1998

Seamless Metacomputing with Bond

Ladislau Bölöni
Kyung Koo Jun
Mihai Sirbu
Dan C. Marinescu

Report Number:
98-010
SEAMLESS METACOMPUTING WITH BOND

Ladislau Boloni
Kyung Koo Jun
Mihai Sirbu
Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #98-010
April 1998
Seamless Metacomputing with Bond

Ladislau Bölöni, Kyung Koo Jun, Mihai Sirbu, and Dan C. Marinescu
(Email: boloni, junkk, sirbu, dcm@cs.purdue.edu)
Computer Sciences Department
Purdue University
West Lafayette, In 47907, USA

Abstract
We introduce a metacomponent based architecture for network computing. Bond is an infrastructure for building domains associated with groups with common interests. In this paper we present the basic elements of the Bond architecture then we discuss the execution model, security and scheduling on a computer grid with autonomous nodes.

Contents:
1. Introduction
2. Bond architecture
3. The execution model and security
4. Bond contracts and models for scheduling on a computer grid
5. Conclusions

1. Introduction
Bond is a metacomponent architecture for network computing on a grid consisting of autonomous nodes. The Bond space is populated with metaobjects consisting of information and network object pairs. The basic philosophy of the system is to accommodate the inherent heterogeneity of the computing hardware and the diversity of computer software by emulating the lock and key mechanisms used by proteins to recognize each other. Reflection mechanisms allow objects to discover each other’s properties and core agents and services provide scheduling, user-level resource management, security, fault-tolerance and other functions we have been accustomed to in existing systems [17].

During the past few years several projects in the areas of metacomputing have achieved a level of recognition. The list includes the Legion, Globus, FRIENDS and Infospheres projects surveyed below. They all share the vision of seamless metacomputing and propose to achieve high performance via parallelism using loosely coupled distributed systems [7], [8], [10], [12], [13], [14], [16]. The Legion project started in 1995 at the University of Virginia envisions a system capable to schedule transparently applications on processors, manage data transfer and coercion and provide communication and synchronization [14]. Its design objectives are: site autonomy, an extensible core, a scalable architecture, a seamless computing environment, high performance via parallelism, a single global name space, security for users and resource providers, management and exploitation of resource heterogeneity, multi-language support and inter-operability, and fault
tolerance, [16]. The actual implementation of the Legion system was expected to start in late
1997.

The Globus project is "a multi-institutional research effort that seeks to enable the construction of
computational grids providing pervasive, dependable, and consistent access to high performance
computational resources, despite geographical distribution of both resources and users" [13]. In
Globus a low level toolkit provides communication, authentication, and data access. The goal is to
build an Wide Area Resource Environment, AWARE, an integrated set of high level services that
enable applications to adapt to heterogeneous, dynamic changing environments [12]. Components
of the Globus project have been demonstrated as early as 1997 and some of them are used by dif­
ferent research groups.

The Friends system is a metalevel architecture providing libraries of metaobjects for fault-toler­
ance, secure communication and group-based distributed applications [10]. Metaobjects can be
used transparently and be composed according to the needs of an application. The system
attempts to support compositability of mechanisms dealing with different fault classes as needed by
an application. Metaobject protocols give the user the ability to adjust the language implementa­
tion to suit their particular needs. Reflections expose the language implementation at a higher
level of abstraction [23].

The Infospheres project at Caltech [6], [7] has the goal to develop network centric technologies
and design information infrastructures that support virtual organizations. The Infospheres proto­
type has been distributed to a number of organizations. An early version of Bond 2.0 uses the
mailboxes and the transport mechanisms provided by the Infospheres.

The Bond project was triggered by a collaboration with structural biologists who provided the
problems, the motivation, and need to learn some basic facts about the structure of biological
macromolecules. The complex procedures needed for data acquisition, data analysis and model
building for x-ray crystallography and electron microscopy are discussed elsewhere [10], [18].
Here we only note that processing of structural biology data involves large groups and facilities
scattered around the world, complex programs that are changed frequently. Most computations
are data intensive, they require the use of parallel and distributed systems.

Our first step to automate the complex data collection and analysis process in computational biol­
ogy was to define a scripting language, SBL, the Structural Biology Language, suitable for exe­
cuting iterative computations on a parallel computer [9]. SBL supports mechanisms to determine
if convergence criteria have been met and if so execute a different sequence of programs, or carry
out more iterations involving the same group of parallel programs. SBL includes a checkpointing
facility as well as a log. This approach has obvious limitations that became evident once we
attempted to execute parallel programs using the MPI communication library, on a cluster of
workstations. Another problems we faced, were the constant need to modify the programs, and
the need to ensure compatibility between different program versions and data. Different members
of the group used different versions of programs and merging partial results was a difficult task.
After weeks or even months of calculations, it was often impossible to determine the "genetic"
information, to establish what parameters of the model were used in computations carried out in
the past, unless one kept a very detailed diary of each step.
The next step was to define descriptors of all objects, programs and data and to design a remote execution shell [21]. The Bond shell uses these descriptors to support user level management of resources distributed over a computing grid and to ensure the compatibility between different program and data formats. Each user has a database of program, data, and hardware descriptors, used to start the execution of a program on a remote host. The Bond shell is written in Tcl/Tk and Expect. Though well suited for the needs of a single individual, the Bond shell does not support group activities.

Bond 1.0 (http://bond.cs.purdue.edu) inherited the descriptors as well as the mechanisms for remote execution, from the Bond shell, but made further steps towards supporting group activities. The new system is built around a Bond server, an HTTP server, which provides persistent storage for all descriptors, or Bond objects in our terminology. The system is accessed from a Web browser and provides a uniform and location independent execution environment. Once a user with a Bond account connects to the Bond server, a collection of applets is downloaded and the user can build a workspace, the collection of Bond objects visible during a session and execute flow graphs remotely. The sandbox security model used to support applets in Java is very restrictive it does not allow local file access, it only allows an applet to communicate with the host it has been loaded from. Therefore we created an applet server running on the same host with the Bond server and all operations required to manipulate Bond objects were carried out at the site when the two servers were running. Bond objects could be shared among the members of the group.

A number of services for seamless remote execution are provided in Bond 1.0, including data migration, password, and scheduler agents. The scheduling agent [22] implements a dataflow execution paradigm and maps computations to the platforms of a computing grid with autonomous nodes. The data migration agent replicates data from the producer site to the consumer site on the grid. The knowledge base of the password agent identifies groups of machines which share the same password for a given user. Bond is a facilitator not a distributed system, the basic philosophy of the system is to defer security and resource allocation mechanisms to the individual nodes of the grid. Bond initiates an action, say ftp or rexec in behalf of a user, on a node of the computing grid using information stored in its internal knowledge bases, but does not attempt to store confidential information or interfere in any form with the resources management of individual nodes. When the password or even the user id are necessary, the system requests the information from the user. By August 1997, the Bond 1.0 system consisted of about 80,000 lines of Java 1.0, Tcl/Expect, and Clips code.

In this paper we discuss ideas and concepts pertinent to the new version of the system, Bond 2.0. In Section 2 we discuss the architecture of the system, in Section 3 we introduce the Bond execution model and discuss security and in Section 4 we discuss contracts and several models for scheduling on a computer grid.

2. Bond architecture
Bond is a Message Oriented Middleware, MOM, whose main ingredients are a lightweight mechanism to establish long term relationships amongst objects and a “universal” language which enables objects to communicate attitudes about information, being indifferent to the format of the
information itself [3], [5], [17]. Shadow objects provide a high level abstraction for a unidirectional communication channel linking two objects together. When an object needs to communicate it involves the directory service and, when the object is found, a shadow of the remote object is created and the connection is established.

A contract defines a temporary relationship amongst metaobjects expected to collaborate towards the goal of the contract. A Virtual Object Network, VON, is an abstraction for a set of metaobjects involved in a contract. The shadow mechanism described in [17] allows Bond to establish a network of cooperating objects that need to inform each other about the progress of the individual components of the contract.

The second important decision was to use KQML, Knowledge Sharing and Manipulation Language, as the native language of Bond. KQML is a product of the Knowledge Sharing Effort supported by DARPA, NSF, and AFOSR, for organization and coordination, [11], [15]. Intended as an inter-agent communication language by its designers, KQML is used in Bond as an inter-object communication language. In Bond all objects can receive messages regardless of their state.

KQML is indifferent to the format of the information itself and KQML messages may contain information in the so-called "context language". The meaning of a KQML message is defined in terms of the constrains of the message sender and allows the message receiver to choose a course of action compatible with other aspects of its function. The richness of KQML is most appealing to us, it allows us to define subprotocols, sequences of messages needed to carry out a desired function.

KQML messages, called performatives encode basic abstractions like asking, replying, achieving, subscribing or notifying. There are several classes of performatives: informative like tell and deny, database performatives e.g. insert, and delete, basic query performatives as evaluate, reply, ask-if, ask-about, ask-one, ask-all, sorry, effector performatives as achieve and unachieve, notification performatives as subscribe and discard, networking performatives as register, unregister, forward, broadcast, pipe, facilitator performatives e.g. broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all.

A Bond object allows dynamic definition of new properties. Properties defined at run time are stored in hashtables attached to the objects, while the properties defined at compile time are regular Java fields. Performance considerations force us to define all known properties at compile time, reserving the dynamic definition for the unexpected cases. The remote access to the properties of an object requires a transparent access to both types of properties as well as "access by name". The problem was solved using the reflection mechanism available in Java 1.1. The local "get" and "set" methods and the remote property access subprotocol provide access to the regular Java fields and the dynamic properties in a uniform way, while the compiled code can still access the fields at the full speed of a compiled language.

The basic element of Bond architecture is a cell, a collection of atoms, or Bond objects, b, coexisting on a given host. A cell consists of a local directory, dir, a resident r, the main thread of control of the cell, other threads spawned by r, including a messaging thread, mt and two mailboxes, in box and an out box, as shown in Figure 3.
The messaging thread pools its in box to determine if there are messages to be delivered, then invokes the message parser to determine the destination and contents of a message, and finally it invokes the corresponding method on the target object. Outgoing messages from all objects in the cell are placed in the out box by the messaging thread. Cells communicate with one another using a transport mechanism that removes messages from the out box of a source cell and places them in the in box of the destination cell.

There are different types of cells in the system. The function of a cell is determined by the configuration of the additional threads spawned by the resident. For example, the system monitoring cell is responsible to start server cells supporting the basic functions of the system, an external cell is created once a user connects to the system.

Bond cells are organized into domains or Intranets. The cells in a domain connect to one another according to service patterns, as shown in Figure 2. An external cell, E, first connects to the dispatching service cell, D, to gain access to the system, then accesses the persistent storage and global directory service, P, and other services distributed throughout the system, e.g. brokerage, B, authentication, A, software distribution, S. Control and supervisory services, are ensured by the system monitor, M, and the quality of service monitor, Q. Each service may be provided by the...
one or more service cells. New cells are created as needed. For example a contract scheduler cell, \( C \), is created in response to a request from an external cell to execute a contract. A contract is a Bond object describing a complex activity that can be carried out on a computer grid, it is a metaprogram in execution, the same way a process is created to run a program. A contract consists of components, internally represented as bondComponent objects, each describing an atomic action, for example the execution of a legacy application. Legacy applications are not aware of the fact that they executed as a part of a contract. The interface between the Bond system and the legacy application is provided by a wrapper, which, like other Bond executables is running in a cell, and controls the legacy application.

A domain is generally associated with a group of users with common interests. The cells of a domain are physically distributed over a computer grid. A domain grows and shrinks depending upon the needs of the community using it. The only cells active at any time during the lifetime of a domain are those providing Bond services, e.g. P, M, Q, D, B, A and so on. Domains may be connected with one another by establishing links between their persistent storage and global directory service cells as seen in Fig 2b.

3. Bond execution model and system security.
In this section we introduce the Bond execution model based upon the architecture of the system discussed in the previous section and discuss at some length solutions for the security problems inherent in a metacomputing environment.

Each Bond executable runs within a cell. In order to run Bond executables on a host, a cell must be started and configured as a member of the Virtual Object Network. Cells are started by small bootstrap programs called initiators either from command line or by the remote execution methods provided by the operating system. The initiator initializes the cell, and then terminates, leaving the cell running with only two threads, a resident and a messaging thread. Other threads running Bond executables are started remotely by messages using the execution control subprotocol.

In a Bond domain we recognize two components, a coreVON populated with permanent servers and agents providing core services, and userVONs populated with temporary servers and agents providing services for a given user. The specification of the precise functions of core servers and agents can be found elsewhere [18], here we are only concerned with the mechanisms to start up core services.

Locally, the Bond system relies on the security features of the local system (passwords, access rights, quotas). The most severe security challenge for a Bond object receiving a command sent by a remote object is to authenticate the source of the message before executing the command. Otherwise a potential intruder sending a message from a guest account can pretend to be a core agent and instruct the agent to execute commands that may affect the integrity of the system. This authentication problem is solved by a signature mechanism. Each core agent of the Bond system possesses a pair of public/private keys. The public key is made known to all interested parties and is kept by the authentication server. The private key is known only to the agent. All critical messages must be signed with the private key of the sender. The signature can be verified using the public key of the sender. This scheme raises two questions: how does an object learn the public key of the sender and how secure core agents and services are created.
The answer of the first problem is the existence of the authentication server. Whenever a remote object contacts another agent pretending to be a secure agent, the agent uses the authentication server to verify the public key of the sender. The authentication agent contains a security database whose entries identify secure agents and their public key. If the identity is confirmed, the authentication server sends the public key of the object to the requesting agent. An intruder can generate its own public/private key pair, however, its public key is not registered with the authentication server, so the command is rejected.

This leads us to the second problem. The security features in Bond are based on the fact that only legitimate agents are registered with the authentication server. To start a secure service on a cell, the cell itself needs to be secure, it should possess a key. Typically core services are started by the monitor agent. Figure 3 presents the message pattern to start a core service on a cell.

Figure 3. Starting a secure service on a trusted cell. The triangles indicates the keys of the executables. The message sequence is shown.

The process is initiated by the monitor sending a request to the server to start a new core service. If there is no trust relationship between the cell and the monitor the cell starts a sequence to verify the identity of the monitor, see Figure 4 and [5]. After verification the cell requests a new key from the authentication server. This request should be signed by the cell to prevent intruders to register keys, therefore the need for a secure cell. The authentication server sends back to the cell the new public/private key pair \( k'' \), encrypted with the public key of the cell. Even if the message is intercepted, only the cell can decrypt the message and discover the new key. Then the cell starts the new service and assigns to it a new private and public keys. Once the new service is started and registered, the cell sends the public key of the new service to the monitor in a signed message,
and the public key of the monitor to the new service, establishing a trust relationship between the monitor and the new agent. The initialization process between the monitor and the new service then continues without the intervention of the cell.

Figure 4 presents the authentication pattern. Agent a receives a message from an agent b it has not communicated with previously. To verify the identity of b, a generates a random pattern and sends it to b asking it to encrypt it with its own private key. Upon receiving the encrypted message, a sends it to the authentication server and asks it to decrypt the message. If the decrypted message and the original random pattern coincide, the identity of b is verified.

Once all services are started, the domain is ready, and a user with a Bond id can start a session and request execution of contracts. The security precautions described above ensure that all cells running core agents and services are properly configured. The fact that all cells were originally started by the domain superuser ensures that persistent data stored on each system are properly protected.

![Authentication Flowchart](image)

**Figure 4. Authentication of a trusted agent. The message sequence is indicated.**

Let us now turn our attention to the users. The first time a user connects to a Bond domain from a new platform he/she has to download a primitive cell and add to it a set of critical objects. This cell can be used to start a Bond session following the procedure described below:

- The user first signs on the system user-host using his/her own id and password.
- The user connects to the Bond domain, specifies a Bond id and a password. A message containing the bid, the IP address of the host and the current time encrypted with the Bond password are sent to the Dispatcher, D, running at a known address. In turn D forwards this message to the authentication server.
- The authentication server, A, decrypts the message using the Bond password corresponding to the bid and then it issues a service granting ticket. The service granting ticket provides information about services and capabilities awarded to the user vis-à-vis these services. For example a user may be allowed to start a contract scheduler and a broker but not a monitor. The capabilities for service y are encrypted using ykey.

4. Bond contracts, events, and models for scheduling on a computer grid

In this section we present mechanisms and algorithms used by Bond for scheduling then we discuss generic problems related to scheduling on a computer grid consisting of autonomous nodes. The operating system of each node of the grid is in charge with the management of local resources
and may cooperate towards computational goals involving several systems. The scheduling model we base our work on is the one described in [1]. To carry out complex computational tasks application brokers negotiate with resource brokers. A contract scheduler agent determines when different components are ready to run, then it communicates to an application broker the needs of the application as shown in Figure 5.

The Bond objects involved in scheduling are: contracts, contract scheduling agents, possibly other agents including the QoS monitors, and brokers. A contract is a workflow, a metaprogram in execution. A contract scheduler is an agent which coordinates the execution of a contract, maps service requests to available services and starts wrappers to run application programs, and maps services to platforms available on the grid. In Figure 5, the contract scheduler is created in response to a user request. In turn, a contract scheduler starts wrappers, temporary servers running application programs, and application brokers.

Bond is a message oriented middleware. Different functions are implemented by sub-protocols, dialects spoken by objects that have to collaborate towards a common goal. We now present sub-protocols involved in scheduling. Table 1 presents the subprotocol used by an external agent to start up and control a contract scheduler. In turn the contract scheduler uses the agent control subprotocol presented in Table 2 to start up a wrapper. A wrapper starts up a legacy application using the remote execution subprotocol in Table 3.

Figure 5. Contract execution and resource location in Bond.

QoS monitors, start resource brokers and collect synthetic information about resource utilization using the monitoring subprotocol in Table 4. The contract scheduler subscribes to classes of events generated by wrappers who in turn monitor the execution of an application. Examples of such events are: error conditions, successful completion of program execution, progress reports, creation of a file, and so on.

A Bond event is associated with changing of object properties, and creation and deletion of Bond objects. An external event typically triggers a Bond event. For example a wrapper detects that a legacy program terminated (external event), then the wrapper generates a Bond event by marking this change as a new property of the Bond object describing the component. Typically a monitor-
ing agent subscribes to Bond events occurring in the workspace of another agent and is periodically informed about the current value of each property the agent has expressed an interest into. It is the responsibility of the monitoring agent to maintain a history of past values and trigger actions based upon its assessment of the state of the agent being monitored. We have adopted this delayed-notification model for the sake of efficiency. The alternative is to notify the monitor when the actual change of property occurs but this would imply a high overhead.

<table>
<thead>
<tr>
<th>Performative</th>
<th>:content</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieve</td>
<td>init</td>
<td>:contract contract-id</td>
<td>initialize the contract scheduler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:workspace workspace-id</td>
<td></td>
</tr>
<tr>
<td>achieve</td>
<td>precompute</td>
<td></td>
<td>execute the precomputation</td>
</tr>
<tr>
<td>achieve</td>
<td>setup VON</td>
<td></td>
<td>set up VON of executor agents (wrappers) for the components</td>
</tr>
<tr>
<td>achieve</td>
<td>run</td>
<td></td>
<td>run the program</td>
</tr>
<tr>
<td>achieve</td>
<td>stop</td>
<td></td>
<td>stop the program</td>
</tr>
</tbody>
</table>

Table 1: The scheduling subprotocol.

<table>
<thead>
<tr>
<th>Performative</th>
<th>:content</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieve</td>
<td>start-agent</td>
<td>:agent agent type</td>
<td>start an agent of a specified type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tell</td>
<td>new-agent</td>
<td>:agent agent ID</td>
<td>acknowledgment that a new agent was started</td>
</tr>
</tbody>
</table>

Table 2: The execution control subprotocol.

<table>
<thead>
<tr>
<th>Performative</th>
<th>:content</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieve</td>
<td>legacy-run</td>
<td>:bondComponent comp</td>
<td>start legacy program comp</td>
</tr>
<tr>
<td></td>
<td>legacy-poll</td>
<td></td>
<td>poll wrapper to get status</td>
</tr>
<tr>
<td>tell</td>
<td>legacy-exec</td>
<td>:bondComponent comp</td>
<td>respond to polling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:completed percentage</td>
<td></td>
</tr>
<tr>
<td>tell</td>
<td>legacy-end</td>
<td>:completion-code completion-code</td>
<td>provide completion code of the legacy application</td>
</tr>
</tbody>
</table>

Table 3: Remote execution subprotocol.
Performative :content Parameters Description

| subscribe | monitor-information | :interest events :frequency interval | request sent by the monitor to an executable to send reports on certain events at specific intervals. |
| discard   | monitor-stop        |                                       | stop sending reports to the monitor. |
| ask-one   | poll                |                                       | requests the monitored executable to send an immediate report. |
| tell      | report              | :value messages :error messages       | sends the values for the requested parameters and the error messages (if any). |

Table 4: The monitoring subprotocol.

Scheduling on a grid of autonomous nodes raises difficult problems inherent to any decentralized system. The problems become even less tractable when there is contention for system resources. Further complications arise when the knowledge about the time of need and the amount of each resource is imprecise. We propose to examine several models of resource consumption and availability. These models are summarized in Table 5.

The first resource availability model is based upon the assumption that access to resources is guaranteed. This model corresponds to systems which support resource reservations or to those with dedicated resources. For example in a real-time system critical tasks require exclusive access to resources; a large parallel system is shared using reservation schemes.

The other availability model is based upon non-guaranteed resource access. In this case once a component is ready to run the contract scheduler determines where to execute the component subject to a set of constrains and based upon some knowledge of the state of the system. A network of resource brokers and QoS monitor(s) provide the information necessary to map computations to available resources in an optimal way.

<table>
<thead>
<tr>
<th></th>
<th>Guaranteed Access (G)</th>
<th>Non-guaranteed Access (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic execution</td>
<td>GD</td>
<td>ND</td>
</tr>
<tr>
<td>Non-deterministic execution</td>
<td>GN</td>
<td>NN</td>
</tr>
</tbody>
</table>

Table 5: Scheduling Models for a Computing Grid with Autonomous Nodes.

The first resource consumption model, deterministic execution, is based upon the assumption that resources need for each task and the time when they are needed are known in advance and the actual resource consumption will never exceed our expectations. In case of non-deterministic execution we know only an estimated value of the resources needed and the expected execution time. The scheduling algorithms should be prepared to deal with potentially large deviations from the expected value.
When we combine the models described above we end up with the four cases presented in Table 5. The model studied extensively is GD, guaranteed access and deterministic execution. In this case a schedule is precomputed; the most difficult aspect of this computation is searching a possibly large solution space for optimal schedules. A variety of approximate methods including genetic algorithms, can be used to reduce the time needed to compute a near optimal solution. Schedulers based upon a data flow model are used for the NN case for example the one presented in [22]. Efforts to address the GN model are under way, e.g. the one reported in [4].

5. Conclusions
Substantial changes are underway in information sciences and technologies. Powerful microprocessors, sensors, high speed networks, large volume storage systems, distributed information systems like the World Wide Web, are all part of the current computing landscape. The time seems ripe for network computing, a new computing paradigm which emphasizes network versus local resources. In this framework a Petaflops system can be a "virtual" computer consisting of thousands of autonomous nodes interconnected by a gigabit network.

Bond is a metacomponent architecture capable to accommodate the inherent heterogeneity of a computing grid and to ensure the usability and dependability of such a metacomputer. In our system objects communicate with one another in KQML. Long term relationships among objects are supported by a lightweight abstraction, the shadows, which supports the creation of Virtual Object Networks. In this paper we outline the basic architecture of the system. Then we introduce the execution model and discuss security. In the last section of the paper we present the workflow model and discuss scheduling on a computer grid with autonomous nodes.

Additional information about Bond is available at http://www.cs.purdue.edu/home/mb/main.html.

6. Acknowledgments
The work reported in this paper is partially supported by the National Science Foundation grants BIR-9301210 and MCB-9527131, by the California Institute of Technology, under the Scalable I/O Initiative, by the Intel Corporation, and by the Computational Science Alliance and the NCSA at the University of Illinois.

7. References
4. L. Bölöni and D.C. Marinescu. Robust Scheduling of Metaprograms in a Nondeterministic Environment, Department of Computer Sciences, Purdue University CSD-TR #98-003.
5. L. Bölöni, K.K. Jun, T. Daniels, and D.C. Marinescu. Message patterns in the Bond Distributed Object System, Department of Computer Sciences, Purdue University CSD-TR #98-004.
6. K. Mani Chandy, A Rifkin, Paolo A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka,


