# Agents: an Abstraction Mechanism that Labels and Taxonomizes the Objects it Encapsulates

J. Yelon
Dept. of Computer Science,
University of Illinois,
Urbana Illinois 61801,
jyelon@cs.uiuc.edu

March 3, 1997

**Abstract**

In object-oriented languages, an abstraction often involves a group of objects acting as a single entity. The programmer often has a mental vocabulary by which to label the objects: for example, in a particular LR(0) automaton, one object might represent the state "`P :== R . S`". In a particular instance of a binary-tree, one object might represent the data item "15". In general, each object in an abstraction can be mentally labeled, by the programmer, according to some statement of its purpose. The programmer's vocabulary to taxonomize and label the objects constituting an abstraction may be quite sophisticated and detailed. However, that vocabulary is only in the programmer's mind, it does not exist at the language level. The runtime system has no means to tell two instances of class binary-tree-node apart — it can't even distinguish the nodes of one tree from the nodes of another without sophisticated analysis. Though object-oriented languages make it possible to construct data structures with many different parts, they do not create a language-level vocabulary by which to refer to the different parts of the data structure.

There are several problems in the field of object-oriented parallel processing that have hampered programmers for years, and for which solutions have been elusive. We argue that the lack of solutions to these problems stems from the *absence of a vocabulary* with which to express those solutions. In particular, the inability to refer to objects by name, and the lack of a vocabulary to distinguish objects from each other, makes it difficult to even describe what is desired from a complex algorithm. When the runtime system can't identify which object is which, it becomes becomes impossible to express such ideas as locality, or interconnectivity, or grouping. Clearly, this is a serious problem for parallel programmers. We have designed a new encapsulation mechanism for parallel processing, the `agent` declaration. This mechanism makes it possible to create an abstraction whose subcomponents are labeled and taxonomized. Having done so, we straightforwardly address some of the difficulties that have been facing object-oriented parallel programmers for years. The purpose of this research is to explore the consequences of having a vocabulary by which to distinguish objects, and to experiment with powerful language-level constructs that were previously inexpressible.

# Contents

# 1 Introduction

While object-oriented oriented languages have recently gained tremendous acceptance as a means for sequential programming, their acceptance as a means for parallel programming has been slow. This reluctance is attributable to several inadequacies associated with using the object-oriented model in a parallel context. In short, writing efficient object-oriented parallel programs is quite hard. We have identified several of the factors that make it hard, and have come to a general conclusion: all these factors could be alleviated, in principle, but a single obstacle stands in the way. If that single obstacle were removed, progress in many areas of parallel object-oriented language design could begin to leap forward. It is the objective of this research to remove that one obstacle, and to explore many of the paths that open up as a consequence. The obstacle of which I speak is the *lack of a language-level vocabulary by which to distinguish objects from each other.*

Abstraction mechanisms often group a set of computational entities together, but some also assign a collective naming scheme to the encapsulated entities. For example, the abstract data type `array` not only groups a number of distinct sub-entities (elements) together, those sub-entities are identified by a naming scheme (indices) that tells the sub-entities apart. Using classes as an abstraction mechanism does not impose a naming scheme over the encapsulated objects. Objects are allocated individually, and then assembled together into larger structures in imperative code. It is the imperative assembly that leads to the lack of a taxonomy for the various objects in the abstraction. Though a group of objects may be conceptually related in the eyes of the programmer (e.g., "these objects form a binary tree"), there is no vocabulary for the user to make declarations about the whole tree or branches of the tree. The system itself can only reason about one node at a time.

Array indices are good labeling scheme for objects when those objects are in a rectilinear, finite, grid-like organization. For example, indices are a perfect means to label the objects that form the nodes of a Jacobi relaxation grid. When the objects are interconnected in a non-rectilinear fashion, for example, when the objects are in a hierarchy or a complex graph, a more sophisticated labeling scheme will be needed.

When an encapsulation mechanism creates an expressive naming scheme for the encapsulated elements, an obvious benefit is gained: one can refer to elements by name in the statements of the programming language. This, in itself, is more than enough reason to use an encapsulation mechanism that assigns a naming scheme to entities. However, there is a much more important and subtle benefit to be gained: once objects are distinguished by name, it becomes possible to express concepts which simply cannot be stated without such a naming system.

A good example of a concept that cannot be expressed without a good naming scheme for objects is "locality." Compare the redistribution directives of HPF (essentially, a load-re-balancer for array elements) to the load-balancers for object-oriented languages. Since HPF array elements can be distinguished by their names (indices), it's possible for the redistribution directive to know which array elements are which, and thus, which belong where. So, the load balancer for HPF array elements not only preserves locality, it is often used to improve it. By way of contrast, the load-balancers for object-oriented languages degrade locality. This happens because the load balancer is unable to tell objects apart: they all look the same to the runtime system. Since objects have no

distinguishing marks (except possibly past history, if the system is willing to settle for that), and since the system has no meaningful vocabulary by which to distinguish objects, the load balancer can't even begin to reason about which belong where. Thus, it ends up making more or less random individual-object movements, shattering whatever locality used to exist.

In fact, the very concept of "locality" is based on the idea that that objects have a location within some communication pattern. Thus, objects can be identified by their location. The objects could theoretically be labeled with information about their position in that communication pattern. Having such an explicit labeling system would make locality-preserving load-balancing trivial. Even if the objects were labeled with a naming system not transparently related to locality, it would be possible for the programmer to provide a mapping from the existing labels to labels more visibly related to locality. Almost any labeling system, so long as it is predictable, makes it possible for the programmer to write assertions about locality and other object properties. In current object-oriented systems, such assertions are inexpressible.

In general, a great many concepts pertaining to the structural relationships between objects cannot be expressed without a naming system for objects. Thus, we assert the following principle: an encapsulation mechanism for parallel programming should distinguish and label the objects it encapsulates. It should do so in a way that makes it possible to make assertions about object properties and about the structural relationships between them. The object-oriented encapsulation mechanism, classes and objects, does not taxonomize the objects it encapsulates, however, it can be modified to do so. Once such a modification is made, it becomes possible to make real progress in solving several previously unmanageable problems.

The objectives of this research are:

- To show that there are certain highly difficult foundational tasks which all parallel algorithms must perform. To show that the language cannot provide higher-level support for these tasks without a vocabulary to distinguish objects.
- To develop "agents", a new encapsulation mechanism for parallel processing that assigns a descriptive labeling scheme to the entities it encapsulates. To develop a compiler supporting agents as an infrastructure for exploring the advantages of this model.
- To show how the vocabulary created by agents makes it possible to express support mechanisms for the foundational tasks, thereby alleviating much of the difficulty of parallel programming. To implement those support mechanisms within the agents infrastructure, thereby verifying their expressibility and utility.
- To show how the same vocabulary simplifies the expression of parallel algorithms themselves. To verify this reduction in complexity through comparative programming experiments, which can then be evaluated through the use of software complexity metrics.
- To explore several interesting experimental language constructs that can be added once objects are labeled.

This document attempts to describe the research as it has progressed to date. It is structured as follows. Section 2, as a foundation point, simply lists some general principles to which we feel an

encapsulation mechanism should adhere. Section 3 describes two tasks which must be performed daily by object-oriented programmers, and which are extremely difficult to do efficiently using the standard object-oriented model. Section 4, we describe the *agents* encapsulation mechanism, which taxonomizes the entities it encapsulates. We show how it can provide powerful support to eliminate the problems we set forth in sections 2 and 3. Section 5 describes the work we have yet to perform.

# 2 General Principles

The following subsections briefly list several criteria that an encapsulation mechanism for parallel processing should adhere to. We set these general statements of principle forth as a foundation on which the remainder of the paper can build.

This section begins a convention of explicitly listing *encapsulation principles*, or criteria to which we feel an encapsulation mechanism should adhere. We also catalog a list of *known problems*, or ways in which some encapsulation mechanisms (including class definition and variants) have failed to satisfy these principles. The purpose of such listing is to pinpoint those ideas which we consider important.

## 2.1 Bottlenecks and Efficiency in Encapsulation Mechanisms

Most of the encapsulation mechanisms used in the parallel programming world are derived from similar sequential constructs. This introduces a potential problem: bottlenecks. A good example of such a bottleneck occurs when using a header object to represent a group of objects "behind" it.

*Known Problem — Using an ordinary sequential object as the header to a parallel structure creates a bottleneck.*

This doesn't eliminate all concurrency, but limits it severely. This was quickly recognized as a problem by researchers in the field, and was explicitly commented upon by Chien in [3]. The first solution was the *branchoffice*, which was part of the Chare Kernel[12] parallel programming system (later renamed Charm). BranchOffices are essentially distributed arrays of objects. They support an operation *branchcall* which causes the entire array to appear as a single object on which one may perform method invocations. Such invocations are handled by one element of the array, namely, the element which is situated on the same processor as the invoker. Thus, the array can serve as an interface to a parallel computation.

Without *branchoffice*s or some alternative, the only way to achieve concurrency is to bypass the header object, and communicate directly with the internals of a data structure. This violation of an encapsulation layer illustrates what will be a common theme:

*Encapsulation Principle — An encapsulation mechanism must not introduce undue costs, and must avoid bottlenecks.*

## 2.2 Continuations and Function Pointers

Many parallel language now include what we refer to as *metaprogramming* constructs. This very-loosely defined term refers to those language features which are highly abstract, and which tend to be used only by academicians. This class of features certainly includes first-class continuations and lambda closures, and debatably includes function pointers and first-class method pointers.

Metaprogramming features may be extremely complex, but they can be encapsulated into much more comprehensible tools. For example, first-class continuations are notoriously hard to understand. However, they can be used to implement threads, which are easy to understand, and whose utility is well-known. For this reason, metaprogramming constructs are useful and have their place. The fact that a construct has proven to be too complex for use by many programmers does *not* mean that it should not be included in a programming language. Such constructs enable the more sophisticated programmers to create useful tools for programmers in general.

However, when a construct has been shown too difficult for normal programmers to use, then every effort should be taken to avoid the imposition of that construct on those users who would find it difficult to comprehend. For example, anyone designing a module with a first-class continuation as an input should be aware that his module will be relatively inaccessible to anyone other than an academic researcher in computer science. In particular, such modules will tend to alienate a large subset of the customers of parallel processing, namely, scientists who learned some programming as a means to further their research.

Unfortunately, some parallel programming languages use metaprogramming too pervasively: they require the use of metaprogramming features to implement even the simplest parallel algorithms. Worse, many require the use of metaprogramming constructs in the *interfaces* of parallel modules. The use of a metaprogramming construct in a module-interface violates this principle:

*Encapsulation Principle – The use of continuation-passing in a module-interface is contrary to the encapsulation goal of hiding complexity within the module.*

A good example of a place where this principle was not used was is in the design of the Charm[10][11] language, which is an object-oriented language with support for remote method invocation. The problem arises from the fact that in Charm, like in many other such languages, remote method invocations do not have return values. An object must "return" a value to its caller through a callback mechanism: the callee must invoke a method on its caller. If a method is to be invoked from multiple call-sites, then one must not hardwire the method name used in the callback. Instead, one must pass in a method-name: e.g., one must use continuation-passing style. It is disappointing that all methods which wish to return values must use continuation-passing style. As a consequence, even the most trivial computations must use continuation-passing style, not only in their implementation, but in their *interfaces.*

*Known Problem – The lack of return-values in remote-method invocation often forces one to use continuation-passing style as a substitute.*

A quick look at a repository of parallel programs in Charm reveals that in several cases, the use of continuations was avoided by hardwiring the method-name of the invoker into the code of the

invokee. This practice tells us something important about encapsulation:

*Known Problem – Real-world programmers will sacrifice encapsulation entirely in order to avoid having to use continuation-passing style.*

Of course, the proper solution for this problem is to provide return values to remote method invocation. This reduces the need for continuation-passing style. However, as will be shown later, there are several other problems that lead to continuation-passing style as a partial solution.

## 2.3 Interface Compatibility

Modern parallel programming languages are being designed with a large number of language constructs. Unfortunately, these constructs are often designed in such a way that each construct accepts input in a different manner. As a consequence, one often finds incompatibility between the manner in which one module produces output, and another module expects its input. This makes it difficult to compose the modules.

*Encapsulation Principle — All encapsulation mechanisms in a language should have plug-compatible interfaces to allow maximum compositionality, at least insofar as possible.*

An example of this problem occurs in Chant[8], which contains two major constructs: threads, and ropes (groups of threads). Both kinds of entities interact with the world by sending and receiving messages. Thus, they are *potentially* plug-compatible. Unfortunately, this potential compatibility is damaged by an almost trivial difference at the type-system level: thread IDs and rope IDs aren't the same type. Therefore, a rope cannot be interchanged for a thread. For example, it is common for an entity to accept a thread ID as an argument, allowing the entity's creator to control where the entity sends its output. Unfortunately, this interface precludes directing the entity's output to a rope. If, on the other hand, the entity accepted a rope ID as an argument, it would be difficult to direct its output to a thread.

*Known Problem — potentially compatible interfaces can be made incompatible by trivial differences, such as type-incompatibility or differing accessor-names.*

Because of this apparently trivial incompatibility between Chant threads and Chant ropes, it is difficult design an entity whose output can go to an arbitrary location. One way to get around this problem is to use continuations. In other words, the entity accepts neither a thread ID nor a rope ID, but instead accepts a function pointer that controls every aspect of how it produces its output. Of course, doing so is a surrender to continuation-passing style.

*Known Problem — the best available solution to the problem of interface incompatibility is the use of continuation-passing style.*

The simplest way to avoid this problem is to minimize the size of the language, providing one general construct instead of many specialized constructs. This approach is being used less and less often, as composite languages like CC++[2], Charm++[13], Chant[8], HPC++ and many others are being invented with a large number of features.

The other approach is to provide compatible interfaces despite having multiple constructs. This is not usually difficult. Currently, the only language that we are aware of that uses this approach is *concurrent aggregates*[3]. In CA, aggregates were specifically designed to support the same interface as an ordinary object — method invocation. The type system also allows the substitution of an aggregate for a single object.

Either approach is acceptable, as long as the goal of compatible interfaces is achieved.

# 3 Foundational Tasks performed by Parallel Algorithms

There are certain foundational tasks that every parallel algorithm must perform: receiving input, producing output, allocating memory for data, and so forth. When one of these foundational tasks is difficult, the price is high: almost every algorithm written ends up suffering in terms of complexity or speed.

The following subsections describe two foundational tasks which are very difficult to perform in an efficient manner. Both tasks could be dramatically simplified by the addition of language-level support. However, it is obvious in both cases that such support could not function unless the runtime system had at least some ability to distinguish the objects from each other. Since current object-oriented languages don't make this possible, the language-level support has not been provided.

## 3.1 Data-Driven Output

Often, the purpose of an abstracted computation is to "produce outputs" which are values. Such production can either be data-driven, i.e., initiated *inside* the abstraction, or demand-driven, i.e., initiated *outside* the abstraction. For example, one way to design an object-oriented computation is to produce outputs via retrieval-methods. This approach is inherently demand-driven: abstractions designed in this way must wait until somebody outside the abstraction calls a retrieval method, only then can the abstraction produce one output value. The `return` statement itself is inherently demand-driven. In message-based systems, demand-driven output is recognizable by the fact that a message must travel into an abstraction for each piece of data emerging from that abstraction. Demand-driven behavior is often too synchronous for parallel applications:

*Encapsulation Principle – Data Driven Output: An encapsulated computation must be capable of producing output values not only on demand, but also in a data-driven fashion.*

Data-driven behavior is needed in situations where a producer-module is transferring large amounts of data to a consumer-module, and pipelining is desired (one wants each individual datum to travel into the consumer as soon as its available). Consider, for example, the case where the producer is a matrix multiplier. In this case, much pipelining is possible: the matrix multiplier produces a large volume of output data (an entire matrix), and it produces one element at a time. One can conceptually relay each matrix element into the consumer as soon as it is computed, without

waiting for the other matrix elements to be computed.

Consider how a matrix-multiply routine would be implemented in a parallel language like Concurrent Aggregates[3] (CA for short). CA computations are embodied by aggregates, which are much like BranchOffices: they are arrays of objects, with the ability to invoke methods on the array as a whole. One possible interface for the matrix multiplication would be an aggregate with the following methods: a method `accept-row` for feeding in the rows of matrix A, and another method `accept-col` for feeding in the columns of matrix B, and a method `retrieve-result-matrix` for obtaining the result matrix. This interface denies the opportunity for pipelining.

A better option is to replace the `retrieve-result-matrix` method with a `retrieve-one-element` method. This makes it possible to obtain result-elements as they become available, and pipeline the computation. However, it is demand-driven: one must make $N^2$ invocations to `retrieve-one-element`, thereby sending $N^2$ request messages (demands) into the matrix multiplier to obtain the results. These $N^2$ request messages contain no data, they are entirely a waste of bandwidth. In this manner, the demand-driven approach often doubles the number of messages needed. It also tends to use extra resources to record the requests: in this case, one must create and suspend $N^2$ threads. Overall, demand-driven output can be excessively resource-intensive.

*Known Problem – Demand-driven output uses extra bandwidth, because of the need to transmit requests. One needs data-driven output.*

To implement this pipelined producer-consumer scenario efficiently, one needs data-driven output: one needs to be able to send the $N^2$ results out, one at a time, without first sending $N^2$ requests in.

Even if one is not concerned with the wasted bandwidth associated with the use of demand-driven output, there is a second limitation associated with using demand-driven output. Consider another producer-consumer scenario, involving an AI system. In this scenario, the producer is a module generating plans to achieve a goal, using a search-tree to do so. The consumer is a module that analyzes the plans, double-checking their consistency. Clearly, this should also be pipelined. One possible interface for the plan-producer is a single `retrieve-all-plans` method. As in the matrix-multiplication scenario, this option denies the possibility of pipelining. A better option is a `retrieve-one-plan` method. Unfortunately, there isn't any simple way to implement `retrieve-one-plan` efficiently, because of the *routing problem*. The plans are being generated by a search-tree. They appear at apparently random locations within the objects embodying the search tree. Meanwhile, the `retrieve-one-plan` invocations are being made on the objects embodying the interface of the computation. Somehow, the plans must be routed from the objects in the search tree to the objects that have been asked to `retrieve-one-plan`. This routing can be achieved, but it is extremely complex.

*Known Problem – the use of demand-driven output often leads to a routing problem. To avoid this, one needs data-driven output.*

The only way to avoid the request-overhead (in the matrix example) and the routing problem (in the AI example) is to use data-driven output. One achieves data-driven behavior in CA by passing a continuation into the producer. The continuation forwards the results to the consumer. This

attains the goal of data-driven output, at a cost:

*Known Problem – the lack of any other mechanism for data-driven output leads to the use of continuation-passing style.*

Retrieval methods represent an example of demand-driven output, where one must send a request in before a result can come out. There are many other examples of demand-driven output mechanisms. For example, another common means of obtaining the results of a computation is to preallocate a variable, and to ask the producer to fill that variable. This approach is commonly used in languages like Id, in which computations are often started to fill I-Structures, or in Multilisp, where computations are started to fill futures. This approach is demand-driven: one must pass a pointer to the variable into the computation before one can get the results out.

*Known Problem – The technique of using preallocated variables to obtain the results of a computation suffers the same limitations as other demand-driven methods: wasted bandwidth, and routing problems.*

There is also an intermediate stage between demand-driven and data-driven approaches — some programming models make it possible to send in one request, after which many results can be obtained. We call this *single-demand-driven.* The technique of passing a continuation into a computation is a actually single-demand-driven — it is not truly data-driven at all. Thus, continuations (aside from being against our general principles) do not solve the data-driven output problem.

An abstraction mechanism for parallel programming should support the ability to send data "out" of an abstraction in a data driven manner. Unfortunately, the concept of sending data "out" cannot be effectively expressed without a vocabulary to identify which objects are inside the abstraction-boundary, and which are "out". In particular, the system needs to identify which outer objects are "connected" to the output of the abstraction.

## 3.2 Allocating and Connecting the Computational Entities

The encapsulation device determines not only what kind of entity will represent the computation, it also determines how those entities are allocated and what kinds of structures they can form. For example, the subroutine-definition mechanism determines not only that the computation will involve activation records, but that those records are allocated automatically and are linked together automatically into a hierarchical caller-callee relationship. Conversely, the class-object-method mechanism determines not only that the computation will involve object-instances, but that those instances will be allocated imperatively and then connected together by pointers into graphs of the user's own design. Since the encapsulation mechanism completely specifies the procedure which is used to allocate and connect the computational entities, we make this requirement:

*Encapsulation Principle — The encapsulation entities should be able to connect together into arbitrary graphs, to support arbitrary data flow patterns.*

This seems obvious. However, it doesn't appear to be a problem: the class-object-method paradigm appears to be sufficient for constructing arbitrary graphs, thereby supporting arbitrary dataflow

patterns, because of the degree of manual control it gives the programmer. This turns out not to be the case.

Consider, for example, the case where one wishes to perform a Jacobi relaxation on a 10x10 matrix. To do this in an object-oriented language, it might be desirable to create 100 objects, with each object connected to its four neighbors. If one views the 100 objects as nodes, and the neighbor relationships as edges, then the grid becomes a graph: a very predictable and regular graph, to be sure, but a graph nonetheless.

Most parallel object-oriented languages contain built-in primitives for constructing distributed arrays of objects. In other words, they contain primitives for creating this particular shape of graph. However, it is interesting to consider what would happen if the language did not contain a primitive to allocate this shape of graph. In such a case, one would have to build the grid manually.

The easy way to build this graph is with the help a separate 10x10 array of object pointers. This is an example of building a graph with *auxiliary storage*, where the auxiliary storage can be any storage device (usually an array or hash-table) used to hold pointers to the graph nodes during the construction process. There is a cost-penalty with using auxiliary storage: each node's address must be inserted into the storage, at a cost of one message. When an edge must be set up in the graph, one must fetch the address of the target object from the storage, at a cost of two more messages. Using auxiliary storage makes it easy to allocate a graph, as long as one is willing to pay the extra message overhead.

In this example problem, the overhead is 900 messages: 100 insertions into the array, 400 messages to fetch the edges, and another 400 messages containing the fetch results. If the Jacobi relaxation goes through many iterations, this overhead is probably irrelevant. However, if each edge were used only once, then the overhead would lead to a threefold multiplier in the amount of message traffic. This does not occur in Jacobi, but as will be shown in later examples, there are many computations where the graph edges are used only once.

*Known Problem — Constructing a graph using auxiliary storage to hold the node pointers can up to triple the message traffic.*

As an alternative, one might consider building the graph without auxiliary storage, keeping the object pointers in the graph-nodes themselves even as the graph forms. Until one allocates a graph-node, there is no place to store the pointers to its neighbors. Thus, when using this method, allocation of the nodes in the graph must follow a spanning tree. In our Jacobi grid, it is not exactly obvious what is the best means of defining a spanning tree over the grid, but this much is clear: the tree-structured allocation pass bears no structural relationship to the computation itself, and it will have to occur in a separate pass. In general,

*Known Problem — Constructing a graph without auxiliary storage often involves implementing a separate top-down tree-structured pass, which is both complex and inefficient.*

There is a solution in parallel object-oriented languages supporting distributed vectors of objects. One maps the graph nodes onto an array of objects using a static mapping. The mapping will be many-to-one, if the graph being built contains an unpredictable number of nodes, or if the

language imposes a one-object-per-processor restriction. For example, consider implementing our Jacobi problem by mapping the Jacobi nodes onto the elements of a Charm BranchOffice (array of objects with one object per processor). In this case, each array element must handle several Jacobi nodes. The BranchOffice elements serve as intermediaries, whose job is to allocate the Jacobi nodes and simply forward messages to them.

This BranchOffice is an example of a *graph-node manager*: any vector of objects whose job is to allocate graph nodes and forward messages to them. The objects of the graph-node manager are much like proxy objects in Corba/IDL, in that their primary function is message forwarding. To forward messages, the graph-node manager must have proxy methods — one proxy method for each actual method in the graph-node object. Unless one extends the language in some way to automatically generate graph-node managers, these proxy methods must be written manually. Thus, the total amount of code the user must write is as follows. For each graph-node object, the user must write one graph-node manager object, which must contain the same methods as the graph-node object. Each proxy method contains essentially the same code for static mapping and forwarding. Despite the fact that these proxy methods are all the same, code reuse is effectively nil. There is very strong dependency between graph-node objects and their graph-node manager objects, reducing the degree of cohesion in the program. The amount of redundancy in the code is immense. Thus,

*Known Problem — Constructing a graph by implementing a graph-node manager is bad software engineering, suffering from poor cohesion, poor reuse, and immensely redundant code.*

Thus, it is unfortunate that this is the only efficient way to implement any graph structure other than trees or arrays. The fact that there is no efficient and elegant means to assemble anything other than a tree or an array explains why so many constructs are described as "primarily for tree-structured problems" or "primarily for regular problems". The use of graph-node managers is common practice, it is by far the most popular means of implementing parallel object-oriented algorithms. This demonstrates that programmers will sacrifice software engineering objectives for efficiency.

To avoid this sacrifice, an encapsulation mechanism should provide an automatic means to assemble the entities in the computation into arbitrary graphs. Unfortunately, this is not possible without a means to express the shape of the graph that should be assembled. In other words, it cannot be done without some vocabulary to describe the relative positions and structural relationships between the objects.

# 4 The Agents Programming Model

We have devised an encapsulation mechanism for parallel computation, entitled the "agent". We say that function-invocations have agents "working for them." Agents serve the same essential role as callees. Just as a function-invocation may have several callees, a function-invocation may have several agents working for it. Functions pass arguments to their callees, they send data to their agents. Functions receive return-values from their callees, they receive result-messages from their agents. In both the function/callee relationship and the function/agent relationship, there is a clear

hierarchical structure of whom is working for who.

At a conceptual level, the difference between agents and callees lies in the timing. Ordinary callees perform the following steps, in order: the caller passes input to the callee. The callee's activation record comes into existence. The callee computes and finishes. The callee's activation record is deallocated, and the outputs are sent back to the caller. These 5 steps are supposed to occur in this strict order. Agents break all these sequencing rules.

At a more mechanical level, the difference between agents and callees is that callees are created imperatively (by executing a function-call statement), whereas agents are specified declaratively, and created on demand (when data is first sent to them). The agent declaration occurs within function declarations, like this:

```
void functionname0()
{
  agent agentname1(indices) runs functionname1(arguments...);
  agent agentname2(indices) runs functionname2(arguments...);
  ...
}
```

The agent declarations are intended to represent all the possible agents that the function-invocation might need during its execution. It is as if one were enumerating and labeling all the possible callees that a function might need to invoke. The agentname names a set of agents. Indices are similar to array indices, they can be used to select an individual element in the set. However, unlike array indices, they are not bounded, so the set of agents may be of unbounded size, and it may be sparse. The indices to the set of agents need not be integers, they may be any other type that can reasonably be used as an index into a table. Note that the agents are locally scoped within the function, the agents of one function invocation cannot access the agents of another function invocation.

Initially, the agents are virtual entities, taking no memory and performing no actions. At any time, the function may send data to one of its agents, awakening it: the agent's activation record is allocated, and its computation to be started, and eventually, the data is received and processed. When an agent runs off the end of its body, it returns to the virtual, dormant state.

Data is transmitted using the `send` statement:

```
send tag(value1, value2, ...) to agentname(index1, index2, ...);
```

In our notation, messages are tuples of values with a symbolic tag at the front[1]. We use the notation `tag(value1, value2, ...)` to denote such a tuple. The tag is a single identifier.

[1]The use of the word "tuple" to describe messages should not be construed to imply that such messages enter a "tuple-space", as they do in Linda. It simply means that messages contain a short sequence of values.

Messages, when they arrive, trigger *handlers*, defined by the `handle` declaration. These occur within function declarations:

```
void functionname0()
{
  handle tag(var1, var2, ...) from agentname(var1, var2... ) { code }
  ...
}
```

In the `handle` declaration, the from-specifier may be omitted to accept tuples from anywhere. When a tuple arrives, the variables in the handler declaration are bound to match the data in the message and the name of the sender. The handler may modify variables of the enclosing function.

When a function is executing, its handlers are inactive. The function's body continues atomically without interruption by handlers until the function executes a wait-statement:

```
  wait <boolean-expression>;
```

This causes the function to suspend while messages arrive and are handled. Messages continue to arrive and get handled (in the background) until the condition is satisfied, then the function resumes. Handlers *only* execute during wait-statements: even the message that first awakens the agent is queued until the agent reaches its first wait-statement. Code without wait-statements is always executed atomically.

A function, in addition to sending to its agents, may also send to `self`, `parent`, or `invoker`. `parent` refers to the agent's owner in the superagent/subagent hierarchy. `invoker` can only be used inside a handler, it refers to the agent that sent the message that triggered the handler.

The intention of the agents mechanism is that it provides an analogue to the caller/callee relationship and structure, without the sequencing limitations built into procedure call. With these mechanisms, functions may communicate with their subagents at any time. Unlike with callees, data may pass back and forth entirely asynchronously.

## 4.1 Minor Features

This section describes a number of the "finer points" about agents as a construct.

**Function Call.** Note that function names can be used in agent-declarations, but they can still be used in call-statements. Therefore, there are two ways to create a function-invocation: by calling a function, or by declaring an agent. Every function invocation, regardless of whether it exists by function-call or by agent-declaration, can send and receive messages.

**The Return Statement.** The return statement acts as it does in C. However, when a function is used as a agent, the return statement causes the agent to transmit a `result` message containing

the return-value to the parent. This can be distinguished from other `result` messages using the from-clause of the handler, if necessary.

**Multicast.** There is a variation of the send-syntax for multicast. This transmits the same message to a large number of agents:

```
send <message> to agentname(lo...hi, lo...hi, ...)
```

**Copying Data.** Sending data across address-space boundaries causes the data to be automatically copied. If the data contains pointers, as in a linked list, the copy is in most ways identical to the original. Sending data within address-space boundaries causes a reference to be passed without copying.

**Handers containing `wait`.** When a message arrives that triggers a handler, a thread is spawned to execute the handler, and that thread is executed immediately. If it suspends, and another message arrives triggering another handler, another thread will be created: there is no mutual exclusion. Thus, multiple handlers (even multiple instances of the same handler) can execute concurrently. The fact that code without wait-statements is always executed atomically often leads to mutual exclusion in practice, however.

**Send-from.** Normally, in a handler, the word `invoker` refers to the agent that sent the message that triggered the handler. One may change the value of *invoker* in the handler using:

```
send <message> to <agent> from <invoker>;
```

**Efficient Forwarding.** Note that a function may only send messages to its own agents, just as a function may only pass arguments to its own callees. This seems to suggest that data necessarily flows along tree-lines, as it does in sequential programs. However, in sequential programs, functions often simply relay data from one callee to the next. Similarly, agents often wish to relay data from one subagent to the next. To make this efficient, we add the following optimization rule: any handler that executes without touching a variable of the enclosing agent is a *relay handler*. Relay handlers need not be executed locally to the agent that contains them. Instead, they are executed on whatever processor sent the message that triggered the handler. Therefore, even if a message goes through a chain of relay handlers, it only makes at most one actual processor-to-processor hop, straight to its final destination. With this condition, data can move in any direction through the agents-tree. Note that the addition of relay-handlers also makes it possible for subagents to be allocated before their parents! If the parent relays data to a subagent without touching it, the subagent may in fact be allocated before the parent. This condition is necessary to prevent wrapper objects from becoming bottlenecks.

**First Class Agent-Names.** Agent-names evaluate to agent-references, which are values. Agent references can be stored in and copied from place to place in the usual ways. The only nontrivial use for an agent reference is in a send-statement.

**Objects as Agents**. One may convert an object into an agent using `agentize(obj)`. Sending a message to the object asynchronously triggers one of its methods.

**Agents Scoping.** There is a slightly more general version of the agent declaration:

```
agent subagentname(index1, index2, ...) runs { code }
```

This syntax is sometimes syntactically more convenient than the other. It is more powerful, in one sense: the code can refer to subagent names defined in surrounding scopes. It cannot refer to variables defined in surrounding scopes.

**Local Agents.** Agents can be declared local by putting the keyword `local` in front of the agent declaration. This simple pragma implies that the agent's activation record resides on the same processor and in the same memory space as its superagent's.

**Identifying Address Spaces.** We provide a function `nodecount()` which returns the total number of distinct address spaces in the host. We provide another function `nodecurrent()` which returns the index of the address space in which the agent that called `nodecurrent` resides.

**Controlling Address Spaces.** The following construct controls which address-space a particular agent is allocated in. `address-space-index` may be an expression involving the indices:

```
align subagentname(index1, index2, ...) to <address-space-index>;
```

## 4.2 A Simple Example: Fibonacci

The purpose of this section is to demonstrate the agent mechanism in the simplest possible context. After this example, we will move on to more interesting examples. The computation we show has

the same structure as this sequential program:

```
int fib(int n)
{
    if (n<2) {
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Here is the parallel version, using agents:

```
int fib(int n)
{
    agent sub(int m) runs fib(m);
    int total, count;
    handle begin() {};
    handle result(int x) { total=total+x; count=count+1 };
    if (n<2) {
        return n;
    } else {
        total=0; count=0;
        send begin() to sub(n-1);
        send begin() to sub(n-2);
        wait (count==2);
        return total;
    }
}
```

Note that in the sequential version, each fib-invocation has (up to) two callees, both of which are also fib-invocations. In the parallel version, each fib-agent has (up to) two subagents, both of which are also fib-agents. The first line of the parallel fib-function declares a large potential set of subagents underneath fib named sub(0), sub(1), etc. Of those subagents, only two are ever used.

The design is as follows: a fib-agent will send two messages with the "begin" tag to its two subagents. Each subagent will then begin executing, will compute a fib-value, and will send it back to the parent agent with a "result" tag. The parent will accumulate the results from its two children, and will then generate its own result.

Each fib-agent has two handlers, making it capable of receiving two kinds of messages: "begin" messages from the parent, and "result" messages from subagents. The begin-handler contains no code, but remember: the first message an agent receives always wakes it up (causes it to be allocated and start executing). Therefore, sending a begin message to a fib-agent wakes it up,

causes it to start computing, and does nothing else. The "result" handler is used to accumulate results from children. When the result is received, it is added to the running total, and the counter is incremented to keep track of how many results have been received.

After the declarations, we see the code of the agent. If `n<2`, it functions like a sequential fib function. In this case, none of the subagent or handler-declarations get used. If `n>=2`, the result-accumulator is cleared, two begin-messages are sent off starting two subagents, the agent waits until two results have been received, and then it returns the total. If the agent is the subagent of another, then that return-value becomes an upward "result" message.

Now that we have shown an introductory function, we move on to more interesting examples.

### 4.3 A Second Example: Summation

An interesting problem is implementing a summation tree, which adds up the contents of a large array of numbers. To show compositionality of agents, we begin by defining a simple accumulator agent. This small agent expects one to feed in a number of `add` messages. When it has received a predetermined number of adds, it produces an output in the form of a `result` message.

```
int accum(int count)
{
  int total=0;
  handle add(int n) { total+=n; count--; }
  wait (count==0);
  return total;
}
```

An agent running `accum` will wait until it receives a number of `add` messages equal to the `count` argument. Once it receives the appropriate number of values, it returns the total in a return value (or equivalently, a `result` message).

Note that the way the accum function produces values (by sending out a result message) is data-driven. One does not have to send anything in to the accum agent to get the results out, they simply emerge. This is manifested at the hardware level, one can see that only the add messages travel into an agent running `accum`. The utility of this will be visible later.

Note that there is no question of interface inconsistency in agents. This agent, like all other agents, produces its output as a stream of messages, and expects inputs as a stream of messages. By way of comparison, a similar object in Chant[8] would need to decide whether to produce output by sending it to a *thread*, or by sending it to a *rope element* — an impossible decision to make without contextual information.

Now we move on to the summation-tree proper. Its wrapper expects an array of numbers to be fed in in the form of `input` messages: one value per input message. Once all the inputs have been received, the total pops out in the form of a `total` message.

We decide to implement the summation tree by assembling a large number of `accum` agents together in the form of a tree. The tree is a particular class of graph, thus, this will be an example of graph construction in agents. There are two ways to build this graph in most languages. One can do it using a spanning tree, which is trivial, but unnecessarily slow: the data is moving up the tree, not down it, thus a top-down assembly requires a separate pass, doubling the latency and message traffic. One can also do it using an auxiliary hash-table, but note that each edge in this graph is being used exactly once: the overhead of using this approach would triple the message traffic. Agents will assemble and propagate data up this graph without difficulty.

The design is as follows: the wrapper for the abstraction is a single summation agent. Underneath the summation agent is a group of agents labeled `node(lo, hi, size)`. We assign to `node(lo, hi, size)` the responsibility of producing the total of all array-elements in the range `(lo-hi)`, where `size` is the size of the array. We utilize only some of these nodes (obviously, we don't wish to compute the totals of all possible subranges). At the bottom of the tree, we use the nodes that add two successive elements: `node(0,1,size)`, `node(2,3,size)`, `node(4,5,size)`, and so forth. At the next level we use the nodes that add four successive elements: `node(0,3,size)`, `node(4,7,size)`, `node(8,11,size)`, and so forth. This repeated doubling goes on up the tree until the root handles the entire array. For simplicity's sake, we assume that the size of the array is a power of 2.

```
void summation()
{
  agent node(int lo, int hi, int size) runs accum(2);
  handle input(int index, int size, int val) {
    if (index&1) send add(int val) to node(index-1, index, size);
    else         send add(int val) to node(index, index+1, size);
  }
  handle result(int n) from node(int lo, int hi, int size) {
    int rangesize;
    if ((lo==0)&&(hi==size-1)) send total(n) to parent;
    else {
      rangesize = (hi-lo)+1;
      if (lo&rangesize) lo-=rangesize; else hi+=rangesize;
      send add(n) to node(lo, hi, size);
    }
  }
  wait 0;
}
```

Explanation: the declaration of the `node` subagents constitutes the first line of the summation engine. The first handler is responsible for feeding `input` messages in at the base of the tree. The `input` messages come from outside the abstraction, and get routed to the appropriate node at the base of the tree depending on whether they are odd or even numbered elements. The second handler is responsible for the propagation of values up the tree. It takes the output from a node, and routes it to the next higher node in the tree. This is the first time we truly use the `from` clause: the variables `lo`, `hi`, and `size` are bound to the indices of the node from which the result came.

The main `if` statement in the handler checks if the value was produced by the root. If so, it is sent out as a `total` message. If the value was produced by an internal node, some simple mathematics determine the node's mathematical parent, and the value is routed upward. Note that since this function requires input messages, it cannot be called, it must be used as an agent.

Note that both handlers are relay handlers: all the messages bypass the main `summation` function. In fact, if an agent running `summation` is declared, the agent's activation record will never get allocated because no message ever goes to it. Thus, the `summation` function is not a bottleneck, as it would be if it were an ordinary object-oriented header. Also note that any executable code in the function is irrelevant, as the function will never get executed. We put a wait-forever statement there out of a sense of completeness.

The data-driven manner in which the `accum` agent produces its results are valuable here. The outputs of the `accum` agents were obtained and forwarded *without* sending requests, pointers, or continuations into the `accum` agents. Instead, the outputs simply emerged from the accumulators, were captured, and were forwarded to where they belonged. In comparison, if the `accum` agent had been an accum *object*, then each accum object would need to know where to send its results. Each `accum` object would need to be given a pointer to the object that wants the results. This would be an example of a situation where the lack of data-driven output would lead to continuation-passing style. However, the continuation-based program would still require twice as many messages as the agents implementation, requiring those extra messages to pass the parent-pointers into the `accum` objects. This demonstrates the fact that in some situations, single-demand-driven output is as inefficient as ordinary demand-driven output, and it is not a substitute for true data-driven output.

The agents declaration mechanism makes it possible to assign semantically meaningful names to the agents. Having a clear vocabulary for referring to specific agents, in particular, a vocabulary reflecting the logical function of the agents, makes the expression of the algorithm fairly clear.

This algorithm shows one of the uses of relay handlers. This alternate implementation of the summation algorithm shows a more interesting property of relay handlers:

```
void summation()
{
  agent L() runs summation();
  agent R() runs summation();
  agent acc() runs accum(2);
  handle rinput(int pos, int lo, int hi, int n) {
    if (lo==hi) send total(n) to parent;
    else {
      int mid = (lo+hi) / 2;
      if (pos<=mid) { send rinput(pos, lo, mid, n)   to L(); }
      else          { send rinput(pos, mid+1, hi, n) to R(); }
    }
  }
  handle input(int pos, int size, int n)
     { send rinput(pos, 0, size-1, n) to self; }
  handle total(int n) { send add(n) to acc(); }
  handle result(int n) { send total(n) to parent; }
}
```

This is a much more elegant design than the previous one. In this variation, the agents are in a recursive hierarchy, each agent having two subagents, each responsible for half its range. When an `input` is fed in, it gets converted to an `rinput` message. The `rinput` message rapidly works its way down the tree using a binary-search to find its proper starting point.

Despite several factors which might suggest otherwise, the graph-assembly is just as efficient in this implementation as in the previous. In particular, the `rinput` messages appear to be moving down the tree, the tree is nonetheless allocated from the bottom up. This is because the `input` and `rinput` methods are relay handlers: the value works its way down the tree without actually touching the nodes of the tree. In fact, the only entities that ever do get touched are the `accum` agents: the rest of the structure is an abstract concept only, the accum agents are the only entities that ever get allocated. The binary-search, incidentally, is performed by the processor that tries to send the input into the tree. Unlike the previous implementation, the array does not need to be a power of 2, and thus, the tree is no longer perfectly balanced. This irregularity does not impact the efficiency or simplicity of the graph assembly. This again highlights the fact that agent declarations with relay handlers make it possible to assemble complicated graph structures, and for data to move directly into the graph and through it in the optimal direction.

## 4.4 An Example Utility: A Replicated Storage Agent

Our next example is a replicated-storage agent. Its purpose is to enable the broadcast of data to all processors, where it is immediately accessible to all who want it. Before we can implement it,

we start by writing a simple storage agent with handlers for `set` and `get`:

```
void simplestore()
{
  int done=0;
  misc value = nil;
  handle set(misc v) { value = v; }
  handle get() {
    wait (value != nil);
    send reply(value) to invoker;
  }
  handle free() { done=1; }
  wait done;
}
```

This agent simply contains a variable. It accepts one kind of message to `set` the variable, and another kind to `get` the variable. The `get` handler sticks around in the background as a sleeping thread until the value is set, then it transmits the result. This demonstrates one way to implement a request-reply protocol in agents. The agent sticks around and remembers any stored value until somebody sends it a `free` message. Here is a function designed to fetch from a simplestore agent:

```
misc fetch(agent a)
{
  misc value=nil;
  handle reply(misc x) { value=x; }
  send get() to a;
  wait (value != nil);
  return value;
}
```

This simple agent demonstrates how agents can act like objects that return values. The fetch subroutine *is*, for all intents and purposes, a blocking method of the simplestore agent. Note that the same agent also has a nonblocking interface. Thus, in agents, continuation-passing style is not needed to substitute for the lack of return values.

Given this basic construct, one can implement replicated storage. Each node has a `simplestore` agent like the one above holding a copy of the value.

```
void replicatedstore()
{
  agent node(int n) runs simplestore();
  align node(int n) to n;
  handle set(misc v)
     { send set(v) to node(0..(nodecount()-1)); }
  handle get()
     { send get() to node(nodecurrent()) from invoker; }
  handle free() { send free() to node(0..(nodecount()-1)); }
}
```

This agent keeps a copy of whatever value you set on every processor. It is plug-compatible with simplestore, therefore, fetch works on it too. Whenever you request the value, it obtains it from the nearest copy. Because of the forwarding policies of relay handlers (no unnecessary hops), and the copying policies for messages (no unnecessary copying), the get is guaranteed to be efficient.

This time, taxonomy is used for a different purpose: we assign names to the `node` agents according to what processor they are on. Having semantically relevant names that the compiler can see makes it possible to state the align directive. It also makes the expression of the imperative part of the code simple.

### 4.5 An Example Application: Theorem Proving

As our next example, we show a use of the replicated storage agent. This subroutine is an SLD refutation engine. SLD theorem provers start with a single assertion, and by combining that assertion with a (unchanging) rule-database, generate more assertions. These new assertions are in turn combined with the rule-database, and so on recursively, until finally an assertion is derived that is known to be false. This refutes the original assertion. The problem is almost a plain search tree, with one exception: assertions frequently get re-derived, but they must not be re-processed. The prover consists of a `refute` function, which uses many `tryrefute` children. Each `tryrefute` agent has the task of trying to refute one assertion. A `tryrefute` agent either sends `refuted` immediately, or it derives a set of assertions, transmitting `begin` tuples to initiate their recursive

expansion. It then sleeps forever, thereby refusing to try to refute something twice.

```
void refute(assertion goal, database db)
{
  agent dbholder() runs replicatedstore();
  agent tryrefute(assertion A) runs {
    if (obviously_false(A)) send refuted(A) to parent;
    else {
      database DB = fetch(dbholder);
      for all assertions D derived from A and DB do
        send begin() to tryrefute(D);
    }
    wait 0;
  }
  handle refuted(assertion a) { send refuted(a) to parent; }
  wait 0;
}
```

Again, the agents in this computation form a graph. This time, there is no good word to describe the shape of the graph. Despite its apparently random structure, the graph-assembly takes place without difficulty. In other languages, one solution would be to assemble the graph using a hash-table of assertions as auxiliary storage. This would approximately triple the number of messages, as each edge in the graph is used once. Another option would be to statically map the assertions onto an array of objects, making the array into a graph-manager. Of course, this involves the usual reimplementation of the graph-manager code, and the resulting lack of clarity.

In this implementation, each agent is identified according to the assertion it is supposed to refute. Naming of agents according to their responsibility is one of the things that makes the algorithm so simple.

### 4.6 A Final Example: Matrix Multiplication

Our final example is a matrix multiplication subroutine. We assume the existence of a subroutine `vecmul` that multiplies a row with a column. The subroutine `matelt` is defined to represent a single element in the result matrix. The `matmul` subroutine expects one to feed in a number of `row` and

col messages, which causes it to respond by producing elt messages.

```
void matelt()
{
  vector r=nil; vector c=nil;
  handle row(vector x) { r=x; }
  handle col(vector x) { c=x; }
  wait (r != nil);
  wait (c != nil);
  return vecmul(r, c);
}

void matmul(int size)
{
  agent element(int r, int c) runs matelt();
  align element(int r, int c) to (c/2)+((r/2)*(size/2));
  handle row(int r, vector x)
    { send row(x) to element(r, 0..size-1); }
  handle col(int c, vector x)
    { send col(x) to element(0..size-1, c); }
  handle result(double d) from element(int r, int c)
    { send elt(r, c, d) to parent; }
}
```

In this example, the utility of data-driven output is clear: the element values simply emerge without prompting. There is no need to send request messages into the matmul subroutine: one need only send in the rows and columns. The results will come out as they are computed, and the results can be easily pipelined to a consumer. Because of agents forwarding policy and *relay handlers*, the matmul wrapper agent (which appears to be singlehandedly converting result messages into elt messages) is not a bottleneck.

In the align directive, we see an example of where having a vocabulary to describe groups of objects becomes extremely useful. This declaration makes it possible to explicitly state which processors should hold which agents. In this example, we specify a 2x2 blocking using the align declaration, more as an example than as an efficient thing to do. Note that the agents runtime system can also support dynamic load balancing, by allowing the user to map agents onto virtual processors, which can then be moved about the physical processors.

# 5 Planned Work

The agents language has been fully designed, a compiler has been written. The compiler is currently capable of handling only very simple programs. We estimate that another month or two will be required to make the compiler fully operational and bug-free on arbitrary agents programs.

Once the agents compiler itself is fully operational, we intend to begin testing the agents *construct* itself, attempting to find algorithms that cannot be easily encapsulated in agents. This may take some time, as we wish to assure ourselves that in fact, agents is a fully general encapsulation mechanism for concurrent computation. Therefore, we wish to examine a broad range of algorithms. We do not anticipate serious trouble during this phase, since a great many algorithms have already been implemented successfully on paper. However, we suspect that actual execution will reveal subtle points about these algorithms that we may not have noticed during pencil-and-paper evaluation. Therefore, we expect to encounter minor problems in the agents construct. We plan to make any necessary modifications or extensions to the design, modifying the compiler as well.

By the end of this phase, we expect the agents compiler to be a fully operational compiler, suitable not only for internal use, but also for use by other research groups and programmers in general. As a side effect of the first testing phase, we expect to have developed a fairly large library of agents. This library will include highly abstract control structures like the `replicatedstore` agent shown in a previous section. It will also include more concrete algorithms such as matrix operations.

The next phase will involve comparative evaluation of the agents construct. We will choose a number of small benchmark applications, coding them in each of several parallel languages, including agents. We will code each benchmark in the manner most appropriate for the language being used, subject to the condition that we will not change the underlying dataflow pattern that we wish to implement. For example, if we implement a tree-structured Fibonacci algorithm in one language, the Fibonacci algorithm in the other languages will also be tree-structured. Each program will be coded with the following priorities:

1. sacrifice encapsulation for efficiency where necessary
2. avoid unnecessary dependencies between modules
3. avoiding redundant expression
4. avoid using continuation-passing, especially in module interfaces

Once each program is done, we will count the number of times one of the criteria listed above had to be compromised. In addition, the programs will be evaluated using an appropriate normalized metric of size. Once the numerical evaluation is complete, the resulting information will be compiled into a summary of agents' relative strengths and weaknesses in terms of software engineering objectives, particularly encapsulation. These strengths and weaknesses will be explained in terms of the properties of the languages being tested.

While this paper emphasizes the importance of having an expressive vocabulary for identifying objects according to semantically meaningful names, the utility of such a vocabulary was demonstrated primarily in imperative code. We feel that a significant portion of the value of having a meaningful taxonomy over objects is the possibility of writing declarations about those objects. Therefore, we plan on experimenting with declarations, especially with those pertaining to load balancing: as an example, we anticipate a new form of the alignment directive that aligns agents to points in a Cartesian space, making it possible to load balance by shifting processor-boundaries within that space. We also intend to experiment with declarations that control priority, that manage grainsize,

and handle other tasks. These experimental declarations will be intended to exemplify the types of concepts that become expressible once entities are meaningfully labeled.

# 6 Related Work

The foundation of agents is its ability to assign semantically meaningful names to entities. Such names make it possible to express a wide variety of ideas that cannot be stated otherwise. The need for a vocabulary by which to identify data, and the related need to taxonomize those data, have also been recognized in languages like Linda[5] and D-Memo[15]. In such languages, each item of data is represented by a key (or by tag fields) which can be semantically meaningful. This simplifies the expression of programs. Linda's wildcards taxonomize the tuples into groups, those groups are useful for expressing such concepts as wildcard receives. We are not aware of any object-oriented languages in which the objects are labeled.

Agents, being an abstraction mechanism, focuses strongly on software engineering goals such as modularity and reuse. Early parallel programming models (e.g., pure message passing) did not focus very strongly on the issues of encapsulation, abstraction, and modularity. Currently, there are indications that level of awareness of these issues seems to be rising. Though there are not yet many papers being published on the subject of software engineering in the context of parallel programming, one sees frequent references to such issues within papers on other subjects. In addition, more and more researchers cite software engineering objectives as a goal of their language design efforts.

Much progress toward the goal of modularity and encapsulation has been made within the past several years. The adaptation of the object-oriented model to the parallel arena was a first step in this direction, and clearly represented an intention to achieve modularity within a parallel context. Among the first languages incorporating the object-oriented model was the Chare Kernel[12]. Almost immediately, it was clear that the object-oriented model contained bottlenecks, and this bottleneck was addressed by the addition of the *branchoffice* to the Chare Kernel, making it possible to create abstractions of distributed modules. Agents has taken a few more steps along the road toward true modularity and efficient encapsulation.

Agents, by labeling objects according to their caller/callee relationships, makes the statement "send to parent" meaningful. Thus, it supports true data-driven output. Though other models cannot provide data-driven output, many do provide single-demand-driven output, which is at least an approximation to data-driven. For example, in Fortran-M[4], after delivering one "channel endpoint" (much like a Unix pipe) to a thread, the thread can then write values into the channel. Similarly, in Id, one can pass a global pointer to an I-variable into a producer. The producer can then gradually fill the I-variable with a linked list. In both these approaches, the output channel is a bottleneck. One can eliminate the bottleneck by passing an array of channels into a Fortran-M procedure, or an I-array into an Id function. These approaches are somewhat complicated to use as one's standard module interface. Nonetheless they do achieve, if not data-driven output, then at least reasonably efficient single-demand-driven output.

Agents, by labeling objects, makes it possible to declare the entities which can potentially exist.

Thus, automatic graph creation is straightforward. Insofar as we are aware, the problem of efficient and elegant graph construction has not been addressed in a general manner in other models. The visual petri net languages like TDFL[16] strongly seem to suggest that easier graph construction was a primary design objective. However, though the idea seems to be there at an abstract level, it does not manifest itself in a very practical way, the visual petri net languages are no better at graph construction than most other languages. Some languages are recognizing the importance of the issue by providing primitives that construct specific kinds of graphs. For example, many parallel libraries now provide the reduction tree, and Charm recently added a primitive for building multidimensional arrays. We feel that this approach is limited, but useful.

Many researchers have asserted that continuation-passing style may not be desirable for general programming. For example, the Cid description[14] comments that "continuation-based programming seems quite tricky for humans. We believe that this is because it is difficult to modularize continuation-passing style, causing it to pervade the whole program." Agents makes a concerted effort to avoid the need for continuation-passing style. One major motivation for continuation-passing style is the need to obtain return-values from one's requests. Agents makes this effect possible without continuations using constructs like `fetch` on page22. The second major motivation for continuation-passing was the lack of data-driven output: one can use continuations as a means to achieve, if not data-driven output, then at least single-demand-driven output. The third major cause for continuation-passing style was interface inconsistency, leading to the need for continuations as glue. Agents attacks all three problems. Other language designers have also dealt with one facet of the continuation-problem: many have provided support for lightweight threads, making it possible to implement return-values on remote method invocations[14][1][2][6].

## References


[1] Robert D. Blumfoe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, California, July 1995.

[2] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. Technical Report Caltech-CS-TR-92-13, Department of Computer Science, California Institute of Technology, 1992.

[3] A. Chien and W. J. Dally. Concurrent aggregates. In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–196, March 1990.

[4] I. Foster and K. M. Chandy. Fortran m: A language for modular parallel programming. In *Journal of Parallel and Distributed Computing*, 1994.

[5] David Gelernter, Nicholas Carriero, S. Chandran, , and Silva Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, pages 255–263, Aug 1985.

[6] A. S. Grimshaw and J. W. Liu. Mentat: An object-oriented data-flow system. *Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference*, pages 35–47, October 1987.

[7] Attila Gursoy and L. V. Kale. Dagger: Combining benefits of synchronous and asynchronous communication styles. In H. G. Siegel, editor, *Proc. 8th International Parallel Processing*, pages 590–596, April 1994.

[8] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing 1994*, Washington D.C., Nov 1994.

[9] L. V. Kale and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proc. 27th Conference on Parallel Processing for Scientific Computing*, pages 738–743, February 1995.

[10] L. V. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[11] L. V. Kale, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[12] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, August 1990.

[13] L.V. Kale and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[14] Rishiyur S. Nikhil. Parallel Symbolic Computing in Cid. In *Parallel Symbolic Languages and Systems*, 1995.

[15] W. O'Connell, G. Thiruvathukal, and T. Christopher. Distributed Memo: A heterogenously distributed parallel software development environment. In *Proceedings of the 23rd International Conference on Parallel Processing*, Aug 1994.

[16] Paul A. Suhler, Jit Biswas, and Kim M. Korner. TDFL: A task-level data flow language. *Journal of Parallel and Distributed Computing*, 9(2), June 1990.

[17] J. Yelon and L. V. Kale. Agents : An undistorted representation of problem structure. In *Proceedings of the Conference on Languages and Compilers for Parallel Computing*, pages 551–565, August 1995.