

The PAGIS Grid Application Environment

Darren Webb and Andrew L. Wendelborn

Department of Computer Science
University of Adelaide
South Australia 5005, Australia
{darren, andrew}@cs.adelaide.edu.au

Abstract. Although current programming models provide adequate performance, many prove inadequate to support the effective development of efficient Grid applications. Many of the hard issues, such as the dynamic nature of the Grid environment, are left to the programmer. We are developing a programming model that incorporates a familiar, formal computational model and a reflective interface. The programming model, called PAGIS, provides a desirable abstract computer with an interface to introduce and customize Grid functionality. Using PAGIS, an application programmer constructs applications that are implicitly parallel and distributed transparently. This paper describes the basic components of the PAGIS framework for constructing and executing applications, and the reflective techniques to customize applications for computation on the Grid.

1 Introduction

The Grid is used in numerous scientific disciplines to solve large, complex problems. The Grid provides pervasive access to a geographically-dispersed, large scale execution environment. It integrates heterogeneous resources with dynamic availability and capability connected by an unreliable network. Computational scientists and engineers use the Grid to distribute computation to hosts offering various computation and data services.

Grid-enabled programming systems enable familiar programming models to be used in Grid environments[1]. A programming model defines an abstract machine, user libraries and software tools a programmer uses to interact with a system. Grid programming models are generally adapted from established parallel programming models designed for homogeneous, tightly coupled supercomputers. However, there is growing consensus that these programming models are inadequate to support the effective development of efficient Grid applications[2]. Models, such as MPICH-G2[3], are typically low-level, restricted in their applicability, and lack many essential properties and capabilities required for Grid applications.

We believe that current Grid programming models are too complex for many potential Grid programmers. While providing high-performance, the responsibility for managing the hard issues, such as adapting to the dynamic nature of

the Grid environment, fall to the programmer. As a result, most programmers will lack the skills necessary to exploit the full power of the Grid.

Many projects[4–9] are developing high-level component-based programming models suitable for the Grid. In these models, the programmer’s focus is on *what* program components are executed. Our approach is different, focusing on the techniques used to describe *how* these program components are executed.

PAGIS is a middleware that enables scientists to tap into the Grid with little or no Grid programming skills. PAGIS provides two interfaces that emphasise a separation of what an application does from how it does it. The *application programming interface* uses a formal yet simple, abstract computational model to describe what is executed. The *reflective interface* enables the introduction of new behaviour to the application, such as its execution in a Grid environment and its dynamic control.

In this paper, we present the PAGIS framework. We describe its basic components for building and executing applications, and discuss the reflective techniques used to customize applications for computation on the Grid.

2 A Grid Programming Model

Our work emerges from the PAGIS project[10], an architecture for Grid programming. For Grid computing to become widely accessible, its users must be able to communicate with it on their own terms. To this end, PAGIS is middleware that uses composition of processes (a process network[11]) to specify the execution and remote processing of a set of computational tasks. The architecture is generic, although our prototype implementations focus on the domain of satellite image processing for Geographical Information Systems (GIS).

The PAGIS middleware enables a user to specify the set of tasks as a process network. The middleware supports higher-level tools including a domain-based visual programming environment for the visualization of applications in design and execution. Fig. 1 shows a process network to access a data set, crop, and visualize the result. Other operations might include to geo-rectify and scale the data.

Pragmatically speaking, the process network model is compositional, and provides an intuitive notation for building Grid applications. Our work uses the semantic foundation of the process network model to explore the theme of safe adaptation via reconfiguration, which we believe to be of great importance in a flexible web services architecture. Our compositional framework lends itself also to supporting web services[12], an aspect under investigation.

2.1 PAGIS Middleware

The PAGIS middleware consists of a Java-based computational engine that schedules and manages the process network. The PAGIS middleware is an abstract machine with a language that defines the interface for communicating

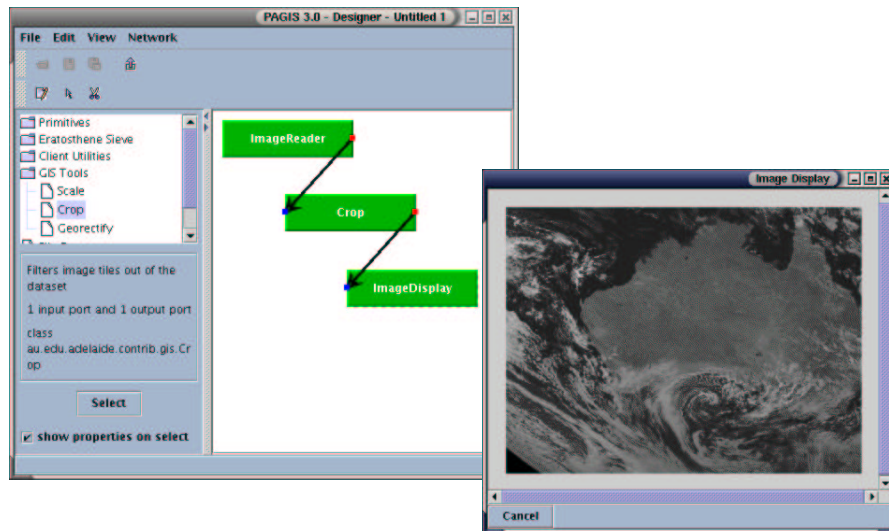


Fig. 1. The PAGIS Visual Programming Environment.

with the machine. The programmer uses the language to instruct the machine to perform certain tasks.

The computational model used in PAGIS is Kahn's Process Networks[11]. Process networks is an intuitive computational model for parallel programming. The model describes a collection of autonomous computational resources connected in a network of unidirectional communication links. A given computational resource computes on data coming along its input communication links, using memory of its own, to produce output on some or all its output links.

The structure of a process network is a directed graph where a node represents a process and an edge represents a channel able to carry data of a given type. A process is a sequential program representing successive passes each of which incrementally transforms a stream of data. Processes communicate through first-in-first-out streams of tokens 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.

The model has a number of desirable characteristics. A process network program is deterministic. Computation depends only on program state. Program structure determines the schedule, relieving the programmer of the burden of scheduling and guaranteeing consistent behaviour across implementations[13]. In addition, the process network model allows pipelined parallel computation. Future input concerns only future output, hence a process may produce output before all its input is available.

The Process Network Application Programming Interface (PNAPI) defines a graph-based syntax for describing the structure and behaviour of a process

network. The API provides abstractions to describe both network construction and execution, including *Network*, *Process*, *Port* and *Channel* abstractions to describe the structure of a process network, and *Builder* and *Framework* to describe building and reconfiguring a network.

2.2 PAGIS and Grid Programming

PAGIS is a lightweight middleware that enables programmers to construct, run and reconfigure process networks. But how do we best approach the execution of process network applications in a Grid environment? One alternative is to change the implementation of the computational engine. The implementation will undoubtedly benefit some Grid application domains, but will very likely adversely affect others. A new implementation for each application domain will diversify the code base, complicating portability and reusability of the system. A second alternative is to leave Grid concerns to the programmer. The programmer contorts application code to use Grid libraries directly. In so doing, the programmer mixes what the application does (its functional concerns) with how it does it (its non-functional concerns), thereby increasing complexity, and reducing portability and reusability.

In investigating a way to alleviate such problems, we chose to explore the application of reflective techniques. We will see that such techniques allow programmers to separate out, and focus on, the characteristics of particular grid behaviours one at a time, and define them as customizations of a well-understood base model. The reflective technique we have used (an example of metalevel programming) allows us to expose selected aspects of the programming model the application programmer is likely to need to introduce such behaviour. In the next section, we describe the technique of metalevel programming, the metalevel architecture introduced to our programming model, and how the metalevel is used to introduce Grid behaviours.

3 Customizing PAGIS Applications

In this section, we report our experience in metalevel design and reflective programming for Grid applications. The semantics of the process network model are defined formally, and execution can be performed by any computational engine conformant with those semantics. Hence, it is desirable to express any given computational engine as a customization of a generic computational engine. In this section, we discuss reflective infrastructure for customizing how a PAGIS application executes. We introduce metalevel architectures, and describe a metalevel architecture designed for the development of Grid applications.

3.1 Metalevel Programming

One view of metalevel architectures, shown in Fig. 2, is to treat the system as a black box, with separate interfaces that address the functional and non-functional concerns of an application. The *baselevel* interface, implemented by

the system, describes the functional behaviour of a program, and comprises “ordinary” objects and classes called baseobjects and baseclasses. The *metalevel* interface exposes particular aspects of the underlying system to customization. The metalevel consists of objects that *reify* elements of the system, and carry out *reflective computation* on them.

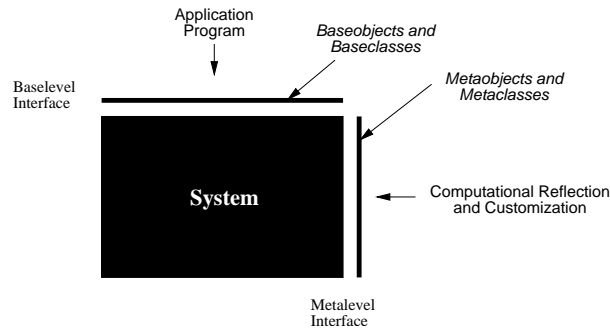


Fig. 2. A system with baselevel and metalevel interfaces.

Reflection is the ability of a system to observe and change its own execution[14]. It is described by Maes [15] as being “the activity performed by a computational system when doing computation about its own computation”. The goal of reflection is to allow programs to reason about their own execution state and alter it to change its own meaning. Reification is essentially the process of converting some component of system state into a representation that may be computed upon[16], using *metaobjects*[15]. Modifications to the metaobject result in actual modifications to the behaviour of the system, through the property of causal connectivity[15].

There are two forms of reification: *structural reification* and *behavioural reification*. Structural reification is the process of converting some element of a system into a metaobject, and might include reification of an object, class or method. Behavioural reification is the process of converting computation or behaviour into a metaobject, such as a method invocation. These metaobjects constitute an interface for introspection and customization called a *Metaobject Protocol*.

In this paper, we describe our work toward the development of a framework for Grid application development that incorporates metalevel programming techniques. We present a metalevel architecture influenced by, and the result of the analysis of a number of other metaobject protocols including ProActive[17], FRIENDS[18] and Coda[19] (further details are available in [20]). We show how this metalevel architecture, called *Enigma*, is suitable for Grid programming and proceed to show how it benefits the development of component-based Grid applications.

3.2 Enigma Metalevel

Enigma is a metalevel architecture that decomposes the method invocation process into several operational phases, and allows the composition of new meta-behaviour at each phase. The metalevel reifies various structures (e.g. objects and classes) and behaviours (method invocations or messages), and exposes these reifications to customization. The metalevel decomposes method invocation into three orthogonal phases. The *marshaling* phase prepares a message to be sent from a base object. The *transmission* phase coordinates how a base object receives a message. Finally, the *execution* phase invokes the message upon the target baseobject.

We have developed a metalevel architecture called Enigma that decomposes the method invocation process into several operational phases, and allows the composition of new metabehaviour at each phase. The metalevel reifies various structures (e.g. objects and classes) and behaviours (method invocations or messages), and exposes these reifications to customization. The metalevel decomposes method invocation into three orthogonal phases. The *marshaling* phase prepares a message to be sent from a base object. The *transmission* phase coordinates how a base object receives a message. Finally, the *execution* phase invokes the message upon the target baseobject.

Figure 3 shows the course of a method invocation through the Enigma metalevel. The metaobject `MetaInstance:a` intercedes messages sent from baseobject `Object:a`. The metaobject marshals the message and directs it to its target, in this case `Object:b`. The message is transmitted to `MetaInstance:b` which executes the method invocation upon the baseobject `Object:b`.

Metalevel programmers introduce new behaviour by customizing one of these phases. New metabehaviour is introduced by adding a handler to a phase. Many handlers can be added to each phase and applied as one combined customization.

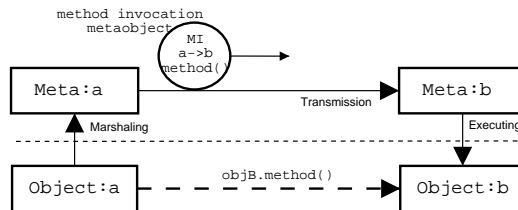


Fig. 3. A model reifying the three phases of method invocation.

To show how to design a new metabehaviour, we now briefly describe a tracing metabehaviour. A tracing behaviour, useful for debugging, is readily defined (similar to [15], for example) in terms of the execution handler: the metalevel program can simply extract and print the method name, invoke the

method code, then extract the results and print those. We now discuss how we apply Enigma to process networks applications.

3.3 Enigma applied to Process Networks

One critical design decision in applying Enigma to process networks is: what is the best way to represent, from the point of view of reflective programming in Enigma, the various aspects of a process network? To make a poor decision leads to a cumbersome and complex meta-program; a separate reification the individual process network components turned out to be the wrong approach.

Foote[21] suggests a key requirement of selecting a set of metaobjects be to mirror the actual structure of the language model. Kahn[11] describes the process network model as a network of autonomous computational resources connected by unidirectional links. Rather than reifying each process network element individually, we instead mirror this structure of computational resources, introducing a new metaobject defined as the composition of these baselevel elements. We call this metaobject a **Computation**.

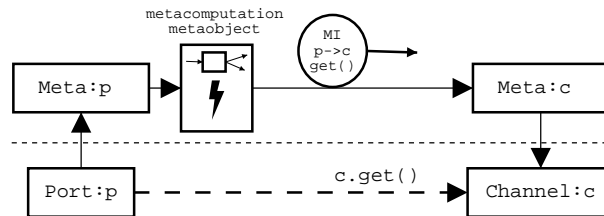


Fig. 4. The **Computation** metaobject reifies a computation's components.

The **Computation** is introduced into the metalevel to compose the process, port and channel elements into a single structure. We use the Enigma metalevel to insert the **Computation** in the transmission phase of relevant metaobjects. The **Computation** metaobject then intercedes all method invocations for the metaobjects of **Computation** elements. Consider Fig. 4. *MI* is the reification at the metalevel of the baselevel method call `c.get()`. Figure 4 illustrates the route of a method invocation *MI* from a port object *Port:p* to a channel object *Channel:c*. This forms the basis of customization.

The advantage of our approach is that we can customize all elements of the **Computation** together as a whole, rather than the more cumbersome approach of applying individual customizations one at a time. By customizing the **Computation** metaobject, which in itself represents such baselevel aspects as process code, ports, channels and threads, we influence baselevel behaviour. The mechanism that we use to customize the **Computation** metaobject is another metalevel (a meta-metalevel) which treats the **Computation** metaobject itself as

a baseobject and makes it customizable by reapplying the view of Fig. 3. The `Computation` serves a functional purpose (the composition of metaobjects) at the first metalevel, and a second metalevel helps us to separate new behaviour from the `Computation`. Figure 5 illustrates the introduction of this second metalevel.

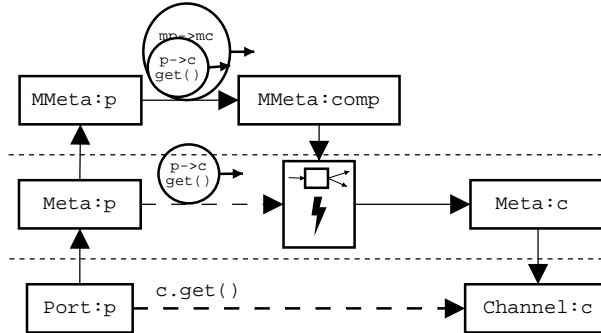


Fig. 5. A meta-metalevel for customizing all `Computation` components.

Finally, we have considered a simplified view of the metalevel, where the metalevel exposes only `Computation` and the interaction between them. The advantage of this approach is simplicity – the metalevel consists of a minimal set of metaobjects, which is less complex and easier to understand, and consistent with Kiczales’ minimalistic view of metaobject protocol[22]. The cost of this simplified approach is flexibility – the simplified view does not allow fine-grain customization of individual baselevel elements. We are still investigating which approach is best for defining Grid metabehaviours.

4 Grid-Related Metabehaviours

In this section, we discuss the implementation of several metabehaviours to deploy a PAGIS application on the Grid. We show how each metabehaviour type can be used to introduce new behaviour relevant to Grid programming. For example, consider the deployment of a satellite image processing application to the Grid. In this example, we may wish to introduce behaviours to migrate computations to Grid hosts, compress the communication of image data between these hosts, and monitor performance to induce adaptation. We now discuss the implementation of these behaviours.

- **Compression.** The compression metabehaviour intercedes method invocations at the marshaling phase of the `Computation` and compresses the reified method invocation. The metabehaviour first serializes the reification to a byte stream, then GZIP compresses the byte stream, and finally encapsulates the compressed data within a new method invocation object. The new

method invocation object decompresses the original method invocation as required, such as when the method is executed. A similar approach could be followed for data encryption.

- **Migration.** The migration metabehaviour affects the transmission phase of the **Computation** to allow the remote deployment of a **Computation**. The metabehaviour enables a third-party (a user or software “agent”) to initiate the migration of the **Computation** and its elements. The metabehaviour implements a migration protocol that delegates method invocations to another (possibly remote) metaobject.
- **Performance Monitoring and Adaptation.** We now consider an execution phase metabehaviour that detects and forecasts large-scale changes in system behaviour to drive computation migration. A “performance monitoring” behaviour, based on the tracing behaviour described earlier, records information about method invocations (its destination, timing, frequency and so on) into a suitable repository. An agent may use this information as the basis for an adaptation event, and trigger a computation to migrate. At the CCGrid 2002 GridDemo Workshop, we demonstrated an Application Builder interface through which a user can generate these adaptation events. The interface enables introspection of implementation attributes such as physical location, searching Grid Information Services for potential hosts, and triggering adaptation events. At this time, the generation of events is invoked through user action, and is a very manual process. In the near future, a software agent using the performance monitoring behaviour and Grid Information Services will initiate this automatically.

5 Conclusions

In light of the low-level nature of current Grid programming models, we have developed a high-level programming model with an interface to a simple, intuitive computational model and an interface to customize aspects its implementation. Programmers use the application interface to describe the functional requirements of their application. Programmers optionally use the metalevel interface describe non-functional requirements, such as the application’s execution on the Grid. The model forms a unifying framework for describing a computational engine independent of its deployment.

We have developed a metalevel with an operational decomposition and form of metabehaviour composition that we believe is useful for deploying existing systems to the Grid, for building new Grid applications, and especially for supporting adaptive software. We intend to continue to build better abstractions for Grid metabehaviour composition, and improve our understanding of how to use them.

References

1. Foster, I.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of Supercomputer Applications (2001)

2. Lee, C., Matsuoka, S., Talia, D., Sussman, A., Mueller, M., Allen, G., Saltz, J.: A grid programming primer. Technical report, Global Grid Forum Programming Models Working Group (2001)
3. Foster, I., Geisler, J., Gropp, W., Karonis, N., Lusk, E., Thiruvathukal, G., Tuecke, S.: Wide-Area Implementation of the Message Passing Interface. *Parallel Computing* **24** (1998)
4. Bhatia, D., Burzevski, V., Camuseva, M., Fox, G., Furmanski, W., Premchandran, G.: WebFlow - a visual programming paradigm for Web/Java based coarse grain distributed computing. *Concurrency: Practice and Experience* **9** (1997) 555–577
5. Shah, A.: Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources. PhD thesis, Virginia Tech (1998)
6. Neary, M., Phipps, A., Richman, S., Cappello, P.: Javelin 2.0: Java-based parallel computing on the Internet. In: Proc. of Euro-Par 2000, Munich, Germany (2000)
7. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B.: Toward a Common Component Architecture for High-Performance Computing. In: Proc. of HPDC8. (1999)
8. Krishnan, S., Bramley, R., Gannon, D., Govindaraju, M., Alameda, J., Alkire, R., Drews, T., Webb, E.: The XCAT Science Portal. In: Proc. of SC2001. (2001)
9. Welch, P., Aldous, J., Foster, J.: CSP networking for Java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, A.G.Hoekstra, eds.: *Computational Science - ICCS 2002*. Volume 2330 of LNCS., Springer-Verlag (2002) 695–708
10. Webb, D., Wendelborn, A., Maciunas, K.: Process Networks as a High-Level Notation for Metacomputing. In: Proc. of 13th Intl. Parallel Processing Sym. Workshops: Java for Distributed Computing, Springer-Verlang (1999) 718–732
11. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: Proc. of IFIP Congress 74, North Holland Publishing Company (1974) 471–475
12. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The physiology of the grid. Technical Report Draft 2.9, Globus (2002)
13. Stevens, R., Wan, M., Lamarie, P., Parks, T., Lee, E.: Implementation of Process Networks in Java. Technical report, University of California Berkeley (1997)
14. Smith, B.: Reflection and Semantics in a Procedural Language. Technical report, MIT (1982)
15. Maes, P.: Concepts and Experiments in Computational Reflection. In: Proc. of OOPSLA 87. (1987)
16. Sobel, J.: An introduction to Reflection-Oriented Programming. In: Proc. of Reflection'96. (1996)
17. Caromel, D., Klauser, W., Vayssiere, J.: Towards seamless computing and meta-computing in Java. *Concurrency: Practice and Experience* **10** (1998) 1043–1061
18. Fabre, J.C., Pèrennou, T.: FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications. In: Proc. 2nd European Dependable Computing Conf. (EDCC-2), Taormina, Italy (1996)
19. McAffer, J.: Meta-level Programming with CodA. In: Proc. of ECOOP85. Volume 952 of LNCS. (1985)
20. Webb, D., Wendelborn, A.: The pagis grid application environment. Technical Report DHPC TR-139, Adelaide University (2003)
21. Foote, B.: Object-oriented reflective metalevel architectures: Pyrite or panacea? In: Proc. ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures. (1990)
22. Kiczales, G.: Towards a New Model of Abstraction in the Engineering of Software. In: Proc. of Intl. Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture. (1992)