An efficient distributed tuple space implementation for networks of workstations

Antony Rowstron⋆ and Alan Wood⋆⋆

Department of Computer Science, University of York,
York, YO1 5DD, UK.

Abstract. In this paper an overview of a novel run time system for the management of tuple spaces, which utilises *implicit* information about tuple space use in Linda programs to maximise its performance. The approach is novel compared to other tuple space implementations because they either ignore the information, or expect the programmer to provide the information explicitly. A number of experimental results are given to demonstrate the advantages of using the our approach.

1 Introduction

Linda is a well known coordination model[1]. For many years implementations have followed very traditional routes based on the work at Yale[2]. However, since the initial traditional implementations the model has evolved; one of the major changes has been the addition of multiple tuple spaces. Most implementations "tack" tuple space names onto tuples in an ad hoc fashion and subsequently treating them like any other field within a tuple. Some implementations[3] use the potential locality information that tuple spaces can provide, but expect the programmer to provide *explicit* information about the "type" or "usage" of a tuple space, thus explicitly declaring the locality of a tuple space.

Our new run-time tuple space management system (kernel) uses *implicit* information to enable it to classify every tuple space as either a *local tuple space* (LTS) or a *remote tuple space* (RTS) on the fly and without extra communication or programmer guidance. A LTS is one which can only be accessed by one process and a RTS is one that many processes can access. What is novel about our approach is the use of implicit information to enable the kernel to transparently move large numbers of tuples around the system to gain optimum performance. It should be stressed that making the distinction between LTSs and RTSs does not alter the semantics of the Linda model, a user sees no distinction between a LTS and RTS.

2 The novel implementation technique

A tuple spaces classification controls where its tuples are stored. If the classification changes then the tuple space migrates to the correct storage place for its

⋆⋆ Contact: {ant,wood}@minster.york.ac.uk

⋆ Funded by an EPSRC CASE grant with British Aerospace Military Aircraft.

new classification. The kernel itself is distributed and Figure 1 shows its general structure. The kernel has two distinct sections, *Tuple Space Servers* (TSS) and *Local Tuple Space Managers* (LTSM).

The TSS is a set of stand alone processes which together act as a tuple server. They receive tuples and requests for tuples. All RTSs are stored in the TSS, but usually distributed over many processes within the TSS. How tuples are distributed between processes is not important for this paper[3]. The LTSM is a set of library routines which are linked into user processes; LTSMs join and leave the kernel with the user processes they belong to. A LTSM is able to find information on whether a tuples space is a LTS or a RTS *dynamically* (with no inter-process communication), and subsequently controls the movement of tuple spaces. A tuple space will only ever migrate from a TSS to a LTSM or vice-versa (*never* from LTSM to LTSM or TSS process to TSS process). A LTS is always stored on the LTSM of the user process which know it as a LTS.

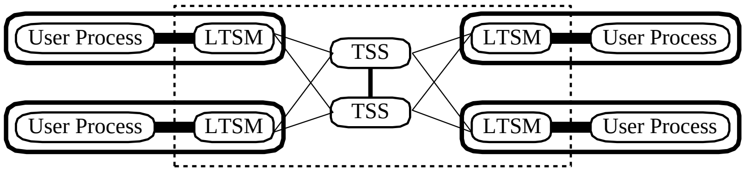

Fig. 1. Diagram showing the layout of the kernel.

When tuple space operations occur the LTSM checks internally to see if the tuple space is a LTS. If so, the LTSM updates itself accordingly, with no communication with the TSS. If it is a RTS the LTSM contacts the relevant TSS process(es). The bulk movement of tuples occurs when either a bulk movement primitive is performed (`collect`[6] or `copy-collect`[7]) or a tuple space handle for a LTS is placed in a tuple in a RTS.

The bulk tuple space primitives require a source and a destination tuple space. If the the classification of both tuple spaces are the same then they are performed in the same part of the kernel; LTS in a LTSM and RTS in the TSS and there is no migration of tuples. If the source tuple space is a RTS and destination is a LTS, then the TSS performs the duplication (if necessary) and migrates (in one or more blocks of *multiple* tuples) the tuples to the correct LTSM. If the source is a LTS and the destination a RTS then the LTSM performs the duplication and migrates the tuples to the TSS. If no movement is required there will not be any movement of tuples. When a tuple space handle for a LTS is placed in a tuple in a RTS the LTSM checks to see if the handle refers to a LTS. If so, the tuple space is migrated to the TSS (thus becoming a RTS).

In this section a very brief overview of the implementation techniques have been presented, for more information see Rowstron et al.[8]. This describes why the bulk movement of tuples is more efficient than the single movement of tuples. The most important point is that the system uses *implicit* information rather than relying on extra explicit information provided by the programmer.

[3] The method used is the same as for previous York kernels[4, 5].

3 Experimental results

In order to demonstrate the performance of the kernel we compare its performance with the LTSM enabled and disabled. When the LTSM is disabled the kernel degenerates into an implementation similar to many more traditional implementations[4, 5] because all tuple spaces are treated as RTSs and therefore stored in the TSS.

Experiment	1		2		3		4		5	
LTSM	Off	On	Off	On	Off	On	Off	On	Off	On
`out`	2.890	2.861	2.769	0.018	2.802	0.018	2.770	0.018	2.769	2.767
`collect`	n/a	n/a	n/a	n/a	n/a	n/a	0.007	0.038	0.007	0.007
`in`	3.224	3.227	3.270	0.017	3.303	3.877	3.258	3.670	3.255	0.056
Total	6.114	6.087	6.039	0.035	6.105	3.895	6.035	3.726	6.031	2.830

Table 1. Performance of the kernel with and without the LTSM.

Table 1 shows the experimental results (in seconds) for a number of experiments using a 10 Mbit/s Ethernet and a TSS distributed over 4 Silicon Graphics Indy workstations. Experiment 1 shows the time taken to place 1000 tuples in a RTS using `out` and retrieve them using `in`. This demonstrates that there is no significant difference in time between the LTSM being on or off (because in both cases the tuple space is stored on the TSS). Experiment 2 is the same except the tuple space can be classified as a LTS. The results demonstrate that when the LTSM is on it detects that the entire operation is local to the process. When the LTSM is off the tuple space is treated as a RTS and the execution times reflect this. Experiment 3 shows the time taken to place 1000 tuples in a LTS, then place in `UTS`[4] a tuple containing the handle of that tuple space (so the tuple space becomes a RTS) and then retrieve the tuples. This shows that the time taken to perform the operation is less when the LTSM is on. This is because, when the movement of tuple occurs (as the tuple is placed in the `UTS`), the tuples are packed into larger packets for dispatching to the TSS. Experiment 4 shows the time taken to place 1000 tuples in a LTS, then `collect`[5] them all into a RTS and then retrieve them. As one would expect the times are comparable to those of experiment 3, as essentially the same tuple space "movements" are occuring. This again demonstrates how the bulk movement of tuples creates a more efficient implementation. Experiment 5 shows the time taken to place 1000 tuples in a RTS, then `collect` them all into a LTS and then retrieve them. The results again demonstrate the effectiveness of the bulk movement of tuples and tuple spaces. This is an important operation, as it represents the communications behaviour of the basic operations that a process has to perform to overcome the multiple `rd` problem[7].

[4] `UTS` is a *universal tuple space*, which all processes have access to.

[5] This primitives moves all tuples which match a given template from a source tuple space to a destination tuple space, and returns a count of the number of tuples moved[6].

These results demonstrate the effect that the LTSM has on the execution time for a number of specific examples, and that the LTSM does not slow the kernel down, and indeed that when used it provides large speed increases for certain classes of operations. Within the scope of this paper it is not possible to demonstrate the performance of the kernel against other kernels. We have compared our kernel with a commercial version of Linda, called C-Linda[6]. For certain classes of algorithms our kernel achieves a speedup of between 10 and 70 times. For more information on the performance of the kernel (including the results for a "real-world" problem) see Rowstron et al.[8].

4 Conclusion

An overview of a new Linda kernel, which transparently uses bulk movement of tuples to achieve performance increases over traditional implementation methods has been presented. The bulk movement of tuples is achieved by using *implicit* information about tuple spaces gathered on the fly, rather than by using either compile time analysis or explicit programmer added "hints".

Currently work is focusing on the development of a multi layer hierarchical kernel, thereby altering the definition of tuple spaces from a discrete categorisation to a continuous one. This kernel uses the same implicit information to provide better locality information to the kernel and will have the potential to be used by hundreds of workstations.

References

1. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
2. R. Bjornson. *Linda on distributed memory multiprocessors.* PhD thesis, Yale University, 1992. YALEU/DCS/RR-931.
3. B. Nielsen and T. Sorensen. Implementing Linda with multiple tuple spaces. Technical report, Aalborg University, Denmark, 1993.
4. A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. In *Transputer and occam developments*, pages 125–138. IOS Press, 1995.
5. A. Rowstron, A. Douglas, and A. Wood. A distributed Linda-like kernel for PVM. In *EuroPVM'95*, pages 107–112. Hermes, 1995.
6. P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.
7. A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem. In *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 357–367. Springer-Velag, 1996.
8. A. Rowstron and A. Wood. An efficient distributed tuple space implmentation for networks of heterogenous workstations. Technical Report YCS 270, University of York, 1996.

This article was processed using the LaTeX macro package with LLNCS style

[6] Available from Scientific Computing Associates, Connecticut, USA.