22C078 Homework 6: Projects with Linda Programming

due 5 April 2004

Instructions: for this homework, email your solutions (attaching programs and files) to me,

`herman@cs.uiowa.edu`

with `[22C078] hw6` in the subject line; if you don't put this in the subject line, my spam-killer might destroy your email. A number of files will be available online from the course web site.

Group and Individual Projects

Unlike previous homeworks, this one has different options for groups of up to four students, or for individual students. The general idea of the project is to implement a version of Python Linda that uses networking (message-passing, sockets, and so on) and then to run parallel programs (such as the earlier exercises of summing an array and generating prime numbers) on the networking version of Python Linda. The *group projects* will be programs and interface specifications for the networked version of Python Linda. The *individual student* projects will be the customers of networked Python Linda: these students will write parallel programs that use the software written by groups of students.

The Need for a Standard Interface

Consider the task of a student doing an individual homework. That student can test a parallel program using the `Linda.py` module and its `ThreadingTupleSpace` class, however this isn't real networking. In order to use the software developed by some student group, the "network Linda" package will likely have a different interface. Unfortunatley, if each student group makes a different interface, the testing of a parallel program becomes cumbersome. So we will need a *standard* interface that all the groups use for their new modules. One of the first steps for all the groups is to meet and agree on such a standard interface — we'll discuss this in class.

Beyond just the interface, there is the question of how to run a parallel program on many machines. In a sense, we are just replacing threads by network hosts, but that leaves open the practical problem of figuring out which part of the parallel program executes on which network host, when to start it, how to initialize the tuplespace and so on. Groups also need to think about these issues and propose solutions and possibly standards, so that their "customers" (students doing individual projects) know what to expect.

Group Projects

Each group will choose a particular strategy for implementing tuplespace on a network of hosts that communicate using sockets. Perhaps surprisingly, there are many possible strategies, each with advantages and disadvantages. Here are a few:

- Classic Server-Client. One host is the Tuplespace Server and all other hosts send requests to this server for Linda operations. This is the easiest idea to implement (which is an advantage),

but it causes every Linda operation to send and recieve a message, which can be slow. Of course, to support parallel programming, the server would use threads or the `select` call to allow concurrent client requests.

- Peer-to-Peer Replication. Every host is both a server and a client (and of course, each host also executes some part of the parallel program's code). In the peer-to-peer design, each host has a complete copy of the tuplespace. Therefore, any program executing an `Rdp` or `Rd` operation doesn't require any messages to be sent across the network – the tuplespace is locally available. However, any *change* to the tuplespace (such as `Out`, `In`, or possibly `Inp`) has to be performed at all hosts, so that they all continue to have identical copies of the tuplespace. This implementation has the potential added advantage of *fault tolerance*, since the loss of one copy of the tuplespace doesn't really lose the underlying data.

- Cached Tuplespace. Another idea could be that each `Out` operation puts its argument into the local tuplespace only and sends no message. Now, when a parallel program starts a `Rd`, `Rdp`, `In`, or `Inp` operation, either the local tuplespace has the matching tuple or requests have to be sent over the network to other hosts, to examine their tuplespaces. This idea is somewhat simpler than Peer-to-Peer Replication, and potentially has some performance benefits too. There are some fancier adaptations of cached tuplespace; some advanced implementations actually move tuples around from one local tuplespace to another, in an attempt to balance load or bring data closer to where it is likely to be useful.

Individual Projects

Each student not participating in a group should write some new parallel program using Linda. If you've got some idea for an interesting application, please discuss it with me, it may be a good thing to try with Linda. If you don't have any ideas, then you might consider doing the following two problems:

Problem 1: Sorting

The problem is to sort values in tuples. Initially, the tuplespace contains tuples of the form `(i,string)` where `i` is an index value and `string` is a character string. For simplicity, the index values range from 0 to $n-1$. So an example of the initial tuple space for $n = 4$ could be something like

```
[ (0,'once'), (1,'we'), (2,'found'), (3,'it') ]
```

Your program should change the tuple space so that it is "sorted" in the sense that for any pair of tuples (i, a), (j, b), it should be that if $i < j$, then $a \leq b$. For the example above, this means that the tuple space should become

```
[ (0,'found'), (1,'it'), (2,'once'), (3,'we') ]
```

(this puts the strings in alphabetic order).

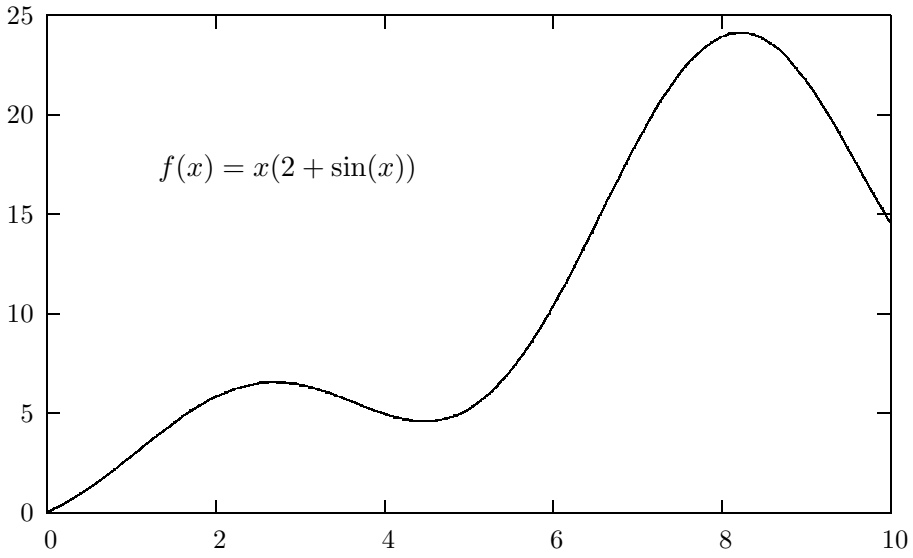
A good solution to this problem will sort in parallel: in some sense, multiple threads can concurrently contribute to the problem's solution.

Problem 2: Approximating an Integral

Compute, using Linda, the value of

$$\int_0^{10} x(2+\sin(x))\ dx$$

In other words, your program should approximate the area under the following curve:



Your program should compute the integral by summing the areas of rectangles. Usually this is done by splitting up the integration interval [0,10] into small intervals, computing the area of a rectangle for each interval, and then summing up all the rectangle areas. Consider some small interval $[a, b]$. For points a and b, one can imagine two rectangles that approximate

$$\int_a^b x(2+\sin(x))\ dx$$

Both rectangles have width $b - a$; one rectangle has height $a(2 + \sin(a))$ and the other has height $b(2 + \sin(b))$. Let us call these two rectangles "approximately equal" if the absolute value of the difference between their areas is at most 10^{-6}.

Unlike many algorithms that simply divide the starting interval [0,10] into, say a thousand equal-size units, your program will use an ***adaptive*** strategy. Whenever there is an interval $[a, b]$ to be integrated, your program will first check to see if the two rectangles for this interval are approximately equal: if they are approximately equal, the area of one of the rectangles (it doesn't matter which) is considered to be the answer for that interval; otherwise, the interval $[a, b]$ is divided into two smaller intervals $[a, \frac{a+b}{2}]$ and $[\frac{a+b}{2}, b]$. Then these two smaller intervals are integrated in the same way. Therefore, the exact number of intervals depends on how things work out with the function $f(x) = x(2 + \sin(x))$.