ACT-Concurrently Concurrency Work-Around for ACT-R

Frank Tamborello
National Research Council
Postdoctoral Research Associate
U. S. Naval Research Laboratory

"ACT-R models are often computationally expensive. For example, a model I was recently working on takes ≈15 s per run on a 2.6 GHz i7. 'Scaling up' the field is likely to involve even more expensive models. However, computers aren't getting faster so much as they're becoming more parallel, so we could make life a little easier on ourselves by parallelizing. Furthermore, I something that would take minimal setup and load with ACT-R."

Overview

Manager

- List of workers
- Model

- Sends model & run command to workers
- Receives data

Workers

- Wait for Manager
- Run the model
- Send data back to manager

“ACT-Concurrently is meant to be configured with one node operating as a ‘manager’ and all others as ‘workers.’ Each node runs in its own ACT-R instance. The manager coordinates the workers; sending the model and the run command to all workers and then receiving their return values, which could for example be the model data. So the system sort of prefers a functional style of computation for running the model, but that’s not necessary. So for example if your model writes data to a file or prints to the top level, you should still get that, but it’ll be on each worker separately.”

"I'll demonstrate two workers and one manager, all on this local one local 4-core machine. The setup and usage procedures are the same for remote workers."

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Manager
;;; If this is the manager,
;;; then set *manager-p* to t,
;;; place just below the model's run call as well as
;;; the workers' addresses and ports.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar *manager-p* t) ; <- set manager status here
```

"Here's where you tell ACT-Concurrently whether it's going to run as a manager or as a worker. You can see there are a lot of comments here and I really tried to make it clear for people who might not have much Lisp experience. *manager-p*'s value of t lets act-concurrently know this node is the manager. "

```
;; Worker addresses and ports
;; Format them as a quoted list of "dotted pair" lists,
;; like '(("127.0.0.1" . 4321) ("192.168.2.8" . 4322))
;; where each dotted pair is the address and port of one
;; worker.
(defvar *worker-addresses-and-ports*
  '(("127.0.0.1" . 4321)
     ("127.0.0.1" . 4322)
     ("127.0.0.1" . 4323)))
```

"List of dotted pairs of strings and integers."

```
;; Put your expression to start your model
;; here as a quoted expression,
;; like: '(bst-experiment 10)
(defvar *model-run-expression* '(bst-experiment 10))
```

ACT-Concurrently wants to call your model’s run function separately, rather than in the model file. The reason has to do with setting each worker’s random seed between defining the model and running the model. I’ll talk a bit more about that later, but that's the reason.

I’ve slightly modified that Unit 8 tutorial building sticks model for demonstration purposes.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Worker
;;; If this is a worker node,
;;; then place the worker's address and port here
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Worker's address replaces nil, format it as a string, like
;; "127.0.0.1"
(defvar *worker-address* “127.0.0.1”)

;; Worker's port replaces nil, format it as an integer, like
;; 4321
(defvar *worker-port* 4321)
```

"For localhost, okay to use either 127.0.0.1 or the machine's external-facing IP address. Just be sure to be consistent for that worker in itself and in its entry in the manager."
"Port is an integer"

```
;; Uncomment this following line to automatically start
;; the worker into its ready state:
; (worker-listen-for-job)
```

Call worker-listen-for-job to start the worker into its ready state. Once you've edited those three lines for a worker and placed that copy of ACT-Concurrently into ACT-R's user-loads folder, it'll start automatically with ACT-R and go into its ready and listening state.
The worker will then listen for a job to run and return the job expression's value to the manager.
At this point, ACT-Concurrently is ready to load automatically with ACT-R. The workers will be listening for jobs, and all you have to do to run is tell the manager (send-out-jobs). When you revise your model, just call (read-model-file) and then (send-out-jobs) to run again.
If your model writes something to a file, it'll do that on each worker. Whatever value(s) your model run function returns will be returned from each worker to the manager and collected into a list.

Performance

- Theoretically: speedup factor –> # workers
- Single-Machine (4-core i7) ≈ 3x speedup
- Network (7 workers) ≈ 4.5x speedup

Single machine: one manager, three workers
Network: 4-core i7 MBP, 2-core i5 MBP, 2-core i5 MM
execution time: as fast as slowest machine (≈70s, i5MBP)
…but still 700 runs in 70s vs 100 runs in 45s on a single i7 core
More and faster cores should only help

Compatibility

- ACT-R: 6.0 & 6.1
- Lisps: Any supported by ACT-R: ACL, MCL, CCL, LW, CMUCL, CLISP, SBCL, ABCL
- OS: Mac OS X, Windows, Linux

Should work with any environment that ACT-R works with… at least that's what the libraries I used claimed.
Only tested briefly w ACT-R 6.0 & 6.1 in CCL 1.10 on Mac OS 10.10

Security

- Executes arbitrary Lisp code
- No encryption
- No authentication
- Run on machines behind a firewall using ports closed by that firewall

Stop workers when not in use

Randomness

- Each run should be independent
    - …but that assumes each uses a unique random seed
- Seeds each worker
    - current time * worker "ID" * 500

Per Dan's suggestion parallel model running runs into a potential issue with trying to ensure randomness: multiple workers might start with the same random number generator seed. My solution is to have the manager assign seeds to workers at the start of a job. This is why it needs to call the model's run function separately from the model code, to give it each worker a chance to define a model, then call SGP for the seed, then run.
Where "ID" just means position w/in manager's list of workers.
Doesn't guarantee seed uniqueness, but should be okay. A revision might redefine some of ACT-R's random module so that each worker's random module must request its seed from the manager at the time that the worker's random module is created.

To Do

- Guaranteed unique random seeds
- Security
- More informative status & error messages
- Load balancing