Process Networks as a High-Level Notation for Metacomputing

Darren Webb, Andrew Wendelborn, and Kevin Maciunas
{darren,andrew,kevin}@cs.adelaide.edu.au

Department of Computer Science, University of Adelaide
South Australian 5005, Australia
Fax: +61 8 8303 4366, Tel +61 8 8303 4726

Abstract. Our work involves the development of a prototype Geographical Information System (GIS) as an example of the use of process networks as a well-defined high-level semantic model for the composition of GIS operations. Our Java-based implementation of this prototype is known as PAGIS (Process network Architecture for GIS).
Our process networks consist of a set of nodes and edges connecting those nodes assembled as a Directed Acyclic Graph (DAG). In our prototype, nodes represent services and edges represent the flow of data (in this case sub-processed imagery) between services. Services are pre-defined operations that can be performed on imagery, presently selected from the Generic Mapping Tools (GMT) library. In order to control the start and end-point of the DAG, we define an input node (the original image) and an output node (the result image).
To exploit potential parallelism, we extend our idea of a process network to a distributed process network, where each service may be processed on different computers. A single server coordinates computation and computation is performed by any number of workers. The server and workers together can beseen as a metacomputer. The server takes a process network from a client and distributes work to the workers. Each worker applies for work and decides if it is capable of performing the work offered. In this way, scheduling is essentially dynamic, and computation can be performed without client intervention.

1 Introduction

In this paper, we introduce the concepts of process networks, distributed process networks, and metacomputing, and the application of these concepts in the development of a prototype Geographical Information System (GIS). Process networks provide a well-defined high-level semantic model for the composition of processing steps, here used in a GIS.

PAGIS, our prototype GIS, provides a sophisticated user interface for domain experts to apply transformations to selected satellite imagery. The user selects and orders transformations in a process network from their local computer. This process network is then submitted to a metacomputer for computation across

geographically distributed high-performance computers. Scheduling is dynamic so the user requires no intervention in how processes are distributed. This factor is especially important for systems such as a GIS, as domain experts are generally not interested in learning parallel programming paradigms.

Section 2 of this paper defines and explains important characteristics of process networks and introduces distributed process networks. Section 3 is a brief discussion of metacomputing and metacomputing issues. Section 4 provides an overview of the PAGIS GIS, and Section 5 discusses some future work.

2 Process Networks and Distributed Process Networks

2.1 A Semantic Model for Composing Computations

The Kahn model of process networks [6] can be used to semantically represent transformations to be applied to data. Process networks are directed acyclic graphs where a node represents a process and an edge represents the flow of data from one process to another. Kahn defines a process to be a mapping from one or more input streams to one or more output streams. Concurrent processes communicate only through directed first in - first out streams of tokens with unbounded capacity such that each token is produced exactly once, and consumed exactly once. Production of tokens is non-blocking, while consumption from an empty stream is blocking.

Stevens, et al [8] present a Java implementation for dataflow networks. In this case, a network is defined to be a directed graph, comprising a set of threads connected by a set of unidirectional first in - first out queues. Each thread executes an infinitely executing sequential program. To enable each thread to run in finite memory each edge in the graph represents a bounded queue that nodes can read and write data tokens. A thread blocks if the queue is empty on a get or full on a put. A master thread executes to detect deadlock in the network - solved by increasing the size of the offending queues.

We propose a process network model that is a hybrid of Kahn's process network and the dataflow network presented by Stevens, et al. Our interpretation of a process network is a directed acyclic graph where nodes represent conceivably infinitely executing sequential lightweight processes, and edges represent bounded queues. Our implementation blocks if the queue is empty on a get or full on a put, but does not require a master thread to resolve deadlock as the process network is acyclic. The only time deadlock can occur is when the all queues are empty, which means there is no input into the network.

There are several properties that make process networks desirable as a computational model. The following are basic properties noted in [3]:

1. Each process is a sequential program that consumes tokens from its input queues and produces tokens to its output queues.
2. Each queue has one source and one destination.
3. The network has no global data.

4. Each process is blocked if it tries to read a communication channel with insufficient data. The read proceeds when the channel acquires sufficient data.
5. Writing to a communication channel is generally not blocking, with each queue capable of storing an unlimited amount of data. We utilize a blocking write to enable our process networks to operate in bounded memory.

Given these basic properties, more advanced properties can be derived. These are:

Concurrency. This is safe in process networks because the process network has no global state, and hence concurrent computation of the network nodes will not affect the state of other nodes. More important to a programmer is concurrency is implicit, exploited by overlapping communication with computation and concurrently executing independent processes.

Figure 1 shows a simple UNIX script that takes 2*(*s1*+*s2*+*s3*), where *s1*, *s2* and *s3* are the time taken calculating the three results. Concurrently executing these processes and caching their results, shown diagrammatically in Fig. 2, takes just *s1*+*s2*+*s3*, a parallel speedup of approximately two.

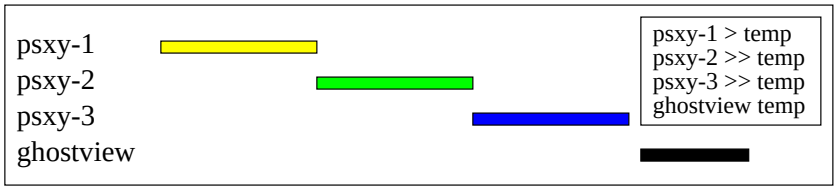

Fig. 1. Computation time of a trivial UNIX script

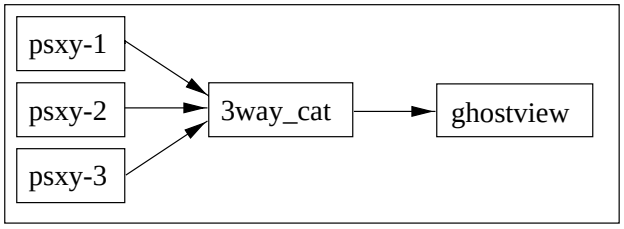

Fig. 2. The pipeline in Fig. 1 as a process network

Scheduling. In an information system such as PAGIS, it is important that scheduling be handled transparently, as domain experts composing a computation should not be required to understand how the components of a computation are scheduled. Process networks satisfy this criterion. All processes in our process networks are started independently; those processes that have no data to consume block. A process is eligible to be scheduled if it wishes to read data from an input port, and data are available on that input. In this way, scheduling is data-driven and dynamic. Scheduling in PAGIS is based on the dynamic ordering of process input and output operations. It may be possible to regard network operators as dataflow nodes, with firing rules that specify the data items required for an individual firing of the node. The scheduler could then operate by scheduling such activations in a data-driven, demand-driven, or hybrid, manner.

Determinacy. A concurrent system requires some guarantee of consistent behavior across implementations. Our process networks are acyclic directed graphs that can be shown to never deadlock. Deadlock due to lack of storage is the only source of deadlock in cyclic networks. Cyclic process networks can be defined (and executed by the PAGIS scheduler), but we have not used them so far in our application. Kahn has shown that some cyclic networks may deadlock in a unique state regardless of scheduling method [6]. In a non-deadlocking process network, method of implementation and order of computation do not affect the result.

Hierarchy. A node of a network can be expressed in any suitable notation provide it satisfies the conditions of Section 2.1. Thus, we can use the process network notation, or a cached result from such a network, itself for a node. Such a process network may be implemented differently from the outer-level PAGIS process network; for example, a node allocated to a multi-computer might implement process networks in a fine-grained manner. We are exploring issues of notational support for hierarchical networks, and of interfacing between different implementation techniques (at node boundaries). Note also that such process networks can (compared to the PAGIS model of process network) more readily exploit computations local to a high-performance computer, because latencies are smaller, and can be masked by finer-grained units of computation.

2.2 Distributing Process Networks

Process networks executed on a single-processor machine can exploit multiprocessing to emulate concurrent execution, but to take full advantage of any parallelism, we may need to execute each of the processes on different processors. We can do this by distributing the processes of the process network such that we have a distributed process network. Distributed process networks can be used to provide an abstraction over potential parallelism, such that the user is implicitly programming a computation that is executed in parallel. The five processes in Fig. 2 can be executed on different machines, with results communicated over a

high-speed network. In this way, each process is executed in parallel, successfully exploiting parallelism available in the problem.

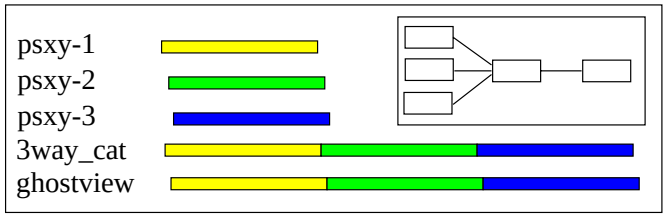

Fig. 3. Computation time of the process network in Fig. 2

2.3 Summary

A GIS requires a simple mechanism for the construction of generic computations by domain experts. Process networks provide us with an elementary model for representing complex computations in a GIS. It is a simple yet powerful semantic model for representing the composition of certain computations and higher-order functions.

Concurrency in process networks is implicit, achieved through concurrent computation of independent processes, and overlapping communication between processes with computation of others. To derive parallelism, we extend the concept of a process network to a distributed process network where processes are executed across distributed computers.

As computations are implicitly concurrent and scheduling is dynamic, user intervention in such matters is not required. These aspects make process networks beneficial for many applications, especially for metacomputing. We now proceed to show that our process networks are also a useful abstraction over the resources of a metacomputing system.

3 Metacomputing

3.1 An Overview

A metacomputer consists of computers ranging from workstations to supercomputers connected by high-bandwidth local area networks and an environment that enables users to take advantage of the massive computation power available through this connectivity. The metacomputing environment gives the user the illusion of a single powerful computer. In this way, complex computations can be programmed without requirement for special hardware or operating systems. There are a number of difficulties that arise when designing and implementing such systems. These issues include:

1. **Transparency.** This is especially important in a heterogeneous system and is generally solved by installing an interpreter on each node or by using a standard cross-platform language.
2. **Synchronization and Scheduling.** These issues define when parts of a larger computation should begin. These should preferably be transparent so user intervention is not required.
3. **Load balancing.** This is the technique or policy that aims to spread tasks among resources in a parallel processing system in order to avoid some resources standing idle while others have tasks queuing for execution.
4. **Security and Robustness.** These issues define policies for what resources are available and how they are accessed by particular users, and how the system should recover when a computation terminates prematurely.

Metacomputers are intended only for computationally intensive applications. Applications that are not computationally intensive can be disadvantaged when processed on a metacomputer, as the cost of communication and synchronization between co-operating processors may outweigh the benefits of concurrent processing.

3.2 Metacomputing and Distributed Process Networks

Without easy-to-use and robust software to simplify the virtual computer environment, the virtual computer will be too complex for most users [7]. Process networks, as a semantic model for composing computation, can provide the simplification required so users can compute efficiently and effectively. The user interface can be made very simple, requiring only a mechanism for graphical construction of a graph such that the user can compose networks of processes and the flow of data between them. In this way, the user can be offered an easy-to-use, seamless computational environment. The process network model can also facilitate high-performance through implicit parallelism, in cases (such as the abovementioned computationally intensive applications) where nodes can mask large communication latencies by overlapping intensive computation.

Distributed process networks provide a simple model for distributing work in a program. The nodes of our process networks can be easily mapped to metacomputer nodes. Edges of the process network represent communication between nodes of the metacomputer.

3.3 Summary

A metacomputer is an abstraction for the transparent grouping of heterogeneous compute resources. The abstraction is provided by a simple user interface that facilitates the effective and efficient use of these compute resources as a single homogeneous system. The distributed process network model allows us to easily distribute work to processing nodes in a metacomputer system. Java is a suitable mechanism for performing the distribution as it assists us to effortlessly communicate between distributed heterogeneous machines.

4 PAGIS: A Prototype GIS

4.1 Background and Overview

A Geographical Information System (GIS) is intended to aid domain experts such as meteorologists and geologists make accurate and quantitative decisions by providing tools for comparing, manipulating, and processing data about geographic areas. Such a system can help interpret what is shown by the image because it relates the image to known geographical areas and features. Images can help a domain experts interpretation and understanding of features of an area, showing aspects of the areas that are not noticeable or measurable at ground level.

Our project involves the development of a prototype GIS for processing geostationary satellite data obtained from the Japan Meteorological Agency's (JMA) [4] GMS-5 satellite. GMS-5 is a geostationary satellite positioned at approximately 140 °E at the equator with an altitude of approximately 35,800 kilometers. GMS-5 provides more than 24 full-hemisphere multi-channel images per day with each image 2291x2291 pixel-resolution. These images require approximately 204MB of storage capacity each day. The Distributed High-Performance Computing project (DHPC) [1] currently maintains a GMS-5 image repository capable of caching recent image data on a 100GB StorageWorks RAID with older data stored automatically by a 1.2TB StorageTek tape silo. We utilize this repository as the source of data for our prototype GIS.

Processing satellite data represents a time-consuming application we use to model real-world problems and test non-trivial computational techniques on distributed high performance computers. In our GIS application, we use the Generic Mapping Tools (GMT) [2] library to perform manipulation of the satellite imagery. This forms the basis of our "computational engine". GMT is a library of approximately 60 utilities for the manipulation and display of geographic data. These utilities enable the domain expert to perform mathematical operations on data sets, project these data sets onto new coordinate systems, and draw the resulting images as postscript or one of a few raster formats. GMT was chosen as out computational engine because of its modular and atomic design. Each utility serves a specific function, but together provide a powerful suite of utilities for a GIS.

We utilize the Java programming language to tie together our model, data and computational engine. The fact that Java is a distributed, interpreted, architecture neutral, portable, multithreaded, and dynamic language makes it highly desirable for the implementation of a metacomputer system such as PAGIS. Communication between distributed machines is possible using Java's RMI (Remote Method Invocation), essentially an evolution of RPC (Remote Procedure Call). RMI uses object serialization to enable Java objects to be transmitted between address spaces. Objects transmitted using the object serialization system are passed by copy to the remote address space [7].

There are drawbacks in using Java. Java is an interpreted language, and as such can not take advantage of architecture-dependent code optimizations. In

addition, code interpretation is performed on a Java Virtual Machine (JVM), which needs to be available for all architectures in the metacomputer. In our case, our Thinking Machine's CM-5 does not have a native JVM that can take advantage of its 128 nodes. This reduces the ability to use certain machines in the metacomputer system optimally. Java is, however, a step in the right direction for distributed systems.

4.2 An Introduction to PAGIS

PAGIS is a simple yet sophisticated proof-of-concept metacomputer system that uses process networks as a semantic model for the composition of complex tasks in a GIS. Users are able to select and view a start image, compose a process network, submit the process network to a server for computation, then display the results. These actions are abstracted by a sophisticated interface that facilitates the composition and submission of a process network to a metacomputer for computation.

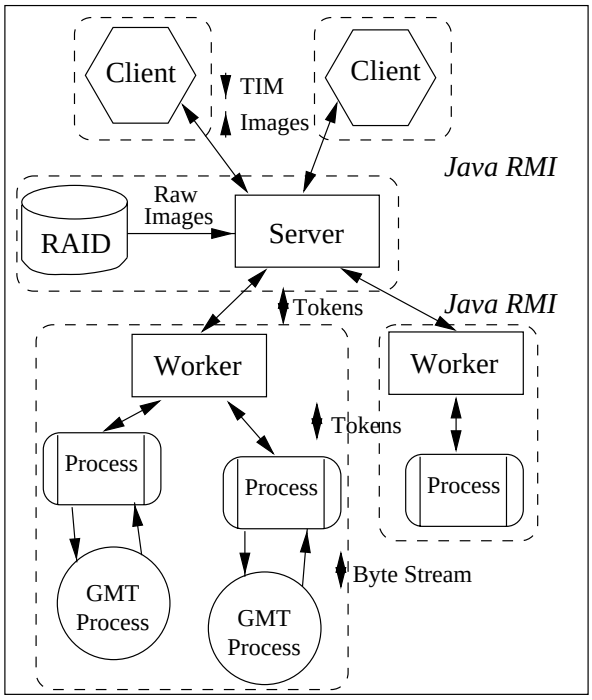

Fig. 4. The PAGIS system architecture, with dashed rectangles used to represent processing at potentially different nodes

The PAGIS metacomputer system represents an end-user-oriented approach to computation, rather than a program-development oriented architecture. In an end-user-oriented system, the user effectively programs applications to be executed on the metacomputer. By utilizing distributed process networks, issues of scheduling and parallelism are abstracted so the process of deciding how computations are to be performed is drastically simplified.

The end-user approach is facilitated by a service-based architecture. The domain expert is able to construct computations consisting of services to be applied to data. One service could be the conversion of satellite data from one format to another, another might be georectification. Services need to be sufficiently coarse-grained that the user is not swamped with a plethora of services, but fine enough that the system be effective.

The PAGIS system comprises three main levels: the Client, Server, and Worker. The correlation between these three levels is shown in Fig. 4, and discussed in detail in the following sub-sections.

4.3 The Server

The server is the central point of communication within the PAGIS system. It can be regarded as the front-end to our metacomputer system. It consists of a collection of Java classes used to coordinate various requests from any number of clients and workers.

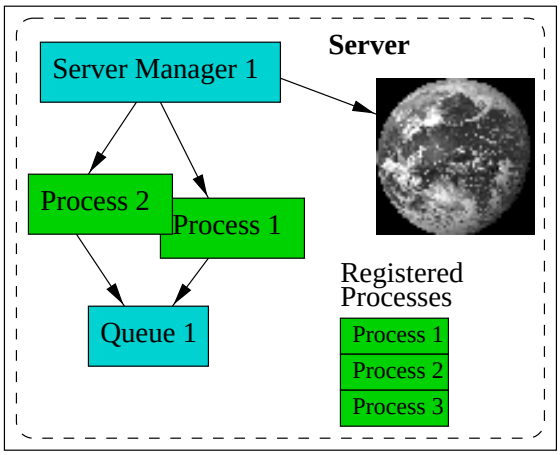

Fig. 5. The PAGIS Server

Each process network has a corresponding server manager object on the server. The server manager is a thread object that contains references to all the processes and edges that make up the process network, as well as references

to the original and result images. This relationship is shown in Fig. 5. This architecture allows the server to be multithreaded such that it is capable of concurrently handling multiple process networks from any number of clients.

When a server manager thread is started, it parses a textual representation of the process network to a collection of *PAGISProcessID* objects representing nodes, and *Queue* objects representing edges. The processes are registered with the server as available for distribution to the workers, and the thread suspends itself.

We derived *get()* and *put()* methods to manipulate *Queue* objects from [3]. The server channels all requests to put and get data from the remote process object to the appropriate *Queue* object on the server. This requires the server to find the appropriate server manager the *PAGISProcessID* object belongs to, and the server manager to find the appropriate *Queue* object to modify. *Queue* objects are currently kept on the server but they could be kept on one of the communicating workers to reduce the impact of network latency. We are currently investigating a number of ways this could be done.

When a process signals its termination, the appropriate server manager thread is resumed. The server manager thread checks for process network completion and suspends itself if there are processes still active. If the process network is complete, the thread performs some clean-up operations, then notifies the server of its completion. Once all the processes have complete, the client is able to download the result image. The results of the process network are cached for later access and reuse.

4.4 The Workers

A worker represents a metacomputer node, sitting transparently behind the front-end. Workers are responsible for polling the server for work and, when available, accepting and executing the given process. Work can be rejected should the processing node be incapable or too busy to perform the work. This is a simple policy used for transparency and load balancing. The workers are multithreaded so they can accept more than one job at a time, ensuring that the server never holds onto processes that may block the entire network, keeping scheduling completely dynamic.

Work supplied by the server takes the form of a *PAGISProcessID* object that is used by the worker to create a *PAGISProcess* thread object. This object, whose thread implementation is shown in Fig. 6, triggers the execution and communication of process network nodes.

The *PAGISProcess* class creates and starts new instances of the *ProcessWriter* and *ProcessReader* threads. These objects are threads so the *ProcessWriter* may continue if the *ProcessReader* blocks on a read. The *ProcessWriter* thread, shown in Fig. 7, reads data from the server and writes that data to the process. The *ProcessReader* thread shares a similar structure except it tries to read results from the process, blocking if none are available, and writing this data to output queues on the server. This is a source of parallelism, where data is sent to the server while processing continues. The node continues consuming

```
public class PAGISProcess extends Thread {
  Server s; PAGISProcessID pid;
  public Service(Server s,PAGISProcessID pid) {
    this.s = s; this.pid = pid;
  }
  void run() {
    Process p = start_process();
    (new ProcessWriter(p.getOutputStream(),s,pid)).start();
    (new ProcessReader(p.getInputStream(),s,pid)).start();
    p.waitFor();
  }
}
```

Fig. 6. The PAGISProcess class

```
public class ProcessWriter extends Thread {
  OutputStream w; Server s; PAGISProcessID pid;
  ProcessWriter(OutputStream w, Server s, PAGISProcessID pid) {
    this.w = w; this.s = s; this.pid = pid;
  }
  void run() {
    Packet p;
    while ((p = s.server_consume(pid)) != null)
      process_write(w,p);
    w.close();
  }
}
```

Fig. 7. A thread for reading data from a server and then writing to a process

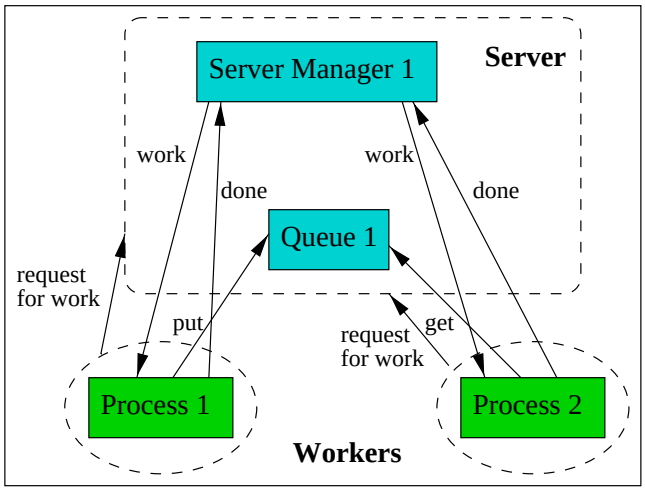

Fig. 8. Processes communicating with the server

and producing data until the process is complete. When the node has completed processing, the worker signals to the server that the process is done. This ensures the server knows the progress of the network.

Figure 8 presents an example of two workers communicating with the server through their server manager. The workers request *PAGISProcessID* objects from the server and obtain *Process1* or *Process2*. The worker then creates new *PAGISProcess* objects and their threads are started. The two processes communicate through a shared *Queue* on the server. The software currently does not take advantage of locality, so two communicating processes on the same node still communicate through the server.

Once the processes have completed, they notify the server manager of their completion, and the threads terminate. From this example, scheduling can be seen to be dynamic, dependent only on the availability of data, and pipelined parallelism is implicit, requiring no intervention by user or server to obtain.

4.5 The Client

The client is at the user end of the metacomputer system. The client consists of two pieces of software: the user interface and communication system. Our user interface design is influenced by a system proposed by Roger Davis, an interpreter capable of constructing arbitrarily complex process network topologies [10]. This interpreter is known as the Icon Processor or *ikp*. *ikp* allows the user to graphically build process networks. Users can also coordinate processes to run on remote hosts, allowing the user to distribute the processes across a number of machines. Each remote machine must be running an *ikp* daemon, in order to perform user authentication procedures similar to rlogin and rsh. The user must be aware of which remote hosts are running this daemon in order to distribute processes.

Distribution in a metacomputer needs to be transparent. *ikp* is deficient in this aspect. Our server implementation described earlier provides transparency by making the remote hosts apply for work and decide if work can be performed. The graphical nature of *ikp* did give us some ideas on the requirements for a process network interface, and we used this as the basis of our evaluation of different user interface systems.

The user interface system we use is Tycho - an extensible application that allows developers to inherit or compose essential functionality. Tycho was built for the Ptolemy project at the University of California at Berkeley [3]. It was developed using ITCL/ITK, an object-oriented version of TCL/TK with C and Java integration support. Tycho is a collection of classes that represent various graphical widgets such as slates, buttons, lines (or textboxes), and menus. These objects can be extended to provide extra functionality. User interfaces are classes just as widgets are classes. This means we can take an editor application class, inherit its properties and methods, and extend it to include extra functionality.

The *EditDAG* class provided with Tycho is a user interface that allows the user to construct directed acyclic graphs. These graphs are constructed by creating a start node, selecting that node, then creating a connecting node. A *DAG*

class handles cycle checking on the fly so users can never build an illegal graph. The same class allows us to access and manipulate a Tycho Information Model (TIM) that textually represents the graph designed by the user. This TIM is used as input to our metacomputer.

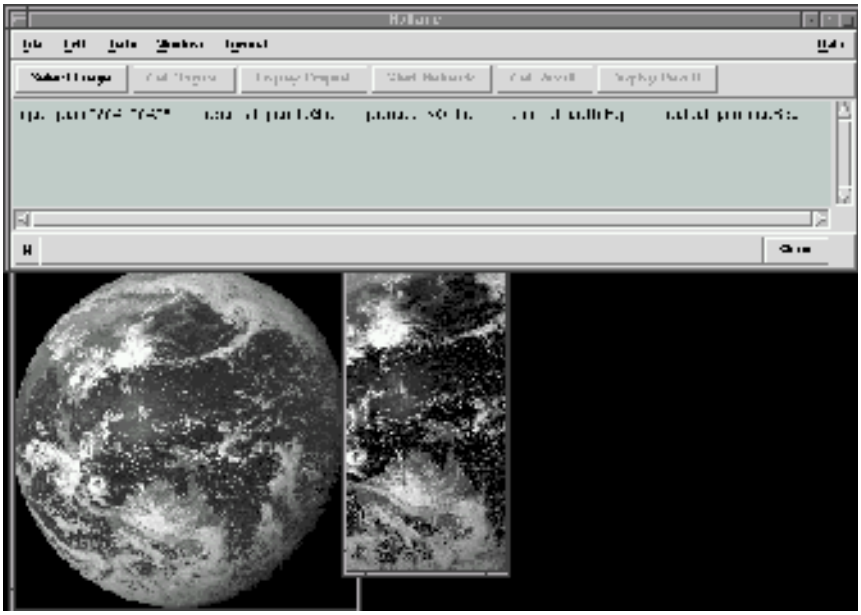

Fig. 9. The PAGIS graphical interface showing georectification of GMS-5 imagery

Our extension of the *EditDAG* interface involved adding a toolbar of standard commands and a menu of standard services. The user uses the menu to compose a network of services, and the toolbar to select and view original images, submit the TIM to the server, and select and view result images. The PAGIS user interface is demonstrated in Fig. 9.

As the server is written in Java, we need to utilize the Tycho-Java interface so that Tycho callbacks correspond to communications with the server. We have a *UserSide* class that performs method invocations on the server and some rudimentary graphical operations. (This class is also used by a simple Java AWT interface we developed that enables us to test the user-side software independent of the Tycho interface.) The *TclPAGIS* class implements the *TychoJava* interface. Calls to this class result in corresponding calls to the *UserSide* class. This separates any Tycho-related code from our communications code. Figure 10 describes the relationship between these entities.

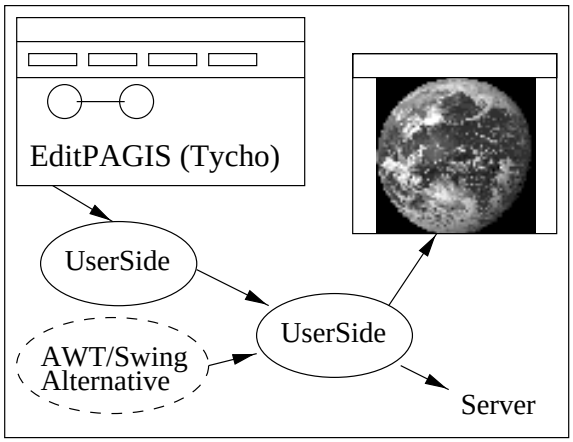

Fig. 10. PAGIS Client architecture

4.6 Communication Mechanisms

Communication is achieved with the use of Java RMI. The server implements two Java interfaces: one for client-server communication, another for worker-server communication. These interfaces act as an API for communication between the various entities. The interfaces provide access to relevant functions of the appropriate server manager. The client-server interface allows the client to create a new server manager instance and select an image to be retrieved from the DHPC image repository. The client interface also allows the client to submit a textual representation of the process network, start the process network, request the status of the network, and retrieve the result image. The worker-server interface provides workers with the ability to apply for and retrieve work, get and put data to the queues, and signal the completion of work.

4.7 Summary

The use of process networks has given us a simple solution to several of the metacomputing issues raised such as transparency, scheduling, and ease of use. Process networks offer implicit parallelism, dynamic scheduling, determinacy, and trivial work distribution. We are able to offer a high level of transparency to the user as the model implicitly manages issues such as parallelism, scheduling, determinacy, and distribution.

The PAGIS server software controls communication between any number of clients and metacomputer nodes. Each metacomputer node requests *PAGISProcessID* objects from the server and, if accepted, creates a *PAGISProcess* object that executes the desired process. The process consumes tokens from and produces tokens to *Queue* objects on the server. The *PAGISProcess* objects represent process network nodes and the *Queue* objects represent edges. The client

enables domain experts to graphically compose computations, without needing to address issues such as parallelism, scheduling and distribution.

We are considering issues of load balancing. Process-level load balancing can be implemented by enabling workers to reject jobs based on their inability to perform them. Fine-grained load balancing is not appropriate within our coarse-grained computational engine, but a finer-grained model will be possible within an implementation of hierarchical networks as mentioned in Section 2.1. A simple implementation for hierarchy may simply pre-process the TIM and substitute (non-recursive) sub-networks with their TIM representation. An interesting issue is that of caching sub-network results. Tycho lacks support for building hierarchical networks so we are also currently examining other user interfaces, such as Ptolemy, that are capable of this. We hope that PtolemyII's version of Tycho will provide direct support[9]. We are also experimenting with Swing and 'Drag and Drop' to implement our own client as part of the DISCWorld infrastructure. Robustness and security have also not been implemented.

The nodes that constitute our process networks are selected from a configuration file. This list can be altered so any generic process can be executed; thus, the system is not limited to a GIS.

5 Discussion

A performance analysis of PAGIS is in progress. Our main performance concern currently is Java process IO. The Java implementations we have used for Sun Sparc and Dec Alpha appear to buffer output from processes by default. This means that calls to read data from a process block until the data is fully buffered, meanwhile the opportunity for latency hiding is lost. The run time of a process network has hence become $T+l*t$ where T is the run-time of a process network on a single machine, l is average message latency, and t is the number of tokens transmitted during execution of the process network. This is obviously sub-optimal, and is our current focus of attention.

The use of RMI from a prototyping view-point was highly successful. RMI has enabled us to easily manipulate the server configuration and communication protocol. We hope to implement a version in the near future that transmits raw tokens over sockets to isolate the effect of using RMI. In the mean time, an obvious communications improvement, as highlighted in Section 4.3, will be removing our centralized queue system. Communications between two nodes currently requires two remote method invocations, one *put* and one *get* from the shared *Queue*. Remote method invocations can be halved just by moving the queue to the sending or receiving node.

It is worth noting that the architecture used is not the only possibility. PAGIS shows how a simple semantic model can be used to build metacomputing applications. The deterministic nature of our process network implementation means we can safely use other architectures preserving process network semantics to get the same results, and likely more efficiently. An example of this is the future use of PAGIS in the DISCWorld infrastructure. DISCWorld itself is a meta-

computing system that executes services available from other nodes or resource administrators that have chosen to provide and advertise these services. DISCWorld provides a common integration environment for clients to access these services and developers to make them available [12]. Our hope is that PAGIS may be integrated with DISCWorld to utilise this integration environment as well as DISCWorld's distribution and scheduling mechanism.

We are also investigating the possible use of a Java package for concurrent and distributed programming developed by [11] called Java//. Java// abstracts over the Java thread model and RMI to provide an API for the implementation of concurrent and distributed systems. This system natually supports latency hiding because of its asynchronous will provide us with a transparent notion of active object, abstracting the distinction between local and remote nodes. Also, Java//'s asynchronous model naturally supports latency hiding.

Acknowledgements

This work was carried out under the Distributed High-Performance Computing Infrastructure (DHPC-I) project of the On Line Data Archives (OLDA) program of the Advanced Computational Systems (ACSys) Cooperative Research Centre (CRC), funded by the Research Data Networks CRC. ACSys and RDN are funded under the Australian Commonwealth Government's CRC Program.

References

1. Distributed High Performance Computing Project, University of Adelaide. See: `http://www.dhpc.adelaide.edu.au`
2. "Free Software Helps Map and Display Data", Paul Wessel and Walter H. F. Smith, EOS Trans. AGU, 72, p441, 445-446, 1991.
3. "The Tycho User Interface System", Christopher Hylands, Edward A. Lee, and H. John Reeckie, University of California at Berkeley, Berkeley CA 94720.
4. Japanese Meteorological Agency. See: `http://www.jma.gov.jp`
5. JavaSoft Java Studio. See: `http://www.javasoft.com/studio`
6. "The Semantics of a Simple Language for Parallel Programming", G. Kahn, Proceedings of IFIP Congress 74.
7. "RMI - Remote Method Invocation API Documentation", Sun Microsystems Inc., 1996.
8. "Implementation of Process Networks in Java", Richard S. Stevens, Marlene Wan, Peggy Laramie, Thomas M. Parks, and Edward A. Lee, 10 July 1997.
9. Ptolemy II. See: `http://ptolemy.eecs.berkeley.edu/ptolemyII`
10. "Iconic Interface Processing in a Scientific Environment", Roger Davis, Sun Expert, 1, p80-86, June 1990.
11. "A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming", Denis Caromel and Julien Vayssière, INRIA Sophia Antipolis. See: `http://www.inria.fr/sloop/javall`
12. "DISCWorld: An Environment for Service-Based Metacomputing", Hawick, James, Silis, Grove, Kerry, Mathew, Coddington, Patten, Hercus, and Vaughan. Future Generation Computer Systems Journal, Special Issue on Metacomputing, to appear. See: `http://www.dhpc.adelaide.edu.au/reports/042`