Agents: an Undistorted Representation of Problem Structure

J. Yelon and L. V. Kalé

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

Abstract. It has been observed that data-parallel languages are only suited to problems with "regular" structures. This observation prompts a question: to what extent are other parallel programming languages specialized to specific problem structures, and are there any truly general-purpose parallel programming languages, suited to all problem structures? In this paper, we define our concept of "problem structure". Given this definition, we describe what it means for a language construct to "directly reflect" a problem structure, and we argue the importance of using a language construct which reflects the problem structure. We describe the difficulties that arise when the language construct and the problem structure do not fit each other. We consider existing language constructs to identify the structures they fit, and we note that language constructs are often designed with little regard for such generality. Finally, we describe a parallel language construct which is designed specifically with the goal of being able to reflect arbitrary problem structures.

1 The Importance of Problem Structure

It has often been noted that problems have "structures". Problem structure is made concrete in the dataflow graph, which is defined as follows. When executing, a program performs many primitive operations. Each operation uses one or more values as input, and produces one or more values as output. In the dataflow graph, each primitive operation is represented as a vertex. Wherever a value is transmitted from one operation to another, an arc exists in the dataflow graph. Therefore, the dataflow graph is a representation of an execution showing only the operations performed and how the values computed flowed from operator to operator.

We observe that the more closely the shape of the language construct fits the shape of the dataflow graph, the more comprehensible the program turns out to be, and the more feasible it becomes to perform macroscopic optimizations (section 1). We then consider many common notations for problem representation, and discuss the limitations of those approaches from the point of view of how well they reflect the shape of the dataflow graph (section 2). We discuss a set of constructs we have developed to make it possible to write programs whose data structures directly mirror the shape of the dataflow graph, for arbitrary graph structures (section 3).

1.1 Optimization: Extracting Knowledge about Problem Structure

Optimizers rely on data-dependency knowledge, which is essentially knowledge of the *relationship between the data structures in the program and the nodes in the dataflow graph.* Optimizations that involve multiple procedures, multiple objects, or multiple threads will often require knowledge of how the dataflow graph spans the procedure, object, or thread boundaries. For example: a parallelizer proving that two subroutines are independent is proving that there are no dataflow arcs connecting the dataflow subgraph of the first subroutine with the dataflow subgraph of the second. A load balancer which wishes to preserve locality must move objects around such that distant objects are working on (relatively) unconnected regions of the dataflow graph. A scheduler that wishes to prioritize tasks on the critical path must be able to predict the existence of critical dataflow arcs between tasks. In general, there are a tremendous number of intelligent large-scale manipulations that could be performed on parallel programs, if only the data dependency relationships between the elements in the program could be predicted.

The simpler the relationship between the dataflow graph and the elements of the program, the more likely an optimizer is going to be able to predict those relationships. For example, consider a divide-and-conquer implementation of Factorial. In this formulation, Factorial(N) is defined as the result of multiplying all the numbers in the range 1 to N. A parallel version can be achieved by splitting the range in two, generating the products of each subrange, and multiplying them together. The dataflow graph for this algorithm turns out to be an initially branching, then collapsing, tree (Fig. 1), wherein the top half computes the subranges over which to multiply, and the bottom half performs the actual multiplication.

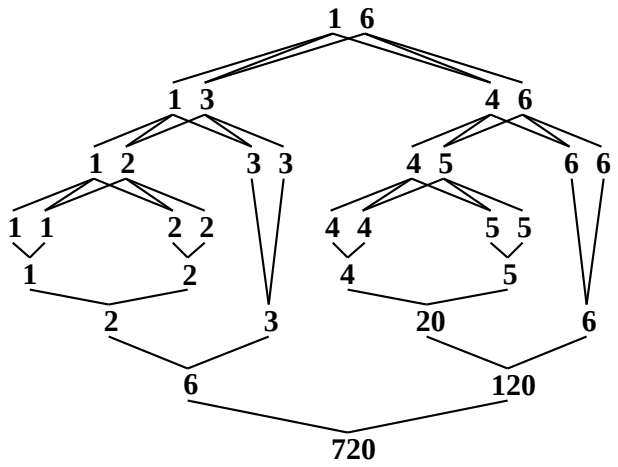

Fig. 1. dataflow graph for factorial(6) using recursive formulation

The simplest implementations of this algorithm will utilize constructs that also have a branching-then-collapsing tree structure. Perhaps the best example of such a construct is function invocation in a parallel functional language. Each function invocation can in turn spawn more function invocations. Eventually, a large tree of invocations form. As the invocations return, the tree collapses. Since the construct fits the shape of the dataflow graph so well, the implementation is trivial.

The close fit between the graph shape and the program structures yields a program that is easy to manipulate. Indeed, compilers for functional languages usually do such manipulation, effectively parallelizing, load-balancing, and scheduling tree-structured problems, all without user intervention.

Attempting to implement an algorithm with a construct that does not "fit" can be a disaster. Consider the familiar example, the data-parallel languages, as they apply to this factorial algorithm. It might be possible to express this recursive algorithm concurrently using arrays, do-loops, and barrier synchronization. However, doing so would require transformations which, as other researchers have noted, tend to leave the actual problem structure completely obscured [5]. The program would run in parallel (perhaps with poor speedups), but only because humans gritted their teeth and did the necessary convoluted analyses on behalf of the compiler.

However, the data-parallel languages are by no means the only ones that tend to distort problem structures. Many languages heavily utilize constructs which are vector-shaped.[1] For example, to process a 1000x1000 matrix on a 32-node machine using PVM, the matrix must be decomposed into a vector — in this case, a 32-element vector of irregularly-sized rectangles. Representing a 1000x1000 square as a vector of irregular rectangles almost totally obscures its otherwise simple structure. Variable-sized vectors help in rectilinear problems like this one, but mapping a non-rectilinear problem structure onto a vector of any size obscures it. Although this isn't an issue for PVM, which doesn't attempt any analysis, it clearly *is* an issue for the more advanced languages utilizing vector-shaped constructs. Such languages lose a lot of information if the programmer uses a vector-like data structure to represent a non-vector-shaped problem.

The key observation is this: programs whose control and data structures directly match the shape of the dataflow graph are significantly simpler to analyze than programs which rely upon complex mappings from graph nodes to program elements. The inability of current languages to reflect dataflow cleanly will make it impossible for compilers to perform those large-scale manipulations that demand much macroscopic knowledge. This will become an increasingly onerous burden as we seek higher levels of performance and ease of use through smarter compilers. To solve these problems, we advocate the use of language features that enable direct representation of the structure of the dataflow graph.

[1] The "aggregates" in Concurrent Aggregates [4] are vectors, the "branch offices" in Charm [10] are vectors, the "replicated nodes" in TDFL [15] are vectors, etc.

1.2 Programming: Trying to Express Problem Structure

Optimizers benefit from clear expression of the problem structure. But perhaps more importantly, the programmer benefits as well. If the problem and the language constructs do not match, then the programmer must be concerned with two different structures: the shape of the problem, and the shape of the data structures used to represent it. While programming, he must continually perform mental translations between the two representations.

For example, in the case of the 1000x1000 matrix computation, the programmer thinks of the computation as being performed on a square matrix. He mentally conceives of operations such as "multiply this row times this column". However, to implement these operations, he must translate between his simple mental representation of the problem and the physical representation: instead of fetching a row or column of the matrix, he must fetch the Jth row owned by processor K, where J and K depend on some formula expressing how the rows of the matrix were distributed among the processors.

It is a great boon to be able to express problem structure directly. The programmer benefits from such a representation in exactly the same manner that the optimizer benefits: simply put, the absence of a convoluted transformation between the problem and its representation makes the program easier to understand.

1.3 Constructs that Simplify the Dataflow Graph

If the relationship between the program constructs and the dataflow graph must be simple in order for the optimizer to unravel it, then one might be tempted to search for programming constructs that yield the simplest dataflow graphs. Such a search proves futile — the dataflow graph is an inherent property of the algorithm chosen, it cannot be restructured by changing one's choice of data structures or control structures.

For example, suppose that a programmer wishes to compute Fibonacci 5. The dataflow graph will always contain the same six nodes shown in Fig. 2. The graph may contain other nodes as well, which may vary from implementation to implementation, but those six nodes will always be present.

Actually, this is only partially true: there are other algorithms for Fibonacci, and those algorithms have entirely different dataflow graphs. However, for many problems, only one or two algorithms are known. One is essentially stuck with a small choice of problem structures. Since one cannot affect these problem structures by choosing a different language or by making different implementation decisions, it instead makes sense to choose constructs based on the principle of *clearly expressing* the *inherent* shape of the dataflow graph.

2 Evaluations of Existing Languages

In this section, we consider some existing programming languages, and evaluate their programming constructs according to how well they reflect the shape of the dataflow graphs they express.

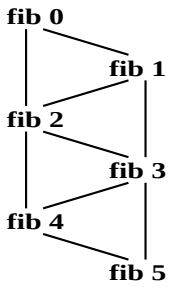

Fig. 2. dataflow graph for Fibonacci(5) using linear-time method.

2.1 Dataflow Languages

It seems intuitive that the dataflow languages would contain constructs that naturally reflect the shape of the dataflow graph. By "dataflow languages" we refer not to those languages with ordinary functional language semantics but unusual runtime systems. (Clearly, such languages are no more or less analyzeable than their non-dataflow counterparts). Instead, we refer to languages that explicitly reveal their dataflow underpinnings at the language level. Probably the best examples of such languages are the visual dataflow languages like TDFL [15], PFG [14], and Poker [13].

These languages are loosely based on the Petri net programming model. If one were to base a language on a restricted class of Petri nets, with the limitations that the net be finite, contain no cycles, process only scalars, and that arcs be used only once, then the programs would be perfect representations of their dataflow graphs. On the other hand, the language would only support finite problem structures. Therefore, languages like the ones named above necessarily extend the model: for example, TDFL adds "replicated nodes" to support vector-shaped problem structures, and recursive nets to support branching-and-collapsing problem structures.

The restricted Petri net model by itself is capable of representing (and ideally reflecting) a small class of dataflow graphs. Each extension adds the ability to express (and again, very directly reflect) another class of problem structures. The resulting language is capable of very naturally and directly expressing those problem structures for which it has constructs. Other problem structures become problematic. As a result, the temptation is present to continue adding new features for new problem structures. We now discuss the effects of such a philosophy.

2.2 Agglomerate Languages

Many languages are being created containing a collection of constructs, each construct appropriate to a different problem structure. We term these languages the "agglomerate" languages. Such languages seem to be quite popular right

now: consider Charm [10], HPC++ [2] [3], and HPF+Fortran-M [6]. By including different constructs for different problem structures, these languages aim to support a broader class of problems than their predecessors.

Some would say that designers of agglomerate languages are haphazardly adding constructs, each of which is only useful for one problem structure. Others, however, might argue that the objective in adding constructs is to create a "basis set", eventually covering the entire space of problem structures. The reasoning is sound. If an agglomerate is built with sufficient constructs, it may indeed achieve a basis set.

Unfortunately, achieving a basis set requires more than the simple ability to encode arbitrary problems: it requires the ability to encode problems *without* mapping their dataflow graphs onto dissimilarly-shaped data structures or control structures. Thus, a basis set is unlikely to be achieved very quickly through haphazard addition of constructs. To our knowledge, no existing agglomerates contain a basis set; for example, no languages of which we are aware can represent both a reduction tree (which grows from the leaves up, not the top down) and a 2D matrix without mapping either onto a vector of data elements. Rather than randomly adding constructs and hoping they eventually form a basis set, it makes sense to be more critical, accepting only a set of constructs that offer us some concrete reassurance that they cover the entire space of possible problem shapes.

Such concrete reassurance is offered by the linked data structures, which can obviously be connected into any shape imaginable. Therefore, linked structures promise to singlehandedly cover the entire space of possible problem shapes.

2.3 Object-Based Languages and Linked Structures

Parallel Object-based languages (Actors [1], ABCL/1 [16], Mentat [8], etc.) rely on linked data structures to represent problem structure. Linked data structures, because of their incredibly flexible shape, are the sequential programmer's tool of choice for clear representation of complicated shapes such as trees and graphs. Unfortunately, linked data structures have two flaws which, from our point of view, make them undesirable constructs for parallelism.

The first flaw of linked structures is that their shapes are hard to predict. For example, in an object-parallel implementation of Jacobi relaxation, the compiler can easily see that the code declares objects of type `jacobi_node`, each containing four fields called `left_neighbor`, `right_neighbor`, and so on; yet the compiler remains oblivious to the fact that those objects form a two-dimensional rectangular grid. The challenge of predicting the shape that linked structures will form tends to be as great an impediment to optimizers (if not greater) than a complex relationship between problem structure and program stucture[2].

[2] Some researchers have attempted to predict the shapes of linked structures which will form at runtime. Such work has seen some success, although only certain kinds of information are made available through these techniques. [12]

The second flaw in linked structures is that they are quite difficult to construct concurrently. Consider the problem of adding a single object to an existing linked structure in such a way that the new object is accessible from (pointed to by) several old objects. One of the older objects must create the new object, obtaining a pointer to it. The creator must then deliver the pointer to all locations where it is needed — and the delivery process is the problem. Potential difficulties in delivering the pointer include identifying the other objects that need the pointer, routing the pointer through the older parts of the graph, and inventing some addressing scheme to get the pointer to the correct destination. The effort involved ranges from significant to extraordinary, even for problems as simple as building a 2D grid. The problem of routing a pointer to a destination in a graph is sufficiently difficult that linked structures are convenient only when such routing is not necessary. Routing can usually be bypassed by storing the pointers in a concurrent array or hash-table... but this, of course, maps the problem structure onto a vector again, obscuring it from the compiler. The only case where such routing is not necessary is when an object is only connected to its parent and its children — in other words, in tree-structured problems. Indeed, the fact that pure objects seem to be best for tree-processing has been noted by other researchers [9].

Both limitations of linked structures are rooted in a single property: dynamic creation. Both can therefore be eliminated by doing away with dynamic creation. Allowing the programmer to statically declare the entire problem structure would relieve him of the burden of creating it, while simultaneously making the structure far more visible to the optimizer. Therefore, our constructs are based on this fundamental principle: *whenever possible, represent problem structure declaratively.*

3 Agents: Undistorted Representation of Structure

We have devised constructs that enable the programmer to declaratively express arbitrary problem structures. Our constructs define a graph of "agents". Individually, these "agents" are very similar to objects: they are small entities that have state, and have code associated with them. Unlike objects, though, they conceptually "exist" in a prespecified pattern from the instant the program begins. In this static layout, they are much like Petri nets. However, unlike the specific patterns allowed by Petri net languages, agent networks can be infinitely large, with arbitrary numbers of connections between agents, in unrestricted patterns.

Agents are declared in groups:

```
AGENT identifier[ indices... ] RUNS function(arguments...);
```

In our model, each agent is a tiny process running a function. On a more theoretical level, agents could have been expressed using any representation of behavior and state, for example, agents could have been C++ objects. We emphasize that the means of expressing agents and their internal behaviors is orthogonal to our ideas on problem structure.

The agent identifier names a set of agents. Indices are similar to array indices, they can be used to select an individual element in the set. However, unlike array indices, they are not bounded, so the set of agents may be of unbounded size, and it may be sparse. To understand the need for unbounded, sparse sets of agents, one need only recall their purpose: each active agent intends to represent a single node in the dataflow graph[3]. The entire set of agents is intended to represent the space of possible nodes that could be used during the execution. The indices to the set of agents need not be integers, they may be any other type that can reasonably be used as an index into a table.

To accommodate unbounded sets of agents, we impose this constraint: each agent is completely passive until it receives its first message. Passive agents send nothing, compute nothing, and have uninitialized state. No memory is allocated to an agent which has not yet received its first message. Likewise, an agent which has finished executing its code it is returned to its passive state.

Our model does not assume shared memory: it is assumed that agents cannot access each other's variables without sending explicit messages to each other.

Every agent can send and receive messages. In our notation, messages are tuples of values with a symbolic tag at the front[4]. We use the notation `tag(value1, value2, ...)` to denote such a tuple. The tag is a single identifier. The `SEND` statement is used to transmit tuples:

```
SEND tag(value1, value2, ...) TO agent[index1, index2, ...];
```

Two special agent identifiers are recognized, `SELF` and `PARENT`. Sending a tuple to one's parent is defined as follows: if an agent A1 is running a function F1, and function F1 contains a declaration of an agent A2, then agent A1 is the parent of agent A2. An index can be a range of integers `low...high`, indicating a multicast. Finally, note that agent identifiers are first-class objects, so the `TO` clause can contain an expression.

The `HANDLE` declaration and `WAIT` statement are used to receive tuples:

```
HANDLE <tuple> FROM <source> { code }
WAIT <boolean-expression>;
```

Once started, agents execute uninterrupted until they reach a `WAIT` statement. When they reach the `WAIT` statement, they block, and their handlers become active. The agent begins receiving tuples, executing the appropriate handler code when each message is received. After a handler fires, the `WAIT` condition is reevaluated, and the agent may unblock and continue execution.

In the `HANDLE` declaration, the `tuple` and `source` fields are both patterns. The `tuple` field is matched against the contents of the tuple. The `source` field is

[3] One will often want to decrease the "resolution" or grainsize of the data flow graph by merging small sets of adjacent nodes. This does not affect our constructs.

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

matched against the name of the originating agent. Both patterns may contain variables, which are bound to the contents of the tuple or the indices of the originating agent. The `FROM` field can specify `SELF`, `PARENT`, or it can be omitted to accept tuples from anywhere. Note the following subtlety: if a tuple matches more than one handler, both fire, and if it matches no handlers, nothing fires, although the tuple still "awakens" the agent that receives it.

Note that every function invocation created by a function call is an unnamed agent, so functions can declare handlers and send and receive messages regardless of whether or not they are explicitly declared as agents.

We begin with a simple example, a Fibonacci program. This problem has the structure shown in Fig. 2. It begins with a function that acts like a binary adder:

```
void binadder()
{
  int total=0, count=0;
  handle value(int v) from parent { total+=v; count++; }
  wait (count==2);
  send value(total) to parent;
}
```

Any agent running `binadder` will receive two `value` tuples from its parent. After adding them, it sends the total back to its parent. Note that it is normal for an agent to receive its inputs from its parent, much like a function receives its arguments from its caller. The parent of the `binadder`s will link them together:

```
1:int fib(int n)
2:  {
3:    int result; int done=0;
4:    agent calcfib[int i] runs binadder();
5:    handle value(int i) from calcfib[j] {
6:      send fib_eq(j, i) to self;
7:    }
8:    handle fib_eq(int i, int j) from self {
9:      if (i+1<=n) send value(j) to calcfib[i+1];
10:     if (i+2<=n) send value(j) to calcfib[i+2];
11:   }
12:   handle fib_eq(int i, int j) from self {
13:     if (i==n) { result=j; done=1; }
14:   }
15:   send fib_eq(0, 0) to self;
16:   send fib_eq(1, 1) to self;
17:   wait (done==1);
18:   return result;
19: }
```

The `fib` function declares a number of `calcfib` agents. The `calcfib` agents form a chain in which `calcfib[i]`'s job is to add fib(i-2)+fib(i-1) yielding fib(i).

The process is initiated by the `sends` on line 15-16, which trigger the handler on lines 8-11. This causes `value` tuples containing fib(0) and fib(1) to be sent to `calcfib[1]` through `calcfib[3]`. These in turn produce `value` tuples, which are received by the handler on lines 5-7, are forwarded as `fib_eq` tuples to the handler on lines 8-11, which again feeds them back to the `calcfib` agents as `value` tuples. The chain continues until the conditions `(i+1<=n)` and `(i+2<=n)` on lines 9-10 cause it to terminate. Meanwhile, the handler on lines 12-14 is looking for the final `fib_eq` tuple. When it catches this tuple, it stores the result, and sets the `done` flag. This causes the `wait` statement on line 17 to terminate, and the `fib` function returns.

I now show a more interesting example: an SLD refutation engine. SLD theorem provers start with a single assertion, and by combining that assertion with a database, generate more assertions. These new assertions are in turn combined with the database, and so on recursively, until finally an assertion is derived that is known to be false. This refutes the original assertion. The problem would be tree-structured, if not for the fact that assertions frequently get re-derived, but they must not be re-processed. The prover consists of a `refute` function, which uses many `tryrefute` children. Each `tryrefute` agent has the task of trying to refute one assertion. A `tryrefute` agent either sends `refuted` immediately, or it derives a set of assertions, transmitting `begin` tuples to initiate their recursive expansion. It then sleeps forever, thereby refusing to try to refute something twice.

```
 1:  typedef string assertion, database;
 2:  void refute(assertion goal, database dbin)
 3:  {
 4:    int refuted=0;
 5:    agent dbholder runs store_string();
 6:    agent tryrefute[assertion A] runs {
 7:      string DB = fetch(dbholder);
 8:      if (obviously_false(A)) send refuted(A) to parent;
 9:      else
10:        for all assertions D derived from A and DB do
11:          send begin() to tryrefute(D);
12:      wait 0;
13:    }
14:    handle refuted(assertion a) { refuted=1; }
15:
16:    store(dbin, dbholder);
17:    send begin() to tryrefute(goal);
18:    wait (refuted==1);
19: }
```

A few minor points: 1. Note that agents can send to agents defined in surrounding scopes, this presents no particular new issues. 2. `store_string` in line 5 is a library function defining a replicated storage agent. The string is stored in

the agent using `store`, and retrieved using `fetch`. `store_string` distributes a copy of the string to all processors to eliminate copying at read-time. We therefore use it as an efficient means to distribute the database. 3. The `begin` tuples, since they are not handled, cause no effect other than to wake up the agents to which they are sent.

Note that the `fib` agent is serving as a dispatcher for large numbers of messages. As pointed out by Chien [4], any individual object acting as a dispatcher or interface for a large number of other objects can be a bottleneck. We avoid this bottleneck: when a handler does not access any local variables, it can be executed on any processor, and is in fact executed by the processor which originated the tuple. This optimization technique guarantees that tuples always go straight from their origin processor to the first agent they actually affect — in the `fib` case, straight from `calcfib` to `calcfib`. Handlers that do nothing but forward tuples are particularly relevant to understanding problem structure, so we give them a special name: "relay handlers".

3.1 Demonstrations: Analyzing an Agents-Based Program

Our new programming constructs were designed to make it easier for both the user and the optimizer to understand the problem structure. We now show two optimizations made possible by these constructs. Keep in mind this caveat: these demonstrations are not very complex. There are merely intended to show the relative ease with which optimizations can be performed in a model where problem structure is declared explicitly and statically.

Static Load Placement By Tiling In this demonstration, the optimizer notices that agents in the program form a 2D matrix. It notices that the communication is nearest-neighbor, and decides that the best alignment scheme for the agents in the matrix is to tile it into subsquares. The optimizer then implements the tiling.

To determine whether tiling a set of agents is desirable, the optimizer first checks whether the set actually forms a 2D matrix. This is done by examining the agent declaration: if they it has two integer indices, then it is a (possibly infinite) 2D matrix. If not, tiling is aborted.

Second, the optimizer checks whether communication is local; if not, tiling is undesirable and is aborted. Locality can thus be verified by checking the `SEND` statements in the matrix agents, plus the `SEND` statements in any relay handlers affecting the matrix agents. If the `TO` clauses all designate agents whose indices differ by only a small constant from the sending agent, then communication is local.

Third, the optimizer must determine the ranges of the indices. Identifying the used subrange is often possible through mathematical induction using the `TO` clauses of the send statements, and the `FROM` clauses of the relay handlers. Agents can (usually) only be awakened (receive their first message) by their parents. So we check the send-statements in the parent to identify the range of agents used.

As an example, consider the `calcfib` agents. There are two `send` statements that are not in relay handlers, these send to constant locations: `calcfib[1..3]` are accessed. There are also the sends in the relay handlers on lines 5-11. Together they provide two induction hypotheses, one of which is: if `calcfib[i]` is used, and `(i+1<=n)`, then `calcfib[i+1]` is used. Therefore, the range of agents that is used is `calcfib[1]` through `calcfib[n]`. Similar inductive proofs can be made for 2D agent matrices, to verify that they are finite, and to determine the ranges of the indices. If the indices are not finite, tiling is aborted.

Finally, tiling is implemented by choosing a size for each tile. The system then generates a function that maps agent names to processors. This function is used by the runtime system to allocate the agents to processors.

Note that it is the compiler's knowledge of which agents may exist, what shape they form (a 2D matrix), and what their communication patterns are (nearest-neighbor) that makes it possible to statically and intelligently distribute the load in a reasonable fashion. Note that even smarter methods, such as following this initial placement by dynamic balancing, are quite feasible.

Vectorization Across Objects In this demonstration, the optimizer notices that a multicast to a range of objects triggers each object to perform a few simple mathematical operations. The optimizer converts the multicast code such that it can take advantage of the CPU's vector units.

To deliver a multicast to a range of objects existing on the same processor, the system uses a loop over the range of agents. Agents' state variables are stored in C `structs`. If the structs for the range of agents are allocated in a contiguous block of memory, then the delivery loop is a loop over a vector of agent `structs`. Each iteration of the multicast-delivery loop executes a handler on behalf of one agent. The handler generally modifies the agent struct of one agent. So the delivery loop is actually a loop modifying an array of structs, each struct in the same manner. If the handler code is sufficiently simple, then the delivery-loop will probably be vectorizable using traditional methods.[5]

In addition to vectorizing the handler itself, it is sometimes possible to vectorize the code after the wait statement. If it can be proved that all agents receiving the multicast are blocked at the same wait statement (for example, if all the agents are running a function that only contains one wait statement), and if it can be proved that all will unblock upon receiving the message, then the delivery loop can also include the code after the wait statement.

Again, it is the system's knowledge that certain ranges of objects form vectors that makes it possible to perform this optimization. In a linked language, the system would not know to allocate the objects in vectors, nor would it be able to express the multicast.

[5] The actual success of such vectorization depends upon the what code the handler contains.

4 Comparisons to Other Work

Superficially, Agents bears a close similarity to the languages based on linked objects, such as Actors, ABCL/1, Mentat, and many others. There is apparent similarity since individual agents are more or less equivalent to objects. However, in object-based languages, the structures formed by the objects must be created dynamically. In a sequential program this is no problem, but creating linked structures with any degree of concurrency is quite clumsy, except in special cases such as when creating tree-structures (see sec. 2.3). Therefore, dynamically created objects are best suited to certain specific problem structures, notably, those which are tree shaped. Also importantly, the structures formed by dynamically-created objects is hard to predict, thereby concealing the problem structure from the optimizer. Agents and the relationships between them are declared statically, alleviating both problems.

Agents is far more closely related to the parallel Petri net languages like TDFL and Poker, which are also based upon statically-declared dataflow patterns. Most Petri net languages necessarily extend the basic Petri net model. (The unextended model only supports a finite, unchanging set of nodes, which is impractical for expressing massive concurrency.) Languages like TDFL extend the Petri net model by adding several specialized constructs: for example, replicated nodes are added to support vector-shaped problem structures, and recursive nets are added to support branching-and-collapsing structures. Each of these extensions is naturally suited for one class of problem structures. Agents also extends the Petri net model, however, it extends it in a manner specifically designed to handle arbitrary problem structures: it allows arbitrarily-sized networks, having arbitrarily large numbers of arcs, with arbitrary connectivity.

One parallel programming system, Concert [4], attempts to predict inter-object data relationships *despite* its programming model, in which every structure must either be built dynamically of linked objects or mapped onto a vector of objects. As a result, Concert relies on sophisticated techniques to extract structural knowledge about linked structures [12]. This is a significant accomplishment. However, we feel that the Concert programming model is not as expressive as it could be, given its requirement that all problems be mapped onto vectors or built from linked structures, and we feel that this limitation hinders both the programmer and the optimizer. Given this consideration, we feel it would be advantageous both to the Concert programmer and to the Concert optimizer to upgrade to a programming model which is more reflective of problem structure.

Some languages deal with arbitrary problem structures by providing shared memory, fast sync variables, M-Structures, or some similar abstraction. Although shared memory doesn't make it any easier for the compiler to extract knowledge of the problem structure from the program, true hardware-based shared-memory does make it less *important* for the compiler to do so. For example, with hardware-based shared memory, the compiler can more or less ignore questions of data layout, load-balancing, scheduling, and other difficult questions. (Or at least, it needs less reliable strategies for these tasks). Since problem structure

is less important to the compiler, it can be dealt with at the user-level, encapsulating the implementation details inside ADT's that neither the programmer nor the compiler need care about.

Although data-parallel languages are not generally intended for all problems, they *do* clearly show off the benefits of matching problem structure to the language construct when they are applied to the right problems. It is widely acknowledged that coding regular problems is easy in data-parallel languages (quite an accomplishment, given the widespread feeling that parallel programming is difficult). Also importantly, compiler technology for data-parallel languages is advancing by leaps and bounds, since it is possible to extract quite a bit of knowledge about the problem structure from the language constructs.

Two parallel programming systems, Linda [7] and Distributed Memo [11], can represent data of arbitrary structure without mapping or distortion. Both have a globally-accessible storage space for data items wherein items can be accessed by name. In both languages, the naming scheme is infinite. Therefore, one can store data of arbitrary shape in the space without having to force it into a predefined structure. However, though the languages have facilities for *storing* data without distorting its shape, neither languages have any facilities for expressing the computation itself as anything other than a vector of processors. It is impossible to infer the structure of the computation from the structure of the storage space, since the structures in storage space are created dynamically. Even so, both languages gain a significant degree of representational clarity from their ability to store and access data without distortion.

5 Conclusions

We started with the observation that problem structure is embodied in the shape of the dataflow graphs produced by that problem. We also pointed out that the more directly the shape of the dataflow graph is reflected in the data and control structures of the program, the easier the program is to understand, and the easier it is to analyze and optimize. We noted that for maximum predictability, the direct representation of the problem's structure must be declared statically, not created dynamically.

We defined the `AGENT` construct, which makes it possible to express arbitrary computation graphs without distortion. Individual agents are like C++ objects, except that they form a static mesh whose structure is declared, not linked together with pointers. An interconnected set of agents can directly reflect the shape of the computation graph. We demonstrated through two examples the ease with which optimizations can be performed in this framework.

We are currently engaged in the work of implementing our compiler, which is an aggressive implementation with tight control of overhead. This compiler will be complete in late 1995. After completion of the compiler, efforts will be shifted towards performance analysis and the implementation of optimizations.

References

1. Gul Agha and Carl Hewitt. *Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism*, volume 206 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, Berlin-Heidelberg-New York, October 1985.
2. F. Bodin, P. Beckman, D. B. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic ideas for an object parallel language. In *Proceedings of Supercomputing '91*, pages 273–282, 1991.
3. K. M. Chandy and C. Kesselman. *CC++: A declarative concurrent object-oriented programming notation.* MIT Press, 1993.
4. A. A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, MA, 1993.
5. A. A. Chien, M. Straka, J. Dolby, V. Karamcheti, J. Plevyak, and X. Zhang. A case study in irregular parallel programming. *DIMACS Workshop on the Specification of Parallel Agorithms*, May 1994.
6. I. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates high performance fortran and fortran M. In *Proceedings 1994 Scalable High Performance Computing Conference*, 1994.
7. David Gelernter, Nicholas Carriero, S. Chandran, , and Silva Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, pages 255–263, Aug 1985.
8. A. S. Grimshaw and J. W. Liu. Mentat: An object-oriented data-flow system. *Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference*, pages 35–47, October 1987.
9. A. Gursoy and L.V. Kale. High-level support for divide-and-conquer parallelism. In *Proceedings of Supercomputing '91*, pages 283–292.
10. L. V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, 1990.
11. W. O'Connell, G. Thiruvathukal, and T. Christopher. Distributed Memo: A heterogenously distributed parallel software development environment. In *Proceedings of the 23rd International Conference on Parallel Processing*, Aug 1994.
12. J. Plevyak, V. Karamcheti, and A. Chien. Analysis of dynamic structures for efficient parallel execution. *Languages and Compilers for Parallel Machines*, 1993.
13. Lawrence Snyder. Introduction to the Poker programming environment. *Proceedings of the 1983 International Conference on Parallel Processing*, pages 289–292, August 1983.
14. P. David Stotts. The PFG language: Visual programming for concurrent computation. *Proceedings of the 1988 International Conference on Parallel Processing*, II, Software:72–79, August 1988.
15. Paul A. Suhler, Jit Biswas, and Kim M. Korner. TDFL: A task-level data flow language. *Journal of Parallel and Distributed Computing*, 9(2), June 1990.
16. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices, Proceedings OOPSLA '86*, 21(11):258–268, Nov 1986.

This article was processed using the LaTeX macro package with LLNCS style