STATIC NETWORKS OF OBJECTS
AS A TOOL FOR PARALLEL PROGRAMMING

Joshua Michael Yelon
Department of Computer Science
University of Illinois at Urbana-Champaign, 1998
Laxmikant V. Kalé, Advisor

A network of objects is a set of objects interconnected by pointers or the equivalent. In traditional languages, objects are allocated individually, and networks of objects are created incrementally. We call such networks *dynamic networks.* We define a new concept, the *static network*, which is a network of objects that is immutable. Entire static networks are created as a unit, atomically and instantaneously. We develop a language construct that allows the user to create static networks. Then, we explore the implications and applications of static networks. We discover that static networks have unique abilities not present in dynamic networks. This dissertation describes a number of ways to create static networks, and a number of ways to leverage them.

STATIC NETWORKS OF OBJECTS AS A TOOL FOR PARALLEL PROGRAMMING

BY

JOSHUA MICHAEL YELON

B.A., Rice University, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

ABSTRACT

A network of objects is a set of objects interconnected by pointers or the equivalent. In traditional languages, objects are allocated individually, and networks of objects are created incrementally. We call such networks *dynamic networks.* We define a new concept, the *static network*, which is a network of objects that is immutable. Entire static networks are created as a unit, atomically and instantaneously. We develop a language construct that allows the user to create static networks. Then, we explore the implications and applications of static networks. We discover that static networks have unique abilities not present in dynamic networks. This dissertation describes a number of ways to create static networks, and a number of ways to leverage them.

To Sasha, who will always be my little brother.

ACKNOWLEDGMENTS

To my parents and all my friends, many thanks for all your support and encouragement. Your words of wisdom kept me going when I was running out of stamina.

To my advisor and my research group members for putting up with my tendency to spout my opinions constantly, and my tendency to sleep during the daytime.

Special thanks to Dan Oblinger. Your enthusiasm for good software engineering helped keep me motivated in my work on the software engineering of parallel programs.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

AN INTRODUCTION TO STATIC NETWORKS

The ideas in this dissertation are founded on a simple observation. Many concurrent object-oriented programs follow three stages. First, they allocate a number of objects and quickly assemble them into a big network. Then, for a long time, they utilize the network without changing any pointers. Finally, they disassemble the entire network and free the objects. For the majority of the program's duration, the network is static, unchanging.

In existing languages, objects are created using dynamic allocation primitives. Networks of objects are assembled piece-by-piece: one allocates objects one at a time, and one links them together using pointers. Therefore, in existing languages, there is no such thing as a completely static network of objects. There is always an initial construction phase during which the network is being modified.

The fact that one can not create a genuinely static network is unfortunate. Immutable data structures typically have unique properties that do not hold for mutable networks. It is much easier to prove things about static data structures.

The objective of this dissertation is to develop ways to leverage the special properties of static networks. The intention is to be inclusive: we are interested in new compiler optimizations that apply to static networks, new language features that utilize static structural information, new load-balancing algorithms for static networks — basically, any advantage that can be gained from the immutability or predictability of static networks.

The first step, of course, is to develop a construct that lets us express static networks. Once we have done so, we can begin to explore all the various ways that static networks can be leveraged.

1.1 Static Networks Defined

In the above, we used an informal definition of static networks. This section defines the concept more precisely. But first, we need to invent some terminology.

If object A has a pointer to object B, then object A can easily invoke methods of object B. But in some languages, it is not always necessary for object A to have a pointer to B to invoke one of its methods. For example, if objects A and B are part of the same array, often they can contact each other by index.

We define a new term *linked to* to describe the potential for communication between two objects. If object A can invoke a method of object B without outside assistance, then we say that object A is *linked to* B. By outside assistance, we mean the help of a third object C. If object A can contact object B only with the help of another object C, then A is not *linked to* B. There are several common kinds of linkage: object A may have a pointer to B, or object A may have a pointer to an array in which B resides, or object A may have access to a global variable containing a pointer to B. In all of these situations, we say that A is *linked to* B. Alternatively, we say that *there exists a link* from A to B.

A *network* is a set of objects and a set of links between those objects. A *static network* is a network with these properties:

- The entire network, both objects and links, was created atomically.
- The entire network, both objects and links, can be destroyed atomically.
- No part of the network, object or link, can be destroyed independently.

In short, a static network is *immutable*. Its objects and its links cannot be destroyed during its existence, except to destroy the entire network as a unit. It should be noted it is *not* proscribed to alter the instance variables of an object in the network. Only destroying an object or one of the links is proscribed.

1.2 Static Networks in Current Languages

Many concurrent object-oriented languages can create one kind of static network: the array of objects. All elements of an array pop into existence at once. An array is also deallocated as a whole. Any member of the array can can invoke a method of any other, they are fully linked from the moment they come into existence. So an array meets the definition of a static network.

There is a specific kind of array that deserves particular mention: the *group*. A group is a one-dimensional distributed vector of homogeneous objects. What distinguishes a group from an ordinary array is that one can invoke methods on a group, as if it were an object. The invocation is routed to a single member of the group. The term *group* comes from the language HAL [1, 2, 3, 4], but similar constructs occur in several languages. For example, in Concurrent Aggregates [5, 6], the equivalent construct is called an *aggregate*. In the languages Charm and Charm++ [7, 8, 9, 10, 11, 12], there is a similar construct called a *branch office*. Throughout this dissertation, we use the term group generically to refer to all of these.

Current languages can also create an endless variety of networks using incremental construction. One can create trees, sparse matrices, graphs, a multitude of shapes and structures. One can create heterogeneous networks containing several classes of object. One can create large networks composed modularly of smaller networks. But none of these can be created atomically. They can only be built a piece at a time. Of all the thousands of kinds of possible networks, only the rectilinear, homogeneous array of objects can be created atomically. [1]

Our objective is to design a new construct that can build static networks of every shape and description. It should be able to create trees and graphs, lattices and grids. It should be able to mix objects of every type. It should be able to create every kind of network we can imagine. It should be modular, allowing us to compose small static networks into large ones. Once we have such a construct, we can begin to explore the possibilities of static networks.

1.3 Origin of Static Networks as a Concept

Static networks are closely related to the idea of pure functional programming. In pure functional programming languages, every data structure is immutable. Hence, every network of objects is a static network. Conversely, static networks cannot be created by imperative code — that would imply dynamic update of the network. Static networks can only be created by side-effect-free computation.

The absence of dynamic update in pure functional languages means that the only effect an expression has is to compute its value. This leads to referential transparency,

[1]This is not precisely true, but it is an adequate generalization for an introductory chapter. The qualifications will be noted in sections 5.3.3, 5.3.4, and chapter 7.

which enables the compiler to transparently replace expressions with equivalent expressions that evaluate to the same results.

The ability to substitute expressions with others that denote the same value represents a license to perform much symbolic manipulation. One can replace an expression with an expression that computes on demand. This makes lazy evaluation possible. One can transparently replace a value with a promise to compute that value (a future). This makes automatic parallelization possible. One can replace a value with an identical copy of that value. This makes automatic message-packing and unpacking possible.

Static networks also have a limited form of referential transparency. In particular, some expressions designate objects in the network and are side-effect free. These expressions can be transparently replaced with other expressions that denote the same object. We can replace an expression that returns a (static) object with a promise to look up the object. We can replace expressions that look up objects with simplified versions of those expressions. We can replace object pointers with symbolic names, and vice versa. We expect that the ability to perform symbolic manipulations will turn out to be as useful for us as it is for the pure functional language community.

Pure functional languages have one further property which deserves mention. In a pure functional language, the value of an identifier depends only on the explicitly-stated terms in the definition of that identifier, not on the state of the rest of the system. This is a vast simplification that makes it much easier for the compiler to predict the value of an identifier, or to predict the type of an identifier, or to predict other properties of that identifier. Proponents of pure functional languages argue that this property not only makes it easier to optimize these languages, but also to write programs in these languages.

Static networks have the same quality: the structure of a network depends only on the network's definition, not on the state of the system when the network is being built.

We expect this simplification to make it easier to predict and reason about the structure of a static network.

CHAPTER 2

DISTRIBUTED JAVA

This dissertation contains a great deal of sample code. To make the examples more readable, we choose a consistent programming language for all our examples. We also choose to use the same language for our prototype implementation. Specifically, we choose Java [13]. Remember that the dissertation is not about Java: it is about adding high-quality static network support to concurrent object-oriented languages in general. However, sticking with a specific language for demonstration and prototyping purposes makes our examples easier to understand.

It was necessary to add "the usual" concurrency constructs to Java to make it into a parallel programming language. This chapter describes the specific constructs that were added. Note that this does not include the static network support, which is described in chapter 4.

2.1 Why Java

It was necessary to choose an object oriented language for our code samples. There were many parallel object-oriented languages to choose from. Charm++ [11, 12], CC++ [14], HAL [3, 4], and Concurrent Aggregates [5, 6] were the obvious candidates.

The C++ based languages were rejected because it would have taken many years longer to implement the prototype compiler. Concurrent Aggregates was not selected because it uses a proprietary notation with which few people are familiar, it was felt that the odd notation would make the sample code hard to understand. HAL was not chosen because its idea of state replacement is significantly different from the ordinary (imperative) object-oriented programming model, we felt that for exposition purposes we should use something more conventional.

Having concluded that none of the usual parallel languages was appropriate, we reappraised the situation, and realized that we could just as easily start from a sequential programming language. Given that we were already adding static network support to an existing language, it was no problem to also add "the usual" concurrency constructs at the same time.

We selected Java as our foundation language because it is a very conventional object-oriented language with no strange features, and a very simple, familiar syntax. That makes it a good language for exposition.

2.2 Remote Objects

The first thing we needed to add to Java was support for remote objects. We added two clauses to the *new* operator allowing the user to put an object on a particular processor:

```
x = new classname(constructorarguments) on processor;
x = new classname(constructorarguments) remote;
```

The ordinary form of the *new* operator creates an object on the same processor as the object doing the creating. Using the *remote* clause causes the object to be placed on a processor of the system's choosing. Using the *on processor* clause causes the object to be placed on a processor of the user's choosing. One can transparently invoke methods on objects regardless of their placement, remote or local. Of course, there may be a performance difference for remote objects.

When invoking a method on a remote object, the parameters must be copied across processor boundaries. Normally, the callee receives copies of the parameters which are indistinguishable from the originals. If the parameters are of numeric types, this is easy. If the parameters are of object types, the callee receives a *proxy* object, a tiny object that masquerades as the original object. When one invokes a method on a proxy, the arguments are shipped to the original, the method is executed, and the return value is shipped back.

It is often desirable, when passing data across processor boundaries, to copy the data instead of creating a proxy. Therefore, we add a parameter-list keyword *copy* that allows the system to copy the parameter (in other words, provide the callee a value that is distinguishable from the original). The copying is done by calling user-supplied *pack* and *unpack* methods.

2.3 Threads

Java already has threads. For conciseness, we added a shorthand form of the thread creation mechanism. The new thread invokes a method on an object and then terminates. Its syntax is:

```
object<-method(arguments)
```

Of course, when object is a remote object, this is implemented efficiently: the thread is created remotely.

2.4 Atomicity and Synchronization

We are experimenting with a slightly unusual model of atomicity and synchronization. Unlike some concurrent object-oriented languages, multiple threads are allowed to execute within the same object at the same time. However, any contiguous sequence of statements is atomic unless it contains a *blocking* statement. There are two blocking statements: method invocation, and the *wait* statement. The *wait* statement is just a single keyword:

```
wait;
```

The wait statement simply suspends the current thread. The thread will resume immediately after any other thread invokes a method on the current object. In other words, it means "wait until some other thread modifies this object."

There is also a conditional version of the wait statement which is defined in terms of the simple version. The following two statements are equivalent:

```
wait condition;
while (not condition) { wait; }
```

Our experience with this atomicity and synchronization model is that it is fairly expressive, though it has its flaws. It is certainly adequate for our purposes, namely, the discussion of static networks.

2.5 Groups

Most concurrent object-oriented languages provide groups to help achieve concurrency. We will add groups to Distributed Java. The primitive to create a group is:

```
x = newgroup classname(constructorarguments);
x = newgroup classname(constructorarguments) [size];
```

This creates a group of the specified class. Each object in the group executes its constructor with the specified arguments. The number of objects in the group is specified by *size*. If *size* is not specified, the system creates one object per processor.

The group handle is indistinguishable from the handle of an object of the same class. Invoking a method on the group handle is equivalent to invoking a method on an arbitrarily-selected group member. In this regard, groups in Distributed Java function like groups in Concurrent Aggregates [5, 6] and HAL [3, 4]. Any member of a group may determine its position in the group by evaluating the pseudovariable *thisindex*. Any member of a group can obtain the group handle by evaluating the pseudovariable *thisgroup*. It is also possible to invoke a method on a specified group member, or all the group members, using these notations:

```
grouphandle[index].method(arguments);
grouphandle[index]<-method(arguments);
grouphandle[ALL]<-method(arguments);
```

2.6 Processor Numbers

Distributed Java provides two pseudovariables:

```
number_of_cpus
current_cpu
```

The first one allows the programmer to determine the total number of CPUs in the parallel machine. The second allows the programmer to determine which CPU is executing the code. These can be useful when trying to control the mapping of work to CPUs.

2.7 Constructors

Normally, the constructor of a class is named after the class. For example, the constructor of a class C is a method named C. However, for Distributed Java, we allow the constructor to be named *init*. We introduce this variant because our static network constructs will make it possible to define anonymous (unnamed) classes.

Normally, the scope of the constructor arguments is such that it only includes the constructor itself. We broaden the scope of the constructor arguments such that they are visible to all the code in the object. This may appear odd, but is no problem implementationally or semantically. The reasons for this modification will be made clear in section 4.2.

CHAPTER 3

NETWORKS WE WOULD LIKE TO EXPRESS STATICALLY

We now examine the question: what kind of programs could be converted to use static networks? This section's objective is twofold. First, we are looking for pseudostatic networks in use in existing programs. Second, we are interested in programs where the existing dynamic network could be replaced with a static network. We are not yet interested in asking what purpose will be served by such replacement. Instead, we are taking the point of view that static networks are inherently simpler than dynamic networks, and we will be able to find good use for that simplicity later. Later sections will answer the question "what does it buy us." For now, we are just looking for places where we can use static networks in our applications. In particular, we are interested in broad classes of applications where static networks can be used. We are also interested in the indicators that a program is a member of that class: obvious program characteristics that suggest that static networks could be applied.

3.1 Pattern Computation

The most obvious class of problems where static networks are potentially useful are the programs that already use a pseudostatic network. In these problems, the network is created early, and then the network serves as a "pattern" through which a lot of data flows. This section looks at some algorithms that use such "pattern computation."

A good example of pattern computation is the Jacobi method for solving a system of linear equations. In some implementations of Jacobi, the program starts by building a 2D grid of individual objects. Each object is given pointers to its four neighbors. Once the grid is there, the actual computation begins, and the objects pass numbers back and forth to their neighbors. The computation never changes any pointer in the grid.

Some languages provide distributed multidimensional arrays of objects as a language feature. Arrays, when provided by the system, have the defining characteristics of static networks. All elements are created together, atomically. They are all linked to each other. None can be deleted until they are all deleted. In short, whenever one sees a grid or matrix of objects, the problem is almost certainly using a static (or pseudostatic) network.

Reduction trees are another good example of pattern computation. It is typical to implement reductions by building a tree of objects, and then using the tree of objects to perform several reductions. This is a prototypical example of pattern computation. The network is created early, and then data flows through it.

3.2 Task Parallelism

This section describes a second class of applications that can be implemented with static networks. This class is characterized by task parallelism: an object is created to

```
class fib
{
  fib sub1, sub2; int result;
  int getresult() {
    wait (result!=NIL);
    return result;
  }
  void init(int n) {
    if (n<2) result=n;
    else {
      sub1 = new fib(n-1);
      sub2 = new fib(n-2);
      result = sub1.getresult() + sub2.getresult();
    }
  }
}
```

Figure 3.1 Task Parallel Program using Dynamic Network

perform a task, and the object is deleted when the task is done. Such problems can be trivially transformed into problems that use static networks. This section uses an example, parallel Fibonacci, which is prototypical of much task parallelism.

If one were to use an object-oriented parallel language to implement recursive Fibonacci, one would typically create a tree of objects to do the job. The code might appear as in figure 3.1.

The user of the fib object is supposed to create the object, which begins working immediately. When the user of the fib object needs the result, the user calls *getresult*, which blocks until the result is there, and then returns it.

The first thing the object's constructor (init) does is to create two children: one to compute fib(n-1), the other to compute fib(n-2). The children start working independently. The parent fetches both results, adds them, and stores its own result.

```
class fib
{
  fib sub1, sub2; int result;
  int getresult() {
    wait (result!=NIL);
    return result;
  }
  void calc(int n) {
    if (n<2) result=n;
    else {
      sub1<-calc(n-1);
      sub2<-calc(n-2);
      result = sub1.getresult() + sub2.getresult();
    }
  }
}
```

Figure 3.2 Task Parallel Program using Static Network

If you do not include any garbage collection code in the program, you end up with a large tree of objects that reflects all the tasks that were created. Suppose that somehow, you could create that entire tree in advance. In that case, you could remove the object-creation code from the fib object, since the objects are already there. The resulting code is shown in figure 3.2.

This code differs from its predecessor in that it assumes that the tree is already there: it assumes that *sub1* and *sub2* are already initialized (somehow). We have removed the constructor and replaced it with a *calc* method that spurs the object into action. The calc method no longer creates *sub1* and *sub2*, instead, it just activates *sub1* and *sub2*. As you can see, the differences between the two versions of the code are minimal.

The code transformation we just performed on Fibonacci can be performed on any task-parallel program. Whenever one has a tree of objects where an object is created to perform a task, and is deleted when the task is done, one performs these steps:

- 1. Remove the object creation code from the program.
- 2. Pre-create the entire tree of objects statically.

Of course, step 2 is easier said than done. We need to be able to build large trees of objects, instantaneously. Those trees of objects need to be capable of representing the task tree, even when it is irregular or contains heterogeneous objects. *If* we can somehow pull off this static tree-creation, then the rest is easy: any task-parallel program can effortlessly be transformed into a program using static networks. In sections 4.1 through 4.7, we will show how this is possible.

3.3 Graph Exploration

This section shows another class of problems which can be implemented using static networks. The class of problems is characterized by "expanding graphs." These programs start by creating a small graph data-structure, and then they expand that structure a piece at a time. Many such programs could be implemented with static networks. This section shows an example, a resolution theorem prover.

In a resolution theorem prover, the input is a database of postulated assertions, and a goal assertion that should be *disproved.* The theorem prover proceeds by combining assertions pairwise, generating new assertions. If an assertion is ever generated that is obviously false, then the original input is disproved. If you keep a record of which pairs of assertions were combined and which assertions were generated, the result is a directed graph known as the derivation graph, which might appear as in figure 3.3.

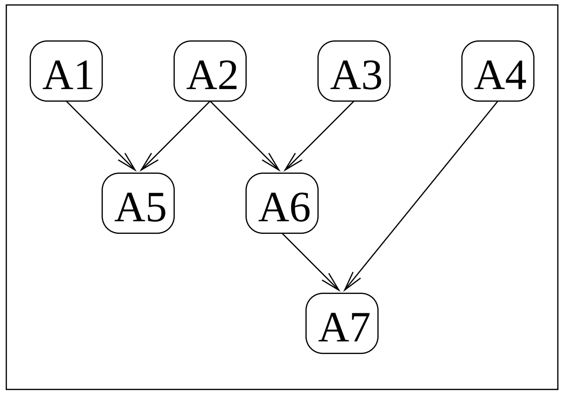

Figure 3.3 A Graph-Exploration Problem: Theorem Proving

If you had infinite computational resources, you could combine every possible pair of assertions, and derive every assertion that could possibly be derived. The resulting derivation graph would be "complete." The complete derivation graph is a function only of the input assertions. The complete derivation graph is very useful, as a concept: most proofs about the effectiveness of mechanical theorem-proving reason about the complete derivation graph. Of course, the complete derivation graph is usually infinitely large, so you can not *usually* build it as a data-structure. However, in a few lazy functional languages, the complete derivation graph can actually be built, even if it is infinite. Of course, that data structure is a static network. Therefore, in the lazy-functional version of this problem, one could start by defining (instantaneously) this large static network, and then the actual work is to explore it.

Other programming languages cannot express the complete derivation graph as a data-structure. Instead, one typically builds an initial subset of the derivation graph, and continuously expands it. Naturally, this data-structure *is not* a static network. But it still has several characteristics that should lead us to think "potential static network." In particular, whenever we see a graph that is continuously growing, or where the bulk of the work is building a graph, the problem can almost certainly be expressed using static networks.

3.4 Data-Parallel Fortran: A Static-Network Language

Some parallel languages are actually only capable of expressing static networks. One of the most important of these is data parallel Fortran 77, e.g., Fortran-D [15] and HPF [16]. Data parallel Fortran focuses on grid computation, which is almost always static-network computation. Data parallel Fortran can do a little more than just grid computation. Despite this, we will show that all data-parallel Fortran 77 programs can use static networks.

Of course, to do this, we have to loosen our terms a little. We will assume that numeric variables are actually small objects — this allows our definition of static networks as networks of objects to apply here. We will also assume that all the array elements are linked, since data can move arbitrarily from any one to any other. With these assumptions, Fortran arrays satisfy our definition of static networks. It is only a small step to see that the entire set of common variables in a Fortran program, taken as a set, also constitute one large static network.

In Fortran (as in most languages), when you call a function, you create a set of local variables. Those local variables also constitute a static network: they are created instantaneously, the set of local storage locations is immutable once created, and the connectivity between the locations is fixed (data can move from any location to any other).

Fortran 77 is capable of creating a limited kind of dynamic network. When you invoke a function, you are creating new storage locations. This function can call other functions, which in turn creates more locations, and so forth. This is a limited form of dynamic network creation. However, this style of dynamic network is covered thoroughly in the section on task-parallelism. As that section showed, this sort of tree-structured task creation and deletion can easily be transformed into a static-network computation.

The conclusion we can draw is this: any program written in data-parallel Fortran 77, no matter what the code, can be expressed with static networks only. The only transformation needed will be the simple one described in the previous section.

CHAPTER 4

A CONSTRUCT FOR STATIC NETWORKS

In chapter 3, we identified three classes of problems that can be conceivably be expressed with static networks: the pattern computations, the task parallel problems, and the graph exploration problems. Now we need an actual language construct to create static networks. We designed a construct for this purpose, the *agent* construct.

This chapter is a walk-through of the engineering process that we went through in creating the *agent* construct. We start with a simple design, which turns out to be flawed. We evaluate that design in terms of our objective, and modify it incrementally until it achieves our goals. We end up with a construct that can express all three classes of static-network algorithms. All constructs will be shown as they would appear in Distributed Java, though they could be added to any object-oriented parallel language.

4.1 The Type System

Most programming languages already have a language for describing ordinary (dynamic) networks of objects: the type system. The code in figure 4.1 shows a set of class declarations. These type definitions *constrain* a network of objects. Given these class declarations, we could build several networks. We could build an object of class A point-

class A	class B

<pre>class A
{
  B sub1;
  B sub2;
};</pre>
	<pre>class B
	{
	  int x;
	};</pre>

Figure 4.1 Class Declarations Describing Small Dynamic Network

ing to two objects of class B. We could build an object of class A with two pointers to a single object of class B. We could build an object of class B with nothing else. We could build an object of class A with two null pointers. And so forth. But we can not build an object of class B containing a pointer to an object of class A. In short, a type system defines a *space* of possible networks.

So, the starting point when defining a notation for static networks is going to be the type system. We will modify the type system, making it possible to express the idea of static networks of objects.

4.2 We Add Immutable Instance Variables

The basic property of static networks is that they are immutable. If we create objects with immutable fields, and those objects are linked to more objects with immutable fields, and so forth, then the entire network will be static. The obvious place for us to start, then, is to add a keyword *immutable* that can be placed in front of an instance variable declaration.

Clearly, if an instance variable is immutable, that variable must have an initial value. So our next step is to add initializer expressions. We will allow the declaration of the

class A	class B
<code>{</code> <code> immutable B sub1 = new B;</code> <code> immutable B sub2 = new B;</code> <code>};</code>	<code>{</code> <code> int x;</code> <code>};</code>

Figure 4.2 Class Declarations Describing Small Static Network

instance variable to be followed by an equal sign, then an expression that computes an initial value for that variable. Figure 4.2 shows some code using this combination.

Whenever an object of class A is allocated, two objects of class B are automatically allocated and stored in the instance variables *sub1* and *sub2*. From then on, the variables *sub1* and *sub2* cannot be modified. So, when you allocate a single object of class A, you have allocated an entire static network consisting of three objects.

Next, we need to make it possible to parameterize the initialization expressions. Without this ability, it is difficult to create static networks of varying sizes. To achieve this end without adding any complexity to the language, we allow all the code in the class including the initialization expressions to refer to the constructor arguments. This may appear odd, but it does not create any semantic or implementational problems. Initialization expressions are evaluated before the constructor executes.

It should be fairly clear that this design can only create trees of objects. In section 3.2, we showed how any strictly task-parallel problem could be converted into a static-network problem by preallocating the task-tree. That suggests that the construct will be useful for expressing task parallelism statically. It is not clear whether it will be useful for the other kinds of problems we identified in chapter 3. We will continue to focus on task parallelism for now, we will return to the other kinds of problems in section 4.9.

```
class fib
{
  fib sub1, sub2; int result;
  int getresult() {
    wait (result!=NIL);
    return result;
  }
  void init(int n) {
    if (n<2) result=n;
    else {
      sub1 = new fib(n-1);
      sub2 = new fib(n-2);
      result = sub1.getresult() + sub2.getresult();
    }
  }
}
```

Figure 4.3 Fibonacci Code using Dynamic Network

We will demonstrate the construct using a prototypical example of task-parallel computation: Fibonacci. Admittedly, it is not a challenging problem, but it is sufficiently representative of its category (the task-parallel problems) that following it leads to a solution that's effective for all task-parallel problems. Recall the parallel code that uses a dynamic network (figure 4.3), building a tree of objects as it computes. If we do not add any garbage collection code, the tree of objects will still be there after the computation.

We can convert the entire tree into a static network. We build the tree instantaneously before starting the Fibonacci computation, then we let the Fibonacci computation simply propagate numbers up and down the preexisting, unchanging tree. The result will be inelegant and inefficient in many ways, but at least it will use a static network. We will fix the problems later. Figure 4.4 shows how to build the static network for Fibonacci, using the *immutable* keyword and initialization expressions. As you can see, the initial-

```
class fib
{
  immutable fib sub1 = IF (n<2) THEN nil ELSE new fib(n-1);
  immutable fib sub2 = IF (n<2) THEN nil ELSE new fib(n-2);
  int result;
  int getresult() {
    wait (result!=NIL);
    return result;
  }
  void init(int n) {
    if (n<2) result=n;
    else {
      result = sub1.getresult() + sub2.getresult();
    }
  }
}
```

Figure 4.4 Fibonacci Code using Static Network

ization expressions contain the variable n, this is a reference to the parameter list of the constructor.

Allocating an object using *new fib(n)* for some large N will trigger the allocation of two more objects, each of which in turn will trigger two more objects, and so forth. An entire tree will form in response to that *new* operator. To the user, the appearance will be that the network forms instantaneously and is immutable — a static network. It is exactly the static network we need to perform the Fibonacci computation. Once the tree is built, the constructors all start executing, and Fibonacci is computed.

There are several obvious problems in this design. The next several sections will address the most glaring of these. This will lead to a gradual refinement of our static-network constructs.

4.3 We Add Array Initializers

The first problem with the Fibonacci code is the redundancy in the initialization expressions. We declare two instance variables, *sub1* and *sub2*. Then we initialize both of them in the same way. The code is very repetitive.

If we could create initialized arrays, we could have cleaned this up. We could have declared a two-element array *sub*. We could then have written an initialization expression for the elements of *sub*. Of course, we do not have a syntax to initialize array elements yet. So, let us create one:

```
immutable fib sub[2] where sub[i]=IF n<2 THEN nil ELSE new fib(n-1-i);
```

The first part (before the *where*) declares an array of two fib objects. The part after the *where* declares the value of each array element. This eliminates one bit of redundancy from the problem.

4.4 We add Lazy Evaluation

There is still a lot of redundancy. The problem is, the initialization expressions almost contain the entire problem. They evaluate the comparison $(n < 2)$. They perform a conditional branch. They evaluate the expressions $(n - 1)$ and $(n - 2)$. Essentially 90% of the work of Fibonacci is being performed in the initialization expressions. We almost had to compute Fibonacci just to build the network. Then, we used the network to compute Fibonacci. In doing so, we even had to redo some of the work, like computing $(n < 2)$ again. This is clearly redundant.

Unfortunately, this kind of duplication is inevitable given the constructs we have provided. The initialization expressions are trying to build *exactly* the tree which is needed by the Fibonacci program. To build the tree exactly, you have to understand

```
class fib
{
  immutable fib sub[2] where sub[i] = new fib(n-1-i);
  int result;
  int getresult() {
    wait (result!=NIL);
    return result;
  }
  void init(int n) {
    if (n<2) result=n;
    else {
      result = sub1.getresult() + sub2.getresult();
    }
  }
}
```

Figure 4.5 Fibonacci, using *immutable* Arrays

a lot about Fibonacci — which inevitably means executing a significant portion of the Fibonacci code.

The solution to this is to realize that we do not need a perfect fit. If we simply build a big enough tree of *fib* objects, it will suffice for the Fibonacci computation. That, by itself, does not lead to much simplification. But when you combine it with lazy evaluation of initialization expressions, the initialization expressions become much simpler. Figure 4.5 shows the effect of this alteration on the initialization expressions in Fibonacci.

When you allocate a Fibonacci object, its initialization expressions allocate two more, which in turn allocate two more, and so forth. So we have created an infinitely large tree of Fibonacci objects. With the lazy evaluation, though, only the parts we really need get allocated. The lazy evaluation eliminates the need to perfectly "fit" the tree to the Fibonacci computation. As long as it is a superset of what's needed, you are fine. Note that there is no more redundancy: each of the expressions $(n < 2)$, $(n - 1)$, and $(n - 2)$ and the conditional branch are only evaluated once per object.

Earlier we mentioned that initialization expressions are evaluated before the constructor executes. To make lazy evaluation work, we have to add a few more constraints on the initialization expressions. Recall that we want to present the illusion that the entire network "pops into existence" the instant the allocation is performed. To create this effect with lazy evaluation, the initialization expressions need to be side-effect free. They also can not refer to the state of the object or to external global state. They can still refer to constructor arguments, since those are specified simultaneously with the creation of the network.

So now we have a construct that lets us create static networks in the form of task-trees, and lets us do it without redundant computation.

4.5 We Condense the Syntax

So far, we have developed a construct that lets us express static task-trees. When we use it for *fib*, we write this:

Declaration 1: `immutable fib sub[2] where sub[i] = new fib(n-1-i);`

It is a verbose notation. It can be tightened up considerably by asking "what does it tell us, what information does it contain?" We can enumerate the pieces of information that are in that declaration:

- The names of the instance variables are *sub[i]*.
- The value of *sub[i]* will be a new *fib(n-1-i)*.
- The range of *i* is 0 to 1

The third piece of information is not even important, given lazy evaluation: we can easily ignore array bounds. That leaves only two important pieces of information in that long declaration. Stated that way, it is obvious that we can say the same thing in a much more concise notation. We will make one up now:

Declaration 2: `agent sub(int i) is fib(n-1-i);`

Declaration 2 means almost exactly the same thing as Declaration 1. It declares an array *sub[i]*, and initializes the values of *sub[i]* to new objects *fib(n-1-i)*. This notation is much shorter than the previous one. It differs only in that it does not specify an array bounds. In fact, there is no place to specify an array bounds, all arrays created with the agent declaration are unbounded. This is not a big limitation, given lazy evaluation. The *new* which is explicitly present in declaration 1 is implicit in declaration 2. That means that with the *agent* notation, the initialized value must be a newly-allocated object. This has pros and cons, which will be discussed later. The final version of Fibonacci, using a static network built by the agent declaration, is shown in figure 4.6.

Variants of the *agent* construct could be added to any object-oriented programming language. When we add it to Java, we choose this syntax:

```
agentdecl ::= AGENT name ( formals ) IS classname ( actuals ) ;
```

Agent declarations can occur at the top level of a class declaration, along with the instance variables and method declarations.

4.6 Sending Messages to Agents

Evaluating an agent's name, including the parameters, conceptually returns a pointer to the agent. As far as the user of the language can tell, it does return a pointer to the agent.

```
class fib
{
  agent fib(int i) is fib(n-1-i);
  int result;
  int getresult() {
    wait (result!=NIL);
    return result;
  }
  void init(int n) {
    if (n<2) result=n;
    else {
      result = sub1.getresult() + sub2.getresult();
    }
  }
}
```

Figure 4.6 Fibonacci, using the Agent Declaration

Implementationally speaking, it actually returns a proxy of the agent. The proxy is an object that pretends to be the real agent. When one invokes a method on the proxy object, the proxy arranges that the arguments be forwarded to the real object, that the method be called, and that the return-value be shipped back. The net result is that the proxy gives a convincing illusion of being the actual object, invoking a method on the proxy has exactly the same effect as invoking a method on the real thing.

So, to send a message to an agent, all you have to do is to write an expression like one of the following:

```
agentname(index1, index2, ...).method(arg1, arg2, ...);
agentname(index1, index2, ...)<-method(arg1, arg2, ...);
```

The first notation is old-fashioned method invocation. It can be used across processor boundaries. The latter notation is for asynchronous method invocation, which can also be used across processor boundaries. The asynchronous version creates a new thread to invoke the method. The first notation returns a value (whatever the method returns). The latter notation does not return anything. The latter notation is usually more efficient, since it does not require the communication of a return-value.

It is also possible to send a message to a set of agents at the same time. To do this, one evaluates the name of the agent, with a "subrange" in one or more of the parameter positions:

```
agentname(index1lo..index1hi, index2, ...)<-method(arg1, arg2, ...);
```

The subrange denotes a range of integer indices. This approach can only be used when the parameter in question is of type *integer*. This is called multicasting, and the proxy it uses is called a "multicast proxy."

4.7 Awakening Agents

When you allocate a single object, and that object has agents, and those agents have agents, and so forth, you are allocating an entire agent hierarchy at once. Because of lazy evaluation, the hierarchy could be infinite.

Obviously, we cannot execute an infinite number of constructors at the instant the static network is allocated. This raises a tricky question: when do the constructors get executed? We use the following rule: when an object is allocated by an agent declaration, that object is "asleep." It does not execute its constructor. Any attempt to access the object (e.g., invoke a method) wakes it up. When the object wakes up, the first thing it does is execute its constructor. After that, it opens itself to method invocations.

Usually, this gives exactly the right behavior. The objects execute their constructors right before their first method invocation. But sometimes, there is no method invocation to be done. Sometimes, the object just needs to wake up without any method invocation. For that, we provide a simple construct:

```
class fib
{
  immutable fib sub1 = new fib(n-1);
  immutable fib sub2 = new fib(n-2);
  ...
}
```

```
class fib
{
  agent sub1() is fib(n-1);
  agent sub2() is fib(n-2);
  ...
}
```

Figure 4.7 The Fib Network: two Equivalent Views

```
awaken <object>
```

This merely causes the object to wake up and execute its constructor.

4.8 Why the Construct is Named "Agent"

Figure 4.7 shows yet another way of coding the static network of the *fib* class, this time, without the arrays, but still using *immutable* and lazy evaluation. It shows both the old syntax, and the new (equivalent) *agent* syntax.

While the two notations mean exactly the same thing, the two notations have two different psychological impacts. The old notation (*immutable*) seems to stress operational semantics. It appears to say, "whenever you allocate an object of class *fib*, allocate two more objects of class *fib*. Put them in the instance variables *sub1* and *sub2*." The new syntax (*agent*) seems to stress the end result. It seems to say, "Every *fib* object will have two agents working for it, both of class *fib*, named *sub1* and *sub2*." These two viewpoints are technically equivalent, but taking the latter viewpoint will help one to understand the intuitions in the rest of this paper.

The *agent* construct was designed using task parallel problems as a motivator. Fibonacci is a prototypical example. In all task parallel problems, there is a task tree. The child is always working for the parent. When a *fib* object allocates two children *sub1* and *sub2*, the two children are there to help the parent do its job. This is why we call the construct "agent" — it emphasizes the fact that the construct creates a task tree, a tree where the parents are in charge of the children, and the children are agents of the parents. We feel that the *agent* notation is better not just because it is shorter, but because it emphasizes the owner/agent relationship between the objects.

4.9 Using Non-Tree Communication Patterns

We have been designing a construct around the problem of representing task parallelism in a static manner. It remains to be shown whether or not our constructs are useful for expressing graph parallelism and pattern computation statically. Clearly, to make this possible, we have to allow non-tree communication patterns, despite the fact that agents form trees.

4.10 Superimposing a Grid on a Tree of Agents

Consider the declaration in figure 4.8. It declares the class *jacobi*. If the program uses values of X in the range 0-99, and values of Y in the range 0-99, then each *jacobi* object will access 10,000 children. Each *jacobi* is the root of a very shallow task tree: just one level deep, with a branching factor of 10,000.

Suppose, hypothetically, that we provide a means for the 10,000 children of the *jacobi* object to send messages directly to their siblings. We program *jnode(i, j)* to send messages to *jnode(i+1, j), jnode(i-1, j), jnode(i, j+1), and jnode(i, j-1).* The objects still form a tree. But the siblings of the *jacobi* object also form a grid, at least as far as communication patterns are concerned. In short, agents always form trees. But those trees can have other

```
class jacobi
{
  agent jnode(int x, int y) is jacobinode();
  ...
}
```

Figure 4.8 One Object with 10,000 Agents

structures superimposed upon them by the communication patterns. Those patterns can be completely unlike trees.

The ability to superimpose non-tree communication patterns on the agents task tree is critical. Without it, agents are nothing but a means to represent task parallelism. But if one can superimpose arbitrary communication patterns on top of the tree, then agents become much more flexible. To achieve this, we must support communication in all directions: parent to child, child to parent, and sibling to sibling.

4.11 A Notation for Sending Data Across the Tree

We need a notation to send data up, down, and across the agent hierarchy. The notation to send data down the hierarchy is obvious. An object can simply refer to its agents by name. For an object to send a message to one of its agents, it merely says:

```
agentname(indices).method(arguments)
```

We do not have a notation for sending messages up the hierarchy. But, it is straightforward to add one. We will add a keyword *owner* to the language that evaluates to the agent's owner. With such an addition, an agent can send data to its owner in the hierarchy by simply saying:

```
owner.method(arguments)
```

Sending data from an agent to one of its siblings now has a relatively obvious syntax. C++ allows one to "reach into" an object and extract the value of one of its instance variables using the notation *object.instvarname.* Of course, this is restricted, if the instance variable is declared private then this operation is forbidden. But if it is public, we can use this notation for an agent to send a message to one of its siblings:

```
owner.agentname(indices).method(arguments)
```

But, as we stated, this is only legal if the agent is declared public. But what if it is not? In that case, this notation breaks the scoping rules of the language. Requiring that all agents be declared public would be a bad idea. This sort of indiscriminate "reaching into" another object's scope is obviously a gross violation of encapsulation rules, we should not require it.

This is clearly a problem for the *jacobinode*s. If they cannot send data to each other, they cannot communicate in a grid. The notational problem is that the declaration of the *jacobinode*s is inside the scope of the *jacobi* object. Therefore, according to the scoping rules, only the *jacobi* object can see the *jacobinodes.* To send a message to a *jacobinode*, it follows that one must go through the *jacobi* object. We will do exactly that. When a *jacobinode* wishes to send data to one of its siblings, it sends the data first to the *jacobi* object, using the *owner* construct. The *jacobi* object then relays the data back to a *jacobinode.* This gives us a way to get data from sibling to sibling without breaking the scoping rules.

Notationally, this works, but implementationally, it has a speed problem. The *jacobi* object becomes a bottleneck. To get rid of the bottleneck, we are going to *require* that our compiler perform the "relay optimization." The relay optimization is this: when object A invokes a method of an object B in the same static network, and that method in turn invokes a method of an object C in the same static network, and no instance variables of B are accessed, the compiler *must* bypass object B: it must not access object

B in any way. In other words, it should send the data directly from A to C. This is a required optimization, like tail-call optimization in Scheme. The language will not be useful unless the compiler does this every time.

The relay optimization enables data to flow agent-to-owner-to-agent at the language level, but in fact go directly from agent-to-agent at the implementation level. In other words, it lets siblings in the agent hierarchy communicate efficiently, without breaking the scoping rules of the language.

4.12 The Relay Optimization is Implementable

Obviously, the effectiveness of this approach depends on our being able to implement the relay optimization. This section demonstrates that it is possible. However, to do so, we must use some unconventional data structures.

The obvious way to build any tree-like data structure is to use child-pointers and parent-pointers. But if we use this strategy to implement the tree of agents, we will not be able to implement the relay optimization. Suppose, hypothetically, that we build the Jacobi hierarchy using parent and child pointers. The *jacobi* object would have pointers to all its children, the *jacobinode*s. Each *jacobinode* would all have a pointer to its parent, the *jacobi* object. Now suppose that a *jacobinode*s sends a message to the *jacobi* object, and the *jacobi* forwards it to a *jacobinode.* The compiler cannot short-circuit this and do a direct *jacobinode-to-jacobinode* transmission. It can not, because the *jacobinode*s do no have pointers to each other. The compiler cannot realistically predict which objects will send to which others, so it cannot reasonably set up such sibling-to-sibling pointers.

We need a way for the compiler to send a message from one sibling to another, without the bottleneck, even if the siblings do not have pointers to each other. A completely different data structure is required. We now describe such a data structure.

`class C`
`{`
`  agent A(int n) is D;`
`  void print_nth(int n) { A(n).print(); }`
`}`
	`X = new C;`
	`X.print_nth(4);`

Figure 4.9 Agent ID Sample Code

The runtime system assigns each object an internal ID, which is a string. There are two disjoint kinds of objects: those allocated with *new*, and those which are agents. The runtime system makes up the identifiers according to the following rules:

ID Rule 1. The ID of any object allocated with *new* is of the form *classname #seqno@processor*. The *seqno* is an arbitrarily-generated unique number to distinguish objects from each other. The *processor* number identifies the location of the object.

ID Rule 2. The ID of any object which is an agent is of the form *owner.name(args)*, where *owner* is the agent's owner's ID, *name* is the agent's name (as it appears in the source code), and *args* are the agent's arguments (indices).

Allow us to show these rules in action. Consider the class declaration shown in figure 4.9, and the short excerpt of some executable code to its right.

When the first statement executes, an object is created using *new*. According to *ID Rule 1*, the object's ID is of the form *classname#seqno@processor*. Suppose, for example, that this code executes on processor 5. The identifier for the object stored in X might be *C#12345@5*.

When the second statement executes, the lazy allocation mechanisms of agents activate and an agent is allocated. According to *ID Rule 2*, this object has the internal identifier *C#12345@5.A(4)*.

These rules are designed so that each part of the static network has a unique identifier which is predictable, and the identifier of one part is straightforwardly related to the identifiers of the parts near it. It is easy for an object to compute the ID of its owner, of one of its children, or one of its siblings.

Note that the IDs are hidden entirely within the runtime system, the user is never aware of their existence. When the user refers to his *owner* or to one of his agents, he is implicitly manipulating agent IDs. However, the user will not be directly aware of it.

We define a mapping function from IDs to processors. The compiler emits this mapping function. The runtime system obeys the mapping function: when it is about to allocate an agent, it first computes the agent's ID. It maps the ID to a processor number. It sends a message to that processor, demanding that it allocate the agent. The net effect is that an agent is always on the processor specified by the mapping function. Given an ID, you can mathematically predict which processor the agent will be on.

One can therefore send a message to an object, even if one does not have a pointer to the object. All one needs is the object's ID. One simply maps the ID to a processor number, finding out which processor the object should be on. One forwards the message to that processor. Each processor maintains a hash table containing all of its objects. So when the message arrives, the processor merely looks in the hash table for the specified ID. If the object is not there, lazy allocation happens at this time. The processor allocates the object, and puts it in the hash table. Then, the message is delivered.

Since one can send a message using only an ID, this makes it easy for an agent to send a message to its parent, child, sibling, or in fact any relative. It starts by obtaining its own ID. It can then perform string-manipulation on that ID, following *ID Rule 2*, to obtain the ID of a relative. It can then send a message using only the ID. In short, an object can send a message to any of its relatives, without having a pointer to the relative. This ability will be used by the compiler, to implement the relay optimization.

The ID-to-processor mapping function could be replaced with a distributed directory that maps IDs to processors. This would enable us to choose object placement at runtime, and migrate objects. On the other hand, it would add some overhead, as the directory lookups take some time. There are a few extant systems that use directories to map object names to objects [17, 18]. The best approach, in practice, is probably a hybrid approach, using the mapping function for some objects, and a directory for others. For concreteness, the rest of this dissertation assumes the use of a mapping function.

Evaluating an agent name and parameters generates a proxy of the agent. The proxy is a small object containing only three fields: a virtual function table (like all other objects), the agent ID of the real object, and the processor number of the real object. The proxy is generated in four steps. First, the proxy object is allocated. Its virtual function table is initialized according to the appropriate typing rules. The agent ID is determined by string manipulation and filled in. Finally, the processor number is filled in by applying the ID to processor mapping function to the agent ID.

4.13 The Jacobi Example, using Agent IDs

We will now show the relay optimization at work with the Jacobi example. We start with the assumption that every iteration cycle, each *jacobinode* object emits some data to its owner, the *jacobi* object. The *jacobi* object relays that data to the four neighbors of the *jacobinode* object. In effect, each *jacobinode* objects is communicating with its four neighbors, using the *jacobi* object as an intermediary. The code, as it might appear in the program, is in figure 4.10.

The implementation of this code would proceed as follows. Suppose, hypothetically, that the user allocates the *jacobi* object using *new* on processor zero. The *jacobi* object might be assigned the ID *jacobi#567@0*, according to *ID Rule 1.* The IDs of the *jacobin-*

```
class jacobi
{
  agent jnode(int i, int j)
     is jacobinode();
  /* The following method can */
  /* be used to send a piece  */
  /* of data to four jnodes.  */
  /* It is used by the jnodes */
  /* to send data to each     */
  /* other.                   */
  void data(int x,int y,
            double v){
    jnode(x+1,y).data(v);
    jnode(x-1,y).data(v);
    jnode(x,y+1).data(v);
    jnode(x,y-1).data(v);
  }
  /* insert other methods here*/
}
```

```
class jacobinode
{
  int my_x, my_y;
  void calculate(...) {
    /* insert calculation code */
    /* send results to neighbors */
    owner.data(my_x, my_y, value);
    ...
  }
  /* insert other methods here */
}
```

Figure 4.10 Jacobinode Sends Data to Owner, which Relays it to Jacobinodes

odes are *jacobi#567@0.jnode(0,0)*, *jacobi#567@0.jnode(0,1)*, and so forth, according to *ID Rule 2*.

Suppose that *jnode(5,5)* invokes the *data* method on its owner. This, in turn, causes the owner to invoke the *data* method on *jnode(6,5), jnode(4,5), jnode(5,6),* and *jnode(5,4)*. This case fits the conditions of the relay optimization: object A sends a message to object B, which causes object B to send a message to object C, with no reference to B's instance variables. The compiler optimizes it. The compiler implements the method invocation as follows. First, the *jacobinode* object obtains its own ID: *jacobi#567@0.jnode(5,5)*. It truncates everything after the last dot, yielding *jacobi#567@0*, the ID of the *jacobi* object. Then, it does what the *jacobi* object, its owner, would have done: it concatenates *.jnode(6,5)* to the ID, yielding *jacobi #567@0.jnode(6,5)*, the ID of its sibling. It then sends a message directly to its sibling, using the ID. It continues doing what its owner would have done, sending messages to all four of its neighbors in this way.

The fact that an object can short-circuit the task tree and send messages to any relative makes it possible to superimpose arbitrary communication patterns on the agent tree. The *jacobi* example above uses a grid-structured communication pattern, for example.

As shown in section 3, static networks are potentially useful in three major categories of applications: task parallelism, pattern computation, and graph exploration. Agents were obviously designed for task parallelism. But the fact that one can superimpose arbitrary communication patterns on top of the task tree makes them well-suited for pattern computation and graph exploration problems as well.

4.14 Agents Positions and the From Clause

In a traditional object-oriented system, aliasing can occur: there can be two pointers to one object. But in agents, aliasing does not occur: every object has a unique position within the agent hierarchy.

Because of this, it is meaningful to talk about *the* position of an agent within its hierarchy. We emphasize the word "the" because in a traditional language, there could be many pointers to any given object, but in agents, an object only occurs once in its hierarchy. In agents, we can point to an object and ask "what is its position in its hierarchy."

For example, consider the ID *jacobi#567@0.jnode(0,0)*. The runtime system could easily tell you that the object is of type *jacobinode*, and that it is working for the object *jacobi#567@0*. If the programmer were to look at the ID, he could also tell you that the object's job is to compute the (0,0) element of the jacobi matrix belonging to that particular jacobi object. In general, positional information also gives you information about the purpose and responsibilities of the object. In a sense, when you ask "what is the position of this object in its hierarchy," you are asking "what is the role of this object in the overall computation."

The ability to ask this question is only possible because agents networks are static hierarchies. If they were mutable, then an object's position would not be a constant. If the network were not a hierarchy, then an object could occur in more than one place. The fact that we have a static hierarchy is what makes it possible to ask the position of an object and expect a meaningful answer.

Now that we can ask this question, we will discover that it is an extremely valuable thing to be able to do. This will be shown in sections 5.1.4 and 5.3.6. To make it possible at the language level, we need to add a simple construct: the *from* clause. Any method declaration can include a *from* clause. Here is an example method declaration, with a *from* clause:

```
void mymethod() from jnode(int i, int j) { ... }
```

When this method is invoked, the agent ID of the invoker is pattern-matched against the *from* clause *jnode(int i, int j)*. Suppose, for example, that the invoker of the method is *jacobi#567@0.jnode(5,3)*. Since the identifier *jnode* in the *from* clause matches the identifier *jnode* in the agent ID, the method matches. If it had not matched, the method declaration would have been ignored, and a different method declaration would have been sought. But since it does match, the variable i is bound to 5, and the variable j is bound to 3. More generally, the syntax of a method declaration with a *from* clause is:

```
rettype methodname(formals2) FROM identifier(formals) { method-body }
```

When the specified method is invoked, the tail-segment of the agent ID of the object invoking the method is pattern-matched against the *from* clause. The identifier in the *from* clause must match the identifier in the tail of the agent ID. If not, the method declaration is ignored and other methods with the same name are considered. If there is a match, the formal parameters in the *from* clause are bound to the indices in the agent ID.

The *from* clause enables a method to determine the position, and therefore the purpose, of the object which invoked it. This information can be used in many ways, as will be shown later.

4.15 The Relay Keyword

We require that our language perform the relay optimization, so that we can superimpose arbitrary communication patterns on the task tree. We also require that it follow the "awakening" rules described earlier. However, these two requirements are contradictory in one way. We must correct the contradiction.

Suppose an object A invokes a method of object B, which in turn invokes a method of object C. Suppose that the method of object B does not refer to any instance variables of B. In this case, the relay optimization must be performed, according to our previously-stated requirements. Object B must be bypassed. In other words, the invocation must go direct from A to C. However, our awakening rules require that object B be awakened in the process. This is clearly contradictory with the relay optimization's condition that object B not be touched.

To break the contradiction, we introduce the "relay" keyword. This keyword is placed in front of a method declaration. The relay keyword changes the awakening rules: the invocation of a relay method should *not* awaken the object of which the relay method is a part. We also weaken the requirements for relay optimization: the system is only required to optimize out relay methods, not normal methods. Together, these two corrections eliminate the contradiction in the requirements.

4.16 Nesting Classes Within Classes

The relay optimization makes it possible for an agent to send a message to a sibling efficiently. But it can be somewhat clumsy, syntactically speaking. At the implementation level, the message really is going directly from the agent to its sibling. But at the language level, the message still appears to be going via the owner, which means the owner needs to contain explicit methods to relay the data from one agent to the other. We would like to eliminate this data-relaying code, where possible.

To eliminate the awkwardness, we could go back to our original idea, and let agents be declared *public*. This would enable the *jacobinode* object to simply reach into its owner,

```
class jacobi
{
  agent jnode(int i, int j) is {
    /* insert body of class jacobinode here        */
    /* note: can now refer to identifier ''jnode'' */
  }
  ...
}
```

Figure 4.11 Using Nested Classes to Implement Jacobi

and pull out a pointer to one of its siblings, using the notation *owner.jnode(i,j)*. It could then send the message directly. The owner would not need to be involved. The fetching of the pointer to the sibling could easily be optimized out by the compiler. Instead of fetching the pointer to the sibling, the compiler would simply use string manipulation on agent IDs to generate the ID of the sibling. The ID would be used as a substitute for the pointer. In other words, the so-called "fetching of the pointer to the sibling" would require no communication at all. This design is efficient and concise, but it requires that the *jnode* agents be declared public. This is not a good idea, they really shouldn't be public. Fortunately, there is a better way.

Our solution is to allow the following notation:

```
agent agentname(indices) is { class-body }
```

This construct declares a class, and an agent of that class, at the same time. Note that the class-body is syntactically inside another class declaration. Scoping tradition says that an inner construct can "see" the declarations in a surrounding construct. So this notation makes it seem "reasonable" for the inner class-body to refer to the agents of the outer class-body. This is easily implemented using agent ID manipulation. Figure 4.11 shows how this would be used in the Jacobi example.

The ability to nest a class in a class (with this construct) makes it possible for an agent to "see" its siblings. This makes it possible to send messages directly, not just at the implementation level, but also at the language level. This eliminates a lot of relay-code. This notation achieves this goal without the unnecessary promiscuity of the *public* declaration. In those cases where this notation can be used, it can be quite elegant.

This notation is part of the bigger picture of superimposing arbitrary communication patterns on agent trees. While we can superimpose grids and other communication patterns on the agent-hierarchy without it, doing so is inelegant and clumsy. This notation makes it much more natural to set up an arbitrary communication pattern between the siblings of an agent.

4.17 The On Clause

Sometimes, it is desirable to take explicit control over the placement of objects. For this purpose, we provide the *on* clause. It modifies the agent declaration:

```
agent agentname(indices) on processor ...
```

Where *processor* is an expression that evaluates to an integer processor number. This gives the user total control over where the objects are to be placed.

4.18 The Agent Construct Summarized

This section summarizes the constructs that we have defined. These constructs can be added to any object-oriented programming language. For the sake of clarity, we will show them as they would appear when added to Java.

Clearly, the most important addition is the agent construct, as it occurs in figure 4.12 line 1. This construct occurs at inside a class definition, among the method-declarations

```
1. AGENT name ( indices ) IS classname ( actuals )
2. AGENT name ( indices ) IS { class-body }
3. AGENT name ( indices ) ON processor ...
4. OWNER
5. AWAKEN <object> ;
6. RELAY rettype methodname ( formals ) { method-body }
7. rettype methodname ( formals ) FROM ident ( formals ) { body }
8. VOID INIT ( actuals ) { method-body }
```

Figure 4.12 Summary of Language Extensions

and instance-variable declarations. Suppose that this declaration occurs inside a declaration of class C. Whenever an object of class C is allocated, the system responds by creating a set of agents. Each agent is an object of class *classname*, where *classname* is the classname occurring in the agent declaration. The object of class C is called "the owner," and its agents are conceptually there to serve it. The object of class C can get a pointer to one of its agents by evaluating *name(values)*, where *name* is the name occurring in the agent declaration, and *values* is a set of parameters matching the formal parameters *formals* occurring in the agent declaration. The *values* can be thought of as indices selecting a particular agent. An agent can get a pointer to its owner by merely evaluating the keyword *owner*. The agents are allocated lazily — they are not actually there until accessed by method invocation. When accessed for the first time, an agent will initialize itself by executing its constructor. The constructor is passed a set of arguments determined by the *actuals* expressions occurring in the agent declaration. The expressions *actuals* are evaluated in a scope that can see the arguments of class C's constructor.

Figure 4.12 line 2 shows a variant of the agent declaration, in which a class declaration and an agent declaration are merged into one. This variant syntax defines a new (anonymous) class, and defines a set of agents of that class in a single statement. This notation would be mere shorthand, except that it allows the inner class to refer to the agents of the outer class by name. Suppose this declaration occurs inside a class C. Then these agents can get pointers to any agent of their owner directly. In other words, they merely evaluate *agentname(indices)*, just like the object of class C can itself do. It clearly follows that they can get pointers to each other.

Figure 4.12 line 3 shows the *on* clause. It allows the programmer to control the mapping of agents to processors precisely.

Agents are not allocated, and their constructors are not executed, until they are accessed by method invocation. This is usually what is desired. But sometimes, you want an object to begin working right away, without waiting for a method invocation. To achieve this goal, we provide the *awaken* construct, shown in figure 4.12 line 5. This triggers the object into action, allocating it if is not already allocated.

When an object A invokes a method of an object B, and that method in turn invokes a method of an object C, data flows from A to B to C. If B does not access any of its instance variables, the compiler can almost bypass B entirely, sending data directly from A to C. But the optimization cannot be done, because of the awakening rules above. According to those rules, this operation should awaken object B. This prevents the compiler from bypassing B. The *relay* keyword, shown in figure 4.12 line 6, solves this problem. It permits the system to bypass the object holding the relay method, not awakening it. It also serves as a way for the programmer to state his requirement that relay optimization be done, that this object be bypassed.

When an object A invokes a method of another object B, the object B can ask "who invoked me" through the use of a *from* clause, shown in figure 4.12 line 7. The *from* clause is pattern-matched against the tail segment of object A's agent ID. In particular, the identifier in the *from* clause must be equal to the identifier in the agent ID, and the formal parameters of the *from* clause are bound to the indices in the agent ID.

Because of the syntax that declares a class and an agent at the same time, it is possible for some classes not to have names. Traditionally, the constructor is named after the class, this is impossible when the class has no name. We solve this by allowing the use of constructors named *init*, as shown in figure 4.12 line 8.

4.19 The Cost of Manipulating Agent IDs

The use of agents depends on agent ID manipulations. The efficiency of such manipulations strongly impacts the efficiency of the agent construct. Therefore, it is a good idea to use an encoded representation for agent IDs inside the runtime system. The encoded representation should be a compact variant of the textual representation. The purpose of this section is to propose one possible encoding for agent ID strings. This will give the reader an idea of how inexpensive agent ID manipulations can potentially be.

During the compilation process, the compiler must scan the program top to bottom. Each time it sees a class declaration, it must assign it a unique class declaration number. Each time it sees an agent declaration, it must assign it a unique agent declaration number. This is purely syntactic. These class declaration numbers and agent declaration numbers are then used in the encoded version of the agent ID. The compiler emits several tables:

- A mapping from agent declaration numbers to class declaration numbers, indicating the class in the agent declaration.
- A mapping from class declaration numbers to ASCII class names, for debugging printouts.
- A mapping from agent declaration numbers to ASCII agent names, for debugging printouts.

Given this foundation, the following representation for agent IDs would be feasible:

- *2 bytes: length of the entire ID.* The length of the ID is always a multiple of 4 bytes.

- *2 bytes: nesting level of object.* The depth of the object in the agent hierarchy. For example, *jacobi#567@9* is a root object, its nesting is 0, and *jacobi#567 @9.jacobinode(3,5)* is at nesting level 1.

- *4 bytes: hash value of entire ID.* The agent ID is treated as a string and fed into a hash function for strings. The hash value is here because agent IDs are frequently inserted into hash tables, so having the hash value precomputed saves time. The hash function should be *incrementally* computable: after making a small change to the agent ID, the hash value should be easy to update.

- *4 bytes: sequence number of root object.* The topmost object in an agent hierarchy has a sequence number. For example, the agent *jacobi#567@9.jacobinode(3,5)* belongs to a hierarchy with *jacobi#567@9* at the root. The sequence number of that object is 567.

- *2 bytes: processor number of root object.* The topmost object in an agent hierarchy also has a processor number. In *jacobi#567@9.jacobinode(3,5)*, the processor number of the root object is 9.

- *2 bytes: class declaration number of root object.* The class of the root object. The classes are numbered sequentially by the compiler.

- *? bytes: information about nested agent.* This occurs once for each level of nesting. For example, in *jacobi#567@9.jacobinode(3,5).calcobj(3)*, There would be two of these fields: one for *jacobinode(3,5)*, and one for *calcobj(3)*.

The information about nested agents is in the following format:

- *? bytes: agent index.* This is repeated once for each index. For example, if the nested agent is *jacobinode(3,5)*, then this field will be repeated twice: once for the 3, and once for the 5.

- *2 bytes: agent declaration number.* The unique number of the declaration of the agent. By referring back to the agent declaration itself, this tells you the agent's name and type.

- *2 bytes: length of the nested agent part.* The length, in 4-byte words, of this part of the agent ID.

Given this representation, we can determine the class of the object in constant time. To do so, one first fetches the length of the ID. Adding the length to the address of the agent ID yields the end of the ID. By looking at a fixed offset from the end, one can find the agent declaration number. Using the table that maps agent declaration numbers to classes, we can obtain the class of the agent.

For an object to find the ID of one of its agents can be done in constant time if the agent has fixed-size indices. The object fetches its own ID. It finds the end of the ID, and concatenates a new segment to the ID. This concatenation consists of storing the indices and the agent declaration number. These stores can usually be hard-wired as straight-line code. The length of the ID must then be updated, and the hash value incrementally updated.

For an object to find the ID of its owner, it must first find the end of the ID. It loads the length of the last segment of the ID. Subtracting the length of the final segment from the length of the ID yields the substring of the ID which is the agent's owner's ID. It updates the length of the ID, and incrementally updates the hash value after removing the final segment.

The tables emitted by the compiler make it easy to convert the encoded representation back to the textual representation. This is useful for debugging. On an error, the runtime system can name the object that caused the problem in a format the user can read.

This section is intended to give the reader an idea of the asymptotic efficiency of Agent ID manipulations. The actual cost in microseconds will be measured in section 6.4. Since the textual representation of agent IDs is easier to read, we will continue to use textual representations throughout the remainder of the dissertation.

CHAPTER 5

THE CONSEQUENCES OF STATIC NETWORKS

Chapter 1 defined what a static network is. Chapter 3 listed three classes of problem that could conceivably be implemented with static networks. It described the transformations needed to convert those algorithms into static-network algorithms. Chapter 4 described a construct for creating static networks, the *agent* construct.

This chapter is the motivation behind the entire dissertation: it discusses what one can do, given good support for static networks. We will use the *agent* construct to create static networks, and we will see what static networks "buy us." We are going to approach this in an open-ended manner: we are looking for any advantage we can get from static networks. We will intentionally leverage them in any way we can think of to improve efficiency, make code more elegant, or simplify the programming problem. The result will be a list of efficiency optimizations, programming techniques, and other advantages that can be obtained when high-quality support for static networks is available.

We organize this chapter into case studies. Each case study is a programming problem specifically designed to illustrate one of the benefits of agents. Each case study presents a programming problem, and implements it once with agents, and one or more times without. This makes it easy to do side-by-side comparisons.

Note that when we say we wrote a program "in a traditional language," we are specifically referring to Distributed Java without agents. We use the broad phrase "traditional language" because our conclusions generalize to all existing object-oriented parallel languages without agents.

After each case study, we address the following major issues:

1. What benefit was obtained from using agents?
2. What general class of problems can obtain the same benefit?

While the chapter is organized as a list of case studies, recall that the real purpose of the case studies is explanatory: the case study is supposed to make one of the benefits of agents obvious. It is the purpose of the subsequent analysis to identify that benefit in detail, and to comment on the scope and importance of that benefit.

5.1 Case Study: The Three-Matrix Multiplier

We will code a class that performs a simple computation: $A*B*C$, where A, B, and C are all square matrices (we make them square because it avoids some uninteresting trivia). We will assume the standard library already contains a parallel matrix-multiplication class. We will assume that the standard library's matrix multiplier expects the first matrix in the form of rows, and the second matrix in the form of columns. Therefore, we will need to do some regrouping of elements along the way. We will assume that our standard library also includes classes for regrouping the elements of matrices into rows, columns, or individual elements.

Note that this problem does not involve coding any algorithms to speak of. All we will be doing is plugging together some preexisting modules, all we will be writing is glue

```
class matmul
{
  void init(Object outobj, int nrows, int ncols);
  void row_a(int row, vector v);
  void col_b(int col, vector v);
  // callback result(int row, int col, double d);
}

class cols_to_rows
{
  void init(Object outobj, int nrows, int ncols);
  void col(int col, vector v);
  // callback row(int row, vector v);
}

class elts_to_rows
{
  void init(Object outobj, int nrows, int ncols);
  void elt(int row, int col, double d);
  // callback row(int row, vector v);
}
```

Figure 5.1 Library Classes needed by Three-Matrix Multiplier

code. This should be very easy. We do not mean it *will* be easy, we only mean that it *should* be.

5.1.1 Matrix Multiplication: The Code

Figure 5.1 lists the classes that we assume are part of the standard library (assuming a traditional language). The first is the matrix multiplier itself, the two latter classes are responsible for regrouping the elements of a matrix from columns into rows, and elements into rows, respectively.

```
class matmul_alt1
{
   void row_a(int i, vector v);
   void col_b(int i, vector v);
   matrix get_answer();
}

class matmul_alt2
{
   void row_a(int i, vector v);
   void col_b(int i, vector v);
   double get_piece_of_answer(int r, int c);
}
```

Figure 5.2 Alternative Interfaces for Matrix Multiplier Class

All of the objects produce their results via "callbacks." By this, I mean that they do not send their results back in a return-value. Instead, they send their results back to their caller by invoking a method. Since we are using a method call to send data back to where it was requested, it is a callback. The *outobj* parameter is there to tell each one where to send its callbacks. For example, the matrix multiplier's constructor contains the parameter *outobj*. The result matrix is then transmitted to *outobj* by invoking the *result* method on *outobj*.

One might argue that callbacks should not be used. But consider the alternatives, shown in figure 5.2. The problem with *matmul_alt1* is that the answer is computed piecewise, but it is returned as a unit. That is costly in two ways: it forces there to be a global synchronization step, and it denies the possibility of pipelining the output matrix into another computation — you can not start processing the result until the multiplier is finished. The problem with *matmul_alt2* is that it requires twice as much bandwidth as the callback-version. This design implies that N^2 method invocations (to *get_piece_of_answer*)

```
class mulabc
{
  void init(Object outobj, int size);
  void input_A(int col, vector v);
  void input_B(int col, vector v);
  void input_C(int col, vector v);
  // callback result(int row, int col, double d);
}
```

Figure 5.3 Interface to Three-Matrix Multiplier

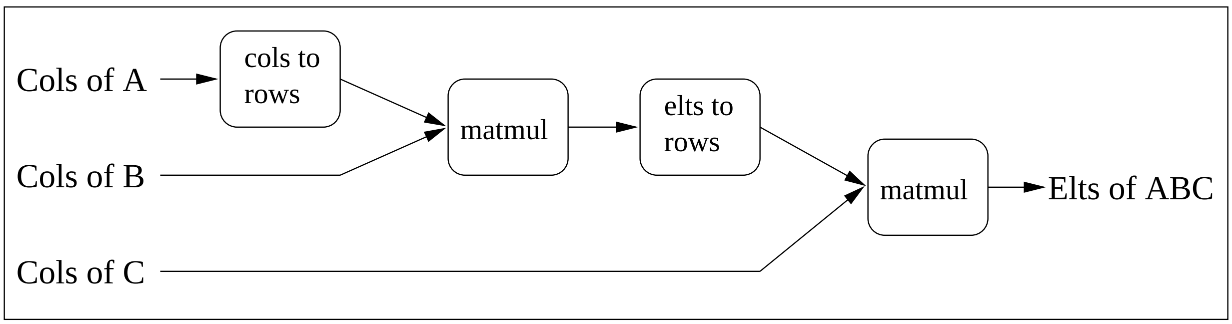

Figure 5.4 Three-Matrix Multiplier, Object Interconnection

must be transmitted into the matrix multiplier (and N^2 synchronizations must happen internally to it) before the N^2 results can come out. Of course, one might be willing to put up with a factor of 2 slowdown to avoid callbacks. We think that's not usually the case — we think that if you are actually going to the trouble of writing explicitly parallel code, you probably care a lot about speed.

Thus, we resume our programming problem with the original (callback-based) library design. The interface we will implement is shown in figure 5.3. A diagram of the computation to be done shown in figure 5.4. As you can see, the data flows in a fixed pattern. If we were to classify this problem into one of the three categories defined in chapter 3, it would fall under the category of "pattern computations."

We start by writing the implementation that uses agents. We will write a class *mulabc* whose job is to coordinate the three-matrix multiplication process. It will own all the other objects. All the other objects will send their output to the *mulabc* object, which will also serve as a central dispatcher, forwarding data to wherever it needs to go. We will eliminate the *outobj* parameters for this version, instead, each object will send its output to its *owner*. We will explain this decision later.

Figure 5.4 shows the four objects that need to be connected together. The first step in writing the agents version of the code is to declare these four objects. Hence, we start our definition of the *mulabc* class by writing four agent declarations. Each sends its output to the *mulabc*. The *mulabc* must forward that data to wherever it is needed. To achieve this, we add one *relay* declaration for each of the edges in figure 5.4. The final result is shown in figure 5.5.

If we try to do the same thing in a traditional language, things become much more complicated. We will convert the agents code shown above into a traditional language, one step at a time. Of course, this will not be valid traditional-language code until we have finished all modifications.

We need to make these changes: first, the *mulabc* class needs to be converted to a group, in order to prevent a bottleneck. (The agents version is not a bottleneck because of relay methods). Second, we need to write code to allocate *mm1*, *mm2*, *cr*, and *er* in the constructor of *mulabc*, followed by a broadcast of those handles to the other members of the group. There is a time period during which those handles have not yet been received. Thus, we need to add synchronization code to make sure that the handles have been distributed. The synchronization needed is shown in figure 5.6.

This extra synchronization can be reduced by restructuring the code, but it cannot be eliminated. To see why this is so, realize that in any callback-scenario (like this one), you have two groups A and B both of which are invoking methods on each other. Assume,

```
class mulabc
{
  agent mm1 is matmul(size, size);
  agent mm2 is matmul(size, size);
  agent cr is cols_to_rows(size, size);
  agent er is elts_to_rows(size, size);

  void init(int size) { };

  relay void input_A(int col, vector v) {
    cr<-col(col, v);
  }
  relay void row(int row, vector v) from cr {
    mm1<-row_a(row, v);
  }
  relay void input_B(int col, vector v) {
    mm1<-col_b(col, v);
  }
  relay void result(int row, int col, double d) from mm1 {
    er<-elt(row, col, d);
  }
  relay void row(int row, vector v) from er {
    mm2<-row_a(row, v);
  }
  relay void input_C(int col, vector v) {
    mm2<-col_b(row, v);
  }
  relay void result(int row, int col, double d) from mm2 {
    owner<-result(row, col, d);
  }
}
```

Figure 5.5 Three-Matrix Multiplier, Agents Version

```
class mulabc
{
  ...

  relay void input_A(int col, vector v) {
    wait (cr != NIL);
    cr<-col(vol, v);
  }
  relay void row(int row, vector v) from cr {
    wait (mm1 != NIL);
    mm1<-row_a(row, v);
  }
  relay void input_B(int col, vector v) {
    wait (mm1 != NIL);
    mm1<-col_b(col, v);
  }
  relay void result(int row, int col, double d) from mm1 {
    wait (er != NIL);
    er<-elt(row, col, d);
  }
  relay void row(int row, vector v) from er {
    wait (mm2 != NIL);
    mm2<-row_a(row, v);
  }
  relay void input_C(int col, vector v) {
    wait (mm2 != NIL);
    mm2<-col_b(row, v);
  }
  relay void result(int row, int col, double d) from mm2 {
    wait (outobj != NIL);
    outobj<-result(row, col, d);
  }
}
```

Figure 5.6 Three-Matrix Multiplier: Adding Synchronization

WLG, that we allocate group A first. In that case, group A cannot receive a pointer to B in its constructor (because B is not allocated yet). Group A needs to receive a pointer to B in some other method. That means that there will be a time-period during which group A exists (and is therefore subject to external method-invocations), but does not have a pointer to B. This situation usually requires explicit synchronization. This does not occur when using agents because of the apparently instantaneous creation of entire agent hierarchies.

The next step in the conversion from agents-code is the elimination of the *from* clauses. This presents another source of awkwardness. Two different multipliers are invoking the *result* method on the *mulabc* group. In the agents version, the *result* method is instantiated twice, with *from* clauses, in order to identify which multiplier sent the data. However, in the traditional language, there is no way to discriminate which multiplier sent the data.

One way to deal with this would be to locate the source code for the standard library and rewrite the matrix multiplication module in such a way that a user of the library can tell which multiplier sent a particular result. The multiplier could achieve this by encoding some tag information along with the callback. We can go into the standard library source code and make this modification to the matrix multiplier, and to the other classes using callbacks as well. The interface of the updated library is shown in figure 5.7. Passing through tags is a particularly unpleasant sort of awkwardness in that would have to be a standard operating procedure: it would have to be done in every class that uses callbacks. This is so universally invasive that we will temporarily avoid this approach. Instead, we see what happens if we deal with the problem in an ad-hoc manner.

Without the tag parameters, the *mulabc* class would have to find some other way to disambiguate the output from the two multipliers. The only way to do that is to have them send their outputs to different places. We could achieve that by adding another

```
class matmul
{
  void init(Object outobj, Object tag, int nrows, int ncols);
  void row_a(int row, vector v);
  void col_b(int col, vector v);
  // callback result(Object tag, int row, int col, double d);
}

class cols_to_rows
{
  void init(Object outobj, Object tag, int nrows, int ncols);
  void col(int col, vector v);
  // callback row(Object tag, int row, vector v);
}

class elts_to_rows
{
  void init(Object outobj, Object tag, int nrows, int ncols);
  void elt(int row, int col, double d);
  // callback row(Object tag, int row, vector v);
}
```

Figure 5.7 Three-Matrix Multiplier: Adding Tag Parameters

```
class mulabc1
{
  matmul mm1;
  elts_to_rows er;

  void distribute_handles(matmul mm1_a, elts_to_rows er_a) {
    mm1 = mm1_a;
    er  = er_a;
  }
  void row(int row, vector v) {
    wait (mm1 != NIL);
    mm1<-row_a(row, v);
  }
  void result(int row, int col, double d) {
    wait (er != NIL);
    er<-elt(row, col, d);
  }
}
```

Figure 5.8 Three-Matrix Multiplier, Traditional Version, Part 1

group to the implementation. Since the *mulabc* group also has a problem telling the output of *er* from the output of *cr*, we let the second group handle the output from *cr* as well. The final code for the Three-Matrix Multiplier in a traditional language, including the *outobj* parameters and synchronization code, is shown in figures 5.8 and 5.9.

Now we have the traditional-language and agents versions of the three-matrix multiplier. The problem specification was to plug together a few preexisting modules. It was expected to be effortless. The following subsections analyze the two versions, pointing out ways in which they they were more complex that they should have been.

```
class mulabc {
  matmul mm1, mm2; cols_to_rows cr;
  void init(Object outobj, int nrows, int ncols) {
    if (thisindex == 0) {
      intermed = newgroup mulabc1();
      mm1 = newgroup matmul(intermed);
      mm2 = newgroup matmul(thisgroup);
      cr  = newgroup cols_to_rows(intermed, nrows, ncols);
      er  = newgroup elts_to_rows(thisgroup, nrows, ncols);
      // broadcast object handles to all members of group.
      intermed[ALL]<-distribute_handles(mm1, er);
      thisgroup[ALL]<-distribute_handles(mm1, mm2, cr);
  } }
  public void distribute_handles
           (matmul mm1_a,matmul mm2_a,cols_to_rows cr_a) {
    mm1 = mm1_a; mm2 = mm2_a; cr  = cr_a;
  }
  void input_A(int col, vector v) {
    wait (cr != NIL);
    cr<-col(vol, v);
  }
  void input_B(int col, vector v) {
    wait (mm1 != NIL);
    mm1<-col_b(col, v);
  }
  void input_C(int col, vector v) {
    wait (mm2 != NIL);
    mm2<-col_b(row, v);
  }
  void row(int row, vector v) {
    wait (mm2 != NIL);
    mm2<-row_a(row, v);
  }
  void result(int row, int col, double d)  {
    wait (outobj != NIL);
    outobj<-result(row, col, d);
} }
```

Figure 5.9 Three-Matrix Multiplier, Traditional Version, Part 2

5.1.2 Graph Allocation and Synchronization

The traditional version contains several pieces of code not present in the agents version: the *init* methods, the *distribute_handles* methods, and the *wait* statements. All three of these pertain to allocating and linking together the objects. The *init* method contains all the allocation statements. The *distribute_handles* method passes pointers around and initializes pointer variables. The *wait* statements are there to ensure that the objects have been linked together before the computation attempts to proceed. All the extra code is to assemble the dynamic network of objects.

Static networks "pop" into existence atomically. There is no network allocation or construction phase. In a program utilizing static networks, all the code needed to assemble the network is automatically eliminated: the system puts the network together for you from its declarative specification. Hence the elimination of all this network construction code in the static version of the three-matrix multiplier.

The elimination of allocation statements and pointer passing gets rid of a lot of code, which is good, but not earthshaking. What matters more is the elimination of the synchronization statements. When we first wrote the traditional-language version of the code, we tried to do it without *wait* statements. We tried allocating the multipliers first, then we tried allocating the *mulabc* first, and so forth. Eventually we came up with a sketchy proof that no matter what order we allocate things in, there are race conditions that have to be eliminated with *wait* statements. Once we realized that wait-statements were inevitable, we designed a strategy for inserting wait-statements into the code. We chose a simple strategy: test every variable for NIL before it is used. We could have used a more complicated strategy, looking for specific variables that didn't need to be synchronized. There may have been other possible designs as well.

Assembling a network of objects is rarely trivial, because of the synchronization issues involved. There are almost always choices involved when deciding which order to allocate the objects in. These choices impact the amount of synchronization needed later, and hence, the efficiency of the program. Once one has chosen an order in which to allocate things, one must figure out where synchronization statements are needed, one must insert them, and one must convince oneself that those synchronizations are adequate. Frequently writing network allocation code involves multiple prototypes: you write a set of allocation statements, and start writing code to pass pointers, and then you realize that if you did it in the other order, things would turn out simpler. In short, construction of a network of objects is not a mindless procedure by any leap, not even for a simple network like this one. Network allocation involves serious programming effort and multiple potential alternative designs.

Static networks eliminate all the network allocation and construction code. Naturally, that reduces the amount of typing you have to do, which is desirable. Frequently, it leads to a 25% code size reduction or more. But much more importantly, it eliminates an entire software design process, a process which is not trivial by any stretch of the imagination.

5.1.3 Callbacks Cannot be Avoided

Much of the complexity in the dynamic version was the result of callbacks. Traditional languages are very poor at expressing callbacks. Because of this, one might be tempted to avoid callbacks altogether.

We already showed you that avoiding callbacks is not feasible in the case of the three-matrix multiplier: we considered the alternatives that didn't use callbacks, and found them both lacking. But one might speculate that the three-matrix multiplier is an exception, that most modules can be nicely coded without callbacks. This turns out

```
class theorem_prover1
{
  void init(rulebase rb, clause goal);
  list get_all_proofs();
}

class theorem_prover2
{
  void init(rulebase rb, clause goal);
  proof get_one_proof();
}

class theorem_prover3
{
  void init(rulebase rb, clause goal, Object outobj);
  // invokes result(proof p) on outobj;
}
```

Figure 5.10 Theorem Prover: Several Possible Interfaces

not to be the case. We will now show you another class, a theorem prover, which also needs callbacks. Then, we will identify what properties of the matrix multiplier and the theorem-prover caused the need for callbacks.

Figure 5.10 shows several possible interfaces for a theorem prover. In each one, you feed in the database and the goal, and the prover starts working. Each produces a set of proofs. In *theorem_prover1*, you use a methods *get_all_proofs* to fetch the proofs. The problem with this interface is that you have to wait for all proofs to be generated before you can get any of them. We could try the interface shown in *theorem_prover2*. But that's not even implementable without bending over backwards. The proofs are scattered about the machine in a search tree, it is extremely expensive and awkward for the get_one_proof method to find one. The only clean interface uses callbacks, as in *theorem_prover3*.

The theorem prover and the matrix multiplier demonstrate two general classes of problems. Callbacks were needed in the matrix multiplication example for this reason: obtaining the output via callbacks requires half as much communication as obtaining the output via fetch-methods (e.g. get_piece_of_answer). Because the results of the matrix multiplication consists of many pieces, this turns out to be highly significant. More generally, any problem where the output consists of many pieces could be speeded significantly by callbacks.

We can generalize that a little further. Whenever we find a problem whose output is large, we as parallel programmers attempt to rework it into a form where the large output can be computed as a large number of independent pieces. Because we intentionally rework problems whose output is large into problems whose output consists of many pieces, we can draw this conclusion: almost any problem whose output is large will be speeded by callbacks.

In the theorem prover, these two reasons do not apply: there may only be a few pieces of output (a few proofs of the goal). But, there are two new reasons to use callbacks. First, the answer consists of an unpredictable number of pieces. We cannot elegantly fetch them one by one, since we do not know how many to request. Second, the results are computed in unpredictable locations (they show up unpredictably in some of the search-tree nodes). Because we do not know where the solutions will be, it would be difficult for a fetch-method to find them.

We summarize the four kinds of problems which require callbacks:

- Problems whose output consists of many pieces.
- Problems whose output is large.
- Problems whose output consists of an unpredictable number of pieces.

- Problems whose outputs are computed in unpredictable locations.

It is interesting to note that none of these reasons makes sense in a sequential program: the lack of communication overhead eliminates the first two, the cheapness of global synchronization eliminates the third, and the ability to use global structures eliminates the fourth. This explains why callbacks are not needed often in sequential programs, but are needed constantly in parallel programs.

Taken together, the situations involving callbacks include a significant percentage of all modules written, perhaps even the majority of all modules. In a sequential language, we might be able to do without callbacks most of the time. But in a parallel language, we need to be able to handle callbacks elegantly.

5.1.4 Callbacks Can be Elegant

Traditional languages are not capable of dealing with callbacks elegantly. Callbacks lead to awkward, confusing code. This section explores the various ways in which callbacks cause traditional languages to break down. As it turns out, static networks provide a fix for all the problems, making callbacks quite elegant when static networks are present.

The first problem with callbacks in traditional languages is the need to pass explicit continuations (the *outobj* parameters). In our experience, continuation-passing causes two problems. First, the continuation parameters are difficult for novices to understand, they usually find the idea of continuation-passing style too abstract. Second, the unrestricted continuations are worse than unrestricted *goto* in terms of their tendency to create spaghetti code. Agents programs use a convention that eliminate the need for the continuation parameters: the object always sends it callbacks to its owner. We will show that this convention works fairly well, and avoids the two problems associated with passing continuations.

First, we show that this convention does not introduce any module dependencies that were not already there. The owner must contain an agent declaration, explicitly naming the agent. Because of this, the programmer must know the interface of the agent when writing the owner. Given that the programmer knows the interface of the agent when writing the owner, he knows that the agent will emit callbacks. Therefore, he knows that he must write methods to catch those callbacks. In other words, the programmer who writes the owner must already deal with the interface of the agent. It adds no additional module-dependency if he must deal with a little more of the agent's interface. Thus, our convention does not create a modularity problem.

Second, we show that this convention does not create an efficiency problem. The owner of the agent can catch the callbacks, and can then forward the data anywhere it wants. For example, all the methods in the *mulabc* object simply forward data to another object. Because of the *relay* optimization, this never introduces a bottleneck or any additional communication.

Our convention forces the *owner* of a set of agents to serve as a relay-station between them. At first, it seems like writing all those *relay* methods is a burden. It seems like it would be more concise to configure the agents to talk to each other directly, without the intervention of their owner. However, that is not usually possible. For example, looking at figure 5.4, it appears as if we could simply have plugged the output of the *mm1* object directly into the *er* object using the *outobj* parameter. Unfortunately, that would not have worked. The *mm1* object invokes the method *result(int row, int col, double d)* on *outobj*, but *er* expects you to invoke the method *elt(int row, int col, double d)*. The method names do not match. A trivial difference, admittedly, but no matter how trivial the difference, it is necessary to interpose some interface glue between the two objects. The *relay* methods, in addition to their task of forwarding data, are serving as that glue code. A quick scan of the *mulabc* class shows that 6 out of 7 of the methods are serving as interface glue. We could have written the interface glue in a different way, but no matter what, we would have had to interpose *something* between the objects. In short, our convention sometimes forces us to write a glue-method which we would not otherwise have had to write, but usually, the glue-method would have been necessary anyway.

The obvious exception is when a set of classes were specifically designed to work together. In that case, their interfaces usually fit perfectly. Glue is usually not necessary. In that case, writing relay-methods would be an annoying chore. However, in that case, it is usually possible to use the nested form of the agent-declaration. One declares one big master class, and nests all the other classes inside of it. All the agents can see each other (by the scoping rules of agents). Hence, they can send to each other directly, eliminating the *relay* methods. An example of this style of programming will be seen in the N Queens example in section 5.2.

It is worth considering whether or not it is possible to use the same convention in traditional languages. Of course, we cannot use it directly, it relies on the *owner* relationship, which is only present because of agents. However, we could try to use a substitute, the *creator* relationship. We define the creator of an object as follows: when one object *O1* invokes code that creates another object *O2*, it could be said that *O1* is the *creator* of *O2*. In fact, the *creator* relationship works quite well if (and only if) you organize your objects as a static hierarchical network. In other words, you do not actually need to have compiler support for agents to use this convention. However, you *do* need to emulate agents by structuring your objects as a static hierarchical network.

We have found that our convention imposes a pleasing regularity on the data and control flow. With this convention, the organization is strictly hierarchical. This is very structured, it significantly reduces the likelihood of spaghetti code caused by unrestricted continuation passing. It shortens the parameter lists, eliminating the most confusing parameter, the explicit continuation. Hopefully, the elimination of continuation-passing style will simplify life for the novice programmer as well. In short, our convention eliminates the worst aspects of continuation-passing style, solving the first problem associated with callbacks in traditional languages.

The second problem with callbacks in traditional languages is that an object cannot identify where a callback came from. In our case study, this makes it impossible to tell the output of *mm1* from the output of *mm2*. Since we needed to tell them apart, we ended up bending over backwards to keep the two separate.

We are aware of two ways to do this. Neither is elegant. The first is the manner we used: introducing an auxiliary class (*mulabc1*) to disambiguate which object sent which callback. It is obvious that this is awkward. The second solution is to make a habit of passing an arbitrary tag field through every class. We could go through the standard library, and add an initialization parameter called *tag* to every class. When an object makes a callback, it is expected to pass out the *tag* along with the data. By examining the tag, we could tell which object sent which piece of data. This solution universally invasive — it requires us to add this repetitive tag-passing code to every class we ever write. Neither of these solutions is desirable. We strongly need a better way to identify the origin of a callback.

Static networks make it easy to identify the origin of a callback. Agent IDs make it possible to identify an object according to whom it is working for, and what its textual name is. This is a semantically rich way of identifying an object: given an object's owner and its name, you can know a lot about the purpose of that object. If one were to ask "where did this callback come from" in a traditional language, the only answer one could expect would be: "it came from object #5903." That answer does not do you any good. But if the object is part of a static network, you can expect a much richer answer: "it came from an agent that is working for *you*, and its name is *mm1*." We allow the programmer direct access to this information by means of the *from* clause. We used this ability to easily distinguish callbacks from *mm1* and *mm2*.

The third problem with callbacks is simple: the glue methods can become bottlenecks. Static networks eliminate that problem immediately, by means of the relay optimization. With dynamic networks, we have to take explicit steps to deal with it: in our case study, we had to turn the *mulabc* object into a group, leading to considerable pointer-passing and several initialization steps.

In summary, there are three problems with callbacks: identifying where callbacks came from, the abstractness of the *outobj* continuation parameters, and the bottleneck caused by the glue methods. Static networks eliminate all three problems. Without static networks, callbacks are very difficult to use elegantly. But with static networks, callbacks become an elegant and simple programming methodology.

5.1.5 Transparent Relationship Between Network and Code

The three-matrix objects and the connections between them are shown in figure 5.4. The agents code that implements this network is in figure 5.5. The relationship between the two is trivial. In fact, the procedure to convert the picture to the code is:

1. For each vertex in the graph, write one agent declaration, naming the vertex and declaring its type.

2. For each edge in the graph, write one relay method. The from clause specifies which object the data should come from, the body specifies which object the data goes to.

The one-to-one relationship between features of the code and features in the network is useful to programmers. It makes it obvious how to write the code that generates a particular network, and it makes it obvious what network a particular piece of code generates.

In the dynamic network versions, the relationship between the network and the code is not straightforward. There are two things that keep it from being simple. First, the procedure of allocating the network requires complicated code, while this code is nonexistent in the agents version. Second, the callback problems lead to convolutions in the dynamic network versions. The result is that the connection between the code and the dynamic network it implements is opaque. One has to study the dynamic network code carefully to figure out what it does.

Given agents, composing multiple objects is simply a question of listing the objects and connections that you want. The relationship between the code and the network is transparent.

5.2 Case Study: N Queens

The case study is an N-Queens program that counts the number of solutions for a given input value N. It uses a search tree to explore the entire space, and maintains a counter that records the total number of solutions.

5.2.1 N Queens: The Code

In all versions of N Queens, we assume that the queens are placed on the chess board left to right. A partial placement is described by a list of integer row-numbers. For example, the list (5, 4, 7) indicates that three queens have been placed (in columns 1-3), and that they are in rows 5, 4, and 7 respectively. We also assume that somebody has written a *legalmove* function that detects whether or not it is legal to extend a particular

```
newthread procedurename(arguments);
```

Figure 5.11 Thread Creation in Multithreaded Pascal

partial placement into a particular row. We do not show this function in any of the versions, since it is not parallel code.

Agents has some scoping rules that are related to the scoping rules of Pascal. To show the similarity, we will implement N Queens first in Pascal, then with agents. We can turn Pascal into a shared-memory parallel programming language by adding a thread-creation operator and some locking primitives. We will call this "Multithreaded Pascal." For concreteness sake, let us assume the thread creation primitive has the syntax shown in figure 5.11. The new thread starts immediately, executes the procedure, and then terminates.

To implement N Queens in Multithreaded Pascal, we will assume that we have already written two ADTs (abstract data types) for interthread communication. Figure 5.12 shows two ADTs (RECORD types with an interface consisting of a set of procedures). Let us assume that all these ADTs are implemented with appropriate locking so that they work correctly in a multithreaded environment. The accumulator is a variable that many threads can increment at the same time. One can fetch the total at any time. The quiescence counter is intended to check for the expiration of a search tree. One notifies the quiescence counter whenever one creates a search tree node, and whenever a search tree node terminates. When the two counts are equal, the search is done. The *quiescence_wait* function waits for the termination of the search tree. We also assume the existence of a *list* type, and some functions like *length* and *append*. This latter type does not need to be thread-safe.

```
1. The accumulator:

   type accumulator = record ... end;

   procedure accumulator_init(var x : accumulator);
   procedure accumulator_add(var x : accumulator, val : integer);
   function  accumulator_total(var x : accumulator);

2. The quiescence counter:

   type quiescence = record ... end;

   procedure quiescence_init(var q : quiescence);
   procedure quiescence_node_created(var q : quiescence);
   procedure quiescence_node_destroyed(var q : quiescence);
   procedure quiescence_wait(var q : quiescence);
```

Figure 5.12 ADTs used by Multithreaded Pascal Version of N Queens

The N Queens algorithm in Multithreaded Pascal is shown in figure 5.13. It uses an accumulator to count the solutions, and a quiescence to determine when the search is done. There will be one thread per search tree node. The body of *nqueens* starts the topmost thread of the search tree. A search tree node receives a partial placement as a parameter. If the partial placement is a solution, it increments the solution counter. If not, it extends that partial placement in every possible way, and creates a thread to explore each possibility. The quiescence object is notified whenever a thread is created or destroyed, so that it is possible to tell when the search is done. Meanwhile, the initial thread that started the whole thing is waiting for the search tree to expire. When it finally does, the initial thread fetches the total from the accumulator and returns it.

Notice we nested the *nqueens_node* procedure inside the *nqueens* procedure. As a consequence, the *nqueens_node* procedure executes in a scope that can see the variables

```
function nqueens(x : integer)
  var
    a : accumulator;
    q : quiescence;
    n : integer;

  procedure nqueens_node(queens : list)
    begin
      if (length(queens) = n) then accumulator_add(a, 1);
      else
        begin
          for i := 0 to n-1 do
            begin
              if (legalmove(i, queens)) then
                begin
                  quiescence_node_created(q);
                  newthread nqueens_node(append(queens, i));
                end;
            end;
        end;
      quiescence_node_destroyed(q);
    end;

  begin
    accumulator_init(a);
    quiescence_init(q);
    n = x;
    quiescence_node_created(q);
    newthread nqueens_node(NIL);
    quiescence_wait(q);
    nqueens = accumulator_total(a);
  end;
```

Figure 5.13 N Queens, Multithreaded Pascal Version

```
class accumulator
{
  init();
  void add(int n);
  int total();
}

class quiescence
{
  init();
  void node_created();
  void node_destroyed();
  void wait();
}

class writeonce
{
  init();
  void set(object v);
  object get();
}
```

Figure 5.14 ADTs used by Agents Version of N Queens

N, A, and Q. This makes it easy for the nodes to increment the accumulator A, notify the quiescence object Q, and access N.

Next, we will convert the Multithreaded Pascal program into agents. The first step is to provide the same ADTs, this time in the form of classes, as shown in figure 5.14. We will need one more ADT, a *writeonce* object, which is a simple synchronization structure. It holds a single integer value. Any attempt to get the value will block until the value has been set.

Having done this, we can write an agents implementation. It is shown in figure 5.15. The one *nqueens* object has several agents working for it: a writeonce object N to hold

```
class nqueens
{
  agent a is accumulator();
  agent q is quiescence();
  agent n is writeonce();

  agent nqueens_node(list queens) {
    void init() {
      if (length(queens)==n.get()) { a.add(1); }
      else {
        for (i=0; i<n.get(); i++) {
          if (nqueens.legalmove(i, queens)) {
            q.node_created();
            awaken nqueens_node(list.append(queens, i))
          }
        }
      }
      q.node_destroyed();
    }
  }

  int nqueens(int x) {
    n.set(x);
    q.node_created();
    awaken nqueens_node(NIL);
    q.wait();
    return a.total();
  }
}
```

Figure 5.15 N Queens, Static Network Version the size of the chess board, an accumulator A to count the solutions, a quiescence object Q to determine when the search is done, and an endless supply of search tree nodes named *nqueens_node* to do the searching. Notice that the agent hierarchy is flat: every other object is working for the one *nqueens* object.

We have indexed the *nqueens_node* objects according to partial placements. If X is a partial placement, we have assigned responsibility for exploring X to *nqueens_node(X)*. When an *nqueens_node* wants to create a child (in the search tree) to search a partial placement P, it merely awakens the object responsible for exploring P, namely *nqueens_node(P)*.

When declaring the *nqueens_node* objects, we used the second form of the agent declaration. It allows us to declare an anonymous class and an agent of that class in one step. Because of this, the *nqueens_node* class is declared inside the scope of the single encompassing *nqueens* class. According to the scoping rules of agents, an inner class can refer to the agents of the outer class. In our case study, the *nqueens_node* objects can refer to the agents of the *nqueens* object. In particular, they can refer to N, A, and Q by name.

Finally, the traditional-language version is shown in figure 5.16. It is easily understood, being only a slight variation of the agents version. The big difference is that the classes are not nested, and hence, there is no way to declare N, A, and Q at the top. Instead, N, A, and Q are propagated from object to object via the parameter lists.

5.2.2 Concurrency and Shared Identifiers

The agents version was written by translating the Pascal version. Each of the constructs in Pascal was mapped directly to an object-oriented construct. We converted both procedures to classes. We converted both procedure bodies to methods. We con-

```
class nqueens
{
  int nqueens(int n) {
    quiescence q = newgroup quiescence();
    accumulator a = newgroup accumulator();
    q.node_created();
    new nqueens_node(q, a, n, NIL);
    q.wait();
    return a.total();
  }
}

class nqueens_node
{
  void init(quiescence q, accumulator a, int n, list queens) {
    if (length(queens)==n) { a.add(1); }
    else {
      for (i=0; i<n; i++) {
        if (nqueens.legalmove(i, queens)) {
          q.node_created();
          new nqueens_node(q, a, n, list.append(queens, i));
        }
      }
    }
    q.node_destroyed();
  }
}
```

Figure 5.16 N Queens, Traditional-Language Version verted every thread to an object. Most of the conversion was straightforward, but there was one aspect that deserves comment: the identifiers N, A, and Q.

In the Pascal code, the three identifiers N, A, and Q were declared at the top of the *nqueens* function. Their scope was such that they were accessible to all the search-tree threads. When we translated to agents, we changed those threads into objects. So whereas previously the scope of N, A, and Q spanned all the search-tree threads, now they needed to be converted into identifiers whose scope spanned all the search-tree objects.

Traditional languages sometimes allow global variables. We could have converted N, A, and Q into global variables, then their scope would have included all the search-tree objects. But that would have made the *nqueens* function non-reentrant. It would have made it impossible to run multiple copies of N Queens at the same time. Using global variables in a parallel program is usually a bad idea, as it prevents concurrent execution.

In Multithreaded Pascal, it is easy to make an identifier whose scope spans multiple threads. But in a traditional object-oriented language, there is no practical way to create an identifier whose scope spans multiple objects, unless you are willing to sacrifice concurrency. Static networks give us a better way to define shared identifiers.

5.2.3 Shared Constants versus Shared Variables

The names of agents are shared identifiers, such as N, A, and Q in the N Queens example. But one often needs shared variables, not shared agents. It is straightforward to implement shared variables on top of shared agents.

The first step is to write a data-storage class, such as the *writeonce* class. One provides a *set* and a *get* method, as well as any other operations that one might want to do to the variable, for example, *increment*. Having done this, one declares an agent of this class.

```
/* before */                      /* after */

class C                           class C
{                                 {
  varagent N is writeonce();        agent N is writeonce();
  void anymethod() {                void anymethod() {
    N = 5;                            N.set(5);
    int i = N;                        int i = N.get();
  }                                 }
}                                 }
```

Figure 5.17 Syntactic Sugar for Varagents

The agent now has the defining properties of a variable: a name, a scope, the ability to store data, and support for the *set* and *get* operations.

Since this kind of agent is so common, we have added a small amount of syntactic sugar to the compiler. The sugar consists of the keyword *varagent*. The *varagent* is the same as any other agent, except that the compiler automatically performs the syntactic conversions shown in figure 5.17. This syntactic sugar completes the effect of full-fledged shared variables.

5.2.4 Propagation versus Shared Identifiers

Certain parallel programming languages like Charm support the equivalent of global variables. Global variables create reentrancy problems: they prevent the concurrent execution of multiple copies of the program. In a parallel programming language, this is a very undesirable quality. As the traditional-language N Queens program shows, the propagation of values from object to object can function as a substitute for global variables. Charm programmers could use propagation instead of global variables to avoid these reentrancy problems. Yet they do not. To explain why programmers choose global variables over propagation despite concurrency problems, we have devised three hypotheses.

First, there are some reliability-related reasons to avoid propagation. For example, propagation creates the potential for propagation errors. If one were to get the values N, A, and Q in the wrong order in some parameter list, the computation would cease working. But that really is not a convincing explanation, in our opinion.

Another hypothesis is that propagation is inconvenient. In the traditional-language version of N Queens, the values N, A, and Q had to be passed from object to object. If the program had been larger, if there had been more than one kind of object, then N, A, and Q would have been in all the constructor parameter lists. The programmer would have had to write code to propagate them each time he wrote a new class — a tedious chore. This is a more plausible hypothesis, as programmers tend to avoid unnecessary coding.

However, our strongest hypothesis about why programmers prefer to put shared values into shared variables is simply that it makes sense. In N Queens, the accumulator handle, quiescence handle, and chess board size *are* shared: every object is accessing the same entities. It makes sense to declare them as shared. It simply is not logical to declare them as parameters, and then try to make sure all the parameters contain the same value. One can imagine the traditional-language programmer wondering how to document his code. Suppose he has written three class definitions, all propagating the values N, A, and Q about. Should he mention next to all three classes that the N, A, and Q parameters are the same N, A, and Q parameters as in the other classes? Should he mention that they are shared constants, and then put the documentation in one place, or should he just put a copy of the documentation next to every class definition? There is a fundamental conflict between the truth, which is the fact that these values are shared, and between the code, which says that they are parameters.

It is hard to know which of these motivations explains why programmers avoid propagation. Nevertheless, experience shows that they prefer shared identifiers over propagation *even when shared identifiers prevent concurrency.* Given this situation, it makes sense to provide a form of shared identifier which has no concurrency problems. Agents provide such a facility.

5.2.5 Groups as Interfaces

Parallel object-oriented languages introduced the *group* construct to serve an important function. The group as a whole appears to the world as a bottleneck-proof object. This bottleneck-proof object makes a perfect interface to a parallel data structure. Agents can also create bottleneck-proof objects. As a consequence, a language with agents does not need to have support for groups.

Consider the agents version *mulabc* object in section 5.1.1. It contains no instance variables. There is no data to store. Its only job is to pass data back and forth — and the relay optimization eliminates even that. There is nothing left for the object to do! In fact, the *mulabc* object is never allocated by the lazy allocation mechanisms, as it is never needed. We imagine there is an "interface object" *mulabc* representing the three-matrix computation, but this is a mere mental convenience, there is no object there physically. Because of this, *mulabc* is bottleneck-proof. When using agents, the vast majority of interface objects are pure abstractions. They do not actually *do* anything, nor are they ever physically allocated. Because of this, they cannot be bottlenecks.

However, there are a few interface objects that actually do need to store a little data and use it a lot. The agents version of N Queens is an example: it needs to store the

```
class writeonce
{
  agent worker(int index) {
    object val;
    void set(object n) {
      val=n;
    }
    object get() {
      wait (val!=NIL);
      return val;
    }
  }
  relay void set(object n) {
    worker(0..number_of_cpus-1)<-set(n);
  }
  relay object get() {
    return worker(current_cpu).get();
  }
}
```

Figure 5.18 A Bottleneck-Free Implementation of Writeonce

chess board size. However, it delegates the responsibility for storing this value to one of its agents, *N*. By delegating responsibility for data storage, the *nqueens* object itself becomes a pure abstraction again. It is bottleneck-proof. Any object can turn itself into a pure abstraction by delegating storage responsibility to its agents. In so doing, it becomes bottleneck-proof.

Note that the *writeonce* object *N* will not be a bottleneck either. A reasonable implementation of the *writeonce* class is shown in figure 5.18. In this implementation, the *writeonce* has agents named *worker(0), worker(1),* etc. Each worker is responsible for storing a copy of the value. The *writeonce* object itself is a pure abstraction, and the workers are concrete. When one invokes the *set* method on the *writeonce* object, it relays the invocation to all its workers. When one invokes the *get* method one the *writeonce*, it relays the invocation to one of its workers, and therefore, the workers divide up responsibility for handling the *get*s. If we somehow arrange that *worker(i)* be mapped to processor i, the *get* will be communication-free.

The fact that agents can serve as bottleneck-proof interface objects implies that they subsume the function of groups. This is a pleasing development. Some parallel programming languages tend to become overly complex, they contain too many constructs. This negatively impacts the learning curve of the language, makes it harder to write the compiler, and makes it harder to get the language's semantics right. It is desirable to reduce the size of parallel programming languages, wherever possible. The addition of the *agent* construct to a language allows us to remove a similarly-complex construct, the group. Since *agents* also subsume global variables and distributed hash tables, we can remove those as well. The net impact of adding agents to a language is a reduction in overall language size.

5.3 Case Study: Pixel Smoothing

The example is an iterative pixel-smoothing algorithm. The input is a pixel array. Each pixel is to be averaged with its four neighbors. This simple smoothing step is repeated for a fixed number of iterations. This problem's structure is much like the structure of any iterative grid computation.

5.3.1 Pixel Smoothing: The Code

We will write two classes: a class to handle a single, fixed-size patch of the pixel grid, and a second class that coordinates all the patches. Instead of using the traditional approach of a two-dimensional grid of patches, We will use a *three-dimensional* grid,

```
class smooth_one_patch
{
  void prev_n(patch p);
  void prev_s(patch p);
  void prev_e(patch p);
  void prev_w(patch p);
  void prev_center(patch p);
  // callback results(patch p);
}

class smooth
{
  void init(int itercount, int nrows, int ncols);
  void inpatch(int row, int col, patch p);
  // callback outpatch(int row, int col, patch p);
}
```

Figure 5.19 The Interface of the Pixel Smoothing Classes

indexed by row, column, and iteration number. This greatly simplifies the logic of the patch, in that it does not have to distinguish messages for iteration i from messages for iteration $i+1$. The specification when using agents is as follows:

The job of the *smooth_one_patch* object is to compute the value of a single region at a single iteration. To do that, it needs the value of that region in the previous iteration, as well as the values from its four neighbors to the north, south, east, and west. We provide 5 methods to feed these values in. Actually, the patch calculator only needs the *edges* of the regions neighboring it. However, the code to extract edges is not very interesting from the point of view of this section and it does not affect the structure of the code, so we will send in entire patches. The *smooth_one_patch* object contains no concurrency, so it is not very interesting. We will not show the code.

```
class smooth
{
  agent calcpatch(int row, int col, int iter) is smooth_one_patch();

  void init(int itercount, int nrows, int ncols) { };

  relay void distribpatch(int row, int col, int iter, patch val) {
    if (iter == itercount) {
      owner<-outpatch(x, y, val);
    } else {

      calcpatch(row, col, iter+1)<-prev_center(val);

      if (row>0)        {calcpatch(row-1,col,iter+1)<-prev_s(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_n(BLANK);}

      if (row<nrows-1){calcpatch(row+1,col,iter+1)<-prev_n(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_s(BLANK);}

      if (col>0)        {calcpatch(row,col-1,iter+1)<-prev_e(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_w(BLANK);}

      if (col<ncols-1){calcpatch(row,col+1,iter+1)<-prev_w(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_e(BLANK);}
    }
  }
  relay void inpatch(int row, int col, patch val) {
    distribpatch(row, col, 0, val);
  }
  relay void results(patch val)
             from calcpatch(int row,int col,int iter) {
    distribpatch(row, col, iter, val);
  }
}
```

Figure 5.20 Smoothing Code, Agents Version

Much more interesting is the *smooth* object, which is supposed to smooth an entire pixel grid for several iterations. The agents version is shown in figure 5.20. The agent declaration at the top declares a 3D grid of objects named *calcpatch*. Each *calcpatch* agent is responsible for computing one iteration of one region. Once a *calcpatch* has done its job, it produces its output via callback, triggering the *results* method. The *results* method uses its *from* clause to identify which *calcpatch* the patch came from, then it sends the patch into *distribpatch*. The *distribpatch* method is responsible for distributing patches: given a patch, it sends that patch wherever it needs to go. The *inpatch* method forwards the input patches into *distribpatch*, so that they too can be sent to where they are needed.

The design of *distribpatch* is as follows. First, we check if the patch is part of the final result. If so, we send it out, as a callback. If not, we send it to the next iteration. Sending it to the next iteration consists of sending it north, south, east, west, and straight. When sending it north, south, east, and west, there are four if-statements to check for the special cases associated with the edge of the grid.

The *smooth* object receives all the patches as they are computed. It uses a *from* clause to determine which patch is which. The traditional-language version will have to use some other approach to distinguish the patches from each other. The problem is similar to the one in the three-matrix problem, where we had to distinguish the output of *mm1* from *mm2*. However, in this case, the option of creating an intermediary class such as *mulabc1* is not reasonable, as we would need one intermediary object per patch.

One solution would be to pass *row*, *col*, and *iter* through the patch object. We could let all this miscellaneous information tag along with the *prev_center* input, along with the *outobj* parameter, as shown in figure 5.21. This design is bad software engineering in two ways. First of all, we have given up on information hiding. The *smooth_one_patch* object does not need to know row, col, or iter, we are just passing it in so it can pass it back out.

```
class smooth_one_patch
{
  void prev_n(patch p);
  void prev_s(patch p);
  void prev_e(patch p);
  void prev_w(patch p);
  void prev_center(patch p,int row,int col,int iter,Object outobj);
  // callback results(patch m,int row,int col,int iter);
}
```

Figure 5.21 Passing Miscellaneous Data through the smooth_one_patch

Second, it flies in the face of the specify/implement project life cycle. We should come up with the interface specification first, and then implement it, but this specification could not have been devised until after the implementation was partly finished.

A better solution is the one we suggested earlier in the *mulabc* example: make it a standard operating procedure to pass a dynamically-typed tag through every class using callbacks. In that case, the specification would appear as in figure 5.22. At least this specification is somewhat respectful of the principles of information hiding, and it could have been written by a non-omniscient project manager prior to handing out assignments, so it is a reasonable choice from a software engineering perspective — it is the one we will use. We will merge *row*, *col*, and *iter* into a three-element list and pass them through this tag-field.

The second thing we need to do to convert to a traditional language is to somehow allocate the 3D grid of *smooth_one_patch* objects. In the agents version, we declared the grid, and obtain an element of it by evaluating the agent name *calcpatch(i,j,iter)*. In the traditional-language version, we will implement a method *calcpatch(i,j,iter)* that does the same thing. The *calcpatch* method in the traditional-language version creates the

```
class smooth_one_patch
{
  void prev_n(patch p);
  void prev_s(patch p);
  void prev_e(patch p);
  void prev_w(patch p);
  void prev_center(patch p, Object tag, Object outobj);
  // callback results(patch p, Object tag);
}
```

Figure 5.22 Passing a Tag through the smooth_one_patch

objects the first time they are accessed, and stores them in a hash table. Subsequent accesses return the objects from the hash table. We will assume that the standard library provides us with a powerful implementation of hash tables.

The third thing we will need to do to convert to a traditional language is to add object-allocation code. The *smooth* object will have to be turned into a group, and the hash table's handle will have to be distributed to all members of the group. We will put these steps in a small static (class) method named *create*.

The traditional code, shown in figures 5.23 and 5.24, differs from the agents code in three ways: we added the *calcpatch* method that builds the 3D grid using a hash table, we added the *create* method that allocates the *smooth* group and associated data structures, and we added the tag-passing to both classes to make up for the absence of *from* clauses.

5.3.2 Issues From the Three-Matrix Multiplier

The pixel smoother contains many of the same issues as the three-matrix multiplier. Since we have already discussed those issues, we will merely touch on each one.

```
class smooth
{
  void init(hashtable tab, int itercount, int nrows, int ncols,
                 Object ntag, Object outobj) { };

  smooth_one_patch calcpatch(int row, int col, int iter) {
    smooth_one_patch p;
    tab.lock(row, col, iter);
    p = tab.lookup(row, col, iter);
    if (p==NIL) {
      p = new smooth_one_patch();
      tab.insert(row, col, iter, p);
    }
    tab.unlock(row, col, iter);
    return p;
  }

  static smooth create(int itercount, int nrows, int ncols,
                            Object ntag, Object outobj) {
    hashtable tab = newgroup hashtable();
    return newgroup smooth(tab,itercount,nrows,ncols,ntag,outobj);
  }

  /* continued in next figure */
```

Figure 5.23 Pixel Smoothing, Traditional-Language Version, Part 1

```
  /* continued from previous figure */

  relay void distribpatch(int row, int col, int iter, patch val) {
    if (iter == itercount) {
      outobj<-outpatch(ntag, x, y, val);
    } else {

      patch(row, col, iter+1)<-
        prev_center(val, new list(row, col, iter+1), thisgroup);

      if (row>0)        {calcpatch(row-1,col,  iter+1)<-prev_s(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_n(BLANK);}

      if (row<nrows-1){calcpatch(row+1,col,  iter+1)<-prev_n(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_s(BLANK);}

      if (col>0)        {calcpatch(row,col-1,iter+1)<-prev_e(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_w(BLANK);}

      if (col<ncols-1){calcpatch(row,col+1,iter+1)<-prev_w(val); }
      else              {calcpatch(row,col,  iter+1)<-prev_e(BLANK);}
    }
  }

  void inpatch(int row, int col, patch val) {
    distribpatch(row, col, 0, val);
  }

  void results(patch val, Object tag) {
    int row = tag.element(0);
    int col = tag.element(1);
    int iter = tag.element(2);
    distribpatch(row, col, iter, val);
    tab.delete(row, col, iter);
  }
}
```

Figure 5.24 Pixel Smoothing, Traditional Language Version, Part 2

The traditional-language version needs to allocate and link up the objects. As before, this leads to a lot of routine coding that is not present in the agents version. In this problem, the allocation code is pretty simple. Explicit synchronization related to object allocation was avoided by putting the hash table allocation and the smooth-group allocation into a static class method.

The traditional-language version of the *smooth* object cannot tell which *smooth_one_patch* object invoked the *results* callback. Because of this, it needs to add the *tag* parameter to the *smooth_one_patch* object.

In the agents version, the object's constructor has three parameters: *itercount*, *nrows*, and *ncols*. All of these are obvious in meaning. In addition to those, the traditional-language versions also need *outobj* and *tag*, neither of which is easy to explain. In the traditional-language versions, we are going to have to add these two parameters to every class that uses callbacks.

In the traditional-language version, the *smooth* object needs to be converted to a group to avoid a bottleneck. The static network version contains no bottleneck because of the relay optimization.

These flaws are the same ones that occurred in the three-matrix multiplier. These flaws will reappear every time we find a problem structured like the three-matrix multiplier: namely, a problem where we need to compose several independent objects together into a single whole. Because these problems recur so often, it would be redundant to mention them every time. We will let this be the last mention. However, look for these issues in other programs.

5.3.3 Hash Tables: Static Networks for Shared Memory

When the *smooth* object receives a patch, the *distribpatch* method forwards that patch to five *calcpatch* objects. The agents version is efficient. The traditional version is not efficient if compiled naively, though it can be made efficient with sufficient optimization.

In the agents version, the forwarding of the patch proceeds as follows. The expression *calcpatch(i,j,iter)* returns an object handle of a proxy. The only data in the proxy is the agent ID of the real object. Building the proxy does not require communication. When this proxy is used to send a message, the machine's communication circuitry must carry that message. The code of the *distribpatch* method obtains five proxies and sends one message to each. The total number of physical messages generated per *distribpatch* invocation is therefore 5.

If the traditional-language version is compiled by a naive compiler, the result would be very slow. The *distribpatch* method invokes the *calcpatch* method 5 times. The *calcpatch* accesses the hash table at least 3 times, each access is a two-way communication. That is a total of 30 communications for hash table accesses, plus 5 more for sending messages to the *smooth_one_patch* objects. Therefore, if the traditional version is compiled naively, the total number of physical messages generated per *distribpatch* invocation is 35.

A sufficiently smart compiler can compile the traditional-language version efficiently. First, the compiler needs to perform the CPS transform. This will effectively move the computation to the data, instead of moving the data to the computation. This will make things much more efficient. If we add the CPS transform to our compiler, the evaluation of *calcpatch(i,j,iter)<-message* will be moved to the processor holding the hash-table bucket, and then it will be moved to the processor holding the *smooth_one_patch* object, for a total of only two communications. This can be reduced to one if the compiler is smart enough to allocate the *smooth_one_patch* object on the same processor as the hash table bucket that stores it. If both of these conditions are met, the result is code that executes as efficiently as the agents version: each *distribpatch* invocation creates 5 physical messages. There are a few compilers, created in research settings, that can achieve this level of efficiency [4, 19, 20, 21].

Since the optimizations required to make the traditional-language version efficient are not simple or common, it makes sense to examine the traditional language version and see whether or not it can be made faster manually. Of course, the hash table's interface layer is the source of the problem. The hash table has four methods *lock*, *lookup*, *insert*, and *unlock*. It sounds like old-fashioned shared memory. Essentially, each slot in the hash table *is* a shared memory location. But we already know we can not implement shared memory efficiently on distributed memory machines. The granularity of the operations is too fine. That explains why it is so hard to get an efficient implementation of hash tables. The hash table's interface emulates shared memory, and thus it is hard to implement it efficiently without hardware support for shared memory. We cannot realistically speed up the program manually unless we change the interface layer of the hash table.

For the sake of consistency in terminology, we will use the term *hash table* to refer specifically to hashing devices with an interface that emulates shared memory. I will use different terms to denote hashing devices with different interfaces.

When you allocate a hash table, it is as if an infinite number of shared memory locations popped existence atomically. When you free the hash table, all the locations collectively vanish. If those shared memory locations are viewed as objects, then the hash table *is* a static network. In short, the hash table is a static network primitive that was designed for shared-memory machines.

The smoothing problem is one of many problems that simply require a static network: there is no reasonable way to implement it without one. If we have an efficient static network construct like agents, we can use it. If not, we have no choice but to use some other static network construct (like the hash table), even if it was designed for shared-memory machines.

5.3.4 Hash Groups

Though hash tables can be efficient in the presence of a sufficiently powerful optimizer, it nonetheless makes sense to design a variant which is easier to compile efficiently. In particular, we will consider a variant called the *hash group*. The hash group is a hybrid between agents, hash tables, and groups. It is obviously more suited for distributed memory machines than the hash table, since it does not present the illusion of shared memory. However, it has two limitations that hash tables do not.

The hash group is an associative lookup device that maps keys to objects. You create the hash group using an allocation statement like this:

```
t = new hashgroup of C;
```

Where C is some class name. The primary operation on a hash group is to invoke a method of one of the objects in the hash group. The notations to do synchronous and asynchronous invocations are:

```
t[key1, key2, key3...].method(arguments);
t[key1, key2, key3...]<-method(arguments);
```

The multiple keys are essentially just a single multipart key. When you access a particular key for the first time, the hash group automatically allocates an object of class C. It then invokes your method on that object. Hash groups can be implemented efficiently, each invocation only costs one message, as an invocation should. Hash groups are fine for distributed memory. Hash groups would work quite well for the pixel smoothing problem.

Besides agents, we now have two more choices for creating static networks: hash tables and hash groups. Hash tables have the obvious limitation that they require shared memory hardware or the presence of a very powerful optimizer. Hash groups do not have this limitation. However, hash groups have two limitations that hash tables do not have: they require compiler support for their special syntax (a limitation they share with agents), and they can only contain one type of object.

Though they have limitations, hash tables and hash groups can do several of the things agents can do. As the case studies continue, we will continue to comment on the abilities and limitations of hash tables and hash groups.

5.3.5 Constructs that Support the From Clause

When a *calcpatch* object invokes the *result* method of the *smooth* object, the *smooth* object needs to know which *calcpatch* object invoked the *result* method.

In the agents version, the *smooth* object wishes to ask the question "what are the indices of the agent that invoked this method." The runtime system can easily look at the agent, extract its agent ID, and determine those indices. So the runtime system can easily answer this question. Agents provides the *from* clause as a means whereby the question can be asked, and the answer given.

In the traditional-language version, the *smooth* object wishes to ask the (almost equivalent) question "what was the hash table key of the object that invoked this method." It is a meaningful question, in the sense that there is a hash table out there, and one of its entries points to the object. But the runtime system cannot look at the object and tell that there is a hash table pointing to it. In other words, it can not look at an object and tell that it is part of a static network. As a consequence, the traditional version cannot support anything equivalent to the *from* clause.

This shows a limitation of hash tables as a static network primitive. Hash tables do enable the programmer to create static networks. But they do not enable the system to look at a member of the static network and identify its place in the network. Because of this, hash tables cannot support the *from* clause.

Hash groups, on the other hand, can support the *from* clause. Since the object is genuinely a part of the hash group (and not just pointed to by the group), the compiler could identify its hash table key in the group and could provide information about it.

5.3.6 Program Trace Analysis

Envision the following hypothetical scenario. The programmer runs the smoothing program with a 1000x1000 pixel grid. He makes his patch size 10x10, for a total of 10,000 patches. He smooths it for 50 iterations. He discovers that processor utilization is much lower than he expected. This is an embarrassingly parallel problem, so it should have near 100% CPU utilization, but the programmer discovers that the processors are only being used 70% of the time. He wants to know why the machine is idling instead of working on his problem.

The agents programmer decides that he would like to see the objects, and watch the messages move from object to object. He reruns the program, this time, with tracing turned on. The runtime system records the agent IDs of the objects and their types. It also records, for each message, the agent ID of the sender and receiver. It generates a huge trace file. The agents programmer then runs his performance visualization tool.

There are many objects in the program. The programmer asks to only see the objects of type *smooth_one_patch*. That leaves exactly 10,000 objects. The programmer tells the system to plot the objects on the screen. The system asks: "where should I position the objects, should I just distribute all 10,000 randomly on the screen?" The programmer says no, that is a bad idea. He wants to see a neatly laid-out grid. So he tells the system to make a 3D plot where X, Y, and Z are derived from the agent's indices. The performance analysis tool can do that easily, since the indices are part of the agent IDs, which are in the trace file.

So the programmer sees a neat grid. The tool then animates the messages, moving them from object to object. They form a wave, flowing from one iteration to the next. He sees something interesting: the wave of messages is not flat, some parts of it are getting ahead. One geographic region, in particular, is getting many iterations behind. Finally, the wave reaches the end of the grid, and most of the machine sits there while one geographic region catches up. He realizes there is nothing in the program that says which patches should be computed first, and the computer, by chance, seems to be neglecting one geographic region.

The programmer comes up with a strategy to fix the problem: priorities. He assigns higher priority to earlier iterations, and lower priorities to later iterations. That encourages the machine to finish one iteration before spending time on the next. He runs and animates again, and this time, the wave of messages moving from iteration to iteration moves much more evenly now.

Meanwhile, the traditional-language programmer decides that he, too, would like to see the objects and watch the messages flow from object to object. But he can not. His trace file contains 10,000 objects, but their indices were never recorded. The system never knew what the indices of the objects were. More to the point, it never knew that there was a grid in the first place. Since the indices were not recorded, the performance tool has no idea how to lay out the objects in a grid. Without a grid, the plot would make no sense at all.

Normally, objects are identified by pointers. Examining a pointer tells you nothing about the object which is pointed to. In contrast, examining an agent ID tells you quite a bit about the object. By examining the agent ID one can usually determine the purpose of the object, one can usually know what task the object is there to perform.

Intuitively, it seems useful to be able to look at an object and know its role in the computation. In fact, it does keep turning out to be useful. The performance visualization tool examines the agent ID to help it decide where to put objects on the screen. The three-matrix multiplier examines the agent ID to disambiguate the outputs from the two multipliers. The smoothing program examines the agent ID to help it tell which patch was computed. In general, it is quite useful to be able to be able to examine an object and determine its role in the computation as a whole. When an object is part of an agent network, this is possible.

5.4 Case Study: The Summation Tree

In this case study, we will implement a summation algorithm. We make the following assumptions. Somewhere, there is a distributed array of objects, and each object contains a number. Somehow, the objects in this array have reached a consensus that those numbers need to be added up. We wish to provide a "black box" that the objects can feed their numbers into, whereupon a total will pop out.

There are a great deal of possible implementations of summation in a traditional language. Though it will take some time, we will explore quite a number of them, including some inefficient ones, which are useful for comparison purposes.

5.4.1 Summation Tree: The Code

All the variants we implement will have the interface shown in figure 5.25. The parameter *size* controls the size of the array that the summation object will add up. The user is supposed to feed all the array elements in by invoking the *input* method, specifying the index and value. When the last value is fed in, the object will invoke the callback *total*.

```
class summation
{
  void init(int size);
  void input(int n, int value);
  // callback total(int value);
}
```

Figure 5.25 Interface to Summation Class

```
class summation
{
  int tot, count;
  agent L is summation(size/2);
  agent R is summation(size-(size/2));
  void init(int size) { count=0; tot=0; }
  relay void input(int n, int val) {
    if (size==1) owner<-total(val);
    else {
      if (n < (size/2)) { L<-input(n, val); }
      else              { R<-input(n - size/2, val); }
    }
  }
  void total(int n) {
    tot+=n; count++;
    if (count==2) { owner<-total(tot); }
  }
}
```

Figure 5.26 Summation Class, Agents Version

Total messages	2N
Critical Path (messages)	log N
Bottlenecks	none

Figure 5.27 Performance, Agents Version

The implementation of the agents version of the summation device is shown in figure 5.26. It is based on a recursive formulation: the sum of an array is equal to the sum of the left half plus the sum of the right half. The summation object divides the array into two halves. It assigns an agent L with the responsibility of adding up the left half, and an agent R with the responsibility of adding up the right half. A binary tree of agents will form, with each agent bisecting the array and giving half to each of its children. The leaf nodes will each be responsible for a subarray of size one. The user invokes the *input* method on the root *size* times. The *input* method decides what to do with the value. If the array is of size 1, then the parameter is the answer, which is output. Otherwise, the element is given to either L or R, depending on whether it is from the left or right side of the array. Eventually, the two agents L and R both produce sub-totals, triggering the *total* method twice. The *total* method takes the two subtotals, adds them, and outputs the final result.

The efficiency of the static network version is as follows. The user invokes the *input* method, which invokes another *input* method, and another, passing the value deeper and deeper into the tree. Each value takes log N steps, and there are N values, for a total of N log N steps. These steps require no communication, since *input* is a relay method. Finally, each value will reach a leaf, and invoke the *total* method. There will be a total of 2N invocations to *total* as the values propagate back up the tree. Since *total* is not a relay method, that involves 2N communications as well. The critical path is the time it takes a value to go down and then come back up the tree: 2 log N steps, but log N communications. The analysis is summarized in figure 5.27.

Next, we are going to implement *another* version using static networks, but using a group instead of agents. We are going to try to make it as efficient as the agents version. We will use a different (but similar) algorithm. The group will have as many elements as there are inputs. We will superimpose a tree on the group, using the binary

```
class summation
{
  int tot, countdown;
  static void create(Object outobj, int size) {
    return newgroup summation(outobj, size) [size];
  }
  void init(Object outobj, int size) {
    tot=0; countdown=3;
    if (thisindex*2 + 1 < size) { countdown--; }
    if (thisindex*2 + 2 < size) { countdown--; }
    if (thisindex != 0) { outobj = thisgroup[(thisindex-1)/2]; }
  }
  void total(int t) {
    tot += t; countdown--;
    if (countdown == 0) { outobj<-total(tot); }
  }
  void input(int n, int val) {
    thisgroup[n]<-total(val);
  }
}
```

Figure 5.28 Summation Class, Binary Heap Version

Total messages	2N
Critical Path (messages)	log N
Bottlenecks	none

Figure 5.29 Performance, Binary Heap Version

heap numbering scheme. Each node will accept one input value and two values from its children, add them together, and send the result to its parent. An exception is made, of course, for those elements that have fewer than two children. The code (which we call the "binary heap version") is shown in figure 5.28.

The efficiency of the binary heap version is as follows. The N input messages are fed into the group, leading to N communications. Each element in the group eventually forwards its result to its parent, leading to another N communications. The critical path is the height of the tree, log N. There are no bottlenecks. The totals are shown in figure 5.29. So the performance of this static network is the same as the performance of the other static network.

Next, we will experiment with some versions based on dynamic networks. By that, we do not mean that we will not use any arrays, but we do mean that we will at least try to make the reduction tree itself a dynamic network. In all subsequent versions, the reduction tree nodes will be connected by pointers.

The agents version looks like it could be directly translated to a dynamic network code. Doing so creates a legal program shown in figure 5.30, which we call the "naive version." It looks almost exactly like the agents version. And technically, it works.

However, the efficiency is not acceptable. The root object creates two objects, which creates two more, and so forth. The entire tree contains 2N objects, and its creation takes 2N messages. This can happen concurrently with the passing of the input values. The root object is being handed all the input values. Therefore, the root is a bottleneck: all N inputs pass through the root. The `input` method then passes those input values down the tree toward the leaves. This process takes log N hops per input value: that accounts for another N log N messages. The reduction (as performed in the `total` method) accounts for another 2N messages. The critical path is determined by the time it takes a single

```
class summation
{
  int tot, count;
  summation L, R;
  void init(int size, Object outobj) {
    count=0; tot=0;
    if (size>1) {
      L = new summation(size/2, this);
      R = new summation(size - size/2, this);
    }
  }
  void input(int n, int value) {
    if (size==1) owner<-total(val);
    else {
      if (n < (size/2)) { L<-input(n, val); }
      else              { R<-input(n - size/2, val); }
    }
  }
  void total(int n) {
    tot+=n; count++;
    if (count==2) { outobj<-total(tot); }
  }
}
```

Figure 5.30 Summation Class, Naive Version

Total messages	N log N + 4N
Critical Path (messages)	2 log N
Bottlenecks	N invocations on root.

Figure 5.31 Performance, Naive Version

input datum to go down and come back up the tree: 2 log N communications. The totals are shown in figure 5.31.

Obviously, the dynamic network versions need to look out for bottlenecks at the root. In the past, we have solved bottlenecks by converting objects into groups. We can do that here, too. To make it as efficient as possible, we will make the groups only as big as they need to be, in proportion to the amount of data they will be carrying. In other words, we will build a "fat tree version." The result is shown in figure 5.32.

The analysis of the fat tree version is as follows. The total number of objects allocated is N log N. To allocate N log N objects requires N log N messages. Then, the *distribute_handles* method is fired: each object receives one such invocation. This accounts for another N log N messages. If the compiler places the objects intelligently, then the *input* data propagating down the tree will not cross processor boundaries. Therefore, we can count that as free, communication-wise. There is no bottleneck this time, because of the fatness of the tree. Finally, the reduction comes back up the tree, creating another 2N messages. Again, if we assume that the downward propagation of data does not cross processor boundaries, the critical path is now shorter: log N communications. The totals are shown in figure 5.33.

The fat tree version has too much overhead: all those objects take much too long to allocate. It is the *input* method that keeps causing problems in the dynamic network version. We will avoid it for our next implementation. We will do so by storing the input data in a big array. Then, the tree leaves can just fetch the data from the array. The code for this "stored input version" is shown in figure 5.34. The summation objects are the array, their job is to implement the *input* method, store the input data, and hand it out on demand. Meanwhile, the summnode objects form a reduction tree, essentially the same tree used in the static version of the problem. The tree leaves fetch their input from the summation array, rather than waiting for it to be *input*.

```
class summation
{
  int tot, count;
  summation L, R;
  static summation create(int size, Outobj outobj) {
    return newgroup summation(size, outobj) [size];
  }
  void init(int size, Object outobj) {
    if (thisindex==0) {
      tot=0; count=0;
      if (size > 1) {
        L = summation.create(size/2, this);
        R = summation.create(size-size/2, this);
        thisgroup<-distribute_handles(L, R);
      }
    }
  }
  void distribute_handles(summation L1, summation R1) {
    L = L1;
    R = R1;
  }
  void input(int n, int val) {
    if (size==1) outobj<-total(val);
    else {
      wait (L != NIL) && (R != NIL);
      if (n < (size/2)) { L<-input(n, val); }
      else              { R<-input(n-size/2, val); }
    }
  }
  void total(int n) {
    tot+=n; count++;
    if (count==2) { outobj<-total(tot); }
  }
}
```

Figure 5.32 Summation Class, Fat Tree Version

Total messages	2N log N + 2N
Critical Path (messages)	2 log N
Bottlenecks	none

Figure 5.33 Performance, Fat Tree Version

The efficiency analysis of the stored input version is as follows. The creation of the summation array can be done in close to constant time. The input values can be sent in with N communications and stored. Meanwhile, the tree forms, creating 2N objects, with one creation message each. Each leaf fetches an input from the array: if the object placement is careful, this will require no communication. Finally, all the values percolate up the tree, requiring 2N more communications. The critical path is 2 log N. The totals are shown in figure 5.35. This is the closest that a dynamic network version can get to the static network versions.

5.4.2 One Task per Object

Both of the static network implementations take 2N messages. That is as good as you can get. The dynamic network versions are slow, with the exception of the stored input version, which is reasonable. The stored input version takes 5N messages.

One of the reasons the stored input version is slower is that it must actually transmit N invocations to the *input* method. In the agents version, those invocations are removed by the relay optimization. This difference does not concern us at this time.

The other reason that the stored input version is slower is that it must build the reduction tree a piece at a time. This incremental construction process takes 2N messages. That is a lot of overhead, considering that the real work only takes 2N messages.

In most problems, the relative cost of object allocation is less. For example, consider the smoothing problem. In the agents version, each *calcpatch* object receives 5 messages.

```
class summnode
{
  int tot, count;
  summnode L, R;
  void init(summnode parent, summation array, int lo, int hi) {
    if (lo==hi) {
      parent<-total(array[lo].getvalue());
    } else {
      int mid = (lo+hi)/2;
      tot = 0; count = 0;
      L = new summnode(this, array, lo, mid);
      R = new summnode(this, array, mid+1, hi);
    }
  }
  void total(int n) {
    tot += n; count ++;
    if (count==2) parent<-total(tot);
  }
}

class summation
{
  int value;
  static summation create(Object outobj, int size) {
    summation res = new summation(outobj, size) [size];
    new summnode(outobj, res, 0, size-1);
    return res;
  }
  void input(int n, int val) {
    thisgroup[n].value = val;
  }
  int getvalue(int n) {
    return thisgroup[n].value;
  }
}
```

Figure 5.34 Summation Class, Stored Input Version

Objects Allocated	2N
Total messages	5N
Critical Path (messages)	2 log N
Bottlenecks	none

Figure 5.35 Performance, Stored Input Version

In the dynamic network version, each *calcpatch* object receives 6 messages. The ratio is less, 1.2, but still quite significant.

One can amortize the creation overhead by building one network and using it repeatedly. That gets rid of the overhead, but it makes the code drastically more complicated. Suppose we did this in the summation example. We could build one summation tree and use it for all summations. Recall that multiple clients may invoke the summation module at the same time (this is a parallel program, after all.) If a single summation tree must handle two clients at once, it must take steps to tell which values belong to which client. Each client will need to be assigned a client ID, and these IDs can be passed up the tree along with the subtotals. Each tree node will need to have a little hash table mapping client IDs to subtotals. The complexity of the resulting code illustrates an important object-oriented design principle: do not attempt to make one object handle multiple independent tasks. In other words, stick to this fundamental rule: one task per object.

The smoothing example follows this design principle: each *calcpatch* object computes one patch. But we could have done things differently: we could have implemented it so that each *calcpatch* object handles many iterations. If we had, we would have paid a dear price in terms of complexity. We would have had to pass an iteration number into the *calcpatch* object. The *calcpatch* would have had to keep the values from one iteration separate from the next. To do this, much buffering code would have been needed. Violating the one-object one-task rule is usually a mistake.

```
root = 0
leftchild(x) = (x*2 + 1)
rightchild(x) = (x*2 + 2)
parent(x) = (x-1) / 2
```

Figure 5.36 The Numbering Scheme used in the Binary Heap Version

But dynamic networks come into direct conflict with this software engineering principle. Dynamic networks charge you one message of overhead whenever you create an object. That is significant if the object only performs one task, if it only receives a few messages.

Dynamic networks essentially leave you with a choice: tolerate a considerable amount of overhead for building the network, or amortize that overhead by sacrificing the software engineering principle of "one task per object."

5.4.3 Numbering Schemes

The binary heap example uses a numbering scheme to map a tree onto a group. The numbering scheme it uses is shown in figure 5.36. Numbering schemes are a venerable tradition, programmers have used them for years to enable existing constructs (like groups) to express shapes of static network that they could not otherwise express (like static trees).

Numbering schemes increase the expressive power of existing constructs, but they do so at the cost of readability. In all the other versions of summation, it is obvious that we are using a tree. In the binary heap version, only careful analysis (or a familiarity with this particular numbering scheme) reveals that a tree is in use. Numbering schemes enable us to express new shapes of static network, but at a readability price.

This is particularly dissapointing considering how many constructs we have already created to help us express static networks: groups, distributed hash tables, distributed arrays, etc. To meet our need for static networks, we keep on adding more and more constructs to express more and more shapes. But despite the battery of constructs we have created, we still have to resort to awkward numbering schemes to stretch their expressive power. These numbering schemes reduce the readability of our code. We should take this as an indication that a more flexible construct is needed.

5.4.4 New Structures, New Algorithms

Our first step in implementing a summation algorithm was to design an algorithm. We chose a simple, logical one: recursive bisection. Since dynamic networks were too slow for this problem, we needed a static recursive bisection tree. To implement a static recursive bisection tree, we have two constructs to choose from: hash tables, and groups.

To use a distributed hash table would have been easy. But hash tables were made for shared memory, using them always incurs a huge communication penalty. It would have cost us a tremendous amount of overhead to use hash tables. The result would have been even slower than the dynamic network versions.

The other way to create a static network in a traditional language is to use a group. To do that, we would have to map the tree onto a group. But a recursive bisection tree is not regular, it just does not map onto a vector cleanly. Trying to shoehorn it would have taken a lot of effort and would have yielded an oversized group with many unused objects. Not only is this wasteful, but it is confusing.

The end result: we gave up on recursive bisection. All the other versions use recursive bisection, but we just could not express recursive bisection statically in a traditional language. The constructs we have available simply are not suited for building a recursive bisection tree. So we started over, using a binary heap instead. That worked OK, but it is disappointing that we had to change algorithms. It is another indication that our language is restricting the set of algorithms we can write by restricting the shapes of the networks we can create.

Looking back, the same thing happened in the smoothing problem. We designed an algorithm around a sparse 3D grid (sparse in the sense that at any given moment, only a tiny portion of the grid contains data). However, the constructs of the language simply were not made to build that *shape* of network. We pressed forward and did it anyway with hash tables, paying a huge efficiency price. If we had wanted speed, we would have had to start over. We would have looked through our catalog of shapes that we *could* build efficiently, and we would have found a 2D grid. Then, we would have designed an algorithm around the 2D grid. We would soon have discovered that the 2D version is far more complex. Again, we had to switch from the right algorithm to an overly-complicated algorithm because we could not build the static structure we needed.

In traditional languages, we have a whole battery of constructs to create static networks. We stretch the expressive power of those constructs using awkward numbering schemes. Despite all this, we still can not create the shapes of static networks that we need. We can create dense 2D grids, but we can not efficiently create sparse 3D grids. We can create trees shaped like priority heaps, but we can not efficiently create trees shaped by recursive bisection. Since we can not use the right structure for the job, we can not use the right algorithm for the job.

Often, we do not even realize how much our freedom to choose a network structure has been restricted. For example, consider the pixel smoothing example. When most people think of pixel smoothing, they do not even think of using a 3D grid. We've been stuck with 2D grids for so long, that it does not even occur to most of us that there is another way. But once you see the three-dimensional implementation, you realize it is dramatically simpler than the two-dimensional implementation. The availability of a new network *shape*, the sparse 3D grid, leads to a small paradigm shift. It is not easy to predict these paradigm shifts. Looking at agents, one would have no idea that they would be useful for iterative grid computations. And looking at iterative grid computations, one would have no idea that they have anything to do with agents. It is only by trial and error that one stumbles upon a new algorithm. Hence, it is only by trial and error that one discovers that agents makes a new algorithm possible.

Agents make it possible to express a broad range of network shapes that we could not express before. But the effect of such freedom is not immediately obvious. It takes a flash of insight to realize that a new network shape allows us to implement a new, better algorithm. We can easily point to one or two new algorithms made possible by agents, like using an explicit time dimension for iterative computations, or using a recursive bisection tree for reductions. But there is no way we can give an idea of the *diversity* of algorithms which are possible given these new network shapes. In the end, that knowledge can only be gained by releasing agents on the public and seeing what new algorithms they implement.

5.4.5 Black Boxes inside Black Boxes

One of the most important principles of modern software engineering is called "information hiding." The principle is that all data should be hidden behind a barrier of sorts, a barrier which does not allow the user to see the implementation details. In object-oriented programs, information hiding is typically achieved by creating a cooperating set of objects with a "representative." The user is only given a pointer to the representative. The representative exports a set of methods whereby the user can communicate with the set of objects. However, the representative conceals the implementation details of how the objects do their work. Information hiding is the key to modularity.

The summation devices (all 5 implementations) utilize the principle of information hiding. The summation object itself is the representative. It provides a method *input* whereby you can feed in data, and *total* whereby data comes out. All the other implementation details of the summation process are hidden to the user of the summation device.

Sometimes, the principle of information hiding is described as creating "black boxes." These boxes have restricted slots in and out of which data can pass. If you think of them as physical boxes, you could imagine breaking them open and looking inside. If you were to break open the agents version of the summation box, you would find that it contains two smaller boxes labeled L and R. If you were to break open one of those, you would find still two more, and so on, recursively. This boxes-inside-boxes structure is one big static network.

The fat tree version also consists of boxes inside boxes. If you were to break open the outer black box, you would find two smaller boxes labeled L and R, just like in the agents version. Each of these contains two more, and so forth. But the tree as a whole is not a static network. All the nodes are connected by pointers. Whenever you use groups to create boxes inside boxes, the result is a dynamic network.

The inability of groups to statically create boxes inside boxes means that they cannot compose static networks modularly from smaller networks. Consider the three-matrix problem. In that case, we started with three prebuilt black boxes: the *matmul* class, the *rows_to_cols* class, the *elts_to_cols* class. We then assembled these black boxes together into one big static network. The agent construct lets you construct a static network by composing smaller static networks. Groups cannot do this.

The ability to embed a module hierarchy in a static network leads to many benefits. In section 5.1.4, we showed that agents dramatically increase the elegance of callbacks across module boundaries. Of course, this would not be possible if agent networks could not span module boundaries. In section 5.2.2, we showed how agents could define a set of lexical names visible to a hierarchy of agents. Of course, this would not be possible if agents could not represent a hierarchy. The ability to embed a module hierarchy in a static network is extremely powerful, it should not be underestimated.

By way of contrast, the *inability* to embed a module hierarchy in a static network leads to a modularity problem. In some problems, you *need* a static network. If you need a static network, and cannot put a module hierarchy in a static network, then you must not create a module hierarchy. You must implement your entire problem in one module.

5.5 The Limitations of Agents

While static networks enable one to do some powerful things, they have certain disadvantages and limitations of which one needs to be aware.

5.5.1 Garbage Collection

Traditional garbage collectors scan for inaccessible objects. That will not work when using agents: the connectivity of the static network is too high, every object is accessible to every other. The garbage collector would never find anything to be inaccessible. At a glance, it appears that we can not write a garbage collector for agents.

That does not mean it is not possible in principle. The purpose of a garbage collector is to free the objects that are not being used. Such objects certainly exist in static networks. If we could come up with a way to find them, we could free them. In theory, it is possible. But we can not do it by proving that the objects are inaccessible. We will have to find some *other* way to prove that they will not be used any more.

Before you conclude that that is impossible, remember that no garbage collection scheme is perfect. There are always objects that have to be freed manually. For example, LISP programmers sometimes have to set variables to NIL to encourage the garbage collector to free certain objects. Such clearing of variables is a form of manual deallocation. A perfect garbage collector would not require any manual deallocation, but such a thing is impossible. To do that, the system would have to figure out which objects are not going to be used any more, regardless of whether they are accessible or not — a problem which is clearly as difficult as the halting problem.

What makes traditional garbage collectors useful is that they get rid of most of the objects that need freeing, and the few that are left behind are easy to deal with manually. We would like to find something equivalent for agent hierarchies. However, we have not yet found any such thing.

It should be mentioned, for completeness, that our current inability to garbage collect agents does not impact ordinary objects. We can still garbage collect dynamically-allocated using traditional methods.

The consolation is that agent hierarchies have fundamentally simpler structure than dynamically allocated networks. Dynamically allocated networks have aliasing and back-pointers, both of which make manual deallocation extremely difficult. But static networks are simple trees. True, during computation, they may communicate in complex patterns. But once the computation's over, all that is left is a simple tree. That makes it easy to deallocate them manually. In particular, we provide two primitives *detach* and *hdetach*. The first releases a single agent, the second releases an entire hierarchy.

In summary, it may, in principle, be possible to garbage collect agent hierarchies. However, we have not yet discovered a way to do this. Until then, I am providing manual deallocation primitives, which so far, have proved to be easy to use. Nonetheless, garbage collection would make an interesting research topic for the future.

5.5.2 Load Balancing Issues

The agent declaration relies on a function that maps agent IDs to processors. The existence of this function is both advantageous and disadvantageous. If the function is well-chosen, it will immediately create a good load balance. However, if the mapping function is poorly-chosen, it will immediately create a bad load balance. This is a potential danger of agents.

The default mapping function provided in the agents runtime system is a hash function: it maps agent IDs to processors in a more or less random fashion. This typically creates a very evenly-distributed load, but poor locality. In other words, it is an adequate mapping function, but hardly ideal. One probably ought to try for better.

Of course, the user can take control of the mapping using the *on* clause. This usually solves the problem, but it requires user effort.

Strictly speaking, agents do not *have* to be implemented with a mapping function. It would also be possible to use system-wide directory that maps IDs to processors. Hybrid approaches would also be possible, in which most agents utilize the mapping function, but for certain kinds of agent, the directory is consulted. The use of directories of course frees the runtime system to make completely arbitrary load placement and migration decisions. The price, of course, is that the directories must be maintained.

Another alternative is to use virtual processors. To use this approach, creates 100 times as many virtual processors as real processors. One then uses a mapping function to map the agents onto virtual processors. One then dynamically load balances the virtual processors across the real processors. This achieves dynamic load balancing without the need for a full-fledged directory scheme (though one still needs a simple directory to find virtual processors).

In short, the mapping function provides you the *opportunity* to design a good static load balance, but it also largely *requires* you to design a good static load balance. The alternative is to use a directory for the mapping, which is more expensive, but it frees you to do arbitrary load migration.

CHAPTER 6

THE PROTOTYPE COMPILER

We have developed a prototype compiler that supports the agent construct within a Java-like framework. The prototype compiler achieves two important ends:

1. When an idea is not realized in software, it is sometimes possible to gloss over important details. Having an implementation makes one more keenly aware of minutiae that otherwise might be overlooked.

2. It is obvious from the design of agents that it is bottleneck-free and fairly inexpensive. Nonetheless, the implementation gives a better idea of what the actual performance numbers will be, and what the efficiency issues are.

Section 6.1 describes the history and evolution of the agents compiler. Section 6.2 describes the agents compiler and runtime system as it currently exists. Section 6.3 describes our observations about the system's performance. Section 6.4 discusses the efficiency that could be potentially be achieved if the system were optimized.

6.1 History and Evolution of the Compiler

The agents work was born out of an older project in which a new programming language was being developed. The primary goal of the new language was to serve as a testbed. We had conceived of several new constructs that we felt were interesting, and we wanted to know if they were useful in practice. We began designing a language to experiment with these new constructs.

It was necessary to choose a type system for the new language. At the time, we had recently learned about a system called Concert [5, 18, 22, 23] that accepts a dynamically-typed input language and converts it to statically-typed code. This design intrigued us, and we decided to duplicate it within our own testbed language.

As we worked on the compiler for the new language, we came to realize that one of the new constructs was more important than the rest: the *agent* construct. We began discussing this construct with the people around us. However, we kept encountering an obstacle: the new language was so unusual that nobody could read our code.

Eventually, it became clear to us that if we wanted people to understand the agent construct, it would be necessary to put it into a more familiar host language. But by this time, the compiler supporting agents was already well underway. Since we had no interest in starting over, we simply retrofitted a Java syntax onto the existing compiler.

The retrofitting was mostly a success, but there are certain ways in which the original design still shows. In particular, the compiler was designed to accept a dynamically-typed input language. Changing such a fundamental design parameter would have required almost a complete rewrite. Therefore, to retrofit a Java syntax onto the compiler, it was necessary to turn Java into a dynamically-typed language. To do so, we simply ignored all the type assertions in the source code. When the compiler sees a declaration like *int x*, it simply simply treats it as if it says *x is a variable*.

The type inferencer has not been implemented, which is unfortunate, considering the extent to which it drove the design. As time went on, though, it became clear that we were overly optimistic about how long it would take to implement such a thing. As we spent more and more manpower implementing the type inferencer, it gradually became clear to us that our efforts were distracting us from the real goal: testing the agent construct. We realized that static typing, though desirable in any language for speed, is actually not necessary to demonstrate the effectiveness of the agent construct. Therefore, we decided to complete the system with dynamic typing.

The compiler is now fully operational, with its new Java-based syntax. It has successfully compiled all the programs in this dissertation.

6.2 Overall Description of the Prototype

This section describes the implementation of the agents language.

6.2.1 Support Libraries

We utilized two libraries, Converse [24], developed in part by the author at the University of Illinois Parallel Programming Language, and DSlisp, developed by the author as part of some work for the US Army Construction Engineering Research Laboratory.

Converse is a runtime system for parallel programming. It supports most of the functions that parallel language designers need. It includes messaging, thread support, queuing, and a large set of other features. In particular, it contains support for asynchronous remote function invocation. This is particularly useful when trying to implement asynchronous remote method invocation.

DSlisp is a partial implementation of Common Lisp, which was designed to be used as an embedded language within a larger C application. For example, one could write a spreadsheet in C, and use Common Lisp expressions in each spreadsheet cell. DSlisp provides a library *liblisp.a* and a C header file *lisp.h*. The header file defines the C type *lsp* which can hold any lisp object. The library contains numerous functions for manipulating lisp objects, including C functions named *cons*, *car*, *cdr*, *eval*, etc. Figure

```
lsp l_nconc(lsp l1, lsp l2)
{
  lsp l = l_last(l1);
  if (!l_cons_p(l)) return l2;
  else { l_cons_cdr(l)=l2; return l1; }
}
```

Figure 6.1 Calling the DSLisp library from C

6.1 shows a typical example of some C code that uses the DSlisp library. This code implements the Common Lisp function *nconc*, which splices two linked lists together. It accepts two arguments *l1* and *l2* which are both lisp objects (lists, to be specific), and returns a lisp object. It uses the lisp library function *last* to find the last cell in the first list. If the result is not a cons cell, then the first list is empty, and it returns *l2*. Otherwise, it sets the *cdr* of the last cell in *l1* to *l2* and returns *l1*. This demonstrates how it is possible to manipulate lisp data structures like cons cells and the like from inside C. We found DSlisp a useful for implementing agents because the language is dynamically typed. DSlisp gives us the tools we need to manipulate dynamically-typed data.

6.2.2 The Agents Compiler and Linker

The agents system consists of a compiler, linker, and runtime system. The compiler and linker are both written in Common Lisp [25]. They both read agents code as input, and generate C code as output.

An agents program may consist of multiple source files, each of which ends with the extension *agt*. These must be compiled one at a time by the agents compiler. It reads one source file, and emits a C translation of that file, and a *lnk* file. The *lnk* file contains

```
1 word: tag field
1 word: name
1 word: sleepers list
? word: first instance variable
? word: second instance variable
...
```

Figure 6.2 Object Layout in the Agents Runtime System

a list of the classes and methods defined inside the C file. For example, when compiling *file1.agt*, the agents compiler emits *file1.c* and *file1.lnk*.

After compiling all source files in this manner, one runs the agents linker, which reads in all the *lnk* files. It then emits the single file *agentlink.c*. One must use the regular C compiler to compile *agentlink.c* and all the C files emitted by the agents compiler. One must then link them all together, including the Converse library, the DSlisp library, and a small agents library. The result is a working agents program.

The agents compiler proceeds in several passes. First, it parses the input file and generates a parse tree. Then, it converts the parse tree into an intermediate language designed for simplicity. This, in turn, is converted into an intermediate language designed for optimization. The code is converted to SSA and then a few minor optimizations (dead code elimination, etc) are performed. Finally, the code is converted to C.

6.2.3 The Data Structures in the Runtime System

This section describes the data structures manipulated the runtime system. Obviously, the most important entities manipulated by the runtime system are the Java objects themselves. The Java classes are registered as "new types" using a mechanism provided by the DSlisp library. Hence, they are valid Lisp objects and can be passed through Lisp routines. The layout of an object is shown in figure 6.2. The *tag* field is a small integer used to identify the type (class) of the object. The *name* field contains a pointer to the agent ID. The *sleepers* list is a linked list of threads waiting for the object to be modified.

For every normal class C, the compiler generates a proxy class (which in the implementation has no name, but we will call it PROXY_C for the sake of discussion). Proxy objects are truncated versions of the objects they mimic. They contain the *tag* field and the *name* field, but the rest is discarded. The tag of the proxy object is not the same as the tag of the real object. In addition, for every normal class C, the compiler generates a multicast proxy MPROXY_C. The multicast proxy is laid out just like a regular proxy, but its agent name contains numeric subranges ranges where the indices would normally be.

The representation of agent IDs depends on whether the object is a dynamically-allocated object or an agent. For dynamically allocated objects, the representation is a Lisp list of the form *((T classdeclnumber processor address))*. For example, suppose that some code executing on processor 5 were to dynamically allocate an object of class *smooth*, and suppose that *smooth* is the 17th class declaration in the source file, and suppose that the object's address turns out to be 340904. In that case, the ID would be *((T 17 5 340904))*.

For agents, the representation of the ID is a Lisp list of the form *((agentdeclnumber index1 index2 ...) . owner)*. For example, suppose that *smooth* object we just discussed were to evaluate the expression *calcpatch(97,98,99)*, and suppose that the calcpatch declaration is the 13th agent declaration in the source file. In that case, the agent ID would be *((13 97 98 99) (T 17 5 340904))*.

Agents relies on a distributed table to map agent IDs to objects. Given our representation of agent IDs, this was trivial: each processor simply allocates a Common Lisp hash table to hold the objects.

Our choice to use Lisp objects for all our data structures was a difficult one. We acknowledge that allocating cons cells for agent IDs is costly, and that Common Lisp hash tables do not use incremental hashing as one would normally want for agent IDs. In fact, we initially tried to use much faster data structures. For example, we chose the optimized representation of agent IDs described in section 4.19. However, we found that our attention to speed was consuming inordinate amounts of manpower. We were spending too much time on issues like message packing, encoding and decoding IDs, type inference, and so forth. Finally, we concluded that the attempts to achieve maximal efficiency were distracting us from the real topic: static networks.

6.2.4 The Emitted Code

The agents compiler converts agents code to C code. This code interfaces to the Converse and DSlisp libraries. The most important translation the compiler performs is to translate agents methods into C functions. The name of the C function is a concatenation of the class name and the method name. For example, if a class FOO contains a method BAR, a function M_FOO__1__BAR might be generated. The M is a prefix common to all methods. The *1* in the name is merely a sequence number indicating that BAR is the 1st method in class FOO. The sequence number is only there in case a class contains more than one method of the same name. The first argument of the C function is always an object handle, the second argument is the ID of the agent who invoked the method (useful for implementing *from* clauses), and the rest of the arguments are the arguments that occur in the source code.

For every method name, the agents linker generates a C function. For example, if any class uses a method named BAR with 2 arguments, the agents linker generates a C function named VM_2_BAR. This function is called a dispatcher. It contains a *switch* statement that checks the type of the object, and invokes the appropriate method. For example, if one passes an object of type FOO to the dispatcher VM_2_BAR, the dispatcher would invoke M_FOO__1__BAR.

If one passes an object of class PROXY_FOO to a dispatcher like VM_2_BAR, the dispatcher checks whether or not BAR is a relay method in FOO. If it is, the dispatcher treats the proxy as if it were a real object of class FOO. In other words, it invokes M_FOO__1__BAR directly on the proxy. On the other hand, if it is not a relay method, the dispatcher arranges for a remote method invocation. It checks the name in the proxy and determines the location of the real object. It packs up the arguments and transmits them to the processor holding the real object. The object is looked up, and the method is invoked. When a dispatcher sees a multicast-proxy object, it arranges for a multicast to occur.

In addition to the regular dispatcher VM_2_BAR, the agents linker also generates a second dispatcher ST_2_BAR. The ST stands for "start thread." The only difference between the two dispatchers is that the VM dispatcher is synchronous. It waits for the method to complete and returns the value. The ST dispatcher is asynchronous. It arranges for the method to be executed in a separate thread. The ST dispatcher always returns NIL.

The constructor is a method like any other, the only difference being that the system knows when to call it. When you allocate an object with *new*, the task is easy. The *new* operator allocates the amount of memory needed to hold the object, initializes the *tag*, *name*, and *sleepers*, and then calls the user's *init* method. The code is straightforward.

`class C`
`{`
`  agent node(int i) is D(i+1);`
`};`
	`class C`
	`{`
	`  agent node(int i) is D [init4893];`
	`  static void init4893(D obj, int i)`
	`    { obj.init(i+1); }`
	`};`

Figure 6.3 Generating Secondary Constructors

The right hand side of agent declarations frequently contain initialization expressions. Consider, for example, the agent declaration in figure 6.3, which contains the expression *i+1*. Such expressions are moved into a method, called a "secondary constructor," as shown on the right hand side of figure 6.3. The secondary constructor accepts an uninitialized agent and its indices, and initializes it.

The agents linker emits an *agent table* mapping agent declaration numbers to agent information. The agent information includes: class size in bytes, class tag, and which secondary constructor to call to initialize the object. When the agents runtime system receives a method invocation for an agent ID, it looks in the in its hash table to find the object. If no such object exists, it looks at the first number in the agent ID. This gives it the agent declaration number. It uses this information to look up the class size in bytes, class tag, and the secondary constructor. It allocates the object, sets the *tag*, initializes the *name*, and clears the *sleepers*. It puts the object into the global agent table. Finally, it calls the secondary constructor, passing in the agent and its indices.

The implementation of the *on* clause is similar to the implementation of secondary constructors. The expression in the *on* clause is moved into a static method. The *agent table* then indicates which static method is to be used for mapping the agent.

The *wait* statement, though orthogonal to our objective (static networks), is nonetheless somewhat interesting. This is one of those cases where the implementation showed us some details that we would have overlooked. Our original implementation was as follows. When a thread executes a *wait* statement, it pushes itself on the list of *sleepers*, and then suspends. The compiler automatically inserts a *wake_sleepers* directive at the end of every method that modifies an instance variable (the directive awakens every thread on the *sleepers* list).

As it turns out, this approach created a deadlock in one of our sample problems. We realized that a method had modified an instance variable. This should have awakened the sleepers, but the method blocked in a *wait* statement before it made it to the *wake_sleepers* directive. The obvious solution was to put the *wake_sleepers* after every assignment statement, but that would have really slowed down all assignments to instance variables. Another solution was to put the *wake_sleepers* directive in front of *wait* statements. This would have usually worked, but it could lead to busy waiting in situations where two *wait* statements both occur inside loops, both concurrently waiting for two different conditions. Our temporary solution was to add the conditional form of the wait statement. It does only one *wake_sleepers* no matter how many times the condition is tested. We do not have an ideal solution to this problem at this time, which is unfortunate, as we find the *wait* statement to be a very convenient form of synchronization.

We built a foreign function interface into our agents implementation that allows us to call into Lisp. This saves us the effort of writing a "standard library" for agents. For example, if an agents program wishes to output something on the console, it can use the Lisp function *print*. Since Lisp functions can call C functions, the Lisp foreign function interface also allows us to call into C, indirectly.

In summary, the implementation of agents turned out to be relatively straightforward. The data structures were not overly complex, and the algorithms needed were not difficult. This is desirable. The simpler a construct is to implement, the more likely it will become part of the mainstream of technology, which is what we sincerely hope will occur someday.

6.3 Performance Results

We have timed the performance of the prototype implementation on two platforms: a network of HP 735 workstations, and on the Sandia machine ASCI RED.

The first program we implemented was N Queens. It is the largest of the test problems, consisting of six primary classes: *summation*, *accumulator*, *quiescence*, *writeonce*, *statistics*, and *nqueens*.

The *summation* class is intended to add up a predetermined number of values. One sends in the values one at a time by invoking the *input* method. When one has fed in the predetermined number of values, the *summation* object invokes the *total* method on its owner. The summation class is described in more detail in section 5.4.1.

The *accumulator* class is intended for adding up values as well, but it is used when one does not know how many values must be added up. It has a method *add* for adding to the total, and a method *collect* to fetch the total when one is done. The total is not kept in a central location. Instead, each processor has a local total, all of which must be added to get the global total. To implement this, the *accumulator* has an agent on each processor. When one invokes the method *add* on the *accumulator*, the invocation is relayed to the local agent of the *accumulator*. Hence, invoking *add* never involves communication. The *accumulator* has an agent of class *summation* to handle its reduction. When you invoke *collect*, the *accumulator* relays the *collect* message to its agents, who respond by feeding their values into the *summation*. When the *summation* calls back with a *total*, the total is returned from the collect method.

The *quiescence* class is intended to determine when a search tree has expired. It has two methods *node_created* and *node_destroyed* which one calls whenever search tree nodes are created or destroyed. It also has a method *wait_quiescent*. Its implementation uses two agents, both of class *accumulator*, to keep track of the two totals. It relays the *node_created* and *node_destroyed* messages to the accumulators. Since adding to an accumulator doesn't require communication, counting a *node_created* or *node_destroyed* doesn't require communication either. The *wait_quiescent* method contains a loop which sleeps for a while, then *collect*s both accumulators to compare the totals. It returns when the totals are no longer changing and are equal.

The *writeonce* class is intended to store a value for fast retrieval. It has an agent on each processor. When you invoke *set* on the writeonce, it relays the *set* to each of its agents. Thus, every processor will have a copy of the value. When you invoke *get* on the *writeonce*, it relays the invocation to the local agent of the *writeonce*, so *get* never requires communication.

The *statistics* class is there solely to keep track of how many objects are created per processor. Its design is much like the *accumulator*, except that there is no collection.

Finally, the *nqueens* object is as it is described in section 5.2. It is slightly more complex, as we added the statistics-counting code and some grainsize control code.

The grainsize control code serves two functions. N Queens creates a very large search tree where each node does very little work. It can easily overflow the available memory by creating such a large tree, and it is inefficient for an object to do so little work. To improve the grainsize, we cut off the search tree at a certain depth. When a search tree node discovers that it is at the depth limit, it uses a sequential subroutine to explore its subtree rather than creating more parallel tasks. This is a typical strategy for speeding up breadth-first AI search problems on parallel machines.

CPUs	13-Queens on HP735s (sec)	15-Queens on ASCI RED (sec)
1	163	-
2	87	-
4	46	418
8	26	211
16	-	107
32	-	54
64	-	27
128	-	14

Figure 6.4 Performance of the Prototype on N-Queens

We ran 13-Queens on our network of HP 735, and 15-Queens on ASCI RED. In both cases, the depth limit was 5. The 13-Queens generated 38680 objects, and the 15-Queens generated 105,370 objects. The results are shown in figure 6.4.

The second problem we implemented was the smoothing problem. This problem was far simpler, it only contained the two classes *smooth* and *smooth_patch*. The implementation is exactly as described in section 5.3. The only addition was the use of an *on* clause to create a block decomposition. We ran a 16x16 grid of patches where each patch was 100x100 pixels on the HP 735s, and a 16x16 grid of patches where each patch was 200x200 pixels on ASCI RED. We smoothed both for 20 iterations. The results are shown in figure 6.5.

Finally, we implemented the three-matrix multiplier. It uses the classes we described in section 5.1: *mulabc*, *matmul*, *cols_to_rows*, and *elts_to_rows*. We did not expect good speedup from *mulabc*, because it requires a great deal of communication. We ran a 16x16 grid of 50x50 patches on the HP 735s, and a 16x16 grid of 85x85 patches on ASCI RED. The results are shown in figure 6.6.

One thing that concerned us about these measurements is that they might be misleading. Dynamic typing reduces sequential thread performance, and ironically, low

CPUs	1600x1600 on HP 735s (sec)	3200x3200 on ASCI RED (sec)
1	130	-
2	68	-
4	35	65
8	22	24
16	-	12
32	-	7
64	-	4
128	-	3

Figure 6.5 Performance of the Prototype on Pixel Smoothing

CPUs	800x800 on HP 735s (sec)	1360x1360 on ASCI RED (sec)
1	102	-
2	54	200
4	32	110
8	-	60
16	-	39
32	-	28
64	-	21

Figure 6.6 Performance of Prototype on MulABC

Cutoff	Objects	Grainsize	Seconds
4	15942	110.4 millisec	55.3
5	105370	16.9 millisec	55.6
6	568408	3.1 millisec	59.4
7	2466110	0.7 millisec	75.0

Figure 6.7 Adjusting the N Queens Cutoff to Vary Grainsize

sequential thread performance makes the speedup numbers look good. For example, if we slow down the patch-smoothing code by a factor of two, the relative speedups will improve. This occurs because slowing the inner loops increases the problem's grainsize, which puts less demand on your communication overhead and latency. Therefore, we decided to stress-test the system, to determine its limits in terms of grainsize. The first step was to rewrite the inner loops in optimized C. We did not rewrite any parallel code, only sequential subroutines. Specifically, we rewrote the pixel smoothing for a single patch, the matrix multiplication code for a patch, and the sequential part of the N Queens search. This reduced the grainsize, making it harder to get perfect speedups. All the numbers we have quoted utilize the optimized C code.

We ran an experiment in which we varied the grainsize and measured its impact on performance. As a first step, we figured out how much real work is involved in computing 15 Queens by running it on a single sequential processor. It took 1760 seconds. Then, by varying the cutoff depth in N Queens, we were able to divide that time up into smaller and smaller pieces. We ran 15 Queens on 32 processors with various cutoffs. The results are shown in table 6.7. We computed the average grainsize by dividing the total work (1760 sec) by the number of objects. With a grainsize of 3.1 milliseconds or greater, we had good performance. At 0.7 milliseconds, we had moderate but not excessive deterioration.

In summary, the system works, and it exhibits reasonable speedups. On a network of HP workstations, it is medium-grained, capable of efficiently handling tasks in the 1 millisecond range or larger.

6.4 Potential Efficiency

It is interesting to ask how efficient agents could conceivably be. By this, we mean: if we were to optimize the agent mechanism as much as possible, how many microseconds would it take to execute a message transmission using the agent primitive? The prototype implementation does not really answer the question, since it was not designed for efficiency. However, it is given our understanding of how to implement agents, it is straightforward enough to manually generate optimized code, and then time it. We will time the code for the following statement:

```
calcpatch(row, col, iter)<-prev_w(val);
```

While this is really two expressions (one that builds a proxy, and one that invokes a method on the proxy), any reasonable optimized implementation would treat it as a single message-send operation. We will assume that a optimized implementation uses the encoded representation of agent IDs described in section 4.19, not the list representation used in the prototype.

The compiler first determines that the proxy is for an object of class *smooth_patch* and that the method *prev_w* is not a relay method. Hence, this is a real remote method invocation, not a relay invocation. This determines what kind of code it emits.

For remote method invocations, the first step is to allocate a message buffer. Assuming we allocate it on the stack, we can do this in one machine instruction (decrementing the stack pointer). Into this buffer we copy the agent ID of the method's invoker. This requires a string-copy. Assuming the *smooth* object is a top-level object, the length of the copied string will be 16 bytes (see the rules of section 4.19).

The next step is to concatenate the *calcpatch* information to the ID in the message. This consists of storing the three integer indices *row*, *col*, and *iter* of the calcpatch into the message, an agent declaration number indicating that it is a *calcpatch* object, and the length of the *calcpatch* information (4 machine words). We update the length field in the ID by adding 4.

In addition to the target's ID, we must also store the sender's ID. Fortunately, the ID of the sender is almost always a substring of the ID of the receiver or vice-versa. In this case it is sufficient to store only one ID and two length fields. This is what we do in this example. The only case when this approach does not work is in the case of direct (non-relayed) sibling-to-sibling transmission. In that case, a different message format (containing two IDs) must be used. Of course, the *from* information can be completely omitted if it can be determined that the method being invoked does not have a *from* clause.

Finally, the hash value in the agent ID must be updated. Of course, there are any number of hash functions to choose from. A reasonable choice would be the hash function described in [26]. It is very carefully designed to minimize hash-bucket collisions. This will be useful if we want to use it for pseudorandom load-balancing. However, it uses quite a few CPU cycles in its attempt to be as perfectly pseudorandom as possible. Hence, it is probably overkill for our purposes. We choose it for our timing tests because it is a worst-case scenario: other hash functions will only be faster.

Of course, these steps do not include the work of placing the data in the message or the actual process of sending the message. We will not time those steps, because they are exactly the same as in any other system that performs remote method invocation. We only want to measure the overhead added by agents.

We implemented code that simulates these operations, and timed it. The code executed in 0.5 microseconds on a Cyrix PR233. In other words, the total time needed for remote method invocation using an agent ID should be about 0.5 microseconds longer than a remote method invocation that uses a global pointer.

When the message arrives at the destination processor, the receiver also incurs a small amount of overhead, though less than the sender. In particular, it must locate the object in the agent-ID-to-object hash table. Finding the right hash bucket is very quick, as the hash value is already stored in the message. However, this would be followed by a string comparison between the agent ID in the message and the agent ID in the object. This should take considerably less time than the string copy to put the ID in the message in the first place.

It is our judgment that this amount of overhead (0.5 microseconds at the sender side, somewhat less at the receiver side) is within the acceptable range. There are a few circumstances when the overhead would be larger.

The first situation that would increase the overhead is when a *from* clause and a direct sibling-to-sibling transmission mandate the inclusion of two agent IDs in the message. This will increase the overhead by a small constant at the sender side, and would not affect the overhead at the receiver side.

The costs could jump significantly when an agent's indices are not integers. For example, suppose an agent has an index that is a string, and suppose the string is quite long. In that case, the entire string becomes part of the agent ID. The string will need to be copied into the message and hashed. Because of this, the programmer should be aware of the cost of long string indices: they slow down all messages to that agent and its sub-agents. In N Queens, each agent is indexed by a partial placement, which is a list of integers. The list of integers becomes part of the agent ID. To pick a concrete example, suppose we are running 10-queens, and suppose that the encoding of a list of integers takes 4 bytes per integer. In this case, the index will take 10 machine words, raising the length of the agent ID to 16 machine words. We tested the speed of this operation, and discovered that the sender-side manipulations jumped from 0.5 microseconds to 1.1 microseconds. That is still within the acceptable range. In general, little strings like those used in 10-queens are not a problem. However, one does have to be aware of the costs, and not utilize very long strings as agent indices (or use them with the awareness that they may add a few microseconds to the message overhead).

The second situation in which costs increase is when dealing with deeply-recursive agent hierarchies. To pick an arbitrary example, suppose that we were to create a summation tree that spans 1,000,000 objects using the recursive formulation in section 5.4.1. In that case, the depth of the tree would be about 20. The agent IDs of the leaves would contain 20 segments. The segment contains no indices, so it will be one machine word (4 bytes). The total length of the longest IDs will be 24 machine words. We timed this length on a Cyrix PR233 again, obtaining 1.0 microseconds on the sender side. Again, that is within the acceptable range. In general, agent nesting to depth $log\ N$ does not create a problem. However, recursive nesting beyond $log\ N$ depth might be cause for concern.

In short, the costs of agent ID manipulation in the normal cases are minimal. They could conceivably get large in two cases: when using long strings or long lists as agent indices, or when using unlimited agent nesting. Because of these two possibilities, the programmer must be at least partially aware of the cost model, so he can avoid (or accept the cost of) these cases.

Our evidence seems to indicate that an optimized version of agents could be reasonably competitive in terms of performance. In particular, agents does not appear to introduce excessive overhead into the process of message transmission and message reception.

CHAPTER 7

RELATED WORK

This section describes other systems and other language constructs that can achieve a portion of what *agents* achieve.

7.1 Concurrent Object Oriented Languages

The *agent* construct is meant to be added to any existing object-oriented parallel programming language. Clearly, without the foundation provided by those languages, the agent construct would not be relevant. The vast majority of the implementation techniques for agents are derived from techniques used in other object-oriented languages.

Even static networks are not new, there are many older constructs that can create static networks. For example, consider the parallel programming model called the "Chare Kernel," [7] later renamed to "Charm" [8, 9, 10]. When the Chare Kernel was first invented, it supported only dynamically allocated objects. It was soon clear that something more was needed. The *branch office* (a kind of group) was added to meet the need. It was realized at some point that branch offices were not enough, so another static network primitive was added: the distributed hash table. Later, it was found that those two were not quite enough either, so a third static network construct was added: the multidimensional object array.

There are really only two main differences between the agent construct and the older static network constructs. First, the agent construct attempts to be structurally general: it attempts to support all shapes of static networks. Second, it supports modularly-defined static networks.

7.2 Hash Tables and Hash Groups

Distributed hash tables are an older construct for static networks. A few existing parallel languages provide them, for example, Charm [9, 10] and Multipol [27]. Many other languages can implement them easily enough. They are quite capable of representing arbitrary network shapes. They have several limitations, however.

Since they were designed for shared memory machines, they require the presence of a powerful optimizer to work efficiently on distributed memory machines. Only a few reasearch systems [4, 19, 20, 21] are currently powerful enough to use them efficiently, to our knowledge. Section 5.3.3 discusses this limitation.

The objects which are inserted into hash tables are not part of the static network: they are only *pointed to* by the static network. That makes it impossible for the runtime system to look at such an object and determine where it fits into the network. Consequently, hash tables cannot support the *from* clause. Nor can they support any other analysis that requires looking at an object and identifying its position within the overall computation. For example, the structure-driven load balancing we described in section 5.5.2 and the program trace analysis we described in section 5.3.6 are not possible with hash tables.

But the biggest limitation of hash tables is that one cannot use them compositionally, to create a large static network from multiple smaller static networks. As a consequence, hash tables cannot be used to create better support for callbacks, as described in section 5.1.4. They cannot improve the compositionality and transparency of the language, as described in section 5.1.5. They cannot be used to implement shared identifiers, as described in section 5.2.2. Finally, they cannot be used in situations were the definition of the static network spans module boundaries, as described in section 5.4.5.

Hash groups are a variant of hash tables. They were designed to be similar to hash tables, but without the shared memory limitation. And indeed, they are naturally faster than hash tables in distributed memory. However, they have some new limitations. The first is that they can only create static networks containing one type of object. The second limitation is that, like agents, they require compiler support for their special syntax and type rules.

Hash groups cannot overcome the most important limitation of hash tables: their inability to compose smaller static networks into a large one. In other words, hash groups cannot be used as callback support either, they cannot improve the compositionality of the language, they cannot support shared identifiers, and they cannot implement static networks that span module boundaries. See sections 5.1.4, 5.1.5, 5.2.2, and 5.4.5 for discussion of these issues.

7.3 Linda

Linda's tuplespace [28, 29, 30] is a disguised static network. This is not obvious at a glance. It becomes fairly obvious, however, once one is familiar with Linda's implementation. The usual implementation of Linda is to use a centralized database, which is of course a bottleneck. However, it is possible to write a scalable implementation. We will summarize the design of a scalable Linda implementation here.

```
void out(tuple t)
{
  for every query tuple q that matches t
    tuplespace[q].push(t);
}
```

Figure 7.1 Straightforward Implementation of Linda "Out" Primitive

```
[1,  2,  3]        [1,  *,  *]
[1,  2,  *]        [*,  2,  *]
[1,  *,  3]        [*,  *,  3]
[*,  2,  3]        [*,  *,  *]
```

Figure 7.2 Query Tuples Matching the Data Tuple [1, 2, 3]

```
void out(tuple t)
{
  for every query tuple q that matches t
    if there is an in-statement that could make query q
      tuplespace[q].push(t);
}
```

Figure 7.3 Optimized Implementation of Linda "Out" Primitive

```
tuple in(query qu)
{
  atomically {
    tuple t = tuplespace[qu].head();
    for every query tuple q that matches t
      if there is an in-statement that could make query q
        tuplespace[q].delete(t);
  }
}
```

Figure 7.4 The Pseudocode for *In*

In Linda, the *in* directive can be seen as creating a "query tuple," a tuple with wildcards in it. When implementing Linda, the first step is to create a hash group mapping query tuples to queues. The queues should be general purpose FIFO queues with a broad range of methods like *push*, *pop*, *head* (to examine the head without popping it) and *delete* (to scan the queue and delete a specific item). Let us assume that this hash group of queues is stored in a global variable named *tuplespace*.

The idea is that when an *in* statement issues a particular query tuple, all we need to do is look in the tuple queue associated with that query tuple. For this to work, our *out* primitive needs to follow the pseudocode shown in figure 7.1.

The problem with the straightforward approach in figure 7.1 is that there are too many query tuples that match any particular tuple t. For example, consider the tuple [1,2,3]. That tuple is matched by eight query tuples, shown in figure 7.2 (those asterisks are wildcards). To *out* that tuple, we would have to store it in eight places in the hash group. That is too many.

The solution is to add one more if-statement, as shown in figure 7.3. Of course, this implies that the Linda compiler must scan the source code and generate a catalogue of all the *in* statements. That is not too difficult, though. That one optimization is all we need. With that if-statement, tuples will only be stored in a few queues.

The implementation of *in* simply looks in the queue associated with the appropriate query tuple. There is one complication, since there are multiple copies of the tuple in multiple queues. The entire set of copies needs to be removed atomically by the *in* primitive. Rather than show the atomicity protocol, we will simply show pseudocode in figure 7.4. It chooses a tuple, then deletes it from all the places it was stored.

As you can see, the cost of a tuple is proportional to the number of queues it goes into. If a tuple goes into just the right queue, the one from which it will be retrieved, that is ideal. If it goes into a second queue as well, the program's communication cost doubles!

As observed by Thomas Christopher in [31], skilled Linda programmers have made themselves aware of the underlying hash group. They have to, because the speed of the Linda implementation is extremely sensitive to the number of queues a message goes into. Programmers must be *very* careful about making sure their tuples go into exactly the necessary queue and no others. They know the tuple queues are there, and they manipulate them consciously. To them, the *in* and *out* operators are nothing but *pop* and *push* operators that manipulate queues. They have little choice in this: they must see through the abstraction to the implementation underneath, or their programs will run many times slower. Novice Linda programmers may see a tuplespace with *in* and *out* operators, but skilled Linda programmers see nothing but a hash group of queues with *push* and *pop* operators. Therefore, we will view Linda tuplespace as they do: as a hash group of queues.

A hash group of queues has the same basic limitations as a hash table. First, it is fine-grained, like a hash table. In other words, takes a pop and a push operation to update a piece of data in the tuple space, and therefore multiple two-way communications. Tuple space is much faster in a shared memory setting than in a distributed memory setting. With hash tables, we could rely on a powerful optimizer to fix the problem. However, tuple space is so much more complex than hash tables, implementationally, that even a powerful optimizer is not likely to help.

The other problem with tuple space is that, like hash tables, it cannot be used to compose small static networks modularly into a larger one. This makes it unable to provide elegant callbacks, better language compositionality, shared identifiers, or networks that span module boundaries. See sections 5.1.4, 5.1.5, 5.2.2, and 5.4.5 for discussion of these issues.

7.4 AMDC

AMDC, developed at the Illinois Institute of Technology by T. Christopher, is the successor of MDC and LLMDC/C [31, 32]. AMDC defines the term *location*, which is primarily a storage location. The location contains a *table*, a device designed as a repository for arbitrary user data. AMDC allows one to send a message to a location. The message contains a C function pointer. The arrival of the message triggers the execution of a C function, which is passed two arguments: the contents of the message and the handle of the location. The function can then update the table in the location.

Each location is denoted by a location name consisting of a *symbol* and three integers. The symbol is an opaque data type intended solely to function as the identifier of a set of locations. The runtime system provides a function to generate unique symbols. Locations are lazily allocated, in the sense that their tables take up space proportional to the amount of data inside. Initially, the tables are all empty, hence they take no memory.

Although AMDC is not written in an object-oriented programming language, locations bear a certain resemblance to objects. They have the equivalent of instance variables (the table), and they can execute methods of a sort (messages trigger a C function). In a sense, then, the locations in AMDC form a static network of "location objects." AMDC is slightly greater in power than hash groups. Hash groups are restricted to one kind of object. AMDC is too, but the "location object" is a general-purpose object, it can be used for anything. So in a sense, AMDC networks support arbitrary hash groups without the burden of the type restrictions.

AMDC's static network primitive is probably the most powerful that can be created without compiler support, making it a reasonable tradeoff. The primary limitation of AMDC is that one cannot compose static networks modularly from smaller networks. See sections 5.1.4, 5.1.5, 5.2.2, and 5.4.5 for discussion of these issues.

7.5 ActorSpace

Agent IDs consist of a sequence of segments. For example, the agent ID *jacobi#567 @0.jnode(5,3)* consists of two segments separated by a dot. This segmented representation enables agent IDs to represent a hierarchy of agents.

There is another system, ActorSpace [33, 34], which also assigns segmented IDs to objects, arranging them into a hierarchy. ActorSpace was developed by G. Agha and C. Callsen at the University of Illinois, and was based on the already-existing actors [1, 2] programming model.

An actorspace is essentially a table mapping strings to actors [1], very much like a hash table. There are a few differences between an actorspace and a hash table, however. One does not look up actors in an actorspace, instead, one sends a message to a certain key and the message goes to the actor associated with that key. Actorspaces support

[1]The formal specification of actorspace is ambiguous about whether the key is a string, making it clear that there may be versions of actorspace in which the key is more structured. The prototype implementation uses a string.

pattern-driven multicast: one can send a method-invocation to a subset of the actors in an actorspace by specifying a regular expression matching a subset of the keys [2]. Actorspace also supports a pattern-driven nondeterministic send: one specifies a pattern, and the system delivers the message to an arbitrary actor whose key matches the pattern. Actorspaces can be inserted into other actorspaces, creating a hierarchy. One can then send a message to the root of the hierarchy, along with a sequence of patterns, separated by slashes. The patterns cause a traversal of the hierarchy to find a particular actor. Suppose, for example, that an actorspace A maps the key "foo" to an actorspace B. Suppose that actorspace B maps the key "bar" to an actor C. Sending a message to A with the sequence of patterns "foo/bar" will cause the message to be delivered to C.

Pattern sequences like "foo/bar" look remarkably like agent IDs. A first glance might lead one to the mistaken conclusion that agents and ActorSpace could serve the same function. However, actorspace hierarchies are assembled piece-by-piece. Actors are inserted and removed at will, they can move about the hierarchy, and even insert themselves in multiple locations. Whole branches can be moved from one place to another. In short, actorspaces are very dynamic. They may form hierarchies, like agents, but they are not static networks in any sense of the word.

The static structure of agent hierarchies is what make it possible to do the relay optimization, to support the *from* clause, to provide the callback and compositionality support, to eliminate synchronization during network construction, to support shared identifiers, and in general, to do most of the special things that agents can do. Actorspaces, since they are not static, cannot do those things. Of course, ActorSpace does many things that agents cannot: ActorSpace serves well as a directory mechanism for servers and clients in an open system, and it also supports powerful pattern-directed in-

[2] Again, the formal specification leaves open the possibility of a more powerful or structured pattern-matching facility. The prototype uses regular expressions.

vocation mechanisms not available in agents. There is some overlap between the two, but overall, the goals and abilities of the two systems are largely orthogonal to each other.

7.6 Letrec

Certain functional programming languages contain a construct "letrec" which binds a set of variables to a set of expressions. Some implementations of letrec make it possible to create circular data structures. Such circular structures are not static networks because they are not immutable. However, they do come into existence atomically.

Because letrec can create structures atomically, it is useful for parallel programming. For example, the matrix multiplication code implemented with letrec is shown in figure 7.5. We have translated letrec construct into a java-like syntax and placed it inside a class method *create*. The *create* method allocates all the objects and links them together using letrec. This simplifies the code significantly. Notice that most of the pointer-passing and synchronization code is absent, in particular, the *distribute_handles* method and the *wait* statements are gone.

The mulabc example showed that agents can be used to improve the compositionality of the language. Figure 7.5 shows that letrec can be used similarly. Clearly, it does nothing about the problem of distinguishing where a callback came from, but otherwise, it is helpful.

Letrec is different from agents in that it does not include any lazy allocation mechanisms. As such, it is only useful in situations where all the objects can be allocated in advance. For example, letrec does not help us to implement the smoothing problem, where there are too many patches to simply allocate them all at once.

In fact, the matrix multiplication problem might also be considered a problem where eager allocation is wasteful. Assuming that the system appropriately gives precedence to

```
class mulabc1 {
  void init(matmul mm1, elts_to_rows er) {};
  void row(int row, vector v) {
    mm1<-row_a(row, v);
  }
  void result(int row, int col, double d) {
    er<-elt(row, col, d);
} }

class mulabc {
  static mulabc create(Object outobj, int size) {
    letrec {
      matmul mm1 = newgroup matmul(intermed);
      matmul mm2 = newgroup matmul(mabc);
      cols_to_rows cr = newgroup cols_to_rows(intermed,size,size);
      elts_to_rows er = newgroup elts_to_rows(mabc,size,size);
      mulabc1 intermed = newgroup mulabc1(mm1,er);
      mulabc mabc = newgroup mulabc(outobj,mm1,mm2,cr);
      return mabc;
  } }
  void init(Object outobj, matmul mm1, matmul mm2, cols_to_rows cr){};
  void input_A(int col, vector v) {
    cr<-col(vol, v);
  }
  void input_B(int col, vector v) {
    mm1<-col_b(col, v);
  }
  void input_C(int col, vector v) {
    mm2<-col_b(row, v);
  }
  void row(int row, vector v) {
    mm2<-row_a(row, v);
  }
  void result(int row, int col, double d)  {
    outobj<-result(row, col, d);
} };
```

Figure 7.5 Matrix Multiplication Code using Letrec the first matrix multiplier, it will largely complete before the second multiplier begins. Allocating the second multiplier eagerly wastes memory. It may not seem a large waste, at first. But consider composing two of the *mulabc* classes together. There would then be a total of four multipliers, only one of which is in use at any given moment. Using an eager allocation strategy pervasively throughout the program will lead to a situation where all objects are allocated eagerly at the beginning and only a few are in use at any moment. This is not a design that one can use pervasively.

Letrec (and let) create new binding environments, but they do not make it possible to create environments which are visible to multiple objects. However, we could provide a new operator *(make-object* $< class - body >$*)* which defines an anonymous class and creates an object of that class in a single step. We could allow an object created in this manner to refer to the lexical environment in which it was created. Given such a construct, we could use it to write code like that shown in the lower half of figure 7.6. The code bears an obvious resemblance to the agents code above it, which we copied from section 5.2. It appears that this combination is able to create binding environments which are visible to multiple objects, much like agents can.

Unfortunately, it would be difficult to implement the latter code efficiently. Agent hierarchies are not just created atomically, they are also immutable. With agents, we do not have to perform a remote variable access each time we access N, A, or Q, as their values do not change. In fact, we can simply compute their agent IDs. In the letrec-version, we must perform a remote variable access each time we access N, A, or Q. So, while letrec (and let) can create new binding environments, those bindings cannot be accessed across boundaries without creating a bottleneck. A programmer using letrec would have to use the usual workaround: he would have to propagate values from object to object, rather than using the lexical scoping rules.

```
class nqueens {
  agent a is accumulator();
  agent q is quiescence();
  agent n is writeonce();
  agent nqueens_node(list queens)
          { ... code that refers to n, a, q ... }
  void init() { ... awaken nqueens_node(nil); ... }
}
```

```
(letrec ((a (make-accumulator))
         (q (make-quiescence))
         (n (make-writeonce))
         (nqueens-node (lambda (queens)
             (make-object ... code that refers to n, a, q ...))))
  ... (nqueens-node nil) ...)
```

Figure 7.6 N Queens using Agents (top) and Letrec (bottom)

In short, letrec, while it does create networks atomically, does not create true static networks. As such, it cannot be used to improve compositionality in large programs, to implement lexical scoping across objects, or to handle sparse networks of objects.

7.7 Arrays of Objects

A great many object-oriented parallel languages and libraries provide distributed arrays of objects [35, 36, 37, 7, 11, 5, 3]. Arrays are one of the thousands of possible shapes of static network.

The data parallel numerical languages such as Fortran-D [15] and HPF [16] do not create networks of objects, at least not in the usual sense of the word "object." However, one can loosely interpret an array element to be an object. Under this interpretation, data parallel languages manipulate static networks. Since these languages only support loop parallelism, the "objects" must all act in lockstep. That makes them a restricted form of arrays.

Languages like Chant [38] support the rope construct, which is a distributed array of threads. It is not too much of a stretch to interpret a thread as equivalent to an object. Both store data, both receive messages and act on them. Under such interpretation, the rope is a one-dimensional object array.

Object arrays are the simplest and most restrictive construct for static networks. They cannot be used in most situations where hash tables, hash groups, or agents can.

CHAPTER 8

SUMMARY

A static network is a network of objects which is created atomically, destroyed atomically, and immutable. We defined a new construct for static networks, the *agent* declaration. It was designed to be able to represent any kind of network statically. We specifically wanted it to be able to represent the following:

- networks of any shape
- arbitrary communication patterns
- heterogeneous networks with a mix of object types
- networks of unbounded size
- task parallelism and the task hierarchy
- networks composed modularly from smaller networks

After designing the *agent* construct, we then explored what could be done with agents by experimenting with case studies. There were four case studies: the three-matrix multiplication problem, the pixel smoothing problem, the N Queens problem, and the summation reduction. Each was designed to illustrate several of the things that could be done with agents. It was discovered that agents could achieve many goals, listed below.

Elimination of Construction Code. Agent networks are created automatically by the compiler and runtime system. No code is needed to allocate them. In particular, all the synchronization and pointer-passing associated with the incremental creation of object networks is eliminated. Potential race conditions are negated. This frequently leads to a 25% code size reduction. (See section 5.1.2)

Elegant use of Callbacks. Callbacks are ubiquitous in parallel programs. But dynamic networks make callbacks incredibly awkward: the *outobj* parameter is confusing, the inability to tell which object invoked the callback is a problem, and the glue methods that handle callbacks are inevitably a bottleneck. Agent networks eliminate all three problems, and callbacks become an elegant methodology. (See sections 5.1.3, 5.1.4)

One Task Per Object. Dynamic networks must be created a piece at a time, which takes time. The overhead may be significant, or even prohibitive, if one is following the software engineering principle of one task per object. Traditional language programmers amortize these costs by reusing objects, which leads to dramatic complexity increases. Agent networks are created instantaneously, eliminating the overhead, and allowing the uninhibited use of one-task objects. (See section 5.4.2)

Transparent Relationship between Network and Code. When composing pre-existing agents, there is a transparent equivalence between the network and the code. One draws a graph of how one wants the objects connected. One then writes a class declaration with one agent per vertex, and one relay method per edge. In traditional languages, the code does not directly represent the network. (See section 5.1.5)

Shared Variables. Traditional languages have support for global variables, which lead to nonreentrant code. Aside from that, they have no support for shared variables. Agents adds shared variables to the language without sacrificing reentrancy. The scoping of these shared variables is as flexible as the scoping of Pascal variables. (See sections 5.2.2, 5.2.3, 5.2.4)

Eliminates Weird Numbering Schemes. Parallel programmers are constantly inventing peculiar math formulae as a means to map their trees, graphs, grids, and other data structures onto vectors. Such numbering schemes tend to obfuscate the code. The presence of agents eliminates the need for such convoluted programming. (See section 5.4.3)

Simplifies the Language. The addition of agents to a language allows you to remove hash tables, groups, and global variables. The elimination of three constructs and their replacement by one leads to a simpler language. (See sections 5.2.2 and 5.2.5)

New Structures, New Algorithms. Agents can efficiently create networks in shapes that previous constructs could not. This makes it possible to use new kinds of algorithms. To pick an arbitrary example, we discovered that iterative grid computations can be efficiently expressed in three dimensions, which simplifies their design considerably. (See section 5.4.4).

Specialized Load Balancing. The compiler can read a program that uses static networks and can easily determine the structure of the network, for example, it can determine if the user is using a 3D grid. It can use this information to manipulate object placement. (See section 5.5.2).

Debugging and Trace Analysis. When looking at a program trace or debugging dump, the objects will not only have addresses, they will also have agent IDs. This will enable the traceback tools to not only say that *an* object did something, it will enable them to say *which* object did something. This could make it possible to build powerful debugging and analysis tools. (See section 5.3.6)

It is our intuition that this is not the end of the list, that there are many ways to utilize agents that we have not discovered. We discovered many of these ideas by accident: the idea of smarter program trace analysis merely occurred to us one day when we were debugging a difficult program. The idea of shared variables came to us rather by accident, when we were experimenting with nesting classes for a different reason. In short, we have stumbled across many of the ways to leverage and utilize static networks. But we do not know how many more ways there are, waiting to be discovered. Our intuition tells us that there is no limit to what can be done with the structural knowledge provided by static networks, that it can be used to drive optimizations, to make new language constructs possible, and to generally make the language more elegant, efficient, and modular. I hope that this short sampling will lead you to the same intuition.

APPENDIX A

THE EXECUTABLE CODE

The following listings contain the executable code as it was actually fed into the agents compiler.

A.1 Matrix Multiplication

```
class matmul {
  agent node(int r, int c) on (r/8) {
    vector rv; int r1, c1;
    public void init(int r, int c) { rv=nil; r1=r; c1=c; };
    public void row_a(vector v) { rv=v; };
    public void col_b(vector v) {
      wait (rv != nil);
      owner<-result(r1,c1,lisp(vecprod rv v));
    };
  };
  public void init() { };
  relay void result(int r,int c,double d) { owner<-result(r,c,d); };
  relay void row_a(int r, vector v) { node(r, 0..15)<-row_a(v); };
  relay void col_b(int c, vector v) { node(0..15, c)<-col_b(v); };
};
```

```
class conv2rows {
  agent node(int r) on (r/8) {
    vector result; int row, countdown;
    public void init(int r) {
      countdown=16; row=r;
      result=lisp(vecmake 16);
    };
    public void elt(int c, vector v) {
      lisp(vectransfer v result 0 c);
      countdown = countdown-1;
      if (countdown == 0) { owner<-row(row, result); detach; };
    };
    public void col(int c, vector v) {
      lisp(vectransfer v result row c);
      countdown = countdown-1;
      if (countdown == 0) { owner<-row(row, result); detach; };
    };
  };
  public void init() { };
  relay void elt(int r, int c, double d) { node(r)<-elt(c,d); };
  relay void col(int c, vector v) { node(0..15)<-col(c, v); };
  relay void row(int r, vector v) { owner<-row(r,v); };
};
```

```
class mulabc {
  public void init() { };
  agent mm1() is matmul();
  agent mm2() is matmul();
  agent er() is conv2rows();
  agent cr() is conv2rows();
  relay void input_a(int c,vector v)            {cr()<-col(c, v);};
  relay void row(int r,vector v) from cr()      {mm1()<-row_a(r,v);};
  relay void input_b(int c,vector v)            {mm1()<-col_b(c, v);};
  relay void result(int r,int c,double d)from mm1{er()<-elt(r,c,d);};
  relay void row(int r,vector v) from er()      {mm2()<-row_a(r,v);};
  relay void input_c(int c,vector v)            {mm2()<-col_b(c, v);};
  relay void result(int r,int c,double d)from mm2{owner<-result(r,c,d);};
};
```

```
class main
{
  agent m() is mulabc();
  agent c() is conv2rows();
  int countdown; double start;
  public void init() {
    vector v; int c,r;
    countdown = 16;
    start = lisp(wallclock);
    for (c=0; c<16; c=c+1) {
      v = lisp(vecrand 16);
      m()<-input_a(c,v);
    };
    for (c=0; c<16; c=c+1) {
      v = lisp(vecrand 16);
      m()<-input_b(c,v);
    };
    for (c=0; c<16; c=c+1) {
      v = lisp(vecrand 16);
      m()<-input_c(c,v);
    };
  };
  public void result(int r, int c, vector v) {
    c()<-elt(r,c,v);
  };
  public void row(int r, vector v) {
    countdown = countdown-1;
    if (countdown == 0) {
      double delta;
      delta = lisp(wallclock) - start;
      lisp(mprint "done at time " delta);
    };
  };
};
```

A.2 N Queens

```
class statistics {
  agent node(int i) on i {
    int total;
    public void init(int i) { total=0; };
    public void add(int n) { total=total+n; };
    public void print() {
      lisp(mprint "work on " node_current " = " total);
  };};
  public void init() { };
  public relay void add(int n) { node(node_current)<-add(n); };
  public relay void print() { node(0..(node_count-1))<-print(); };
};
```

```
class psummation {
  agent node(int i) on i {
    int total, countdown; object up;
    public void init(int i) {
      countdown = 3; total = 0;
      if (i*2 + 1 >= node_count) { countdown = countdown -1; };
      if (i*2 + 2 >= node_count) { countdown = countdown -1; };
      if (i) { up = node(lisp(truncate (i-1) 2)); }
      else { up = owner; };
    };
    public void add(int n) {
      total = total + n;
      countdown = countdown - 1;
      if (countdown == 0) { up<-add(total); detach; };
    };
  };
  public void init() { };
  public relay void add(int n) {
    owner<-total(n);
  };
  public relay void input(int i, int n) {
    node(i)<-add(n);
  };
};
```

```
class quiescence {
  agent count_created() on 0 is accumulator();
  agent count_destroyed() on 0 is accumulator();
  public void init() { };
  public relay void node_created()   { count_created().add(1); };
  public relay void node_destroyed() { count_destroyed().add(1); };
  public relay void wait_quiescent() {
    int c0, d0, c1, d1, done;
    c1 = 1; d1 = 0; done=0;
    while (done==0) {
      sleep 1000;
      c0=c1; d0=d1;
      c1=count_created().collect();
      d1=count_destroyed().collect();
      if (c0==d0) { if (c0==c1) { if (c0==d1) { done=1; }; }; };
    };
  };
};
```

```
class writeonce {
  agent node(object i) on i {
    object x; object node;
    public void init(object i) { node=i; x=NIL; };
    public void set(object v) {
      x=v;
    };
    public object get() {
      while (x == NIL) { wait; };
      return x;
    };
  };
  public void init() {};
  public relay void set(object v) {
    node(0..(node_count-1))<-set(v);
  };
  public relay object get() {
    return node(node_current).get();
  };
};
```

```
class accumulator {
  object total;
  agent sum() on 0 is psummation();
  agent node(int i) on i {
    int total, nodeno;
    public void init(int i) { nodeno=i; total=0; };
    public void add(int n) { total=total+n; };
    public void collect() { sum()<-input(nodeno, total); };
  };
  public void init() { };
  public relay void add(int n) {
    node(node_current).add(n);
  };
  public int collect() {
    total = nil;
    node(0..(node_count-1))<-collect();
    wait (total != nil);
    detach;
    return total;
  };
  public void total(int n) {
    total = n;
  };
};
```

```
class main {
  agent q() on 0 is quiescence();
  agent a() on 0 is accumulator();
  agent n() on 0 is writeonce();
  agent grain() on 0 is writeonce();
  agent statistics()  on 0 is statistics();

  agent solve(partial p) on lisp(nqueensmap p) {
    public void init(partial p) {
      int n,g,i; partial next;
      statistics().add(1);
      n = n().get();
      g = grain().get();
      if (lisp(length p)==g) {
        a().add(lisp(nqueenscomplete p n));
      } else {
        for (i=0; i<n; i=i+1) {
          next = lisp(nqueensmove p i);
          if (next != nil) {
            q().node_created();
            awaken solve(next);
      };};};
      q().node_destroyed(); detach;
  };};

  public int nqueens(int n, int g) {
    n().set(n);
    grain().set(g);
    q().node_created();
    awaken solve("");
    q().wait_quiescent();
    return a().collect();
  };

  public void init() {
    double begin, end; int count;
    begin = lisp(wallclock);
    count = this.nqueens(13,5);
    end = lisp(wallclock);
    statistics().print();
    lisp(mprint "solution=" count " at time " (end-begin));
  };
};
```

A.3 Smoothing

```
class main {
  agent smooth() is smooth();
  int total; double start;
  public void init() {
    int r,c;
    total=0;
    for (r=0; r<16; r=r+1) {
      for (c=0; c<16; c=c+1) {
        smooth().inpatch(r,c,lisp(randpatch));
      };
    };
    start = lisp(wallclock);
    wait (total==16*16);
    lisp(mprint "done at time " (lisp(wallclock) - start));
  };
  public void outpatch(int r, int c, patch p) {
    total=total+1;
  };
};
```

```
class smooth {
  public void init() { };
  agent calcpatch(int r,int c,int t) on (r/8) {
    lsp n,s,e,w,c; int total;
    public void init(int r,int c,int t) {  total=0; };
    public void prev_n(patch p) { n=p; total=total+1; };
    public void prev_s(patch p) { s=p; total=total+1; };
    public void prev_e(patch p) { e=p; total=total+1; };
    public void prev_w(patch p) { w=p; total=total+1; };
    public void prev_c(patch p) {
       c=p; wait (total==4);
       lisp(smoothpatch n s e w c);
       owner<-result(c);
       detach;
  };};
  public relay void dispatch(int r, int c, int prev, patch p) {
    int t;
    if (prev==20) {
      owner<-outpatch(r,c,p);
    } else {
      t=prev+1;
      calcpatch(r,c,t)<-prev_c(p);

      if (r> 0)  { calcpatch(r-1,c,t)<-prev_s(lisp(edgeN p));   }
      else       { calcpatch(r,  c,t)<-prev_n(lisp(blankedge)); };

      if (r<15)  { calcpatch(r+1,c,t)<-prev_n(lisp(edgeS p));   }
      else       { calcpatch(r,  c,t)<-prev_s(lisp(blankedge)); };

      if (c> 0)  { calcpatch(r,c-1,t)<-prev_e(lisp(edgeW p));   }
      else       { calcpatch(r,c  ,t)<-prev_w(lisp(blankedge)); };

      if (c<15)  { calcpatch(r,c+1,t)<-prev_w(lisp(edgeE p));   }
      else       { calcpatch(r,c  ,t)<-prev_e(lisp(blankedge)); };
  };};
  public relay void result(patch p) from calcpatch(int r, int c, int t) {
    this.dispatch(r, c, t, p);
  };
  public relay void inpatch(int r, int c, patch p) {
    this.dispatch(r, c, 0, p);
};};
```

REFERENCES

[1] C. Hewitt, P. Bishop, and R. Steiger, "A universal ACTOR formalism for artificial intelligence," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1973.

[2] G. Agha, C. Houck, and R. Panwar, "Distributed execution of actor programs," in *Workshop on Languages and Compilers for Parallel Computing*, 1991.

[3] G. Agha, W. Kim, and R. Panwar, "Actor languages for specification of parallel computations," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Specification of Parallel Algorithms*, 1994.

[4] C. Houck and G. Agha, "HAL: A High Level Actor Language and its Distributed Implementation," in *Proceedings of the International Conference on Parallel Processing*, August 1992.

[5] A. Chien and W. J. Dally, "Concurrent aggregates," in *Second ACM Symposium on Principles and Practice of Parallel Programming*, Mar 1990.

[6] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, 1993.

[7] L. Kalé, "The Chare Kernel parallel programming language and system," in *International Conference on Parallel Processing*, Aug 1990.

[8] W. Fenton, B. Ramkumar, V. Saletore, A. Sinha, and L. Kalé, "Supporting Machine-Independent Parallel Programming on Diverse Architectures," in *International Conference on Parallel Processing*, 1991.

[9] L. V. Kalé, 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.

[10] L. V. Kalé, 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.

[11] L. V. Kalé and S. Krishnan, "Charm++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA*, 1993.

[12] L. V. Kalé and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds., MIT Press, 1996, pp. 175–213.

[13] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification.* Addison-Wesley, 1996.

[14] K. M. Chandy and C. Kesselman, "CC++: A Declarative Concurrent Object-oriented Programming Notation," in *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds., MIT Press, 1993, pp. 281–313.

[15] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. W. Tseng, and M. Y. Wu, "Fortran D Language Specification," Tech. Rep. CRPC-TR900749, Centre for Research on Parallel Computation, Rice University, Dec 1990.

[16] High Performance Fortran Forum, *High Performance Fortran Language Specification (Draft)*, 1.0 ed., Jan 1993.

[17] D. J. Scales and M. S. Lam, "The design and evaluation of a shared object system for distributed memory machines," in *First Symposium on Operating Systems Design and Implementation*, Nov 1994.

[18] V. Karamcheti and A. Chien, "Concert — efficient runtime support for concurrent object-oriented programming languages on stock hardware," in *Supercomputing '93*, 1993.

[19] W. Kim and G. Agha, "Efficient compilation of call/return communication for actor-based programming languages," in *High Performance Parallel Computing*, 1996.

[20] R. J. Fowler and L. I. Kontothanassis, "Improving processor and cache locality in fine-grain parallel computations using object-affinity scheduling and continuation passing," Tech. Rep. 411, University of Rochester Computer Science Department, Jun 1992.

[21] W. C. Hsieh, P. Wang, and W. E. Weihl, "Computation migration: Enhancing locality for distributed-memory parallel systems," in *Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993.

[22] J. Plevyak and A. Chien, "Precise concrete type inference for object-oriented languages," in *OOPSLA*, 1994.

[23] J. Plevyak and A. Chien, "Type directed cloning for object-oriented programming," in *Workshop for Languages and Compilers for Parallel Computing*, Aug 1995.

[24] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An Interoperable Framework for Parallel Programming," in *10th International Parallel Processing Symposium*, April 1996.

[25] G. L. Steele, *Common Lisp the Language, 2nd edition.* Digital Press, 1990.

[26] B. Jenkins, "Algorithm alley: Hash functions," *Dr. Dobb's Journal*, Sep 1997.

[27] C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick, "Run-time support for portable distributed data structures," in *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1995.

[28] D. Gelernter, N. Carriero, S. Chandran, and S. Chang, "Parallel programming in Linda," in *International Conference on Parallel Processing*, Aug 1985.

[29] N. Carriero and D. Gelernter, "Tuple Analysis and Partial Evaluation Strategies in the Linda Compiler," in *Languages and Compilers for Parallel Computing*, 1990.

[30] G. V. W. (Ed.), "Linda-like systems and their implementation," Tech. Rep. 91-13, Edinburgh Parallel Computing Centre, 1991.

[31] T. Christopher, "Experience with message driven computing and the language llmdc/c," in *Proceedings of the ISMM International Conference on Parallel and Distributed Computing and Systems*, (New York), Oct 1990.

[32] T. Christopher, "Early experience with object-oriented message driven computing," in *The Third Symposium on the Frontiers of Massively Parallel Computation*, Oct 1990.

[33] G. Agha and C. Callsen, "Actorspaces: An open distributed programming paradigm," Tech. Rep. UIUCDCS-R-92-1766, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[34] G. Agha and C. Callsen, "Open heterogeneous computing in actorspace," *Journal of Parallel and Distributed Computing*, 1994.

[35] E. Arjomandi, W. O'Farrell, and I. Kalas, "Concurrency Suppport for C++: An Overview," *C++ Report*, January 1994.

[36] R. S. Nikhil, "Parallel Symbolic Computing in Cid," in *Parallel Symbolic Languages and Systems*, 1995.

[37] D. Gannon and J. K. Lee, "Object oriented parallelism: pC++ ideas and experiments," in *Proceedings of 1991 Japan Society for Parallel Processing*, 1993.

[38] M. Haines, D. Cronk, and P. Mehrotra, "On the design of Chant: A talking threads package," in *Supercomputing 1994*, (Washington D.C.), Nov 1994.

[39] J. Yelon and L. V. Kalé, "Agents: An undistorted representation of problem structure," in *Workshop for Languages and Compilers for Parallel Computing*, August 1995.

VITA

Joshua Michael Yelon was born in Pittsburgh, Pennsylvania in 1967. In 1990 he obtained his B.A. in Computer Science from Rice University in Houston, Texas. From 1989 to 1998 he served alternately as teaching assistant and research assistant at the Department of Computer Science, University of Illinois, Urbana-Champaign. After completing his Ph.D. he will be a business partner at eGenesis in Pittsburgh, Pennsylvania.