Idea for a Distributed, Parallel Algorithm

Gene Michael Stover

created Friday, 22 October 2004
updated Monday, 8 November 2004

Contents

1 Introduction

As of 2004-Nov-08, this is still under construction. I hope to complete it within a couple of weeks.

I had an idea for an algorithm to distribute a computation job among lots of computers.

I used to be interested in distributed systems, but I haven't kept up-to-date with it since 1992, so I have little doubt that this algorithm has long since been

written up. On the other hand, if the state of the literature now is what it was in 1992, it has been documented in the extremely abstract, not for programmers.

So here is my algorithm, presented by a programmer for programmers.

2 Assumptions

1. You have a lot of *slave* computers.
2. The slaves could be under-powered, unreliable pieces of junk, though when they do a computation, they give a correct answer. So the slaves are *unreliable*, but they are *trustworthy*.
3. You can run a small client application on the slaves. The client does not require a human operator.
4. You have a single *coordinator* computer.
5. The slaves are connected to the coordinator through a network.
6. Slaves do not communicate with each other. They have no knowledge of each other.

3 The Slave's Algorithm

The slaves have a really simple algorithm. Here is the pseudocode:

1. Let *last-answer* = Nil.
2. Until done
   (a) Make a remote procedure call to the coordinator in which we send *last-answer* & receive a *job*.
   (b) Run the *job*, saving the result in *last-answer*.
3. Exit.

4 The Big Idea

You have a Big Job to compute. It could be almost anything.

You divide your Big Job into a lot of smaller Jobs.

My algorithm provides a priority queue into which you place your Jobs.

My algorithm has a loop that doles the Jobs to the Slaves.

5 More Assumptions

RPC functions, but few assumptions about how they are implemented.

6 The Coordinator's Loop

The coordinator's loop of my algorithm assumes that the queue has been primed with all those small Jobs. Here is the loop in pseudo-Lisp.

```
(defun idpa-loop (q done?)
  (loop until (done?)
        do (let ((request (rpc-recv)))
                ;; Handle the request.  Doing so might
                ;; alter the queue or change other things
                ;; so that done? returns true.
                (handle request)
                ;; Send the next job to the slave.
                (let ((job (queue-get q)))
                  (incf (job-outstanding job))
                  (rpc-reply job request)))))
```

7 The Slave's Loop

A Slave initiates communication with the Coordinator. That communication is effectively a Remote Procedure Call (RPC)[1] in which the Coordinator acts as a server & the Slave acts as a client. The Slave always reports the result of the last job it did, with some kind of *nil result* value allowed the first time the Slave contacts the Coordinator.

The Coordinator replies to the remote procedure call with the next Job from the queue, except that we allow a reply that means "There are no more Jobs right now. Ask me again in T seconds." Let's call this special case the *Nil Job*.

8 The Priority Queue

The important stuff is in the priority queue's weight function.

Let's define the relevant parts of a job. A Job has

args Specific to the application domain, not my algorithm. These are sent to a Slave when we tell it what to do next.

prerequisites A possibly empty list of Jobs that must be done before this one.

outstanding-count Number of times this Job has been sent to a Slave without receiving a reply.

id Unique identifier. Possibly implemented as a GUID.

The priority queue must always provide the next Job that we should hand to a Slave. Features of the priority queue's ordering of Jobs include:

[1] http://en.wikipedia.org/wiki/Remote_procedure_call

1. no Job is starved[2],
2. take advantage of the parallelism possible by having multiple Slaves,
3. efficiently take advantage of timeouts on Jobs that are necessary because the Slaves are unreliable.

These goals turn out to be easily accomplished.

I assume that the priority queue's implementation makes us of some kind of an *isLess* function to determine the relative order of the Jobs in the queue. Here is pseudocode for an *isLess* function.

1. Given x & y, two Jobs.
2. If x is a prerequisite of y, then x has priority & should appear in the priority queue before y.
3. Otherwise, if y is not a prerequisite of x, & if x's count of oustanding requests is less than that of y, then x has priority & should appear in the priority queue before y.
4. Otherwise, y has priority & should appear in the priority queue before x.

Here is a possible implementation of that pseudocode in pseudo-Lisp.

```
(defun is-less (x y)
  "Return true if & only if Job X should appear
in the priority queue before Job Y."
  (or (prerequisit-of x y)
      (and (not (prerequisit-of y x))
           (< (job-outstanding x) (job-outstanding y)))))
```

9 Abstract Example

Let's see how the algorithm works in theory with a small number of jobs & slaves.

Assume we have a Big Job to compute, & we have already converted it into smaller Jobs. For this example, assume there are only three such Jobs: J0, J1, & J2. None of the three Jobs are prerequisites for the others, so they may be executed in any order.

We create a priority queue that uses the appropriate queue-ordering function ([**?**,)], & we stuff the three jobs into it. Then we start the loop

Let's assume we have three Slaves: S0, S1, & S3.

[2] I assume there are no circular dependencies between the Jobs. If the tree formed by the *prerequisits* lists in the Jobs contain circularities (& therefor do not form a tree), Jobs will be starved unless the priority queue is omniscient in the sense of a nondeterministic finite automaton.

The first Slave, S0, requests a job, & the main loop hands it J0. Then S1 requests a job, & the main loop hands it J1. Then S2 requests a job, & the main loop hands it J2.

When S0 finishes cits work, it sends the reply to the Coordinator & requests

A Other File Formats

- This document is available in multi-file HTML format at http://lisp-p.org/idpa/.
- This document is available in Pointless Document Format at http://lisp-p.org/idpa/idpa.pdf.

References