

Paraflow: A Dataflow Distributed Data-Computing System

(From Browsing to Teraflops)

Roy Williams and Bruce Sears
California Institute of Technology

roy@cacr.caltech.edu, bruce@srl.caltech.edu

Abstract

We describe the Paraflow system for connecting heterogeneous computing services together into a flexible and efficient data-mining metacomputer. There are three levels of parallelism: a dataflow paradigm provides functional parallelism, connecting services by channels; each service may be a SPMD parallel program; and each node of the SPMD service may be multithreaded. Paraflow is easily integrated with web-computing, using MIME types, HTTP forms, a Java GUI for setting up the network of services, and a simple API to allow web servers to be utilized in the computation. Emphasis is on providing a bridge from straightforward, public, unauthenticated browsing of data to collaborative, high-performance data-computing.

Modern high-performance computing and communication is capable of providing enormous processing speeds and high-speed data transfer between processing engines, while RAID disk and tape robots can store the vast quantities of data that drive these systems. But effectively using such hardware entails learning how to write or port code and how to submit jobs to a queue, as well as other mundane details such as what port number the FTP server runs on. Collaborating on a software project often involves inventing a communication protocol at the byte level, then writing raw socket code. These problems are exacerbated when the code must be portable between different machines.

But to many people, computing means something quite different: a web-client machine at home which can browse remote databases, carry on a chat-group, or download movie previews. Under the surface, complex events occur that may involve several machines, but these are hidden from the user, enabling him to think about an objective rather than its implementation. Web servers already provide much of the software infrastructure that makes distributed computing easier: the MIME mechanism to deal with heterogeneous data types, HTML forms for data input, multithreading, and connectivity to databases. But perhaps the most important advantage of a web interface is the recognition of a web browser by a neophyte user and the consequent feeling of control.

The idea of Paraflow is to bring some of the ease of use of the commodity machine to the high-performance world, allowing scientists to think about their data, not about the computer. Such a system is in operation as a public, purely web-based browser for Earth-observation images, called the Synthetic Aperture Radar Atlas [1], and many of the elements for Paraflow are derived from this project. SARA is a collaboration with San Diego Supercomputing Center and the University

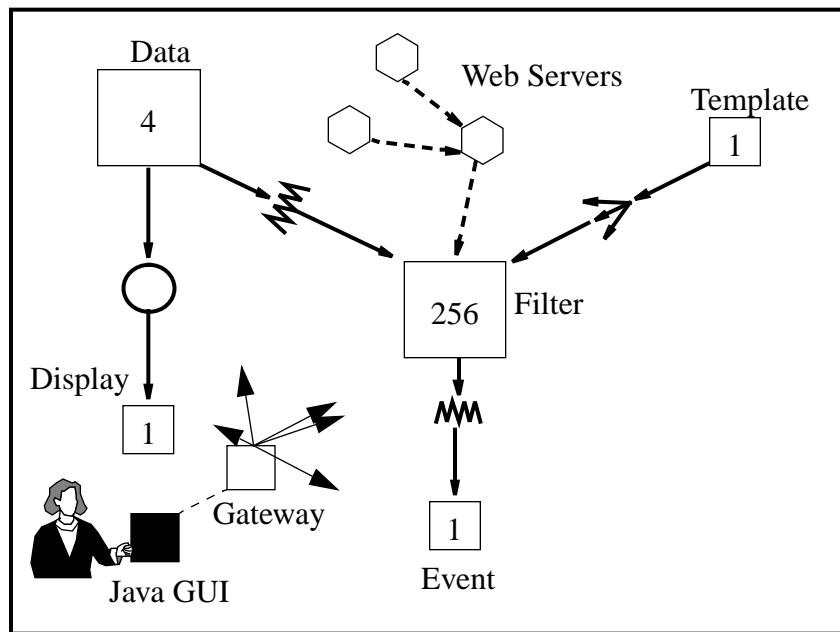


Figure 1: A network of services that might be used for data-mining. The data is delivered from a 4-process *Data* service to a *Filter* service on a parallel supercomputer; the data is also sent to a monitor running a *Display* service that the user can see. The *Filter* service uses a collection of *Templates* to be used in pattern-matching, and a web-based DBMS to mine the data. Any events it finds are sent asynchronously to an *Event* service, which may be a DBMS or a flat file. The network is controlled by a Java-enabled GUI via a *Gateway* service.

of Lecce, Italy, and it is being extended to do on-line processing of SAR data. Future applications include an archiving and processing system for LIGO (Laser Interferometer Gravitational-Wave Observatory) [2] and a proposed Digital Sky Survey.

Paraflow can be used as a web service: multiple web servers provide data, multiple processes run the functional parts of the computation; and each can be tested, debugged, and linked with others using a freely available Java GUI. Computing occurs at the client in Java, or, with suitable access permissions, on other machines.

At the next level, a user can connect computing services into a persistent, heterogeneous network that can take advantage of high-speed data storage and communication devices, doing sophisticated datamining and data filtering. The interface may be (if desired) from the same Java applet that was used for initial exploration, the only difference being that authentication is required for users that will run the supercomputers.

It should be emphasized that Paraflow is to be built in strong collaboration with real applications, including the LIGO and Digital Sky projects mentioned above, always trying to deliver a simple, efficient solution to user requirements — we do not intend to present a finished product before finding applications. This document is merely a snapshot of a plastic road-map.

How it Works

Paraflow can take advantage of a persistent, distributed network of parallel supercomputers to run large datamining and data-processing tasks that may last hours or days, as shown in the example of Figure 1. Three levels of parallelism are provided, to allow maximum utilization of data and computing services:

- Dataflow functional parallelism allows complex tasks to be broken into simpler ones, it allows easy reuse of existing software modules, and it provides a simple conceptual framework for combining those modules;
- SPMD message-passing achieves maximum scalability and efficiency for data- and

numerical-intensive computations, and furthermore provides reuse of existing SPMD parallel applications;

- Thread parallelism encourages efficient use of Symmetric Multiprocessors (SMPs) that are gaining an increasing share of the computing marketplace.

Ease-of-use is achieved because:

- The system is based on two conceptually simple paradigms: dataflow between off-the-shelf modules, or *services*, (as in AVS and other data visualization systems); and SPMD parallelism, where each service may be a set of loosely synchronous processes on the nodes of a parallel computer.
- Programs and data can be brought together in new and powerful ways through a point-and-click Java GUI. The output of the GUI is a human-readable text-file which can also be created with a text-editor, so the GUI is an option, not a necessity.

Portability is achieved because:

- A user's first, trust-building, use of the system only involves Java and web-servers to do computation, without persistent channels, finding machine-specific executables, or setting up a dataflow network.
- The executable programs that run on different kinds of machines need not be available locally, but can be fetched from a web-server that already has a catalogue of services compiled for different machine types.

Computational speed and efficiency is achieved because

- The real work is done with the proven model of SPMD parallelism, where a service may have a thousand nodes of a parallel machine working together, using optimized message-passing for inter-node communication.
- Each node program in a service may be multi-threaded, allowing compilers to get the greatest possible speed from the compute node. It may also be that the "node" is in fact a shared-memory multiprocessor (SMP), in which case multithreading is the preferred route to parallelism in place of the SPMD model.
- The data channels between services are explicitly parallel, so that multiple high-bandwidth physical links can be used as a (conceptually) single channel.

Elements of Paraflow

In Paraflow, *services* are connected together by *channels*. Services are computing units, connected by the dataflow paradigm to form a directed acyclic graph. The data that flows between services is assumed to move as *packets*, or messages, from service to service. In an application, a packet may be a large binary object, or it may be metadata tables, like a table in a relational database. In either case, the packet has an HTTP header with information such as packet size and what type (MIME) the data is.

To make the system accessible, the dataflow graph and control parameters may be created with a Java applet that can be downloaded from a web-server, together with documentation. This means that the system may, if desired, be usable by anyone in the world with an Internet connection. On the other hand, each initiation of a service and a stream may occur only with a suitable

authentication, and all data transfer may be encrypted.

Paraflow may be used either with persistent services and channels, or by using only channels which are of the server type. In the latter case, the services are arranged in a tree-like network. The services could be CGI scripts running on web servers, each making HTTP requests to other web servers, processing the results, and passing them back to the root of the tree: the gateway service.

The gateway service is the coordinator for the network, communicating with the user's Java console, replicating output to other terminals for conferencing and collaboration, initializing and tearing down the network. It runs on the same machine as the web server that delivered Java applets, so that we are not handicapped by Java security rules.

Services

Services provide computing, filtering, and data I/O. We can think of these as groups of processors of a parallel computer, CGI programs such as database interfaces running on a web server, visualization engines, or other kinds of functional units or processing elements. Services communicate with each other through the channels attached to them: they can process packets of data, read or write data to and from file systems or other devices such as scientific instruments and framebuffers, or consume and provide metadata using database management systems (DBMS).

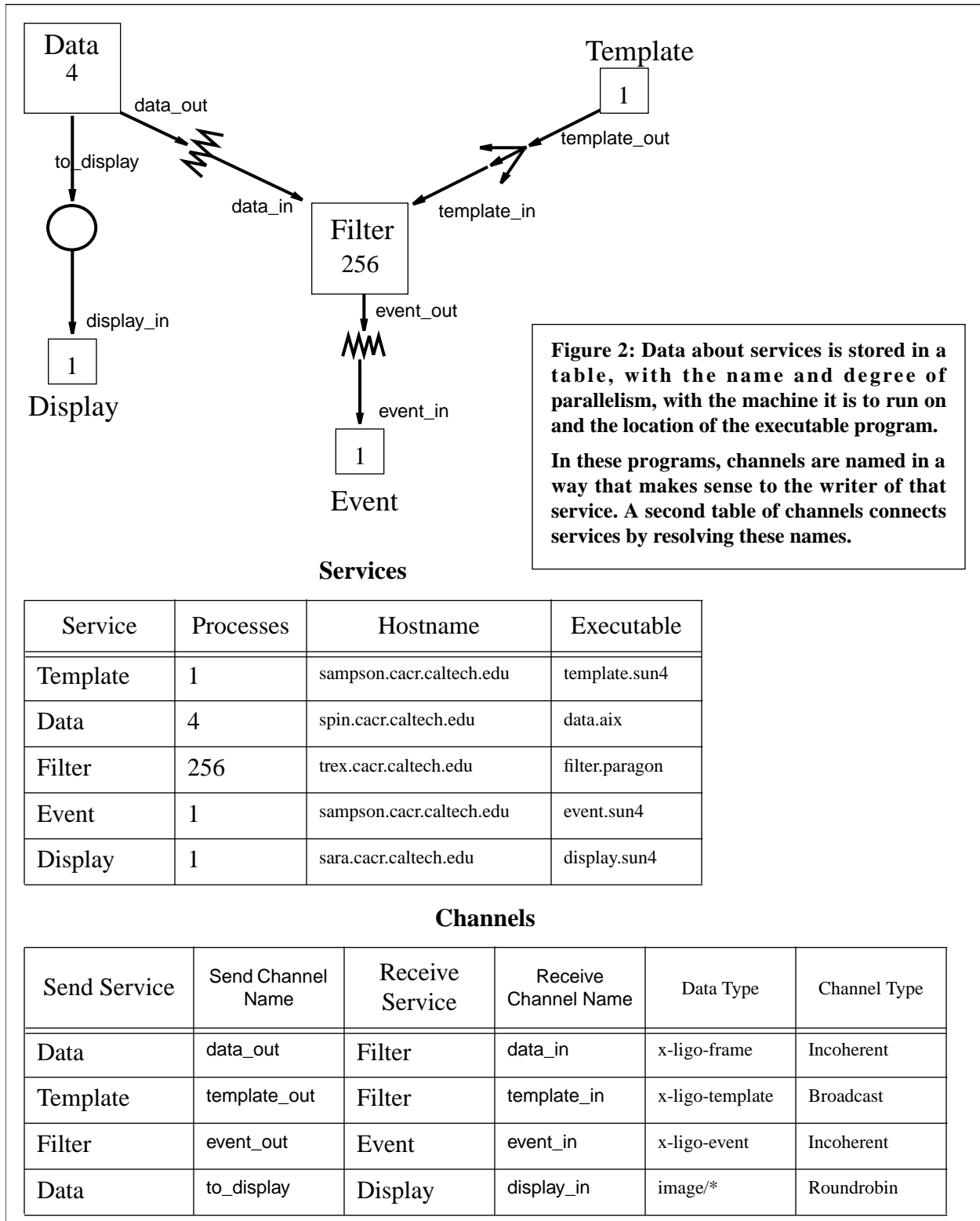
A service can be created by a user by writing C or Fortran or other programming languages, then linking it with a library. This library provides the initiation of the program (main program), then passes control to the user code; on exit, the user code should pass control back to the library. The user code is given the number and nature of the input and output channels that are available: a channel is semantically similar to a file pointer or file descriptor to C programs, and as a logical unit number to Fortran. Blocking, data flow, and progress of the computation are described in the next section.

A service may be parallel, using a number of processes or processors of a distributed-memory (or other) machine. The programming model is *SPMD* (Single Program, Multiple Data), so that each process of the service is running the same program. In an *SPMD* program, there may be occasional Loose Synchronization: a point in the program where all processes block until the last one reaches the synchronization, at which point a communication of some kind occurs between all the processes.

When a service is instantiated, it is given a metadata table containing such information as its processor number within the service, a name for the service, parameters related to its execution that were input to the Java GUI, and a set of channels. The channel includes the names of the sender and receiver services, as well as a local name that each service gives to its incoming and outgoing channels, as shown in Figure 2. The channel provides methods for sending (receiving) packets to (from) the other end of the channel.

As part of the initialization of the Paraflow network, there will be MPI communicator groups for each service and each channel, with packet movement layered on the communicator groups. When the service provides its name, it gets a communicator group connecting all the processes of that service. When a service provides the name of a channel, it gets a communicator group containing the processes from both sending and receiving services.

Each service also has a diagnostic output available, which is assumed to contain only small



quantities of text, frequently flushed, to communicate with the user of the system: every effort is



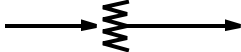
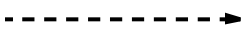
made to make sure that these diagnostics from the services are interleaved and appear in front of the user, or a log file, with maximum reliability and minimum latency.

Channels

Channels connect services, passing large amounts of data at potentially high bandwidth, or passing metadata forms in text format. Channels may be thought of as passing data in one direction only, at least for the purposes of creating a dataflow graph of services connected by channels. In fact, however, there is generally a “back-channel” in the opposite direction for small data such as acknowledgments or (in the case of a server channel) the request to the server.

When the sender and receiver services are serial, there are only two kinds of channels: persistent and server. A persistent channel is initiated when the service graph is instantiated; the sender can send a packet (pointer + length) to the channel, and the receiver can receive a packet from the channel. The receive comes in two parts, first a probe to find the length of the incoming packet, then the actual receive, where a pointer to a buffer is given by the receiving process that is long enough to take the packet. If the receiver is not ready when the packet is sent, the sender blocks; if the receiver probes for a packet and the sender is not ready, the receiver blocks. It need not be the whole of the sender or receiver process that blocks: we assume that in general these can be multithreaded, so that only a thread is blocked, not an entire process.

When the sender and receiver processes are parallel, there are several ways in which packets can be directed from the processes of the sender to those of the receiver, and different strategies for synchronization and blocking. We have chosen to implement the following useful cases:

-  **Broadcast:** If the sender is a serial process, each packet is given to all receiver nodes. If the sender is parallel, each node contributes a packet, forming a barrier synchronization of the sender, then an arbitrary one of these is selected and it is broadcast to the receivers.
-  **Round-robin:** Packets are distributed in processor-number order from sender to receiver. If there are S processes sending to R receiving processes, then packet n is taken from sender node $n \bmod S$, and send to receiver $n \bmod R$. This channel is completely deterministic and synchronizes both the sender and receiver services.
-  **Incoherent:** Packets are taken from the sender when available and given to the receiver nodes when they are ready. One algorithm for implementing this channel might be the “manager-worker” scenario as follows. Each sender process despatches packets until each receiver process has a packet, then waits for an acknowledgment message from one of the receiving processes, at which point another packet is sent to that process.
-  **Server:** Any receiver (client) node can issue a request to the sender (server), which responds with a packet. When many simultaneous requests come in, the

server can spawn threads (see below), and thus achieves the same degree of parallelism as the receiver. Servers can call on other servers to help satisfy requests, forming a tree structure extending away from the network of persistent services.

In addition to communicating with other services, there may be *intraservice* communication, where the processes that constitute a service use a message-passing or other language to exchange data, and each of these multiple processes may be multithreaded, as shown in Figure 3.

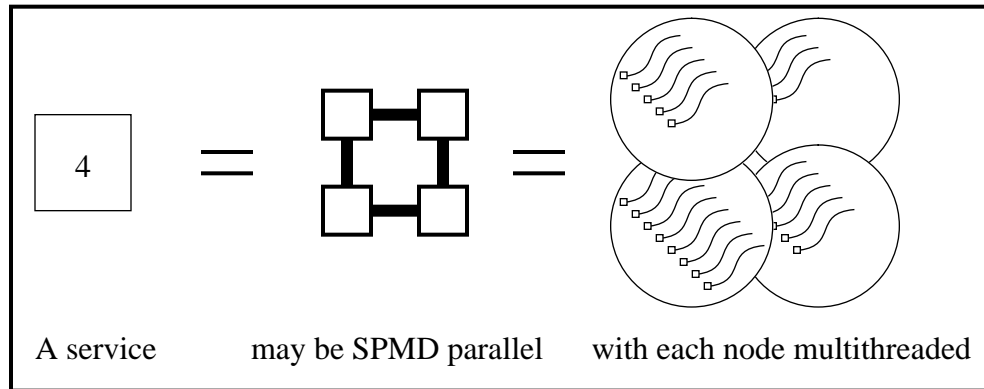


Figure 3: A service may be represented as a square with a number of processes: each of these processes runs the same program, and they may communicate by message-passing in addition to communication through channels. Each process may be multithreaded for efficiency in spite of blocked messages, and for utilization of SMPs.

The Gateway process

The *gateway* is a unique, serial process. It interacts with the user, or with a conference of users, providing input from users to the metacomputer and displaying the results of a computation. The functions of the gateway process are:

- To pass results and diagnostics to the human user; all processes in a Paraflow network have at least a diagnostic connection to the gateway;
- To orchestrate the persistent network, doing setup and tear-down;
- To establish socket connections from Java clients for communication with the metacomputer: the gateway is on the same IP address as the web-server that serves the Java applets which can be used to operate the Paraflow system, meaning that the Java security model allows remote creation of the gateway process and socket connections.
- To replicate results to many clients, to enable Paraflow to be used as a conferencing engine.
- To handle security and encryption.

Packets

Paraflow services communicate with each other by sending *packets*: chunks of data that are large enough for the coarse-grained nature of the metacomputer, but small enough to fit in memory. A packet is a stream of bytes with a length, the assumption being that the entire packet can be taken by the memory of the processor that is to receive it. Furthermore, this length must be known in

advance, since it is given to the receiving process before the first byte flows. If we add one more bit to this simple protocol, we can implement streams of unknown or very long length: the bit expresses if the packet is the last one of the stream or not.

The HTTP protocol provides much of the header structure we need here. In addition to the packet size, it provides a MIME type (what sort of data this is), and other strings, together with the ability to add fields such as the “last-bit” field mentioned above. It also provides a strong link with web technology, so that the “server channel” mentioned above can connect to a real web server: cheap commercial software makes it easy to have web servers interface with commercial databases and other powerful software.

Metadata Tables

A packet may be subclassed to a *metadata table*, which is a collection of records, such as a table in a relational database. The first line of the table is a comma-separated set of field names, while subsequent lines are records, each of which is a comma-separated list of the values of the fields. Such forms can be easily derived from web-mounted databases, or can be created from the output of HTML forms. Requests to web servers, as well as setting parameters and other input for services, can be made by filling in a web form. Thus, applications can be initially debugged by making them web services; a service can remain as a web service, so that members of a collaboration can test the parts of a network, assigning bugs expeditiously to the correct member of the software team.

Multithreading

Threads are lightweight processes that share memory, or at least share address space. Many multiprocessor workstations and parallel computers are configured as symmetric multiprocessors (SMPs) with a shared memory and fast crossbar. The threads model fits here naturally, providing access to the parallelism of these machines.

If a service has multiple channels coming into it, we might like to take the first packet that comes, and process it, in which case we would create several threads, each blocked on its own channel. When a packet arrives, a user-provided *handler* is called to deal with the packet. In this case, the user has relinquished program flow control to an *event loop* function from the Paraflow library.

Alternatively, a user may recognize parallelism in a coarse-grained loop in the code, and put in a compiler directive to encourage spawning of threads. Of course, a user can often take the best advantage of an SMP architecture by writing the code in terms of an FFT or matrix-matrix multiplies, in which case turbocharged functions for these should be available from the machine vendor.

Implementation

Currently (97Q1), cluster computing and parallel computing are coming closer than ever before. Let us think of a *cluster* as a heterogeneous collection of processors where each has an IP address, and the data routing is the responsibility of a system administrator who programs the routers. On the other hand, let us define a *parallel computer* by the presence of a highly-tuned communication backplane connecting a homogeneous collection of nodes. The cluster uses Unix or NT sockets, it would have less nodes than the parallel computer, and would be comfortable with a web-server on each node; the parallel computer has one or two IP addresses even for a thousand nodes.

A communications fabric for Paraflow must unify these points of view. One possibility is the MPI standard [3], which runs on either a cluster (MPICH [4] or LAM-MPI [5]) or on a parallel computer (usually vendor implementation of the MPI standard). Unfortunately, it is difficult to get this so-called “heterogeneous MPI” — however a committee is addressing this problem. One way to do this would be to use IP sockets and MPI, with gateway nodes that translate between the two standards. The Nexus protocol from Caltech and Argonne will soon deliver a heterogeneous MPI layer above the Nexus fabric, allowing both messaging between very different machines, and the efficiency of multithreading and multiprocessing on each machine.

References

- [1] SARA, <http://www.cacr.caltech.edu/sara>, or <http://sara.sdsc.edu/> or <http://sara.unile.it/>
- [2] LIGO, <http://www.ligo.caltech.edu>
- [3] MPI, <http://www.mcs.anl.gov/mpi>
- [4] MPICH, <http://www.mcs.anl.gov/mpi/mpich>
- [5] MPI-LAM, <http://www.osc.edu/lam.html>
- [6] Nexus, <http://www.mcs.anl.gov/nexus/>