a non-empty list. The process guarded by this join-pattern removes one value from the buffer and returns it to the caller of get. The last join-pattern state(_::_ as xs,[]) & get() is satisfied when there is a message on get and a message on `state` that matches a pair whose first component is a non-empty list and second component is an empty list. The corresponding guarded process transfers elements from one end of the queue to the other and performs `get` again. Notice that there is no join-pattern that satisfies `state([],[]) & get()`. As a consequence, a call to `get` is blocked when the buffer is empty.

To initialize the buffer, a message `([],[])` is sent on `state`. The spawn construct is here necessary, since the message sending appears in expression context (the body of the function `create_buffer`).

4 Distributed computation

The join-calculus provides a transparent model for distributed computation. Guarded processes always execute on the site where they are defined but can be fired from any site. More precisely given a channel c, the sending of a message on c can be performed on any site (provided c is known), while the reception on c can occur only on the site where c is defined. This is by design, and comes in sharp contrast to the model of the π-calculus, where it is sufficient to know c to perform emission and reception on c.

Obviously, the join semantics is much easier to implement than the π semantics in a distributed setting. Basically, message sending to a remote site decomposes into a transport phase and a synchronization phase (join-pattern matching), the latter being performed locally on the receiving site.

However, performing the transport phase (and the related global naming of sites and channels) does not upgrade concurrent JoCaml into distributed JoCaml as if by magic. Two important issues arise that are not really expressed in the join model: channel publication and failures. We addressed those pragmatically, so as not to delay the release of the new JoCaml.

When they start, sites (JoCaml programs) have nothing in common. But, so as to initiate communication, sites need to share at least a few channel names. To that aim, JoCaml provides a *name service* that basically is a repository of channel names, indexed by plain strings. In contrast to the JoCaml language, there is no type safety at all. As to failures, our treatment is rather unsophisticated as we rely exclusively over direct routing: communicating sites are connected by a bi-directional link (a TCP socket). Then, the failure of the link, is interpreted by one partner as the failure of the other partner. We plan to improve these two points in future releases.

5 Conclusion

JoCaml is one amongst many recent language that offer serious support for concurrency and distribution (Erlang, Cω, Alice, Scala to cite a few). In our view, JoCaml main contribution resides in the programming style it favors: a smooth integration of functional programming for concurrent and distributed applications. Our tool demonstration will focus on this point.

Tool demonstration

Introduction

Our presentation consists in running a non-trivial distributed application written in JoCaml: a ray tracer. We shall describe the core component of the application, a task pool structure. The task pool is concise, in the spirit of functional programming, and offers a functional style interface (a "fold" function). We address the issue of failures.

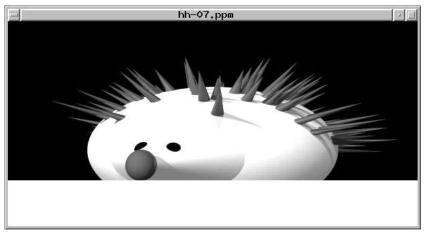
Ideally, if an Internet access is available, we can propose a demonstration with about 40 machines taking part to the computation. Otherwise, a less ambitious presentation on an ad-hoc network of two laptops is possible.

1 Ray Tracing

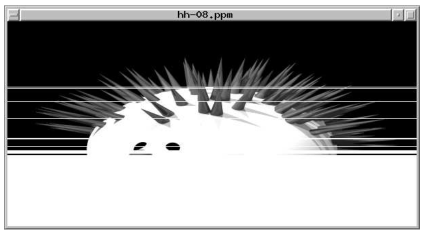
Ray tracing is a well known technique for producing (bitmap) images from 3D scenes. The general principle of ray tracing is to cast light rays from the viewpoint of an observer into the scene. To render a scene as a $w \times h$ bitmap image, one casts the $w \times h$ rays defined by the observer viewpoint and the position of the $w \times h$ picture elements (*pixels*) of the *image plane*, which stands in front of the observer. One then computes the intersection of this *primary* ray with the scene. Reflections on scene objects and the computation of illumination imply casting more rays. The whole process involves much computation.

Our demonstration will start by first running the sequential ray tracer, and then the distributed ray tracer (with about 40 participating computers, which we call *slaves*). Our programs offer a demo mode that shows the images being built (cf. Figure 2). The runs will immediately demonstrate that the distributed

Fig. 2. First runs



Sequential ray tracing



Distributed ray tracing

ray tracer is much faster than the sequential one (about 20 sec. against several minutes).

By considering the distributed ray tracer, it clearly appears that images are computed on a line per line basis. In the sequential case (left image), lines are computed sequentially. As a result, the image grows regularly. The distributed ray tracer offers a more chaotic view of image construction. Namely, slaves perform their work concurrently, and they do not compute every line at the same pace, due to varying complexity in images and slave heterogeneity. As a result, the lines are not completed in enumeration order, as clearly shown by the right image above. Obviously, such a change of computation order does not matter, as long as image completion can be detected.

After the demonstration we shall briefly present the code of the sequential ray tracer that uses a higher order fold-like function. We then present our objective of organizing slaves in a similar programming style.

2 Functional programming style

In a sequential implementations of ray tracing, pixel colors are computed one after the other, by means of, for instance, two nested for-loops. However, we aim at abstraction and generality. Thus, we divide an image as several *sub-images*. We assume a function `add_subimage` of type `subimage -> image -> image` that given a sub-image and a partially constructed image, extends the image with the sub-image. Typically, `image` above is a matrix of pixel colors; `subimage` is a smaller such matrix and a position; and `add_subimage` copies sub-image colors to their final place.

Scenes are divided as images are. A sub-scene simply is a scene and a sub-image description (for instance a line number). We assume a function `subrender` of type `subscene -> subimage` that performs the rendering of a sub-scene by casting rays through all the pixels of the corresponding portion of the image plane.

The sub-scenes in a scene can be enumerated by means of the following functional enumerator.

```
type enum (* left abstract *)
val start : scene -> enum
val step : enum -> (subscene * enum) option
```

Function `start` takes a scene as argument and returns an enumerator. Function `step` applies to such an enumerator and returns either the next sub-scene and a new enumerator, or `None` when enumeration is over.

In any functional language, it is routine to write a general purpose "fold" function on scenes. Here is such a function in OCaml.

```
let fold work add y0 sc =
  let rec fold_rec y enum = match step enum with
  | Some (x,enum) -> fold_rec (add (work x) y) enum
  | None -> y in
  fold_rec y0 (start sc)
val fold : (subscene -> 'a) -> ('a -> 'b -> 'b) -> 'b -> scene -> 'b
```

As shown by its polymorphic type, `fold` is quite generic. Given a scene s whose sub-scenes are x_0, x_1, $\ldots x_n$ (following enumeration order), "`fold` w add y_0 s" computes $add(w(x_{n-1}), add(w(x_{n-2}), \ldots add(w(x_0), y_0)))))$. By supplying the right arguments (`add_subimage` is add etc.), the rendering of a complete scene can be made by calling the generic fold.

In the distributed implementation, the execution of the "worker" function w is performed by a variety of *slave* computers that act under the control of a distinguished *master* computer. The master is in charge of controlling slaves, of performing *add* operations, of enumerating sub-scenes etc. Furthermore, we wish to retain functional style, by having the master to call a fold-like function. To that aim, we introduce a *pool* structure, whose type follows.

```
type ('a,'b) pool =
  { register: (subscene -> 'a) Join.chan;
    fold: ('a -> 'b -> 'b) -> 'b -> scene -> 'b; }
```

The master creates the pool and then calls its `fold` component as follows.

```
let render sc =
  pool.fold
    add_subimage (make_white_image sc.width sc.height) sc
```

In contrast to the sequential `fold`, the master does not supply a worker function. Instead, those are supplied by the slaves, which send a synchronous channel name on the `register` component of the master's pool. Such registered channels act as a proxies for the slaves resident worker functions.

It is perhaps to be noticed that master and slaves are different programs running on different computers. Slaves have access to the master's pool by means of JoCaml *name-service*, which we shall not describe. Instead, we shall focus on the implementation of the pool.

3 Concurrent fold implementation

One key issue in the implementation of the concurrent fold is detecting that the image is completed. In the following, a *sub-task* is the computation of a sub-image from a sub-scene. It should be observed that the functional enumerator does not allow us to know in advance how many sub-tasks will occur. The *monitor* collects the results of sub-tasks and returns the image when all sub-tasks are completed.

```
type ('a,'b) monitor =
  { enter: unit -> unit; leave: 'a Join.chan;
    wait: unit -> 'b; finished: unit Join.chan; }

let create_monitor add y0 =
  def state(n,y) & enter() = state(n+1,y) & reply () to enter
  or state(n,y) & leave(v) = state(n-1,add v y)
  or state(0,y) & finished() & wait() = reply y to wait
  in spawn state(0,y0) ;
  { enter=enter; leave=leave; wait=wait; finished=finished; }
```

During the presentation, we shall show the code above. The monitor is a typical join-definition, with a hidden state expressed as a message `(n,y)` on the internal channel `state`. One easily sees that channel `state` can hold at most one message, which protects the monitor against concurrent access. For instance, a message on `enter` is always consumed jointly with the message on `state`. Then, the internal state is changed and re-emitted on `state`. As a result, the internal state remains consistent.

Component `y` of the internal state is the image being built, it is managed as in the sequential case : sub-images are added to it (`add v y` above), until the image is completed and returned (`reply y to wait` above). Component `n` is new, `n` is an integer that counts the number of active sub-tasks. More precisely, `n` is the total number of messages received on `enter` minus the number of messages received on `leave` (cf. `n+1` and `n-1` above).

The monitor exports all its other channels as fields of the `monitor` record. The distributed fold will obey the following discipline: (1) send a message on `enter` when starting a new sub-task; (2) send the result of any sub-task as a message on `leave`; (3) send a message on `finished` when no more sub-tasks are available. Then, the call on `wait` will be answered when no more sub-tasks can be started and all active sub-tasks are completed, as neatly expressed by "`state(0,y) & finished() & wait() = reply y to wait`" above.

We shall also show the rest of the JoCaml source code for the pool:

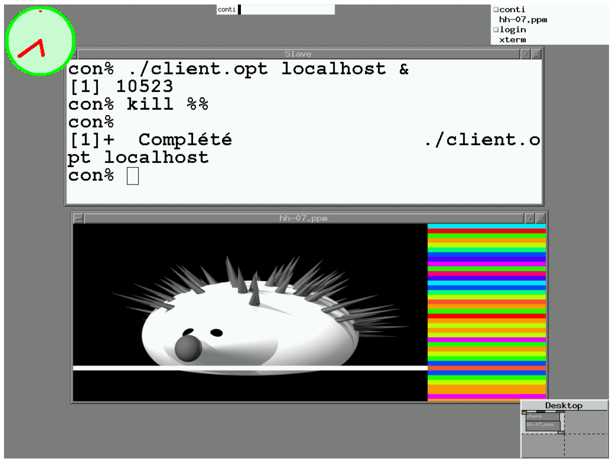
```
let create_pool () =
  def loop(monitor,enum) & agent(worker) = match step enum with
    | Some(x, next) ->
        monitor.enter() ;
        loop(monitor,next) & call_worker(monitor, x, worker)
    | None -> monitor.finished() & agent(worker)
  and call_worker(monitor,x,worker) =
    let v = worker(x) in monitor.leave(v) & agent(worker) in

  let fold add y0 sc =
    let monitor = create_monitor add y0 in
    spawn loop(monitor, start sc) ;
    monitor.wait () in
  { fold=fold ; register=agent ; }
```

The definition of `fold` first creates a monitor, then starts the iteration, and finally calls the `wait` function of the monitor. The `loop/agent` definition essentially performs an iteration on the enumerator `enum` and organizes meetings between available slaves (whose `worker` functions are messages on `agent`) and sub-scenes (`x` above), as expressed by the join-pattern `loop(...) & agent(worker)`. It should be noticed that several images can be computed concurrently: it suffices to call `pool.fold` several times concurrently.

4 Failures

To illustrate the impact of failures, we shall then start another run of the distributed ray-tracer. Figure 3 shows a screen capture of the final state of this run. Two windows are of interest. In the shell window, we have launched a slave program (`client.opt`) and then have killed it. The image window is under control

Fig. 3. Slave program killed, image delayed for ever



```
con% ./client.opt localhost &
[1] 10523
con% kill %%
con%
[1]+  Complété                   ./client.o
pt localhost
con%
```

of the master program, it shows the allocation of sub-tasks with a color code (one color per registered slave). The display demonstrates that image computation does is not completed when some of the slaves fail.
Then, we show again the code that performs the remote call of a `worker` function.

```
and call_worker(monitor,x,worker) =
  let v = worker(x) in monitor.leave(v) & agent(worker)
```

If a slave program is killed, as we did in Figure 3, the call to `worker` does not return a sub-image `v`. As a consequence, no message on `leave` is sent and image completion is delayed for ever.

However, the killing of a local process, certainly can be detected by the underlying JoCaml runtime system. Such a slave failure is transmitted to user programs by raising an exception. The code above simply ignores the exception, and apart from a message on the console, nothing happens.

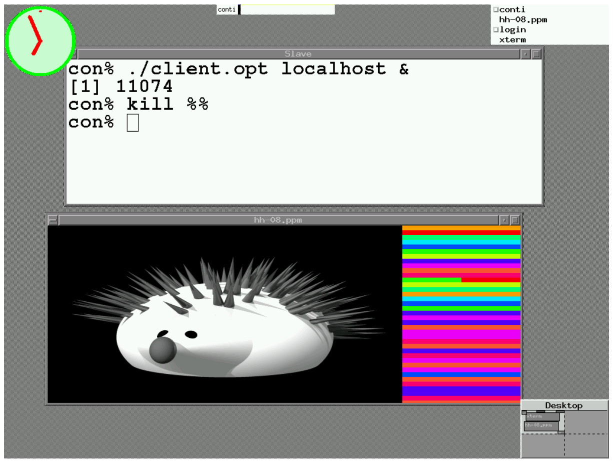
The following, corrected, code catches the exception, so as to take appropriate measures.

```
def agent(worker) & loop(...) = ...
or agent(worker) & compute(monitor,x) =
    call_worker(monitor,x,worker)

and call_worker(monitor,x,worker) =
  let v = try Some (worker(x)) with _ -> None in
  match v with
  | None -> compute(monitor,x)
  | Some v -> monitor.leave(v) & agent(worker)
```

In case an exception is raised, the uncompleted sub-task `x` is simply put again into the pool, as a message on the new channel `compute`. Such re-issued sub-tasks are allocated to available slaves by means of a new join-pattern `agent(...) & compute(...)`, which competes (`or`) with the previous join-pattern `agent(...) & loop(...)` for available slaves.

We then run the corrected program in the same conditions as the failure-unaware program (Figure 4). The color code shows that the previously pending

Fig. 4. Slave program killed, image completed



```
con% ./client.opt localhost &
[1] 11074
con% kill %%
con% 
```

sub-task is now completed by another slave.

5 Undetected failures

We shall complete the presentation by considering undetected failure.

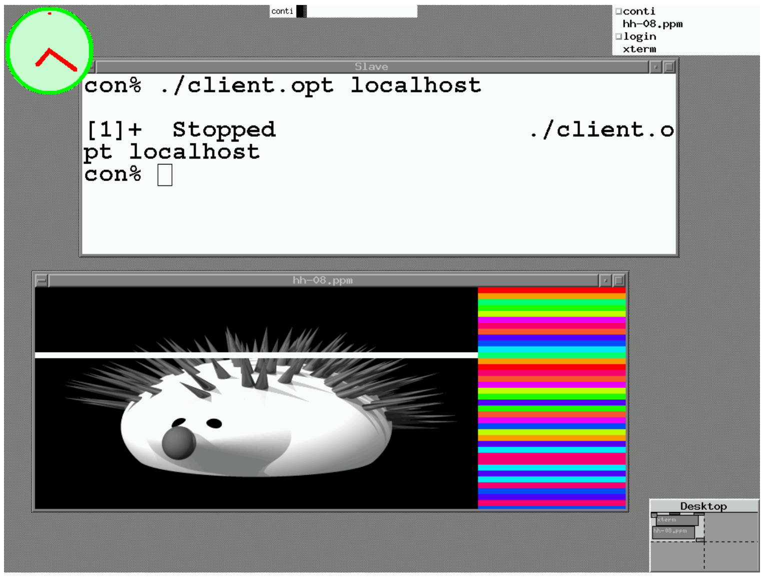
We shall proceed to a second run of the failure aware program, but we shall stop the local slave in place of killing it. Doing so, we produce an undetected failure. This second run will show that an undetected failure delays image completion for ever (Figure 5).

We shall only give the principle behind our technique for tackling the issue, which resides in the following master's quote:

> *I'd rather have n slaves working on the same task, than one slave working and $n-1$ idle slaves looking at him.*

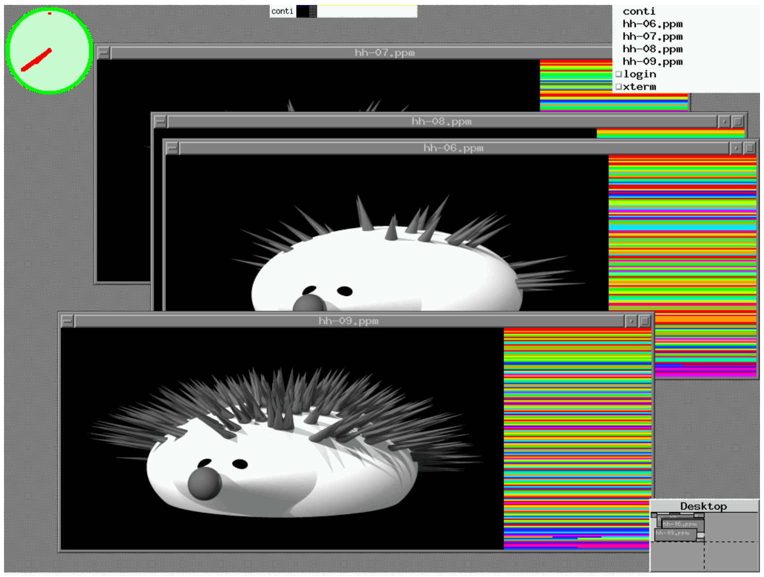
More precisely, we shall sketch a refined monitor that records the list of active sub-tasks, while the previous monitor recorded the count of active sub-tasks. The new pool extends the previous one by querying the monitor for active sub-tasks and allocating those to slaves that otherwise would be idle. We shall illustrate the principle by a final run, first with sub-task granularity as before, and then with sub-task granularity reduced, so as to minimize duplicated work. Figure 6 shows that the bottom lines of the image completed last have been processed by up to three slaves. Performance figures show that this is a small price to pay for guaranteed image completion.

Fig. 5. Slave program stopped, image delayed



```
con% ./client.opt localhost

[1]+  Stopped                      ./client.o
pt localhost
con% 
```

Fig. 6. A serious run, four images at a time


conti
conti
hh-06.ppm
hh-07.ppm
hh-08.ppm
hh-09.ppm
login
xterm
hh-07.ppm
hh-08.ppm
hh-06.ppm
hh-09.ppm
Desktop