Abstract—The properties of a parallel programming model are largely determined by the model of communication/interaction upon which it is based. Parallel programming models, languages and implementations have historically been formulated in terms of shared memory and/or message passing communication models. This paper motivates parallel programming models based on broadcast communication and defines and illustrates a parallel programming system based on associative broadcast. Associative broadcast is a dynamic multicast where the set of recipients for a message is dependent on the state of the processes in the broadcast network and is determined at runtime. Associative broadcast supports direct implementation of fully distributed (peer to peer) control. It enables dynamic composition of programs from components by integrating discovery, linking and invocation. A programming language and runtime system implementing the associative broadcast programming model are discussed including example programs and results. Parallel programs formulated in this programming model are intrinsically dynamically structured and fault-tolerance via replication is readily obtained.

Index Terms—Broadcast communication, distributed computing, parallel architectures, peer-to-peer networks.

I. INTRODUCTION

Parallel programming models and languages have historically been formulated in terms of shared memory and/or message passing communication models. This paper motivates parallel programming models based on broadcast communication and defines and illustrates a parallel programming system based on associative broadcast. Associative broadcast is a communication model which enables direct implementation of fully distributed (peer to peer) control. The associative broadcast programming model is based on a theoretical foundation in naming models [11], [12]. It enables dynamic composition of programs from components by integrating discovery, linking and invocation. Program development by dynamic composition is a paradigm shift from conventional parallel programming to a peer to peer implementation of what is now fashionably called the “web services” model of parallel computation. Parallel programs formulated in this programming model are intrinsically dynamically structured and fault-tolerance via replication is readily obtained.

Parallelism is emerging as a means of attaining goals other than “speed up” of a single program such as fault-tolerance by a parallel implementation of N-Version programming [13] and “greedy reuse.” [1] Parallel programs formulated in the associative broadcast programming model provide a convenient approach to these goals.

Formal models of broadcast communication [14], [15], [16] which incorporate dynamic composition are establishing a theoretical foundation for broadcast-based parallel programming models.

The subsections which follow give context on conventional parallel programming models and motivate attention to broadcast-based parallel programming by noting the increasing pervasiveness of broadcast communication.

A. Conventional Parallel Programming Models

A parallel programming model is defined by specification of an executable entity and a mode of interaction among instances of executable entities. While there are several variants of definitions of executable entities, such as processes, tasks, and run-to-completion actions, the properties of a parallel programming model are, to a large extent, determined by the model of communication and interaction upon which it is based. Most parallel programming models and languages have been formulated in terms of either shared memory and/or message passing communication models. (Skilllicorn and Talia [2] survey models of parallel programming.) The major exception is the family of programming systems based on Linda [3] and tuple spaces. It will be seen in “Related Work” that the Linda and associative broadcast programming models derive from the same family of naming models [11].

Broadcast-type operations in the form of group multicast operations are embedded in many message based parallel programming systems including the highly pervasive MPI. These operations are powerful and useful in the implementation of algorithm steps requiring group communication but implementing them in an intrinsically point-to-point message model may limit both their performance and their role in algorithm formulation.

B. Broadcast Communication

Parallel programming based on broadcast communication will become significant if efficient and reliable broadcast communications operations can be implemented for widely
used execution platforms and if significant benefit to motivate a paradigm shift accrues from their use.

Interconnection of computer systems via broadcast communication is becoming widespread, over mediums such as wireless, IP multicast, and Ethernet-connected cluster architectures. Ethernet (which is physically a broadcast) is the most widely used means of physical interconnection for workstations and personal computers in organizations. Ethernet is also widely used as the message media in clusters. It is straightforward to implement a reliable broadcast directly on an Ethernet segment. Broadcast abstractions and operations such as IP multicast are available to extend broadcast to multiple local area networks and beyond. There is a large and growing literature on implementation of wide area reliable broadcast and reliable multicast though overlay networks [17]. Wide area broadcast can and is often implemented in “gossip algorithms.”[18] There is significant research on wireless interconnect for multiprocessors on scales ranging from on-chip to all of the processors in a compact geographical location [19]. It is becoming clear that much of the Internet is partitioned into “small worlds” some of which are interest group affiliated, which will be connected by multicast. [20] The rising pervasiveness of broadcast communication suggests systematic consideration of broadcast-based parallel/distributed programming models.

It will be seen in the next section, that some broadcast-based distributed/parallel programming models lend themselves to implementation of “peer to peer” systems.

II. ASSOCIATIVE BROADCAST AND ASSOCIATIVE INTERACTION

A. Introduction

A detailed specification of the associative broadcast programming model is given in [4]. Here we informally introduce the associative broadcast programming model through a partial analogy with distributed (“peer to peer”) keyword word search [5], [6] to demonstrate the relationship to peer to peer systems. Consider a peer to peer system consisting of a set of hosts coupled by a broadcast or multicast communication capability. Each host stores a set of executable components and inverted indexes for a set of keywords over which the components it stores can be qualified and selected. Call the set of keywords associated with each component the profile of that component. Broadcast of a keyword set to the hosts engenders a parallel search of the set of distributed indexes stored by the hosts. A host desiring to compose and execute a parallel program broadcasts a message containing a set of keywords and the arguments for the signatures of the components. Each host in the broadcast or multicast domain qualifies and selects the set of components it stores whose profiles match the set of keywords. If the components located are procedures formulated in a remote procedure call interface, and if, instead of returning the components, the qualified procedures are executed and the results returned to the invoker of the “search”, then broadcast-based keyword search becomes a parallel remote procedure call mechanism for invoking sets of procedures: a primitive distributed programming model. This process combines discovery of services with linking and execution. It can viewed a parallel implementation of remote procedure calls or as a peer to peer implementation of parallel “web services.”

The analogies can be continued. The components which are invoked may also invoke further procedures through the “key word” search remote procedure call mechanism. Or as will be discussed in the next section, if the invoked components are formulated with asynchronous message passing semantics and the components which are invoked invoke further components then execution generates a data flow graph.

Keyword search is far too simplistic a mechanism for composing and structuring of parallel/distributed programs. Let keyword search be extended to include qualification over expressions in propositional logic (associative search) so that what is broadcast is an expression in propositional logic. Then, a component matches a search if and only if the keywords classifying the component cause the expression in propositional logic upon which the search is based to evaluate to true. We refer to the expression in propositional logic upon which the search is based as a selector.

This simple associative interaction programming model can be viewed as a broadcast-based implementation of distributed “peer to peer” keyword search where search is based on the selector model and where the result of a match for a selector is the invocation of a specified action.

B. Associative Interfaces and Component Interactions

Each component implements one or more functional behaviors and an interaction behavior, each of which is separately specified but coupled. The interaction behavior is specified as an associative interface. A family of components is defined over a domain defined by a set of attributes. Attributes are types with enumerated value sets.

Each component is encapsulated with an associative interface which includes an accepts interface and a requests interface. The accepts interface includes the profile of attributes, signatures for the functions it implements and the protocol (call-return or asynchronous message passing) by which it interacts. A profile is a set of attribute/value pairs. Profiles may be, and typically are, modified at runtime to reflect the current state of the component. The requests interface consists of a dynamic set of selectors specifying the

1 Very large clusters may have behavioral uncertainties similar to small distributed networks. In such cases the ready dynamic structuring and fault-tolerant properties of broadcast-based programming models may have advantage.

2 The results of the executions can be returned directly to the invoker of the search. However, the invoking component is also classified by a set of keywords. The results of the invocations could also be returned by each executing component broadcasting a search for the invoking component.
set of procedures or functions required by this component and the signatures and interaction semantics for them. A selector may be modified at runtime and selectors may be added to or deleted from the requests interface of a component at runtime. A selector, as described in the previous section, is an expression in propositional logic over the attributes. Each message is broadcast with a selector and messages are received only by those components whose profile causes the selector with the message to evaluate to true.

The behavior of a component is defined (implicitly or explicitly) by a state machine. The state machine sets the profile of the component to conform to the message(s) it should receive in the current state. When a component has only one action the state machine may have only a single state. A component receives a message (or a set or sequence of messages) and takes some action that is a part of its functional behavior (possibly null) in response to the message or messages.

Most practical parallel algorithms involve steps where a component requires multiple interactions before it is enabled for execution. Associative interactions have therefore been extended to incorporate data flow firing rules. The interface of a component may include a firing rule which specifies the set of messages which it must receive before it is enabled for execution.\(^3\)

Replication is another feature that must be included in associative interaction specifications to enable facile specification of parallel programs. In associative broadcast, replication of functionality can be made transparent and synchronization-free. If an initiating component starts several replicas of a given component to insure success in an unreliable environment and each of the replicated components responds by associative broadcast, then the initiating component can safely proceed after the first successful result and set its profile to ignore the other completions. Thus fault-tolerance based on replicated functionality is intrinsic to associative broadcast programming model. A component can be replicated by adding an index attribute to its profile and instantiating replicas in conformance to the index range.

“How greedy reuse” [1] is implemented by specification of selectors which will evaluate to true for multiple profiles. Parallel compositions which can be employed in N-Version programming [13] for fault-tolerance can be similarly implemented.

\(^3\) A firing rule is a state machine which determines when a component is in the enabled state.

C. Programs and Execution Model

A program consists of a configuration file, an initiating component and a dynamically generated, directed graph of self-managing components which propagate from the initiating component. Assume the components are active as daemons on hosts coupled by a broadcast network. Each component executes in the context of a host resident runtime system which implements associative interactions as an overlay to an implementation of reliable broadcast [9]. Consider the execution that results if the initiating component broadcasts a selector and components interact by asynchronous message passing. Each component whose profile matches the selector sent by the initiating component executes and may also initiate broadcasts of selectors (which may include the result of its execution) to discover and invoke further services. The execution terminates when components which do not require further services are reached. The result is a peer-to-peer implementation of a data flow program.

D. Properties of the Associative Broadcast Programming Model

The associative broadcast programming model introduced in the previous section:

- enables dynamic composition of distributed/parallel programs from components by integration of discovery, linking and invocation.
- directly implements fully distributed control of distributed processes,
- enables simple and robust joining and leaving protocols for interacting sets of processes, and
- enables fault-tolerance via synchronization-free replication.

These properties suggest that there may sometimes be an advantage in implementing a broadcast overlay to utilize these properties of broadcast based programming models, even when the physical network is point-to-point, or when the accessible communication interface is TCP/IP overlaid on a broadcast medium.

Additionally there are performance advantages in utilizing broadcast media when it is available. Broadcast communication is parallel communication. Computational processes formulated in a programming model based on broadcast communication may have more efficient communication than the same computation formulated in process to process communication or the same process with broadcast implemented on top of TCP/IP.

E. Algorithm Formulation

There is, however, at least one serpent in every paradise! Most parallel algorithms explicitly or implicitly are formulated on the assumption of central control. Use of a programming model based on broadcast communication will in most cases require development of new algorithms based on distributed control and may require a shift in program development paradigm in order to gain benefit. This is good news and bad news. The bad news is that use of broadcast programming models cannot become significant until a body of algorithms based on broadcast communication are established. The good news is that this requirement opens a whole new field of research in algorithms. There has been relatively little research in formulation of distributed/parallel algorithms in broadcast models of computation [8].
F. Development Model

Parallel programs are typically developed by designing a parallel computation structure which consists of components and relationships between components and then implementing the components as sequential programs and the relationships/interactions in terms of message passing and/or shared memory communication primitives. Components may be developed “from scratch,” linked from a library, or modified from some previously developed code. It is seldom that components are sought outside the immediate environment of the developer or development team. The program is usually executed in a master/worker paradigm by partitioning the data and the program and sending them to some set of processors for execution.

The development paradigm for broadcast-based programming models is intrinsically based on component composition. Broadcast-based parallel programming models coupled with fast and cheap communication and cheap memory suggest alternative development processes for parallel/distributed programs. Components may be discovered and linked across multiple platforms. Commonly used components may be made permanently resident, and perhaps running as daemons, on appropriate platforms. This is the “web services” model for composition of business processes.

In typical parallel programs, all components involved in the computation must be known ahead of time and linked into a static configuration of the system. Redundancy and replication of components must be decided and linked-in at compile time, and it is generally infeasible to alter those parameters during the execution of a system. The associative broadcast programming system is intrinsically based on dynamic linking between components. Dynamic linking allows for the runtime discovery of components, enables adaptive structuring of the program, and replication and redundancy for the purposes of fault-tolerance. Of course, connections can be optimized as point-to-point communications or statically bound as required for performance.

G. Status

A runtime system implementing associative broadcast has been implemented. The semantics of the current implementation of associative interfaces is given in [4]. The current runtime system is written in Java. The associative broadcast runtime is overlaid on reliable broadcast implemented based upon the Light-weight Reliable Multicast Protocol [9]. The CoorSet [10] interface definition language for associative interfaces has been designed and implemented. There is a compiler for the CoorSet language. Applications ranging from distributed mutual exclusion [4] and distributed readers/writers with replication [21] through numerical computations such as the one described herein have been programmed and tested.

III. AN EXAMPLE SYSTEM— DISTRIBUTED DATA FITTING

The example which follows is motivated by the concept of “greedy reuse.” [1] “Greedy reuse” uses parallelism to ensure the success of a computation by simultaneously executing multiple implementations of a required functionality when it is not certain which implementation should be used. This is an alternative use of parallelism where the goal is speed to solution rather than speed up of a given computation. Consider an application that collects a set of data points, and requires approximating them by a curve. There are many possible approaches to data fitting. Consider for illustration a case where it is unclear simply from the data set what method will yield a fit with certain properties required by the application. Possible properties are a minimum of error, compactness of representation, and smoothness of curve. It may be that the requirement is satisfied only by a composition of fits.

Using associatively-addressed components, several data fits can be executed simultaneously by addressing a data set with a selector that matches to true for the profiles of all data fitting components. The selector can be made more specific if only certain types of fits are desired.

The dynamic structuring of an associative broadcast network allows all available components to receive the data set in the message and respond without the calling component knowing what data fit components exist. Transparent replication and fault-tolerance is obtained by having several copies of the same type of component running, and when the calling component receives all of the results, it can compare them to choose the ones which meet the requirements, or alternatively, those which are faulty. In an unreliable environment the initiating component might choose to simultaneously execute multiple copies of another component just to insure that a result is computed and successfully received with high probability.

We have, for this illustration of concepts, implemented components that provide an exact Lagrange interpolating polynomial fit, a least squares approximation, and a natural cubic spline fit. Each component maintains a profile that identifies it as a data fitting component for the purposes of addressing, as well as profile entries that allow it to be addressed more directly when an application wants only a particular kind of fitting.

The dynamic structure of associative broadcast allows an application to link to all components available at the time a request is made, and to do so without explicit knowledge of what components are available; simply the knowledge of the accepts interfaces used by data fitting components is sufficient.

There are two possible system configurations for the components. The components can be active as daemons on hosts in the broadcast network in which case the initiating component is invoked on some hosts and discovery, linking and execution proceeds as previously described. Alternatively, the components can be in a file with the initiating component or in a library. In these latter cases the runtime system will distribute the components to hosts in the broadcast network and start the associative interaction runtime...
system.

Figure 1 contains the definitions for each of the types of components in the CoorSet [10] interface description language. These interface definitions are used to create the initial configuration of the component network. Each definition specifies the following:

- A “node” line including the number of replicas of the component to be instantiated at runtime,
- A “class” line, specifying the name of the component to be generated,
- A “target” line, specifying the class containing the user code to be linked into that component,
- An optional “execute” line, specifying a method in the target class to be executed when the component begins to run. If no such line is present, then the component simply waits for incoming requests.
- Zero or more “profile” lines, specifying the initial profile of the component,
- Zero or more “accepts” lines, specifying the initial transactions accepted by this component, the name of the method in the “target” class to handle the transaction, and a list of types specifying the argument format,
- Zero or more “requests” lines, specifying the initial transactions required by this component for its operation, the method in the generated component class to be invoked by the user code to generate the request, the initial selector for the request, and the list of types specifying the argument format.

In this example, we have five types of components. In this case, only one of each type is started, as indicated by the “1” after the “node” keyword; however, replicated instances could be started by increasing this number. The types of components are:

- **CurveRequestor**: A component that has collected some data set, and requires it be fit to a curve. It does not accept any transactions, but makes a “DataFit” request to all data fitting components by way of its selector.
- **LagrangeModule**: An exact Lagrange interpolating polynomial. Its initial profile has one attribute called “DataFitter” to indicate that it is a data fitting component, and a valued attribute called “Method” with a value of “Lagrange” to specify the particular kind of data fitter it is. It accepts the “FitData” request, and makes a “FitDataResponse_Poly” request.
- **LeastSquaresModule**: A least squares polynomial fitter. Its interface is almost identical to that of the Lagrange module, except that its profile reflects its being a Least Squares fitter, rather than a Lagrange interpolating polynomial fitter.
- **NatCubicModule**: A natural cubic spline fitter. It accepts the same “FitData” transaction, but responds with a “FitDataResponse_Spline” transaction, which contains a spline rather than a single polynomial.
- **CurveCollector**: A component that accepts the resulting curve fits from the above components. Its default profile contains an attribute called “CurveCollector,” which also is the default for selectors for the responses from the above components. It makes no requests, but accepts

```
node 1 {
    class CurveRequestor
    target CurveRequestorLib
    execute start
    requests "FitData" Request "DataFitter" (String, Double[], Double[], Integer)
}

node 1 {
    class LagrangeModule
    target LagrangeLib
    profile ("DataFitter", ("Method", "Lagrange"))
    accepts "FitData" processRequest (String, Double[], Double[], Integer)
    requests "FitDataResponse_Poly" sendResponse "CurveCollector" (String, Double[])}

node 1 {
    class LeastSquaresModule
    target LeastSquaresLib
    profile ("DataFitter", ("Method", "LeastSquares"))
    accepts "FitData" processRequest (String, Double[], Double[], Integer)
    requests "FitDataResponse_Poly" sendResponse "CurveCollector" (String, Double[])}

node 1 {
    class NatCubicModule
    target NatCubicLib
    profile ("DataFitter", ("Method", "NatCubicSpline"))
    accepts "FitData" processRequest (String, Double[], Double[], Integer)
    requests "FitDataResponse_Spline" sendResponse "CurveCollector" (String, Cubic[], Cubic[])}

node 1 {
    class CurveCollector
    target CurveCollectorLib
    profile ("CurveCollector")
    accepts "FitDataResponse_Poly" processPoly (String, Double[])
    accepts "FitDataResponse_Spline" processSpline (String, Cubic[], Cubic[])}
```

Fig. 1. Initial configuration of data fitting in CoorSet IDL.
both kinds of curves.

We also have three types of transactions:

- **FitData**: A request for a data fit. This transaction has four parameters: the first, a String, specifies a transaction identifier, so that multiple fits may be requested and the responses can be connected with the appropriate request. The next two parameters are arrays of Double values, representing the X and Y coordinates of the data points. The final Integer parameter specifies the maximum order of the polynomial, for polynomial fitters that can bound the polynomial degree.

- **FitDataResponse_Poly**: A response to a data fitting request, containing a polynomial fitting to the data. It carries a String with the transaction identifier for which this is a fit, and an array of Double values representing the polynomial coefficients.

- **FitDataResponse_Spline**: A response to a data fitting request, containing a natural cubic spline fitting to the data. It also carries a String transaction identifier, as well as two arrays of Cubic polynomials. The first is a piecewise parameterized representation of the X coordinates of the spline, and the second is a piecewise parameterized representation of the Y coordinates.

This configuration only represents the initial state of the network. Further components can come online later if need be. Profiles and selector interfaces can, and frequently will, change during the computation as is the case with the application component. These values all depend on the course of the execution, and will be updated during runtime.

### IV. AN EXAMPLE SYSTEM – MATRIX MINIMUM FINDING

The second example, a distributed algorithm for finding the minimum element of a matrix requires the compiler to compose multiple component types and illustrates replication for both parallelism and fault tolerance. Note that the Min-Finder component has been formulated to be transparent to replicated computation of the minimum value of the submatrices. The Min-Finder components can therefore be replicated for fault-tolerance as well as parallelism. In an actual execution, the italicized parameters \( K \) and \( L \) would be replaced with actual numbers, set to reflect the desired number of each kind of replicated component. The Initiator component can send the same submatrix to multiple instances of the Min-Finder component without synchronization. There may also be multiple instances of the Merge component. In this example, the state machine for the minimum-finder component is given explicitly.

In this example, we have three types of components:

- **Initiator**: This component divides a matrix into various submatrices, and distributes them amongst the Min-Finder components.
- **Min-Finder**: This component accepts a submatrix, and computes its minimum value. It then broadcasts this value.
- **Merge**: This component accepts the minimum value for various submatrices. As more and more submatrices of the total matrix are computed, it merges those values together to compute the minimum value for larger and larger submatrices, until the entire matrix is covered.

The attributes defining the domain of the Min-Finder algorithm are:

- **NodeType**: Initiator/Min-Finder/Merge

Min-Finder nodes also have the following attributes:

- **MinRow** and **MaxRow**: Range of rows processed at this node
- **MinCol** and **MaxCol**: Range of columns processed at this node
- **State**: Waiting/Computing/Completed

The transactions used in the computation are:

- **MatrixBlock** with a selector that matches nodes that accept this block’s subrange and a content of a submatrix.
- **Min-Value** with a selector that matches all Merge nodes and other Min-Finder nodes computing along an overlapping submatrix, and a content of the indices of the subrange computed as well as its minimal value.

The algorithm for the system is:

The Initiator component is given the matrix. It divides it up into various (possibly overlapping) submatrices, and sends each submatrix out in a MatrixBlock request. Each Min-Finder component receives its submatrix, and begins searching for the minimal element. It simultaneously listens for Min-Value requests, and if any such requests contain a solution for a submatrix of its work unit, it optimizes its computation by using that result and skipping that submatrix.

Each Merge component is given the dimensions of the matrix, and listens for Min-Value requests. When it receives one, it marks a submatrix as completed and stores the result value. As more Min-Value requests come in, completed submatrices are merged and the minimal value is stored until the entire matrix is marked as completed and one minimum value remains.

The state machine for the Min-Finder task is:

- **Waiting**: A Min-Finder task is waiting when it has not yet received a work unit. When it receives a MatrixBlock request,
it transitions to the Computing state. The Status profile entry is set to “waiting.” MinRow, MaxRow, MinCol, and MaxCol are assigned values out-of-band.

Computing. The Min-Finder task is searching the matrix for the minimal value in this state. While computing, it also receives any Min-Value requests that cover submatrices of its work unit, and if any such requests give results not yet computed by the local node, it incorporates that value into its computation and skips searching that submatrix. The Status profile entry is set to “computing,” and the requests interface

```java
node 1 {
    class Initiator
    target InitiatorLib
    execute startup
    profile ("(NodeType", "initiator")
    requests "MatrixBlock" sendMatrixBlock "" (Matrix)
}

node K {
    class MinFinder
    target MinFinderLib
    profile ("(State", "waiting"). ("NodeType", "MinFinder")
    accepts "MatrixBlock" processBlock (Matrix)
    accepts "Min-Value" examineMinValue (Double)
    requests "Min-Value" sendMinValue
    "(NodeType == MinFinder && State == computing) ||
    (NodeType == Merge)" (Double)
}

node L {
    class Merge
    target MergeLib
    profile ("(NodeType", "Merge")
    accepts "MatrixBlock" processBlock (Matrix)
    accepts "Min-Value" processMin (Double)
}
```

Fig. 2. CoorSet Program for Matrix Minimum-Finder Algorithm

is updated so that the Min-Value request’s selector reflects the range of the work unit.

Completed. When the Min-Finder has processed the entire matrix (possibly incorporating results received from other Min-Finder tasks as noted above), it sends out a Min-Value request with the range of its work unit and its minimal value. The node then sets its Status to “complete.”

The CoorSet definition for the initial system configuration of the matrix min-finder algorithm is given in Figure 2. The compiler binds the instances of Initiator, Min-Finder and Merge and instantiates the specified number of replicas of each component.

V. RELATED RESEARCH

Bayerdorffer [11] has developed a lattice taxonomy of name models and characterized the models of communication which can be implemented in a given model of names. The lattice taxonomy is defined with properties of name models as axes. It is shown in [11] that direct implementation of fully distributed, fully symmetric, minimal communication algorithms among dynamic sets of entities requires a model of names at the top of the lattice. Associative broadcast [12] is at the top of the lattice taxonomy of naming systems and thus supports direct implementation of fully symmetric and fully distributed algorithms for managing membership in dynamic sets. Linda [3] and its derivatives are the only other implemented communication mechanisms known to us corresponding to the highest point of the name lattice taxonomy.

Cast in Linda terminology, the relationship between Linda and AB can be given by analogy. The data of a process or component is the content of a tuple. The profile contains the match variables of a tuple. Selectors play the role of tuple templates. Each process or component which is activated by a selector/tuple match may send messages as a result of the action it executes which may bind it to other components. The relationship to Linda-like programming systems is that in associative broadcast the tuple space is fully distributed and the components of the tuple space may be procedures as well as data or function evaluations. Selector-profile matching is more general than Linda tuple space matching operations. Thus associative broadcast interactions can viewed as being based on implementation of a fully distributed tuple space with active linked tuples where all execution is associated with active tuples rather than separately defined processes.

Alternatively, associative broadcast can be viewed as implementing fully distributed name resolution [7] combined with remote method invocation.

Ostrovsky [16] sketches an implementation of associative broadcast as an instance of a higher order broadcast system.

The associative broadcast programming model can be viewed as computational-oriented a peer to peer implementation of “web services.” Associative interfaces combine information similar to that carried in the XML descriptors of web services and SOAP messages used to invoke web services. Simple Web Services are a standards-based implementation of remote procedure calls. Proposals for extending Web Services to programming systems (WSFL, XLANG, BPEL4WS) introduce some aspects of distributed control but are not designed to development of high performance codes. Similar concepts are explored in [22].

VI. SUMMARY AND CONCLUSIONS

The associative broadcast programming system which realizes some of the benefits inherent in use of broadcast communication has been described. An illustration that programs written in associative broadcast can realize many of
the potential benefits of broadcast-based parallel programming has been given.

Broadcast-based parallel programming has a growing role to play in high performance and high reliability computation. This role will increase as systems utilizing broadcast communication become increasing common and as the need for application systems which combine reliability and performance increases.

There is a need for further research in the role and application of broadcast-based programming and for development of broadcast-based algorithm for important computational processes.

**REFERENCES**


