Web-based Interaction and Monitoring for Parallel Programs (Via Conspector)

Parthasarathy Ramachandran and Laxmikant V. Kale'

June 21, 1999

Abstract

Although sequential applications are becoming predominantly interactive, parallel applications remain in the primitive "batch-based" mode. New broad-based parallel applications will require interactive behavior. Even the traditional science/engineering applications can benefit from online monitoring and interactive control. We present Conspector, a tool that supports Web-based submission, monitoring, and interaction with parallel applications. With the ubiquity of the Web, this tool and the associated software technology makes it possible to access huge computational power from any desktop, in an interactive manner.

1 Introduction

Parallel machines are now widely available. Very large machines, with hundreds of processors, are operational at several national centers, and laboratories, and available for scientists and engineers. Departmental machines with 16 or more processors are also becoming commonplace, and clustered workstations, either in dedicated mode or utilizing the existing desktop workstations, are increasingly being used as parallel computers. In spite of this ubiquity, the current methods of running parallel programs remain antiquated, awkward, and inefficient. For running large scientific simulations at the national centers, the situation is not unlike the past when batch jobs on punched cards used to be submitted. One submits the job to a queue, and doesn't know when the job actually starts running; when the job is running, there is no simple way of monitoring its progress. While the sequential programs have become increasingly interactive, with GUI based interfaces, the parallel programs have remained in batch mode. At best, one is able to carry out some run-time visualization, but ability to interact with a parallel program is mostly missing. For a "killer application" of parallel computing technology to emerge for the broad market, such interactiveness is clearly necessary.

In this paper, we describe a tool named **Conspector** and an associated software framework that attempts to bridge this gap. Specifically, our objectives were:

- To provide a Web-based submission process for parallel jobs. In a typical computational research lab, one has access to several different parallel computers. Some of these may be local, whereas others may be at remote sites. With the current emphasis on portability, each of these installations may have the ability to run the particular application a user wants to run. However, the mechanics of running the program on each machine are tedious to remember and execute. A Web-based submission process will alleviate this tedium, and facilitate effective use of resources.
- To provide a Web based monitoring system: when a parallel application is running, the user may want to monitor it continuously or periodically. For example, for a job that runs for tens of hours, a user may want to monitor system characteristics, such as processor utilization, to ensure that the job is continuing to run effectively, or application characteristics (e.g., total energy of a biomolecular system being simulated), to ensure that the

simulation has not entered some undesirable regime. One should be able to submit a job from one location, and monitor it from anywhere else via the Web.

- To be able to interact with a running parallel program, and steer it as desired. Such steering may be necessary to improve performance (e.g., to trigger a load balancing phase), or for interactive functionality (to apply forces to particular atoms of a biomolecule, or to refine a region of simulation space that requires higher resolution), or to support fully interactive applications such as Distributed Interactive Simulation (DIS) or 3-D interactive games.
- To provide a resource sharing facility: usage of multiple resources available from a location should be monitored by the job submission process, with appropriate queuing facility for the local jobs.
- To facilitate management of a set of parallel computers: one should be able to keep track of the usage statistics for each individual resource automatically.

The Conspector meets these objectives. It provides general purpose Web-based submission, monitoring, and system level steering functionality, and provides software components to facilitate development of application-specific functionalities.

Several other projects share some of these objectives. The Autopilot project [10] at the University of Illinois is aimed at providing real-time performance visualizations, supported by an instrumentation framework, and fuzzy logic based automatic resource management. The Globus project [3] which aims at supporting a computational grid and applications that can span multiple different supercomputers, includes several components that are relevant. For example, its GRAM component [2] provides a uniform interface for the resource management. A "heart beat monitoring" to continuously check if the involved computers/programs have not crashed is also part of Globus. Paradyn's [9] approach is to dynamically instrument the application and automatically control the instrumentation in search of performance problems. The Performance Consultant module, which automatically directs the placement of instrumentation, has a knowledge base of performance bottlenecks and program structure so that it can associate bottlenecks with specific causes and with specific parts of a program. Visualization Tool (VT) [11] for the IBM SP Parallel Environment - uses the same technique, but allows only system statistics to be viewed. VT is definitely not portable, since it only works on the IBM RS6000 and SP plat forms. Falcon [4] is one of the early and ambitious monitoring and steering system for parallel applications. It includes a set of "sensors" for monitoring purposes, and uses the steering interface . Breezy [1] (BReakpoint Executive Environment for visualiZation and data DisplaY) is a tool that provides the infrastructure for a client application to attach to a data-parallel application at run time.

The Conspector distinguishes itself from individual approaches above in in or more of the following aspects: It is a web based system and supports general-purpose monitoring, steering and interaction interfaces. Its load balance steering is supported by a powerful object-migration based load balancer. Further, it supports job submission, queuing and asynchronous attachment protocols, and allows clients to inject arbitrary messages into the parallel program, triggering desired actions.

2 Conspector Functionality

The Conspector is a tool that streamlines the effort involved in submitting parallel programs for execution, and monitoring and steering their progress. Its main strengths lie in its inherent portability, resource efficiency, and extensibility. The only software piece needed by a user is an applet-enabled web browser. Users visit a program submission page, and after authenticating themselves with a password, request a parallel program to be executed on the machine they decide to use. The Conspector back-end then determines the admissibility of the request. If the request is inadmissible, the user is prompted to try again with a possibly different set of program parameters. Otherwise, an applet window with the Conspector display is brought up. Which applet to choose is based on which parallel program has been started, and the appropriate applet

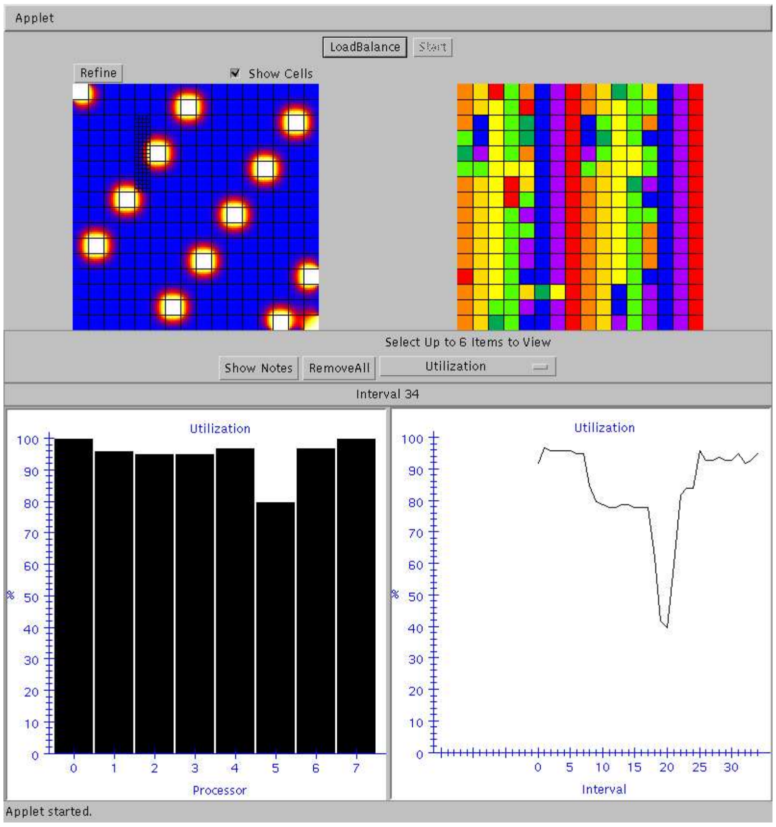

Figure 1: Conspector Display of Jacobi Relaxation Program

is chosen by the Conspector back-end from a repository of pre-written applets. A *Start* button on the Conspector display allows the user to start executing the program.

A similar mechanism exists for users to attach to currently executing parallel programs. Users visit a web page that lists the usage statistics of all machines accessible to the Conspector subsystem, and information about the parallel programs running on them. Users are then free to choose a program they wish to attach to, and the browser is loaded with the Conspector display. After a short initialization delay, monitoring data starts showing up on the display, and steering mechanisms get enabled.

The Conspector front-end consists of two parts - the first part is an application specific display, entirely definable by the application writer. Examples of application specific displays are rendering application specific data as images, data plots, application progress information etc. The Conspector allows the application programmer to define a steering mechanism completely specific to the application. This mechanism allows the parallel program run to be steered by inputs from the Conspector console. As an example, figure 1 is the Conspector display for a Gauss-Jacobi relaxation on a 2-D grid of points. The grid is subdivided into an array of cells. Each cell contains the data for its section of the grid. A cell may independently refine its data to increase the accuracy in its region of the grid, by doubling the resolution of the grid points in each dimension. In the North-West corner, is a button named 'Refine'. Clicking on this button during the run of the program causes the running Jacobi program to refine data for the highlighted cells.

The second part of the Conspector display shows diverse kinds of system related information. An example is the set of processor utilization values across all processors, both in the form of a current snapshot, as well as the history of values. Another display is the lengths of system

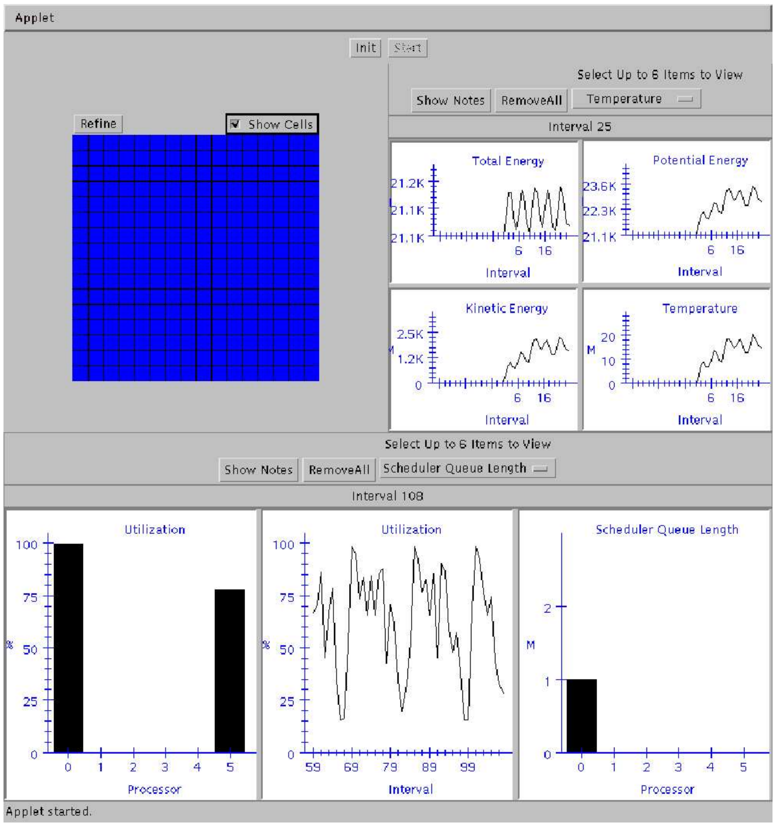

Figure 2: A Conspector display of NAMD (A production quality Parallel Molecular Dynamics Program developed at the University of Illinois)

queues, which is the count of the number of messages on each processor waiting to be processed. This display gives an indication of the amount of work being done on each processor. A steering mechanism for the system can also be defined by the user. An example of this is the Load Balancing button in figure 1. (This is just an illustrative example, since in most production systems load balancing is done automatically).

3 Architecture and Implementation

In this section it is important to distinguish between the roles of the application developer and the user. The user is any individual, who has login access to the Conspector system, and hence can use the utility to submit and monitor pre-written parallel jobs. The Conspector display for these applications is also pre-written, and the user does not have the capability to modify them, or add his own.

The application developer on the other hand, has access to the repository of Conspector applets, and has the capability to develop new parallel programs to run on the system, and write new (and modified) applets that will serve the purpose of monitoring and steering them.

The application developer is free to add monitoring/steering functionality to existing parallel programs, or to develop new programs with such functionality. In either case, development needs to be done both at the applet end, as well as in the parallel program itself. In this section we describe the necessary steps to be performed in both these cases, and the support provided by the Conspector framework.

The Conspector is built on the Client Server Interface to Converse (called the Converse Client Server (CCS) Interface). Converse[7, 5, 8] is a runtime system that provides portable, efficient implementations of all the functions typically needed by a parallel language or library

The Conspector system has a 3-Tier Client Server architecture (figure 3). The first tier consists of the Conspector applet, which displays monitoring information, and presents a steering interface. The second tier is the cluster of intermediary processes, which route data between the applet and the running parallel program. These processes also maintain configuration information, detailing the status of various constituents of the system. The third tier is the executing parallel program. TCP sockets connect the first and the second tier. The CCS Interface connects the second and the third tier.

3.1 Converse Client Server Interface

Ordinarily, a parallel program runs on dedicated nodes of a parallel machine. (This could be a traditional parallel machine, or the more recent network of workstations). The executing parallel program has no interaction with the outside world. The only information gathered about the program execution is the data generated by the program as part of log files. This information can be only used for post-mortem analysis.

The Converse Client Server (CCS) interface allows an external program to contact an executing parallel program. Once this connection is established, it can pass data into and get data out of the parallel program. The client can connect to the parallel program if there is no firewall between the two.

CCS Client side Protocol

- The parallel program prints out its IP address and port number to be contacted, on its standard output.
- The client calls `CcsConnect()`. This step is a two-way handshake between the client and the parallel program. This call allows the client to get information about contacting the individual node programs. This also serves to register the client with the parallel program. As far as the client is concerned, the interface to the call `CcsConnect()` is uniform. Hence the client need not know whether the parallel program is executing on a network of workstations or a traditional parallel machine.

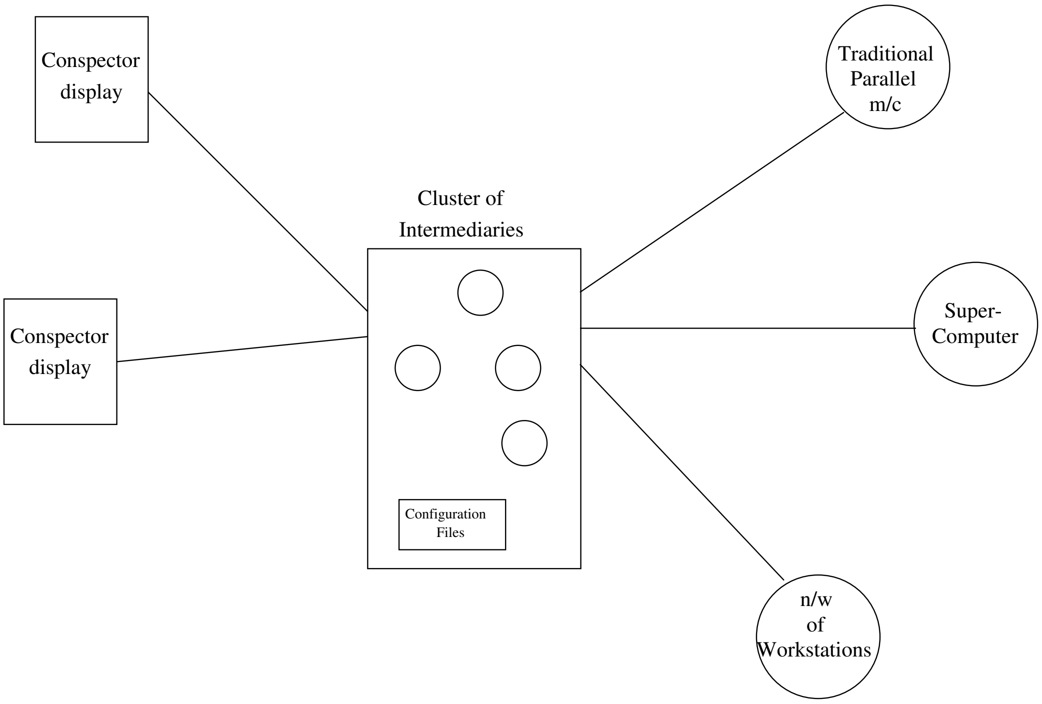

Figure 3: Conspector Architecture

- The client does a `CcsSendRequest()` to one (or more) node programs of the parallel program. Each call to `CcsSendRequest()` sends a message (signifying a particular request) to the specified node program. This call is non-blocking.
- The client does a `CcsRecvResponse()` from a particular node program. In this step, it waits for a reply to a previously issued request. It is important to note that the CCS Interface does not maintain any state about issued and pending requests. It is the responsibility of the client to wait at an appropriate time for an appropriate reply to an earlier request. The `CcsRecvResponse()` call could be blocking or non-blocking. In case of a non-blocking call, it takes an additional parameter specifying the timeout delay after which it ceases to wait for a reply.

CCS Server Side (Parallel program side) Protocol

- The parallel program needs to implement a handler that will carry out the action on receiving a particular CCS message. This handler could be an existing Converse handler, or there could be new handlers defined solely for the purpose of acting on CCS messages.
- The parallel program needs to register the above described handler with the system. This step associates the handler with the message type specified during registration. The call used for this purpose is `CcsRegisterHandler()`.
- In the code for the message handler, code to send a reply to the client should be implemented. (This is of course user-level protocol dependent). In case a reply is desired to be sent, the call to be used is `CcsSendReply()`.

3.2 Intermediary Processes

Applet security specifications stipulate that applets cannot communicate with any host other than the one they are downloaded from. This necessitates the middle tier - a cluster of *intermediary* processes. These processes run on the webserver machine. These processes act as intermediaries for the data transfer between the applet and the running program. Its responsibilities are two-fold. One is to transfer monitoring data coming from the parallel program to the Conspector applet, and second to pass steering information from the applet to the parallel program.

Configuration files are used to maintain information about currently running programs. Information maintained spans the following:

- Current set of programs running, and their attachment related information
- Current set of processors/machines in use
- Load information on all processors/machines
- Set of intermediaries that are currently running a parallel program. (both with a Conspector applet attached to it and not)
- Authentication information

When a submit (or attach) request is made on the web page, it results in a HTTP POST request being sent to the webserver. The associated CGI program then accesses the configuration files to determine if the request may be processed, and which intermediary processes are free to handle the request (in case of a submission request).

Concurrency control is an issue when multiple requests can be made at a time. Absence of adequate concurrency control mechanisms in accessing the configuration files may lead to over allocation of processor resources, leading to degraded performance. In this case, file locking is used to synchronize access to common configuration files.

Apart from password protected authentication while submitting requests, secure login is used to encrypt monitoring and steering data that flows between the parallel program and the Conspector applet.

3.3 Applet Client

The application developer needs to write a few classes that conform to certain guidelines, and then link in these classes with certain system specified classes to create the Java Archive file, which is then used by the Conspector display.

The application panel layout and content is completely upto the developer, and system classes do not require any internal information about it. The only necessary step is the registration of the panel with the system.

To enable application specific monitoring, the developer registers *handler objects* with the system. These objects are responsible for handling different kinds of application specific data. These objects can belong to any arbitrary class, but must implement the `AppsInterface` interface. The `AppsInterface` interface defines a function that must be implemented by the class of the object. This function is responsible for handling application data that is prefixed by a unique *tag*. The object and the tag that the object seeks to represent are registered with the system. The system class instance has a hash table which maintains the mapping between the tag and the handler object reference.

The system class instance contains a communication thread, whose responsibility is to receive all the messages bound to the applet from the intermediary process. Every message it receives is given to the appropriate handler object for processing.

To enable application specific steering, Java GUI events are listened to, using facilities provided by the Java API. The system class instance provides a method to be called to request a steer action to be performed. This call specifies the action to be taken and an identifier for the server side function performing the action.

The system-steering mechanism is implemented in a similar fashion to application-specific steering, except that the server side function identifier is restricted to a predefined set, provided as a utility to the developer.

3.4 Parallel Program side

The server (i.e the parallel program) side has two components : one to enable monitoring, and the other to enable steering.

The `CWebPerformance()` interface in the Converse runtime is responsible for providing system specific monitoring information. Currently, system information comprises the set of processor utilization values across all processors, both in the form of a current snapshot, as well as the history of values. Another display is the lengths of the Converse runtime scheduler queues, which is the count of the number of messages on each processor waiting to be processed. This display gives an indication of the amount of work being done on each processor.

The `CWebPerformance()` interface is extensible - a system provided registration function can be used to register additional functions that provide system specific information. However this facility is available only to the system developers, with a view to providing flexible system information mechanisms for the future.

The application interface allows the implementation of application specific monitoring, i.e it allows the application to deposit arbitrary data which is then transported back to the Conspector applet. This interface provides functions for the following:

- Every node program of the application can deposit data that is then collected at node 0 (via concatenation), and then sent as one data item to the Conspector applet. It is useful in cases where aggregate kind of information is needed to be viewed on the client side.
- Any node program can deposit data, which will be transported to the applet immediately - no collection takes place. This function is useful in cases where isolated data segments need to be viewed at the client end.

The CCS interface is used to implement application specific steering. The `CcsRegisterHandler()` call is used to register a function that will carry out a steering action when a command to steer is received. The action performed by the function is totally upto the application.

4 Summary and Conclusions

We describe Conspector, a Web based tool that simplifies the process of submitting a parallel job to one of many available parallel computers, and more importantly makes it possible to monitor and interact with the parallel application. Automated instrumentation in the Converse runtime system makes it possible to capture performance characteristics of a running application. Converse's object migration strategies allow the user to trigger load balancing actions interactively. Software libraries on the client (browser) end as well as the parallel application end facilitate development of interactive parallel applications. Although some of the initial applications of this technology involve steering — i.e. a corrective action, such as an adaptive refinement, we believe that its real impact will be in truly interactive parallel applications of the future. The interfaces provided by the tool are sufficiently flexible to support such applications. Faster, compression based communication (possibly using SDDF) between the parallel application and the Java applets, is one of the issues that needs further work.

References

[1] D. Brown, A. Malony, and B. Mohr. Language-based Parallel Program Interaction: the Breezy Approach. In *Proceedings of the International Conference on High Performance Computing (HiPC'95)*, New Delhi, India, December 1995. IEEE Computer Society, Tata McGraw–Hill.

[2] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 4–18. IEEE-P, March 1998.

[3] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[4] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. In *Proceedings of the Symposium on Frontiers of Massively Parallel Computation*, pages 442–29, February 1995.

[5] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[6] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[7] Robert Brunner L. V. Kale, Milind Bhandarkar and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Co nverse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel P rogramming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[8] Robert Brunner L. V. Kale, Milind Bhandarkar and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[9] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T.Newhall. The Paradyn Parallel Performance Measurement Tools. Technical report, University of Wisconsin at Madison, Department of Computer Science. available from `www.cs.wisc.edu/paradyn/papers.html`.

[10] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, Chicago, IL, July 1998.

[11] Jerry Yan, S. Sarukhai, and P.Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software – Practice and Experience*, 25(4):429–461, April 1995.