RC24003 (W0607-048) July 13, 2006
Computer Science

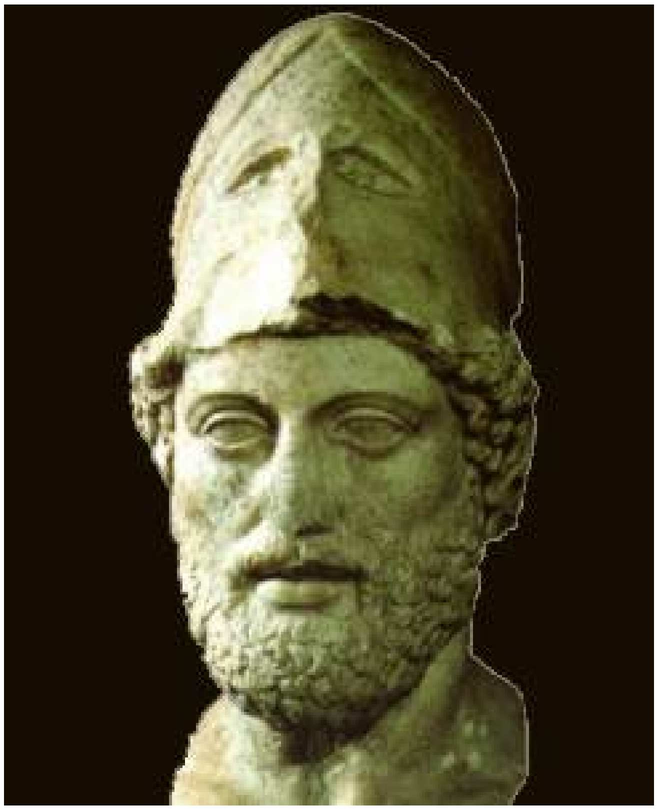
IBM Research Report

Pericles: An Object-Oriented Parallel Programming Model

Donald P. Pazel
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Pericles: An Object-Oriented Parallel Programming Model

Donald P. Pazel
IBM Corporation



What you leave behind is not what is engraved in stone monuments, but what is woven into the lives of others - Pericles

1. Introduction

The purpose of this document is to provide a technical overview and understanding of the Pericles object-oriented parallel programming model.

The need for Pericles derives mostly from a usability viewpoint. The types of operations found in distributed operations such as scatter and gather, for example, require considerable preparation work in setting parameters for data distribution. Providing flexibility in managing the asynchronous nature of parallel programming is also crucial. These concerns are addressed in Pericles mainly through the addition of new classes for data distribution and futures, for example, and through some supporting middleware.

Pericles also simplifies distributed programming using a very simple idea. The messaging paradigm of other distributed API's such as MPI transformed into a parametric messaging paradigm in Pericles. In this paradigm, distributed messaging is achieved through argument passing to remote calls. This simplifies distributed messaging, and returns the user to programmatic terra firma. This idea is not new in the literature, and has been very successfully used by the ProActive project at INRIA.

2. The Pericles Programming Model

2.1 The Pericles Programming Model – Overview

The Pericles programming model provides a set of sophisticated features to facilitate parallel and distributed programming. This facilitation is achieved though the introduction of a number programming concepts including:

- Parallel Complex – The application base class representing a ParallelWorkGroup.
- Parallel interfaces – Interfaces implemented by parallel complexes that identify and provide means for scatter and peer-to-peer operations.
- Distributions – A template class that facilitates the means for scattering data through parallel complex, without explicit message creation.
- Futures – A class that allows asynchronous management of distributed operations launched with scatters.
- Reductions – A class that allows information from a scatter to be incrementally processed in a background thread.
- Role topologies – Attribution on parallel complex members that allows them to be referenced to tasks specific to the application.

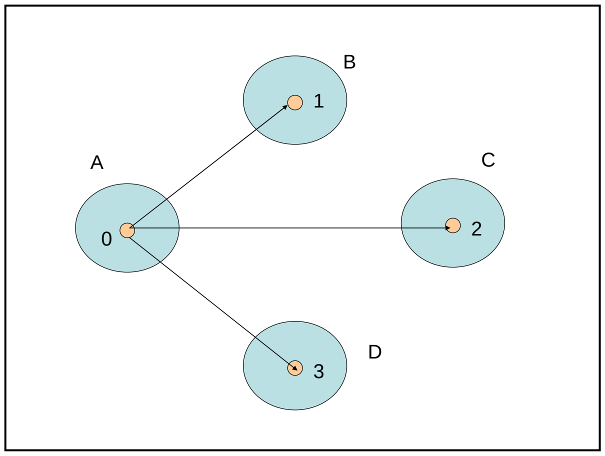
The objective behind the Pericles programming model is to facilitate parallel and distributed programming by taking over and managing many of the data and messaging aspects of parallel programming model.

3. Basic Concepts

3.1 *Parallel Complex*

3.1.1 What is a Parallel Complex

Pericles describes a notion of a parallel workgroup concept through an object representation called *parallel objects*, instances of which aggregate into a *parallel complex*, or *complex*, representing a full parallel workgroup. Each endpoint of a workgroup is represented by a parallel object, and the full set of these, one per endpoint of a workgroup, is a parallel complex. The parallel objects of a complex are *mutually-aware* in as much as each has means for identifying and communicating with each or with a subset of parallel objects collectively.



By example, and as illustrated in the diagram, {0, 1, 2, 3} represents a parallel workgroup with 4 endpoint members. Pericles provides and object framework for representing these as four parallel objects A, B, C, and D, with one-to-one mapping between objects and endpoints. The aggregate set of parallel objects {A, B, C, D}, covering all the endpoints, is the parallel complex.[1] In this example, A is initiating a scatter operation to the other members of the complex.

3.1.2 Characteristics of a Parallel Complexes

Complexes enable both *singular messaging*, as in peer-to-peer communication, as well as *collective messaging*, such as broadcast, scatter, gather, and reduction. Complexes invoke messaging using a call-based paradigm. That is, instead of issuing a message send/receive protocol, passing a data payload, which receivers receive through a messaging notification, Pericles messages are invoked through application interfaces through methods, with arguments being data payload. Receivers receive the call-based message as parameters on their implementations of those method interfaces.

Thus, the contractual agreement on complex messaging is that each parallel object of the complex implements the same messaging interfaces. These interfaces are called *parallel interfaces*, which will be further discussed in a succeeding section.

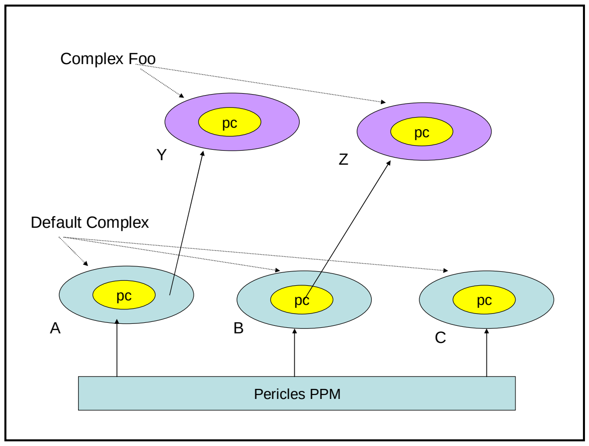
While parallel interfaces define the messaging contract for a complex, it does not constrain the implementation of parallel objects for the complex to be identical. In other

[1] Sometimes any one parallel object is referred to as a parallel complex. While this is an abuse of accurate terminology, it is nonetheless acceptable when the context is understood.

words, while the parallel interfaces define a uniform means for passing information within a complex, the parallel objects themselves can be each implemented uniquely to the intended computational collaboration intended by the complex. After all, each member of the complex may need special computational needs, objectives, and goals based on the design of the aggregate computation. In this sense, Pericles is a MIMD model, in distinction to the somewhat informal SIMD[2] model found in MPI. While this does not constrain the class types of a complex's parallel objects to be identical, in practice it would be practical to do so. With proper design, this should not impact greatly the variation in role that each object instance would carry for the aggregate computation.

3.1.3 How Parallel Applications and Complexes are Created

Launching parallel applications and parallel complex formation or generation is now briefly described. It is required that all parallel object classes derive from *ParallelComplex* and implement a special interface for parallel messaging called the *ParallelComplexInterface* interface. ParallelComplex provides the essential infrastructure for the parallel object to account for its endpoints, and asynchronous messaging. This interface provides means for intercepting a variety of messages, including formation and destruction of complexes, and changes in a complex's membership, for example. A parallel application is launched with a configuration indicating the number of endpoints, and possibly where the endpoints are located. Upon launching, the first parallel objects that are created are for the parallel workgroup representing the entire set of endpoints, called the *default workgroup*. We call the counterpart Pericles complex and object(s) for the default workgroup, the *default complex* and the *default parallel object(s)*.



As shown in the diagram, when all the default parallel objects are instantiated, so is the default complex, i.e. there is no need to explicitly add members to this complex; the system does that for you. Suppose, in turn, a new complex, called Foo, is to be defined consisting of the first two endpoints of the default complex, {0, 1}. In this example, A constructs a new workgroup containing only endpoint 0, and constructs Foo member Y, passing the new workgroup. Y then adds endpoint 1 to itself. This results in a notification to B, representing 1 in the default complex, that it should create Z, with endpoint 1 added to it. The process for formation of larger sub-complexes of the default complex should be clear now. It should also be clear how Foo could itself launch other complexes which would be totally instantiated by the other members of the Foo complex.

[2] MPI is not a strict SIMD model, allowing variation of computation amongst members of a collective. However, the lock-step protocol of many MPI commands does force synchronization amongst collective members, making MPI a command-based SIMD model. While Pericles requires sends to be received, sends are asynchronous, freeing one from the kind of lock-step synchronization found in MPI.

In all cases, parallel objects implement the ParallelMessage interface, which enables the process described above.

3.1.4 Programmatic Details on Parallel Application Creation

In this section, further details are given on the creation of parallel objects, referencing the figure of the last section. All parallel objects derive from ParallelComplex, which has several construction variants:

```
public ParallelComplex(ParallelWorkgroup parent, int [] members,
                       Class [] parallelIntefaces);
public ParallelComplex(ParallelWorkgroup  wkg, Class [] parallelInterfaces);
public ParallelComplex(Class [] parallelInterfaces);
```

In the first case, the constructor builds a new ParallelWorkgroup from a referenced parent workgroup and a subset of its members. The second has a provided workgroup handed to it. The last case allows the ParallelComplex to build its own ParallelWorkgroup. In each case, the parallel interfaces are defined to the parallel object. More on parallel interfaces is found in a later section. In the example above, A would construct Y with:

```
public Y(ParallelWorkgroup parent, int [] members);
```

which invokes then the first of the ParallelComplex constructors mentioned above. When the members {0,1} are added, B's ParallelComplexInterface notification method is invoked:

```
public boolean invitedToComplex(ParallelWorkgroup wkgp);
```

B then constructs Z and returns true. B can use the name of the parallel workgroup to coordinate to the class Z for the construction. The Y constructor would then be:

```
public Y(ParallelWorkgroup wkgp);
```

using the second of the constructors mentioned above. Parallel object A also gets notification, as its workgroup was the basis or parent for the workgroup. However, since A has already instantiated Y, with sufficient bookkeeping on object creation, it can bypass this notification. However, if A had not instantiated Y, but simply built a workgroup and added members, A would instantiate Y in a manner similar to the way B instantiated Z.

Finally, Y and Z get notifications through their ParallelComplexInterface that they their membership in their workgroup has been finalized:

```
public void complexCreated(ParallelWorkgroup wkgp);
```

3.1.5 Implications for Tooling

The tooling for parallel program construction could include features for helping with the following:

- Definition of a parallel object class, providing assistance with implementing the ParallelComplexInterface interface
- Assist in identifying all object classes that belong to a complex
- Assist in identifying all interfaces affiliated with a complex
- Assist in code generation on spawning a sub-complex from a complex
- Assist in code generation for the de-construction of a sub-complex for a complex

3.2 Parallel Interfaces

3.2.1 What is a Parallel Interface

Parallel interfaces provide the contract for messaging amongst the parallel objects within a complex. A parallel interface is a Java interface defining methods for peer-to-peer and scatter messaging within a complex. A complex may be defined through one or more parallel interfaces, and consequently, each parallel object within the complex must have implementations for each parallel interface. The implementation of each method within a parallel interface defines the kind of task work expected from a parallel object within a collaborative computation in that complex.

Since parallel interfaces are not distinguishable from standard Java interfaces, we require means for invocation of their methods with asynchronous parallelism semantics, as opposed to their standard invocation. In other words, the interface for parallel method implementation or expression should be distinguished from the interface for parallel method invocation. Towards that end, a parallel interface has two forms, one called the *implementation interface*, and the other the *invocation interface.* The parallel object implements the implementation interface for parallel work, but invokes the invocation interface to invoke or initiate a scatter or peer-to-peer task. As will be shown later, the invocation interface distinguished itself with variants on the method definitions that allow for singular and collective messaging, using data distributions and futures. The term parallel interface usually means both interfaces inclusively, however it can mean either separately from discussion context.

3.2.2 Characteristics of Parallel Interfaces

Parallel implementation interfaces specify methods whose implementation usually defines collective tasks. For each parallel object, the method would likely be defined in a manner unique to its role in the collective computation. If the parallel object type is unique amongst the complex's membership, it may be implemented uniquely. Usually however, a complex shares the same object type in which case the distinction in computation is dynamically determined, perhaps by endpoint identity.

An invocation interface, as described, is a counterpart to the implementation interface defining variants of the implementation methods allowing for asynchronous messaging. To its name, invocation interfaces are purely meant for parallel method invocation. Invocation interfaces are tool-generated based on implementation interfaces. A parallel object never implements the invocation interface[3], but instead obtains a reference to a proxy for it, to execute a parallel task. Invocation interfaces manage a number of critical details in distributed computation, including marshalling, communication with other parallel object endpoints, and various error detections.

Each implementation method is recast into a set of variant methods in the invocation interface. The variants allow for peer-to-peer messaging, collective messaging, and asynchronous invocation. For example, suppose the parallel interface *ValueSet* defines the following method:

```
   int getValue(double initValue, String title);
```

The invocation interface would include the following counterparts:

```
  Future<int> getValue(double initValue, String title);                // broadcast
  Future<int> getValue(Distribution<double, String> params);      // scatter
```

Two variants are constructed –for collective broadcast, and for collective scatter. In each case the return type is Future, a generic class that allows for asynchronous tracking of the resultant values. In the scatter variant, we introduce a Distribution generic class which provides a means for passing different data, of the same kind, amongst the parallel object endpoints. Further discussion on these topics will be presented later.

3.2.3 How Parallel Interfaces are Used

We will reference the parallel interface *ValueSet* and the getValue() method from the prior section. The ValueSet implementation interface would be defined as a standard Java interface:

```
interface ValueSet {
   public int getValue(double initValue, String title);
}
```

The ValueSet invocation interface is derived or generated from the ValueSet implementation interface. The new invocation interface's name is based on that of the implementation interface, with "*Invoke*" appended to the front. The result is *InvokeValueSet* in this case:

[3] The reason for not allowing parallel object implementation of an invocation interface is firstly that it contains signatures identical to the implementation interface, except for return type. Secondly, the implementation of the invocation interface is, per method, a generic enablement of a parallel task based on the signature of the method. This is best automatically or dynamically generated, and not handed to the user, per se.

```
interface InvokeValueSet {
  Future<Int> getValue(double initValue, String title);                  // broadcast
  Future<Int> getValue(Distribution<Double, String> params);      // scatter
}
```

To invoke parallel messaging, the parallel object first requires a proxy reference to the required interface:

```
InvokeValueSet ivs=(InvokeValueSet)this.getParallelProxy(ValueSet.class);
```

getParallelProxy is a method from the ParallelComplex class from which the parallel object derives. ParallelComplex is provided by Pericles for various parallel queries and operations. Given access to the invocation interface, it is easy to initiate a broadcast across the complex:

```
   Future<Integer> v = ivs.getValue(5, "Bin");
```

By this invocation, a parallel broadcast has been initiated across the complex. Each parallel object will execute getValue(5, "Bin"), and return their values. The Future<int> will track the receipt of all the int values from the broadcast replies. The call will not block; instead, the user may either query for completion, or wait and extract the value:

```
 int value =  (v.completed() ? v.intValue() : -1);  // or
 int value = v.wait().intValue();
```

The other invocation types will be examined in more detail later.

3.2.4 Implications for Tooling

Tooling for parallel program construction could include features to assist in the following:

- Generation of the invocation interface from the implementation interface
- Consolidate parallel object classes with parallel interfaces
- Generate code (assist) in obtaining interface proxies (getParallelProxy)
- Provide general code assist, completions, etc. as found in Eclipse
- Analysis for missing Futures, or Futures that have not been waited upon for values (within a given method).

3.3 Distributions

3.3.1 What is a Distribution

Distributed scatter operations involve the dissemination of varied data across a potentially large population of parallel object endpoints. With methods as the means for scatters in Pericles, there are natural limits on the argument data that can be sent through any one method signature. In fact, per signature, only one set of data can be specified as arguments, which can only be effective for a broadcast distributed operation, or peer-to-peer operation.

To broaden the means of using methods for scatter operations, an abstraction is introduced for representing an amalgamation of signatures of varied argument data, an abstraction called *Distribution*. Conceptually, a distribution allows many "signatures-worth of arguments" to be bundled together, for passing to the method for a scatter operation. Distribution is introduced in Pericles as an enabling software artifact. As was seen in the prior section, a variant of the original method is generated for the invocation interface that allows a distribution to be passed as an argument.

3.3.2 Characteristics of Distributions

Distribution is a Java generic class for enabling scatter operations. For a given method in a parallel interface, the parameter types of a Distribution used in a scatter-variant match the method's parameter types one-to-one, and in order. For example, given the following parallel method:

int sendInformation(int x, String y, Info z);

the corresponding Distribution that would be used in the scatter variant of the method is:

Distribution<Integer, String, Info>;

As a programming mechanism, a distribution is a set of *Distribution.Element*'s. A distribution element corresponds to a single signature's worth of arguments, and is templated similarly to Distribution. In the above example, we have:

Distribution.Element<Integer, String, Info>

where a Distribution.Element corresponds to on set of arguments to the parallel interface method *sendInformation*.

Thus, in programming a scatter, one builds a set of Distribution.Element's to disburse through a set of parallel object endpoints, adds each of them to the Distribution, and invokes the scatter variant of the parallel method, passing the Distribution.

Distributions are an aid for distributed programming in Pericles. However there are a few restrictions and rules. Since Distribution is implemented as a template, the size of the distribution's parameter type signature (corresponding to the number of parameters in the parallel method) is limited[4]. Java generics also have a number of considerations or limitations. For example, native types such as *int* and *double* cannot be directly used. Instead the Object classes *Integer* and *Double* have to be used. Also, one cannot specify arrays as parameter types, and any of the variety of Collection classes must be used

[4] Presently the proposed limit is 20. However, this will likely change with implementation.

instead. As a means for overcoming these short-falls in actual situations, such as when a parallel method has a very large number of parameters, a non-generic Distribution is provided to facilitate the packaging of information for scatter operations. However, discussion of this facility is out of scope of this report.

3.3.3 How Distributions are Used

We will re-examine the example of the prior section, and show details on how to use distributions for a scatter operation. Suppose we are to do a scatter of N sets of {Integer, String, InfoClass} arguments amongst the parallel objects of a complex. The construction of the distribution would look like the following:

```
Distribution<Integer,String,Info> d=new Distribution<Integer,String,Info>();
for(int i=0; i<N; i+) {
    Distribution.Element<Integer, String, Info> e = new Distribution.Element
                    <Integer, String, Info>(new Integer(i), strings[i], info[i]);
    d.addElement(e);
}
```

Following along the discussions of the prior sections, the actual scatter might look like the following:

```
InvokeProcIFace pi =( InvokeProcIFace)this.getParallelProxy(ProcIFace.class);
Future<Integer> f = pi.sendInformation(d);
```

Details on how the handle the Future and acquire the returned results will be discussed in a later section.

3.3.4 Implications for Tooling

Distributions are meant to facilitate scatter operations. Tooling features would therefore focus on facilitating the use of distributions, such as:

- Generation of Distribution<…> with type parameters, in appropriate contexts, based on the parallel method selected.
- Similar with the generation of Distribution.Element<…>
- Various code assists and completions to help in coding distributions easier.
- Refactoring Distribution’s when method parameters are changed.
- More direct code generative means for building a Distribution from a selected set of data (to be used as arguments).

3.4 Collective Messaging Operations (Broadcast & Scatter)

3.4.1 What are Collective Messaging Operations

The collective messaging operations in Pericles are broadcast and scatter. In a broadcast, arguments specified to the method’s signature are distributed identically to all the parallel object endpoints. In a scatter, multiple argument signatures are distributed amongst the parallel objects. Clearly, the former operation uses the multi-typed method signature found in the parallel interface, while a scatter uses an invocation variant utilizing Distributions.

In both cases, the parallel object endpoints return values commensurate with the return type of the parallel method. However, collective messaging operations are asynchronous in nature. Therefore, a device called "Futures" is used to manage the returns on these operations. Futures are discussed in a later section.

3.4.2 Characteristics of Collective Messaging Operations

By default, broadcasts and scatters target the entire complex membership, including the sender. That means, that the sender endpoint must include for itself data in its send, as part of a group computation, and its implementation method will be invoked asynchronously with that data in its parameters. Often this situation is not desirable, and there is a clean separation between master and worker processes. Pericles offers the option of excluding itself from a collective messaging operation. This facilitates greatly the implementation of master-worker relationships within a complex, when a leader member distributes work to the others. More generally, Pericles allows the sender to exclude specific members from the operation, restricting the operation to specific complex member receivers. Alternative, a specific subset of endpoints can be specified for the operation. In either case, this feature enables a rich master-worker computation to be defined that facilitates partitioning the field of complex members to different roles (see Role Topology section).

The distribution of argument data in a scatter operation follows a round-robin/greedy algorithm. That is, at the start of the operation, the Distribution.Element's of the Distribution are handed-out, in order, to each parallel object in the complex. If there are more elements than members, the next element is handed to the member whose response is most recently received, and so forth, on a first-come first-serve basis.

More generally, the method of data distribution is a variable in collective messaging operations. A *Distribution policy* can be defined and specified for any one collective messaging operation. One may want to ensure that endpoints with larger storage capacity, for example, get higher priority for receiving distribution elements than others, or that distribution elements follow a particular statistical pattern, perhaps based on the distribution element data values themselves. The ability to design distribution policies for collective messaging begins to address needs for resource management and scheduling at the programmatic level.

3.4.3 How Collective Messaging Operations are Used

Broadcasts and scatters have been discussed and exemplified in prior sections. Using our last example and setting the stage for further demonstration, examples of a broadcast and scatter are given:

```
InvokeProcIFace pi = (InvokeProcIFace)this.getParallelProxy(ProcIFace.class);
Future<Integer> bcastResult   = pi.sendInformation(id[k], name[k], info[k]);
Future<Integer>scatterResultf = pi.sendInformation(d);
```

Future's are used to manage the return results, and will be discussed in a later section.

In cases where it is derisible to do any of excluding the invoker, excluding other complex members, or narrowing the scatter to a subset of members, a qualifying member constraint must be specified. This constraint is specified thourhg the proxy. This is possible through the proxy invocation interface with a set of utility methods to deal with inclusions and exclusions. These are generated into the proxy invocation interface. In this case, we have:

```
exclude():                    // exclude invoker from broadcast/scatter
exclude(int [] exclusions);  // exclude specific endpoints
include(int [] inclusions);  // only use specified endpoints
```

When a proxy invokes any of the above, the designated set stays in effect with the proxy until it is further altered or reset. For convenience each method returns the proxy. By way of example:

```
bcastResult = pi.exclude().sendInformation(3, "abc", info);    // exclude self only
bcastResult = pi.exclude(new int [] {1, 5, 7}).sendInformation(…);  // exclude 1, 3, 7
bcastResult = pi.include(new int [] {1, 3, 9}).sendInformation(…);  // only to 1, 3, 9
```

As discussed earlier, distribution policies are user defined policies for defining how distribution elements are scattered amongst complexes endpoints. Distribution policies are implementations of the *DistributionPolicy* interface:

```
interface DistributionPolicy {
   int determineEndpoint(DistributionElement e, int [] targets, int [] availableTargets);
}
```

The determineEndpoint method accepts as input the raw generic type DistributionElement, which is the current element that requires a target endpoint identity, the set of allowable targets (which may have been restricted by inclusion/exclusion mentioned earlier), and the set of targets endpoints available for further calls of a scatter operation. The method returns a valid endpoint id to designate the target, or -1 to let the Pericles middleware decide. Note that if an endpoint is specified that is available, the system will wait on that target, and when it is freed will receive that data element. This is not strictly a synchronous barrier, however, as the policy will be called for other distributed element target assignments.

Policy assignment is achieved through the method:

```
bcastResult.assignDistributionPolicy(new MyPolicy());
```

This method also returns the proxy, and thus a broadcast/scatter can be appended to the above.

3.4.4 Peer-To-Peer Sends

Clearly, the include methods on the proxy are sufficient to execute peer-to-peer operations, by simply specifying the one target member. However, to facilitate peer-to-peer, we introduce the additional method to the proxy interface:

```
to(int targeted);              // only to the targetId endpoint
```

Much like the exclude and include interfaces mentioned earlier, this designated set (on one) stays in effect with the proxy until it is further altered or reset. Also, as before, the method returns the proxy, allowing the target method to be appended.

3.4.5 Implications for Tooling

Programming broadcast and scatter operations could greatly benefit from the usual tooling code assist and code completions. A number of related tooling traits were given in the Parallel Interface section 3.2.4. Along with those, it would be useful to have a means for help in coding distribution policies, as well as a means for locating all distribution policy classes within a code base, selection from which could help in coding the assignment statement.

3.5 Futures

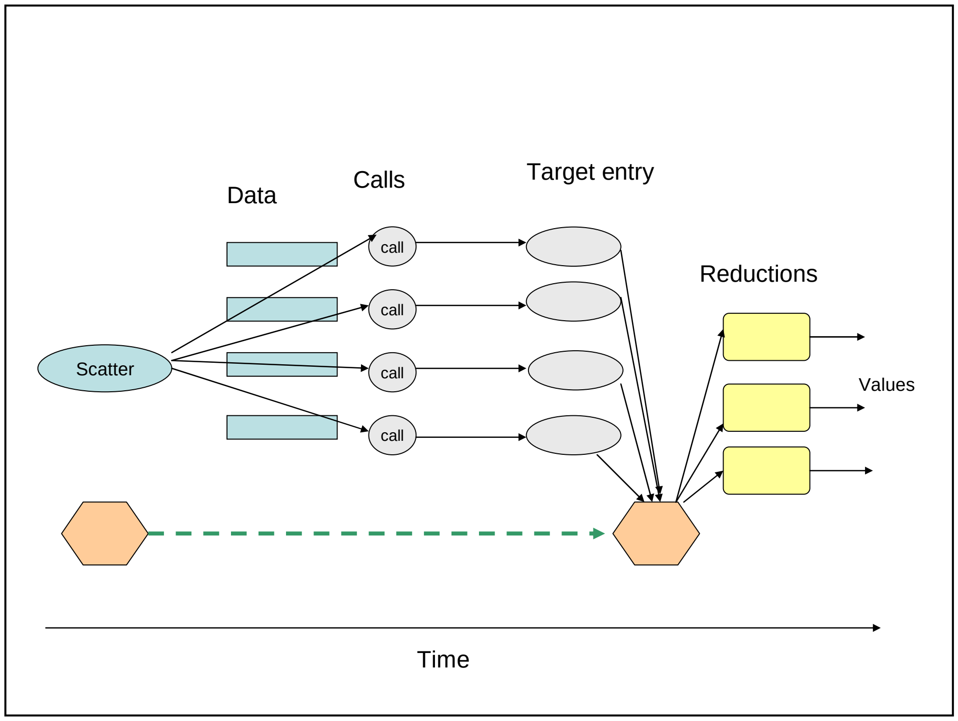
3.5.1 What are Futures

Futures are effectively handles to return results from asynchronous operations. Since Pericles is inherently an asynchronous programming model, Futures are widely used in Pericles programming. A Future is responsible for managing all the returns from the endpoints in an asynchronous operation, whether or not they return actual values. As discussed later, Futures also provide an anchor for means for distributed operation state notification and reduction of returned results into different data forms (see Future Notification & Reductions);

The Future class is implemented as a Java generic. The parameter type on a future must match that of the return type of the method of whose distributed operation the Future receives. In the case of “void” type, the raw Future type is used.

3.5.2 Characteristics of Futures

To gain an appreciation for the characteristics of a Future, it is worth exploring the figure below which examines the execution lifetime of a Future relative to a distributed operation. The timeline charts the events within a complex during, say a scatter. The Future is created at the time the scatter is initiated. In this case, a distribution is specified, and calls are made to all the target endpoints. Each target endpoint processes to the data (parameters) it receives on this operation, and returns a value. During this process, the future incrementally receives the returns from the target call. The Future also has a set of reduction objects which process the received results either as they are received by the future, or at the end of all the receipts (discussed in a later section). The program may retrieve all the reduced values at the end of the process.



Futures then are a managing device for coordinating the receipt of all the asynchronous calls in a distributed operation, as well as coordinating the received values from those calls. However, during that time, the program may proceed with other computations, and poll the Futures for there state, or even intermediate results.

The programmatic operations on Futures involve state query, waiting, and value access. For state query, one would be most interested in querying for the completion status of the distributed operation. The query methods on Futures include:

```
boolean Future::isCompleted();          // indicate whether operation is done or not
int [] Future::getCompletedCalls();     // int [] maps to endpoints for broadcast
                                        //        and to Distributed.Element’s for scatter
int [] Future::getCurrentCalls();       // maps as above, but for current calls
int [] Future::getOutstandingCalls();   // maps as above, but for call not initiated
```

The above methods on Future allow the program to check not only whether or not the full operation has completed, but also how much of the operation has completed. The latter includes queries for completed calls, currently processing calls, and calls that have not been initiated, i.e. when there are more Distributed.Element's than parallel object endpoints. The semantics for these change between broadcast and scatter. Since broadcasts have as many calls as endpoints, the returned array maps to endpoints. That is not the case for scatters, the returned array maps to indices of Distributed.Element's.

There are likely to be times when one needs to wait on a Future for a specific state to proceed with processing. The methods for these are:

```
boolean Future::wait();                      // wait indefinitely
boolean Future::wait(int ms);                // wait for a specified number of milliseconds
boolean Future::waitOn(int [] ids, int ms);  // wait for endpoints/element completions
```

As with the state query calls, the waitOn method interprets the integer array as endpoints or Distributed.Element's depending on whether this is a broadcast or scatter.

The cancellation calls are:

```
boolean Future::cancel();
boolean Future::cancel(int [] ids);    // cancel endpoints/element
```

Value access methods include:

```
<T>  getValue(int id);                 // get the value for endpoint/element call
<T> [] getValue(int [] id);            // get the values for endpoints/elements calls
```

For value accessing methods above, if the value has not yet been received, the method blocks until the value is received. So it is recommended to use any combination of query or wait calls first to assure the value access methods return promptly.

3.5.3 How Futures are Used

Much of how to use futures should be clear to most proficient programmers, from the characteristics section 3.5.2. The practical use of these characteristics is quite varied, and depends on the programming task. In simple cases, one would simply choose to wait on a Future. More complex cases might require querying the Future intermittently for completion, and do other work in the mean time.

3.5.4 Implications for Tooling

Programming tool features that could help in using Futures include:

- Discover distributed operations that do not receive a Future. These could be warnings.
- Re-factor distributed operations that do not receive a Future, into one that does.
- Re-factor all declared Futures when the parameter type changes.

- Provide a set of code generative patterns that can take advantage of the powerful incremental query and processing powers of futures described in 3.5.3.

3.6 Future Notifications & Reductions

3.6.1 What are Future Notifications & Reductions

Futures, as handles to asynchronous operations, can only be used as polling devices for distributed operations status. That is, the user must query the Future object to determine the status. Future notifications provide a means for observing asynchronously the completion status on a distributed operation represented by a Future object. Future notifications follow the standard Java notification mechanisms[5] of notification interfaces, and provide a means for a program to detect the completion of a distributed operation, or its intermediate stages of completion.

Similar to Future notification is the concept of Reduction. A Reduction is a means for computing application-dependent values based on the returned values of a distributed operation. For example, if a parallel method returns an integer value; one reduction might obtain the sum or mean of those values. In fact, it may be desirable to obtain intermediate computations, e.g. running sums, as a computation proceeds. This allows one to cancel or terminate a distributed operation based on early results. Operationally, there is very little difference between Future notifications and Reductions, and in fact the latter can be built from the former. However, conceptually they are distinct in their intended use of the results of distributed operations – the former for notification of state and the latter for computation of related states. Thus, they are related and extremely similar in nature, but described with their own unique settings.

3.6.2 Characteristics of Future Notifications and Reductions

Future notifications provide a means for a program to check not only whether the full distributed operation has completed, but also how much of the operation has completed. The latter includes notification for completed distributed operations, the state of currently processing operations, including states of operations that have not been initiated, i.e. when there are more Distributed.Element's than parallel object endpoints. The notification interface for a notification class on a Future is:

```
interface FutureNotification<T> {
    void finalCompletion(Future<T> f);
    void synchronized  partialCompletion(Future f<T>, int endpoint_DE);
}
```

The two methods provide means for being notified of full completion of the distributed operation, as well as a means for determining partial completion. In both cases, the user is provided a Future to further query the distributed operation. The partial completion method indicates the endpoint or Distributed.Element that most recently completed.

[5] Reference any of the Java 1.4.2 notification related objects, such as EventListener, Observable, Observer.

The implement a Future notification, and its association with a Future is as follows:

```
future.addNotification(new MyNotification() {…});   // example using anonymous class
```

Reductions can be entirely implemented using FutureNotification instances. However, the role of a Reduction has more to do with computing values bases on the returned values from a distributed operation, than with determination of state completion. Therefore, Reductions have a substantial role within the Pericles model, and have their own class. Naturally, the Reduction class looks nearly identical to that of FutureNotification:

```
interface Reduction<T> {
    void finalReduction(Future<T> f);
    void synchronized partialReduction(Future f<T>, int endpoint_DE);
}
```

To use Reductions, one implements the Reduction interface, and associates it with a Future like:

```
future.addReduction(new MyReduction() {…});   // example using anonymous class
```

3.6.3 How Future Notifications and Reductions are Used

How Future Notifications and Reductions are programmatically used should be clear from the above discussion. Notifications, as asynchronous indicators of distributed operation completion, are very useful in enabling multi-tasking while one or more distributed operations are in progress. The actual notification of completion, for example, indicates to the program when the associated Future can be queried for its final state, eliminated wasteful polling techniques.

Reductions are particularly interesting in that “running results” can be gathered in an event driven manner. This is particularly useful for monitoring long-running computations, whose individual tasks contribute to a final value. The program can monitor the computation’s progress, and potentially cancel it if the results are unsatisfactory, or better yet capitalize on the early running results.

3.6.4 Implications for Tooling

Programming tool features that could help in using Future Notifications and Reductions include:

- Reduction template classes for doing common reduction operations such statistics, data collection, etc.
- Coding patterns for managing multiplicities of asynchronous and related distributed operations and notifications.

3.7 *Role-based Topologies*

3.7.1 What is a Role-based Topology

The concept of role-based topology is a generalization of topology found in MPI. A topology provides an alternative means for referencing an endpoint over the usual 0, 1, 2 … indexing scheme. Instead, with a topology, one is able to reference the parallel object endpoints in a domain-specific manner. For example, for a topic such as matrix multiplication mathematical programming, it is convenient to arrange the endpoints into an array, and reference the endpoints by coordinates, or perhaps reference specific rows or columns of endpoints. In other domains, it may be preferable to arrange the endpoints into a graph.

Because of the MPI and numerical legacy of this concept, most discussions on process topology envision them mostly as geometric artifacts. However, conceptually topology breaks down into that of role assignment. In the array case above, the role is designated by coordinates; for a graph, as a graph node.

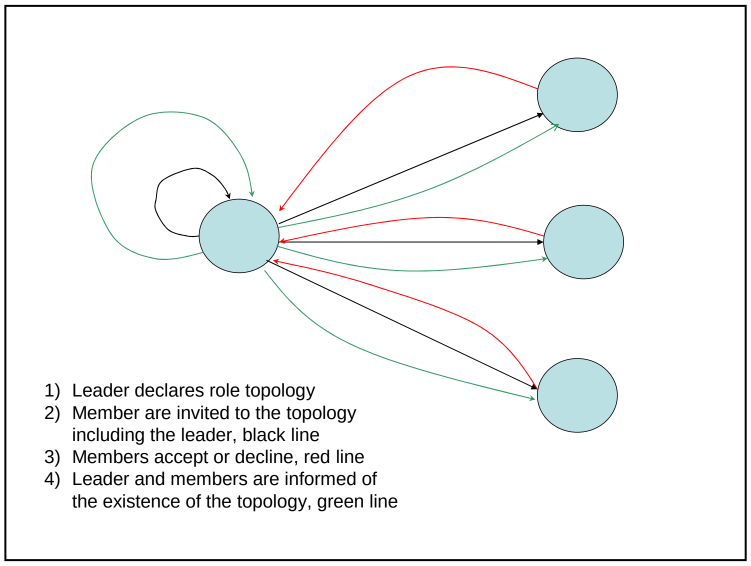
By making a couple of generalizations, a role-based topology could engender broader models. The generalizations are:

- Make the role designations general, e.g. based on strings, combinations of integers and strings, etc.
- Make a role-endpoint mapping one-to-many, allowing many processes to be identified by the same role.

For example, one could have a topology for transaction processing, with different roles, for example, for task receivers, transaction pricing, transaction executors, and report generators. More generally, any process can break into roles, with the roles of designated resources doing specific kinds of tasks.

3.7.2 Characteristics of Role-based Topologies

A *role* is a means of designation for one or more endpoints. It is general in nature, and domain specific. A *role topology* is a set of roles which cover all parallel object endpoints of a complex. By cover, we mean that each endpoint has an assigned role relative to that role topology. A complex may have none or one or more than one role topology.


1) Leader declares role topology
2) Member are invited to the topology including the leader, black line
3) Members accept or decline, red line
4) Leader and members are informed of the existence of the topology, green line

The process for constructing a topology is depicted in the figure. Topologies are constructed through a declaration of the topology by some member, called the leader of the topology, of the complex. The other members are then notified that a topology has been declared for the complex and that they need to accept or decline, as well as state their role in the declared topology. If all the members accept, the topology is validated to the complex, i.e. it formally exists. A second notification is then sent throughout the complex membership indicating that the role topology exists, and restates the role of the endpoint.

The main classes in role topology are:

```
  public Role implements Comparator {  } // User declares attributes for role designation

 public RoleTopology<R extends Role> {
    public int [] getRoleMembership(Role r);
    public int [] getRoleMembership(RoleQuery<R> q);
    public Role  getRoleFor(int endpointId);
    public Role [] getRoles();
 }
```

The programmer declares and implements roles, while RoleTopology only needs to be instantiated. The construction scheme described above utilizes ParallelComplex and ParallelComplexInterface methods:

```
   public ParallelComplex:: establishRoleTopology(RoleTopology topology);

   public Role ParallelComplexInterface::roleTopologyInvite(Class roleClass,
                                                            int     leaderId);

   public void ParallelComplexInterface::roleTopologyCreation(Class roleClass,
                                                              int     leaderId,
                                                              Role yourRole);
```

The establishment of the topology by the leader is achieved through the establishRoleTopolgy method from ParallelComplex. The ParallelMessage notifications include one for invitation, and one for role topology creation. In the notification methods, a role class itself is passed as a parameter indicating the class used in declaring the topology, e.g. RoleTopology<roleClassName>. The return in the role invitation notification is the Role that the endpoint declares for itself, that will be used in the topology. In the roleTopologyCreation notification, the role for the endpoint is passed, saving the programming from bookkeeping chores between invitation/acceptance and creation notification. At this time, there are no underlying mechanisms to ensure that the self-identification of roles (returned in the invitation notification) are consistent. That responsibility lies in the domain of the application.

3.7.3 How Role Topologies are Used

The RoleTopology class is utilized programmatically and through its methods, provides flexibility in describing the target members for scatter distributed operations. As a simple example, suppose all parallel object endpoint belong to a topology wherein they are color coded, and we need a scatter operation to the green members:

proxy.include(topology.getRoleMembership(greenRole)).scatter(Distribution<….);

In cases where, for example, role designation if more complex, and the desired membership for an operation requires more flexibility for determination, the generic class RoleQuery is provided. The essence of this class is a user defined method to accept or decline each role as a member of the distributed operation set.

Proxy.include(topology.getRoleMembership(new RoleQuery() {
public boolean acceptAsMember(Role r) {…}});

The role topology instance an also be used to query the role for a given endpoint id. The full set of roles can be queried through the role topology. The complete set of topologies can be queried through the ParallelComplex object.

3.7.4 Implications for Tooling

From a tool viewpoint to assist in program construction, the following features would be useful:

- Access to RoleTopology instantiations, or variables that hold those instantiations.
- Access to Role definitions, instantiations, or variables that hold those instantiations.

4. Examples

4.1 A Simple Scatter-Gather: Vector Multiply

This example involves the computation of vector multiplication with a matrix. The example is set up such that there is one complex, and a designated master of that complex. The complex holds to the matrix, and a set of vectors that are to be multiplied with the matrix. The scheme is simple; the master first distributes the matrix amongst the workers using row-wise stripping. That is, each of the worker processes gets a number of contiguous rows of the matrix. Then, the master sends each vector in turn to the workers, and collects for each vector from each worker, the portion of the result and stitches them together.

4.1.1 Class Structures

We will assume for this example that the same Java class will be used by the master and the workers. The differentiation of purpose will be clear; the process with rank or id 0 will be the master. We will use one interface called MasterWorker which has methods for receiving a piece of the matrix, and a method for multiplying a piece of the matrix.

The representation of matrices and vectors is with Java arrays, with a matrix being

```
double [] [] matrix;
```

And a vector being simply

```
double []  vector;
```

Since arrays cannot be used in generics, we will freely use some given helper classes that represent part or all of matrix. Detailed constructors for these are:

```
MatrixPart(int [][] matrix, int beginRow, int numRows, int beginCol, int numCols);
VectorPart(int [] vector, int beginIndex, int numIndices);
```

With obvious defaults constructors for representing full matrices or vectors.

The Java class for our example follows along with the parallel interface[6]:

```
class MatrixTask extends ParallelComplex implements MatrixWorker
{
}
interface MatrixWorker
{
   public boolean        deliverMatrix(MatrixPart matrixPart);l
   public VectorPart     multiply(VectorPart   fullVector);
}
```

4.1.2 Matrix Distribution

Matrix distribution is achieved through first building a Distribution<MatrixPart>, followed by a scatter to the workers. Assume that N represents the number of worker tasks in the collective, the distribution's construction following to this outline:

```
Distribution<MatrixPart> d = new Distribution<MatrixPart>();
while(rowsDone!=rowsDim) {
   int             rowsForThisPart = Math.min(rowsPerPart, rowsDim-rowsDone);
   MatrixPart    part = new MatrixPart(matrix, rowsDone, rowsForThisPart,
                                                    0, colDim);
   d.addElement(part);
   rowsDone += rowsForThisPart;
}
```

We then proceed with the distribution. We will not collect the reply Booleans as futures, however more robust implementations would. The general transport has an assumed sequentiality of delivery, meaning the matrix partial will be received before any vector multiplies are dispatched.

```
InvokeMatrixWorker imt = (InvokeMatrixTask)getParallelProxy(MatrixWorker.class);
imt.exclude().deliverMatrix(d);
```

[6] To simplify, we are excluding the application implementation of ParallelComplexInterface.

4.1.3 Vector Multiplication and Results Collection

For a given vector, we send it in its entirety to each worker, and collect the vector partials into a result vector.

```
Future<VectorPart> f = imt.exclude().multiply(new VectorPart(vector));
f.wait();
VectorPart [] vList = new VectorPart[N];
for(int i=0; i<N; i++)  vList[i] = f.getValue(i);
double [] resultVector = VectorPart.toVector(vList);   // concatenate results
```

The detailed implementation of the parallel interface methods is straight forward, and beyond the scope of this illustration.