

## FAST LADY

**Financial software must produce results quickly to keep up with the market. Linda helped Izzy Nelken and Robert Bjornson reduce run times**

**M**any financial applications demand rapid computer calculations in order to take advantage of short-lived market situations. Since users often employ large networks of workstations or other desktop systems, it is worth examining whether run times can be cut by distributing a single application to run in parallel across several processors.

To explore this question, we chose Algorithms' HedgeWatch, and created a parallel version of the code using Scientific Computing Associates' Network Linda. The Linda programming environment proved very convenient – only a small section of the code had to be modified to get marked improvements in performance through network parallelism. For example, a 10-processor network gave a speed-up factor of 7.5.

In this article we show some of the intricacies of HedgeWatch and explain how Network Linda may be used to create a distributed code that runs in parallel on networked workstations.

### HedgeWatch

Given a target portfolio, HedgeWatch hedges by seeking a combination of financial instruments (the replicating portfolio) that will behave identically to it under a set of scenarios. These scenarios might involve various term structures, volatility values and other relevant parameters.

The main task is to solve a stochastic partial differential equation. An important part of this is the evaluation of many financial instruments at many scenario points.

This part of the computation, called the scenario module, is very time-consuming. However, since these computations are independent of each other, it is also a likely area for performance gains through parallel processing.

### Network Linda

Network Linda consists of a few simple operations that can be added to a programme written in a base language like C

or Fortran to produce a parallel version of that programme.<sup>1</sup>

Linda leaves the base language to do the chores for which it was designed – arithmetic, loop control, procedure calling, I/O and so on – and concerns itself solely with interprocess communication and control. It manages these functions using a globally shared, associative object memory called tuple space.

Tuple space consists of a collection of logically ordered sets of data called tuples. There are two kinds of tuples resident in tuple space: process tuples, which are under active evaluation, and data tuples, which are passive.

The tuple space model allows a programme to be visualised as a "bagful of tasks". The "bag" can be used for interprocess communication because tuple space is shared by all the processes. Each CPU in a multiprocessor or on a network can grab programme tasks from tuple space and return results to it. When all the tasks have been completed, the programme terminates.

The greater the number of processors at work, the greater the time saved through parallel processing. Linda programmes are portable because tuple space is logically independent of the underlying computer or network. Linda codes developed on one architecture can be recompiled and run in parallel on other architectures.

There are four basic tuple space operations: out, eval, in and rd. Out and eval add information to tuple space in the form of new tuples. In and rd extract information from tuple space by using a pattern, or template, to search for a matching tuple, and extracting the information contained within it. The operations are atomic and block until a suitable match is available.

Network Linda implements the tuple space concept on networks of workstations. Each workstation acts both as a computational server (computing evals), and also

<sup>1</sup> For a perspective on Linda, see N Carrero and D Gelernter, *How to write parallel programs: A first course* (MIT Press, 1990). Introductions can also be found in D Gelernter, *Getting the job done*, Byte, November 1988; and D Gelernter and J Philbin, *Spending your free time*, Byte, May 1990. Reprints of the two articles are available from Scientific Computing Associates.

as a tuple space server, responsible for a particular disjoint section of tuple space, chosen via a hash function. This distributes the load of handling Linda operations across the entire network, allowing tuple space to be accessed simultaneously by many processes.

A Network Linda computation is invoked by running a utility called `tsnet` on one node, which becomes the master node. The master determines the CPU loads of the other workstations on the network and chooses those that are least loaded. If necessary, it then copies the executable to the other nodes; if NFS (network file system) is available, no copying is necessary. It then invokes the executable on each node via `rsh`.

Tuple space communications are converted into low-level network packets and transmitted to the workstation managing the relevant piece of tuple space. When a request arrives, it will interrupt the `eval`d function being executed, and the new tuple or template will be matched against data already residing in tuple space. If a match is found, it will be sent to the appropriate node. Once the request has been handled, the node will return to processing `eval`d functions.

The process of searching for and matching against tuples is greatly aided by analysis during compilation. The analysis restricts the set of tuples that need to be compared by performing much of the matching during compilation. In addition, the tuples are stored in data structures appropriate to the ways in which they are referenced. These and other optimisations of compile time and run time ensure a high degree of efficiency.

## Parallelising the code

HedgeWatch consists of several phases: initialisation, the scenario module, computation and post-processing. Only the computing-intensive scenario module phase was parallelised, as it accounts for a large part of the total run time when working on large problems.

The parallel strategy employed is a "master/worker" model, in which the master,

on one node, creates tasks for the worker processes, which reside on all other nodes in the network. In HedgeWatch, the master performs the sequential parts of the computation up to the scenario phase. When the scenario phase is reached, the master creates the workers via `eval` and sends them their initialisation data. After all the workers have received this information, the master dumps the first tasks to be executed into a queue in tuple space. The workers allocate tasks among themselves by accessing the task queue.

This master/worker approach provides automatic, dynamic task scheduling and load balancing: faster processors finish tasks sooner than slower processors and reach into tuple space for new tasks more frequently. In one of our experiments, one of the nodes had a slower CPU than the others and, as expected, executed fewer tasks.

The programme's logic is as follows:

1. The master programme starts up a few workers, via `eval` (....).
2. It prepares several vectors with relevant information and sends them to the workers.
3. The master creates the initial task tuples and installs them in a queue in tuple space.
4. It then loops, doing the following in descending order of priority:
  - a) servicing requests from the workers for additional data which is stored only in the master;
  - b) accepting completed tasks from the workers and storing the results;
  - c) executing a task.

Whenever a task is completed, another task is put on the queue.

5. When all tasks have been processed, the master puts "poison tuples" that cause the workers to exit.

It should be noted that the workers sometimes need data from a large table stored in the master. This is the source of type 4a

requests. Since all requests go to the master, this is a potential bottleneck. To reduce the number of requests, each worker has a local buffer in which it stores data elements that have arrived from the master. On subsequent requests, it checks this cache and will only request data from the master if it cannot find it locally.

## Experimental results

We ran the parallel version of HedgeWatch on a network of Sun SparcStation 1s and a purely sequential version of the code on a single machine. These results are presented in the figure above. The workstations used for this experiment were all on a single local area network. Although NFS was available, the executable was copied to each node in order to reduce network traffic and improve repeatability.

The absolute performance levels achieved compare quite favourably with those available from much more costly mainframe systems or time-shared supercomputers.

These results bode well for other financial analytics applications, as many of these display properties similar to those of the HedgeWatch scenario module. Mortgage-backed securities analysis<sup>2</sup>, portfolio optimisations and Monte Carlo simulations of all kinds are quite compute-intensive, yet feature a high degree of independence among the many iterations they perform with different variables. ■

Izzy Nelken is a consultant at Algorithmics, Toronto, Canada, and Robert Bjornson is a research scientist at Scientific Computing Associates, New Haven, Connecticut.

<sup>2</sup> See the report by Stavros Zenios and Leigh Cagan, available from Scientific Computing

Runtime and speedup on SparcStation 1s

