

STRUCTURING DISTRIBUTED COMPUTATION AND VISUALIZATION USING FACET TREES

A.H.DAVIS, J. LU AND C. SUN

*School of Computing and Information Technology
Griffith University,
Brisbane, Queensland 4111, Australia
E-mail: {horatio,jun,scz}@cit.gu.edu.au*

Mathematical computing requires tools for collaborative work backed by the computational power of parallel processing, using existing applications. A framework for structuring distributed computation, based on a tree of shared objects, is presented; the components of this framework wrap around existing serial objects. An implementation, Concurrent Matlab, is presented and discussed.

1 Introduction and Motivation

This paper treats mathematical computation - simulation, modelling, and related tasks using high-performance compiled code steered by an interpreted language optimized for the task, running over a workspace of objects representing numbers and functions operating on those numbers. Matlab, Mathematica, Octave, and Maple are examples. Wood et al ¹³, Chen, Rine and Simon ³, and Saran et al ¹⁰ give good reasons for wishing to share these computations with other members of a research team distributed across a network. From different directions, they propose or describe solutions to the *collaborative visualization* problem: permitting several users to steer and visualize the results of a computation as it is run. To solve this problem, the simulation must run, information from it be abstracted and presented, and steering feedback obtained from users across a network, on a human (near-real) time scale.

The traditional approach is to distribute the computation explicitly. Matlab has the Parallel Toolbox⁹ and the Cornell MultiTask Toolbox ¹⁵ which use wrappers around the Parallel Virtual Machine and the Message Passing Interface to provide explicit shared workspace objects, distributed computation and (by extension) distributed interaction. However, the programmer must concentrate more on the distribution than upon the mathematics, and collaborative interfaces must be hand-built.

Another approach is *conventional collaboration transparency*. One instance of the computation is run on a server and the interface is distributed to the interested parties by tools such as NetMeeting, SunForum, and TechTalk

⁸. The code need not be rebuilt to be collaboration-aware. This widens the applicability but wastes the multiple processors it runs on. Begole et al. ¹ extend this to *flexible collaboration transparency* by replicating the code and state of an application on each node, and synchronizing their inputs. Useability and flexibility are enhanced by viewing an application as a set of objects, some of which are safely replaced by collaboration-aware equivalents.

The framework presented below extends this approach by viewing a computation as a tree of interacting objects. Leaves of the trees at different nodes can be identified as facets of the same conceptual object. These are transparently replaced by objects that are aware of the identity and act to maintain consistency and causality across nodes.

2 Framework and Nomenclature

A *simple object* is a mathematical object (variable, graphics primitive, or piece of code) that exists in one place at one time, where a *place* is a workspace of simple objects, executing independently of other places. Such objects have parents and children: a window has a graph in it, which has some axes, a line, and some labels. They thus structure naturally into trees. A *facet* is a simple object that cooperates with other peer facets to maintain a consistent distributed state, which it presents seamlessly to the objects in its' local workspace. A facet contrasts with an Orca object replica⁴ by being identified with a node instead of cached, destroyed and recreated at need. It contrasts with the *chares* of Charm++ and the *data aggregates* of EPEE⁷ by being a point of presence for the whole object, not a discrete fragment with explicit fragment interfaces.

Collections of facets that refer to the same conceptual object are aggregated into *mirrors*. Mirrors grow by merging; facets are initially single-facet mirrors, and seek other mirrors with the same identifier to merge. This poses the challenge of a distributed, deterministic identifying scheme for facets, mirrors and trees, and algorithms to efficiently search for identified facets.

A *rune* is a facet that encapsulates a value and an identifier, and additional metadata as property/value pairs. The nature of the primary value determines the use of the rune.

A mirror of runes which hold data implements a distributed data structure. The value held in that location must be consistent and fresh (all evaluations after a given assignment yield the same value as was assigned, irrespective of location). Algorithms to guarantee these properties must be implemented *within* the rune, but are a straightforward recasting of existing work⁵.

Following the idea of a computation as a first-class value⁶, the primary value may be a piece of serial code, which the facet insulates from communication and synchronization issues. A mirror of such runes represents a MIMD parallel computation.

A *glyph* is a rune which represents a component of a graphical interface, for example a line, isosurface, window, menu, or an input field. A mirror of glyphs represents a component of a collaborative interface, as each facet exists in front of a different user. A tree of glyphs expresses the hierarchy introduced above, branches of the tree being coherent subsets of the interface. Glyph values are the data they are displaying to or receiving from the user. Consistency maintenance of these values across a mirror is addressed by existing work¹⁴
11.

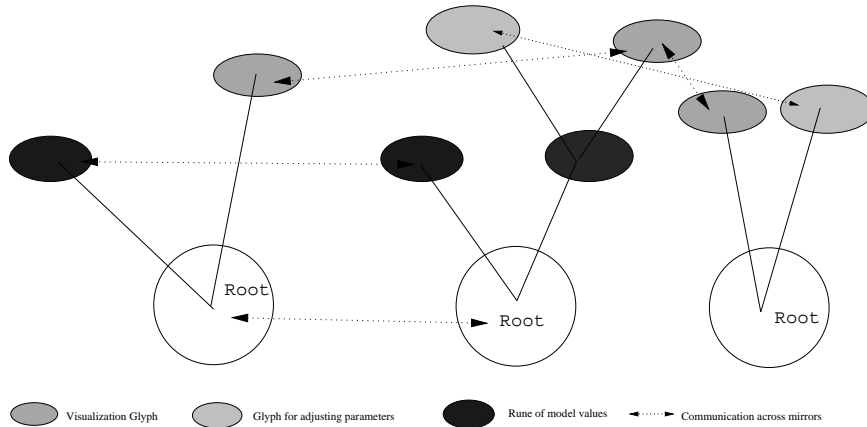


Figure 1. Three facet trees and three mirrors

The collection of facets at a place form a *facet tree*. Facet trees grow by *grafting*, where new leaves replace existing simple objects, or branches of other facet trees are cloned and attached. Growth begins from a facet named **Root**, which manages the place where the tree exists and provides a point of communication for other trees.

An *ensemble* is a forest of facet trees (and places) whose **Root** objects are facets of a single mirror. Ensembles may grow dynamically by seeding trees in new places. Communication across trees is done implicitly, by involving a local facet which then propagates the relevant information across the mirror.

In general, facets of a given mirror occupy different locations in different

trees, and trees in an ensemble each have a different set of facets. The example in figure 1 has one computational mirror, shared across the left two trees, one GUI shared across all three, and a second GUI shared across the right two trees. Note also that the two GUIs are windows in their own right on the visualization tree, but are children of another window on the centre tree. This flexibility is characteristic.

3 Implementation - Concurrent Matlab

Concurrent Matlab is a toolbox that implements glyphs under Matlab. It provides the building blocks for collaborative interfaces to conventional computations. It is a set of C MEX-files for Matlab 5.2, running on an IBM SP2 and connected by the LAM² implementation of the Message Passing Interface. The ensemble is a set of daemons written in C, each controlling the standard input and output of a Matlab engine. The glyphs are wrappers around regular Matlab Handle Graphics objects, each rendered and manipulated by a specific daemon's engine. There is one daemon per user interface. One of these is distinguished as the root node, and presents a command prompt. This daemon's engine performs the computation.

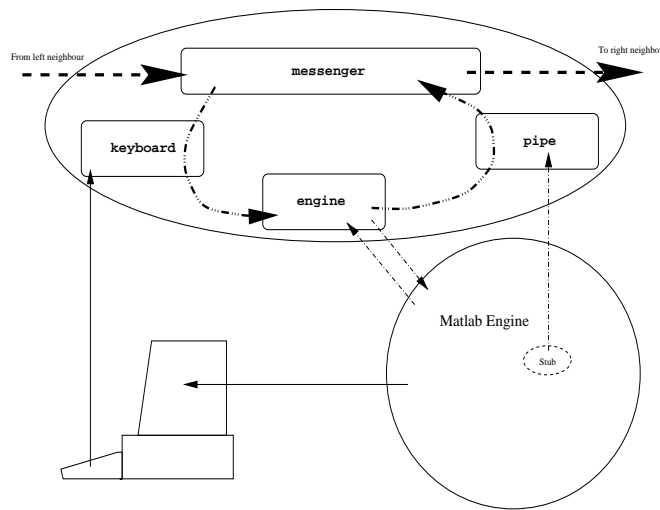


Figure 2. Implementation of a Concurrent Matlab node

Figure 2 shows the internal structure of a node. The daemon has four

threads linked by two FIFO queues, inbound and outbound. The basic quantum of work is the `mxRequest`, which contains enough information to reproduce a single Matlab operation. The `messenger` thread takes `mxRequests` from the outbound queue and passes them via MPI to the next node in the ring. It also receives requests from the previous node and puts them on the inbound queue. The `keyboard` thread takes commands from the user and puts them on the inbound queue as `mxRequests`. The `engine` thread takes `mxRequests` from the inbound queue, uses the Matlab Engine API to reproduce them on the Matlab engine, and passes them to the outbound queue. The `tap` thread listens for requests raised in the local engine to create, delete, set a property or read a property of a HG object, or to execute a callback. The relevant primitives have been replaced with versions which also notify this thread, which places them on the outbound queue for other nodes to reproduce.

All daemons are members of a MPI intracommunicator, whose membership fixes the ensemble at startup. Each `mxRequest` passes from the outbound queue of one daemon to the inbound queue of its neighbour, and is absorbed after circling the ring. Consequently, requests raised at each node, or at different times on different nodes, remain causally ordered without a formal request history or timestamp scheme.

A small application was built with the toolbox for evaluation. Testing used two to four daemons, each running on a node of an IBM SP2, with LAM running in client-to-client mode over the high-performance switch. The goal was for communications latency to be small relative to Matlab computation and rendering times. This was in fact the case. The response time to graphical commands on the root node was excellent, but the round trip for callbacks from other nodes impacted responsiveness. Two limitations became evident. First, conflicting requests on the same mirror execute in inconsistent orders at different nodes. Adding timestamps and serialization is required. Second, there is no group awareness, which ensures that conflicting user requests will arise. Mirroring other users' cursors will supply it.

4 Conclusions and further work

This paper presents a framework that should help solve the collaborative visualization problem by distributing objects in the computation across a cluster of processors. The framework places the communication and synchronization logic within objects called *facets*. The executing serial code is not aware that the other modules it cooperates with are in other places, and the objects it acts on reflect that action elsewhere. The structure of these facets into trees

enables flexible composition of a working set of data and code at each node. The visualization components of this framework have been implemented for Matlab, linked by MPI. The architecture and implementation limits of this system have been discussed, and solutions suggested. Future implementation efforts will focus on building the entire framework as a set of Java classes under Matlab, using appropriate algorithms from the REDUCE(REAL-time Distributed Unconstrained Collaborative Editing) ¹² and GRACE(GRAPHic Collaborative Editing) ¹¹ projects. The end result should be useable for production scientific coding.

Acknowledgments

This research was done using the facilities of the Queensland Parallel Supercomputing Foundation. Their help and support was invaluable. We would like to thank Andrew Lewis and an anonymous reviewer for their insightful comments and suggestions.

References

1. James Begole, Mary Beth Rosson, and Clifford Shaffer. "Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems." *ACM Transactions on Computer-Human Interaction*, 6, June 1999.
2. Gregory D. Burns, Raja B. Daoud, and James R. Vaigl. "LAM: An Open Cluster Environment for MPI." In *Proceedings of the 1994 Supercomputing Symposium*. SC94, June 1994.
3. Jim X. Chen, David Rine, and Horst D. Simon. "Advancing Interactive Visualization And Computational Steering." *IEEE Computational Science and Engineering*, December 1996.
4. Alan Fekete, M. Frans Kaashoek, and Nancy Lynch. "Implementing sequentially consistent shared objects using broadcast and point-to-point communication." *Journal of the ACM*, 45, January 1998.
5. Saniya Ben Hassen, Henri E. Bal, and Cerie J. H. Jacobs. "A task- and data-parallel programming language based on shared objects." *ACM Transactions on Programming Language Systems*, 20, November 1998.
6. John Hughes. "Why Functional Programming Matters." *The Computer Journal*, 30, 1989.
7. J.Jezequel. "EPEE: An Eiffel environment to program distributed-memory parallel computers." *Journal of Object-Oriented Programming*, 6, 1993.
8. Y.N. Lakshman and Michael R. Miller. TechTalk: A Web Based System for Mathematical Collaboration. World Wide Web, <http://penguin.mcs.drexel.edu/~techtalk/>, 1998.
9. Paul Pauca, Kun Liu, Joel Hollingsworth, and Rudy Martinez. Parallel Toolbox for Matlab. World Wide Web, <http://www.mthsc.wfu.edu/pt/pt.html>, 1995.
10. A. Saran, D. Agrawal, A. El Abbadi, T.R.Smith, and J.Su. "Scientific Modeling using Distributed Resources." In *Proceedings of the 1996 Conference on Geographical Information Systems*. Association for Computing Machinery, 1996.
11. C. Sun and D. Chen. "A multi-version approach to conflict resolution in distributed groupware systems." In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*. IEEE, April 1999.
12. "C. Sun, et al. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems." *ACM Transactions on Computer-Human Interaction*. Vol.5, No.1, pp.63-108, March 1998.
13. Jason Wood, Helene Wright, and Ken Brodie. "Collaborative Visualization." Technical report, University of Leeds, 1996.
14. Y.Yang, C.Sun, Y. Zhang, and X. Jia. "Real-time Cooperative Editing on the Internet." *IEEE Internet Computing*, May/June 2000.
15. John A. Zollweg and Arun Verma. Cornell Multitask Toolbox for MATLAB. Cornell Theory Centre, 2000.