Delirium: An Embedding Coordination Language*

*** Best Student Paper Award - Tools (tie)**

Steven Lucco and Oliver Sharp[†]

Computer Science Division, 571 Evans Hall, UC Berkeley, Berkeley, CA 94720

Abstract

Parallel programs consist of a group of sequentially executing sub-computations which cooperate to solve a problem. To exploit existing sequential code and available optimization tools, programmers usually choose to write these sub-computations in traditional imperative languages such as C and Fortran. A *coordination language* expresses data exchange and synchronization among such sub-computations. Current coordination languages support a variety of interaction models. Delirium introduces a new, more restrictive coordination model that provides the benefit of deterministic execution without requiring programmers to re-write large amounts of code. Current coordination languages are embedded; they work through the insertion of coordination primitives within a host language. Delirium is the first example of an *embedding* coordination language. A Delirium program is a compact representation of a framework for accomplishing a task in parallel.

1 Introduction

We have developed an environment for programming MIMD multi-processors which is based on an *embedding coordination language* called Delirium. Parallel programming involves the coordination of sub-computation executions on many processors. Rather than embedding coordination primitives within a host language, our system embeds sequential sub-computations called operators within a *coordination* framework. The operators are written in traditional imperative languages like C or Fortran. A compact Delirium program can express a coordination framework sufficient to parallelize a large application.

Our environment runs on the Sequent Symmetry, Cray-2, Cray Y-MP, and BBN Butterfly T2000. We have developed an efficient run time system for executing Delirium which generally adds less than three percent overhead to the running time of an application. The environment includes an optimizing compiler, various tools for analyzing and improving execution speed, and a visualization tool for coordination frameworks.

We believe that embedding coordination languages such as Delirium offer significant advantages for the expression of parallelism. We give two case studies below which explain how we achieved good speed-up on medium sized applications. The first of these is a convolution-based simulation of the retina for motion detection and the second is the system's own optimizing compiler.

2 Coordination

An application programmer who wants to develop a parallel program can choose among a wide variety of proposed environments, based on many different notations [3] [14] and underlying memory models [4] [27]. Parallel programmers in the scientific community generally choose models that are most similar to their sequential programming environment, a decision that does not result from mere conservatism.

Two practical considerations dictate it:

Existing code When parallelizing an application, a programmer often begins with a working sequential version. The code may have already undergone laborious sequential performance tuning, including vectorization. In our experience, often more than 90% of the sequential code can be re-used in parallel implementations. If the parallel version is written in an entirely new notation, the programmer is forced to rewrite all of that code.

Existing tools Programmers will prefer parallel programming environments which can effectively incorporate existing tools for imperative language compilation, debugging, and profiling.

This is especially true of vectorizing compilers. Vector processors have proven their worth; the majority of scientific applications. from Monte-Carlo simulations [28], to pro-

[†] Steven Lucco Supported by IBM Fellowship, Oliver Sharp by Hertz Fellowship and Lawrence-Livermore National Laboratory

tein folding [23] contain sub-computations which vectorize extremely well. In executing such applications, a single vectorized processor can out-perform an entire scalar multiprocessor. We are convinced that future systems for scientific computing will combine vector and parallel processing. Machines such as the Cray Y-MP already do so on a small scale. Thus, programming environments which can incorporate existing tools for vectorization provide important additional leverage to the parallel programmer.

2.1 Coordination Languages

We believe that *coordination* among sub-computations should be the basic function performed by a parallel programming environment. Programmers can use existing imperative languages to write the sub-computations, drawing on existing libraries and sequential application code. It is awkward and unnecessary to rewrite subcomputations in a radical language notation designed to support parallelism.

A *coordination language* is a notation for expressing coordination directly. A variety of these have been proposed, including Linda [13], RPC systems [6], Sloop [20], and Ada [29].

The coordination language for our system is called Delirium. This language is different from all other coordination languages in that it is *embedding* rather than embedded. Subcomputations are performed by *operators* written in a source language like C or FORTRAN. Delirium is a single-assignment language which directly expresses coordination among operators.

There are many advantages to using an embedding coordination language. First, all the sub-computations are encapsulated. That is, they have unique, well-defined entry and exit points. Debugging is much easier because individual operators can be tested in isolation, both on the target machine or a more familiar sequential one. The decomposition also simplifies the task of load balancing. The programmer can adjust the amount of computation in an operator to minimize overhead and to improve load balance. The language runtime system can make scheduling decisions about the execution of individual operators to try to optimize load balance. Delirium shares this advantage with object-oriented systems which encapsulate sub-computation as operations on objects.

Second, all the coordination required to execute the program is expressed in Delirium. The sequential sections of the program are embedded within this coordination framework. One can express all the glue necessary to coordinate a mid-sized application on a single page of Delirium. This organizing principle makes parallelization easier. Instead of scattering coordination throughout a program, creating a set of ill-defined sub-computations, a Delirium programmer precisely defines operators and embeds these operators within a parallelization framework. One can completely discover the topology of the program's parallel execution simply by reading its Delirium code.

Similarly, one can completely ignore issues of data dependency when writing the code for an operator. The only extra coding requirement is that a Delirium operator must state explicitly whether it might destructively modify each one of its arguments. The Delirium run time system uses this information to enforce determinism. It maintains reference counts in the data blocks, copying them when two or more operators need simultaneous write access. In practice, a Delirium programmer is careful to prevent the copying of large data structures; if two operators need to modify separate parts of the same data block, the Delirium code must arrange to split the data and pass only the relevant parts to each operator. If operators refrain from using global state, the annotation of destructively modified arguments is sufficient to guarantee deterministic execution of the overall program.

Third, since operators are embedded in Delirium, rather than the reverse, only one Delirium notation is necessary. Thus, one can interchange C and FORTRAN operators or replace them all with machine instructions without modifying the Delirium code.

Finally, Delirium need not rely on discrete primitives to express coordination. Since a Delirium program is unified textually, it can express coordination *functionally*. The Delirium notation, discussed in the next section, simplifies and unifies the expression of common topologies for parallel execution. For example, in an embedded language a fork/join construct might look like:

```
a_start=init_fn();

PARALLEL_DO
  a=convolve(a_start,0);
  b=convolve(a_start,1);
  c=convolve(a_start,2);
  d=convolve(a_start,3);
END PARALLEL_DO

return(term_fn(a,b,c,d));
```

This fragment has the semantics that the calls to `convolve` are executed in parallel, after the call to `init_fn`, and that the call to `term_fn` is not made until all calls to `convolve` have finished.

In Delirium one would express this as:

```
let
   a_start=init_fn()
   a=convolve(a_start,0)
   b=convolve(a_start,1)
   c=convolve(a_start,2)
   d=convolve(a_start,3)
in term_fn(a,b,c,d)
```

This code fragment, while no more compact than the preceding fork/join example, avoids the use of embedded directives. Because Delirium is a single assignment language, one can derive a program's communication topology from the data dependencies between its operators. This advantage of single assignment languages is well known [24]; the contribution of Delirium is that it harnesses this property to create clean expression of coordination, without requiring a programmer to express any of the actual computation under the single assignment restriction.

Further, Delirium can compactly express complicated parallel control patterns, such as those found in parallel backtracking, using only a few notational devices. Such complex patterns are difficult to express and understand in an embedded language.

3 The Language

Delirium is based on restricted access to shared memory blocks. No memory block may be destructively modified by an operator unless it possesses the only reference to that block. This restriction corresponds to the idea of single-assignment in functional languages, so we chose to express coordination with a functional notation. The language contains only six constructs; it is described in detail elsewhere [25] so we will only summarize its features here.

A Delirium program consists of a set of functions, one of which is called `main`. These functions are *first-class*, meaning that they can be treated like any other value. They may be passed as arguments, bound to variables, or returned as values. The compiler converts each function into a graph, where the edges represent data paths and the nodes represent sequential operators. Where a function is passed as an argument, the run time system actually passes the corresponding graph. A special *call-closure* operator takes a graph and expands it dynamically at run time.

The language constructs are:

1. atomic values – integers, strings, floats
2. multiple values – packages that can be put together, decomposed, and used as return values
3. let bindings – a binding can be a single value, a decomposition of a multiple value package, or a function definition
4. conditionals
5. iteration – this is compiled into tail-recursive functions which are handled efficiently in the run-time system
6. function or operator application

Because of the flexibility of a functional notation, Delirium can express a broad variety of parallel constructs without requiring special syntax for each one. The `parbegin parend` [3] construct, for example, in which the programmer lists computations guaranteed to be independent, is completely subsumed. In Delirium, any two operations that do not have a lexically expressed dependency can always be executed in parallel and the run time system will automatically detect that.

While the fork-join construct is a particularly useful coordination strategy, it is too simple to demonstrate Delirium's power for compact description of coordinations frameworks. Here is a more interesting example which computes a parallel recursive backtracking solution to the eight queens problem:

```
main()
  let board = empty_board()
  in show_solutions(do_it(board,1))

do_it(board,queen)
  let h1 = try(board,queen,1)
      h2 = try(board,queen,2)
      h3 = try(board,queen,3)
      h4 = try(board,queen,4)
      h5 = try(board,queen,5)
      h6 = try(board,queen,6)
      h7 = try(board,queen,7)
      h8 = try(board,queen,8)
  in merge(h1,h2,h3,h4,h5,h6,h7,h8)

try(board,queen,location)
  let new_board = add_queen(board,queen,location)
  in if is_valid(new_board)
      then if is_equal(queen,8)
            then new_board
            else do_it(new_board,incr(queen))
      else NULL
```

The goal is to find every way that eight queens can be placed on a chessboard so that none can attack any other. We begin with an empty board and wish to place a queen on the first column. In parallel, we call `try` with every possible square in the column. After making a new board with the additional queen, `try` checks to see if the board is still valid. If not, give up and return `NULL`. If we still have a valid board, check to see if we have placed the eighth queen and therefore have a correct solution. If so, return it. Otherwise, call `do_it` again on the next queen to be placed. After all eight calls to `try` have completed, merge the results and return them.

We would like to emphasize two characteristics of this solution. The first is that the parallel decomposition strategy is very clear from the Delirium code because the coordination is expressed explicitly and in isolation. A parallel backtracking algorithm is difficult to express in a model based on tasks and messages because the primitives are not well suited to recursive execution. The underlying coordination is hidden in a tangle of message passing calls and task manipulations which are hard to understand and debug.

The other observation is that a tremendous degree of parallelism is exposed in this program — so much so that it might lead to an unwieldy explosion of schedulable operators without the priority execution scheme explained in section 7. A straight-forward implementation of the operators for this example involves roughly 100 lines of C.

4 Programming Environment

Writing a Delirium program requires the programmer

1. to devise a parallelization strategy and to express that strategy as a Delirium coordination framework and
2. to decompose the sequential computational code into operators, embeding those operators within the coordination framework

The coordination framework of the application, expressed in Delirium, generally makes up a small fraction of the total code. The operators (which can be written in a sequential language like C or Fortran) are compiled normally, providing access to existing optimizing (and vectorizing) compilers. The Delirium code is also processed by an optimizing compiler [25] which is described in more detail below.

As we have already mentioned, the computation performed by a Delirium program is not affected by the number of processors used. We generally debug programs on a single-processor workstation like the Sun, the IRIS 4D, or the HP 300. Once the code is working, we move to a parallel machine; our environment is currently running on the Sequent Symmetry, the Cray-2, and the Cray Y-MP. We have a preliminary version on the BBN Butterfly T2000 as well. In addition to the two medium sized applications discussed in the case studies, we have implemented a number of others including a 10,000 line ray tracer and a simple circuit simulator.

5 Case Study #1: A Retina Model

Frank Eeckman, currently at Lawrence-Livermore, has developed a neural net model, based on the retina, for doing motion detection [11]. David Andes at the Naval Weapons Center in China Lake wrote a Fortan code that implements the model. We took the code, originally written for the Iris 4D, and parallelized it for the Cray-Y/MP at NASA Ames. This section will describe the code, our strategy for parallelizing it within our programming environment, and our results.

5.1 Strategy for Parallelization

The program uses a set of arrays to do the simulation, treating them as a group of layers. During an iteration, each array is computed in turn based on the values in the previous one. After examining the code, we identified the array operations that seemed to be the most time consuming. These passes were modified to run in parallel, yielding the following Delirium code as a first step (note that our first target machine, a Cray-2, has four processors so we chose to have four-way parallelism):

```
main()
  iterate
  {
    timestep=0,incr(timestep)
    scene=set_up(),
      let
        <a,b,c,d>=target_split(scene)
        ao=target_bite(a)
        bo=target_bite(b)
        co=target_bite(c)
        do=target_bite(d)
      in do_convol(ao,bo,co,do)
  }
  while is_not_equal(timestep,NUM_ITER),
  result scene

do_convol(c1,c2,c3,c4)
  iterate
  {
    slab=START_SLAB,incr(slab)
    convolve_data=pre_update(c1,c2,c3,c4),
      let
        <a,b,c,d>=convol_split(convolve_data)
        ao=convol_bite(a,slab)
        bo=convol_bite(b,slab)
        co=convol_bite(c,slab)
        do=convol_bite(d,slab)
      in post_up(slab,ao,bo,co,do)
  } while is_not_equal(slab,FINAL_SLAB),
    result convolve_data
```

Computation begins in `main`. It consists of an loop that goes through `NUM_ITER` iterations (these symbolic constants are replaced with values by the pre-processor). The first time, `scene` is created by going through `set_up`. On each subsequent iteration, we call `target_split` to divide the data into four equal pieces representing a subset of the targets being tracked. Each piece is handed to `target_bite`, which models the subset. The four updated pieces are handed to the `do_convol` function, which performs a convolution.

The function gives the four pieces to `pre_update`, which does some housekeeping chores needed before the convolutions. Because all four of the target pieces are used by `pre_update`, all of the `target_bite` calls must finish first and thus the operator acts as a synchronization point. This is a simple example of a *fork-join* construct, a common and useful way to express many parallel computations.

After `pre_update` is finished, it passes out the merged arrays. Because we are on a shared memory machine, merging is free — all we do is return a pointer to the entire array. As before, we split the data into four pieces, operate on them in parallel, and merge them.

5.2 Improving Load Balance

We ran the program and discovered that our speed-up on a four processor machine was slightly less than two. We were expecting to do better, so we ran on a single processor with individual node timing turned on. This is a feature of our run time system which prints out the amount of time each of the nodes in the graph took to execute. The times are roughly the same whether the system is running on one processor or many, barring cache conflicts and the like. We immediately discovered the reason for the disappointing performance from the following (time is expressed in ticks of the Cray-2's clock):

```
call of convol_split took 10013
call of convol_bite took 1059919
call of convol_bite took 1135594
call of convol_bite took 1060799
call of convol_bite took 1062540
call of incr took 3073
call of post_up took 45672
        ...
call of post_up took 4070365
```

We were getting very nice load balance on the convolutions, but the post_up routine was much more expensive than we thought. Roughly half of its invocations executed in negligible time while half took as long as all the convolutions combined. In the latter case, we could achieve at most a speedup of two regardless of how many processors we used. The solution was simple; we decomposed post_up into a four-way fork-join as well. Here is the final version of the do_convol function:

```
do_convol(c1,c2,c3,c4)
  iterate
  {
    slab=START_SLAB,incr(slab)
    convolve_data=pre_update(c1,c2,c3,c4),
      let
        <a,b,c,d>=convol_split(convolve_data)
        ao=convol_bite(a,slab)
        bo=convol_bite(b,slab)
        co=convol_bite(c,slab)
        do=convol_bite(d,slab)
      in let
          <u1,u2,u3,u4> = update_split(ao,bo,co,do)
          au=update_bite(u1,slab)
          bu=update_bite(u2,slab)
          cu=update_bite(u3,slab)
          du=update_bite(u4,slab)
         in done_up(slab,au,bu,cu,du)
  } while is_not_equal(slab,FINAL_SLAB),
    result convolve_data
```

When we did node timings on the new version, we found that we had almost perfect balance:

```
call of update_split took 16195
call of update_bite took 952171
call of update_bite took 952589
call of update_bite took 1171466
call of update_bite took 953576
call of done_up took 43239
```

5.3 Results

We will report our results in terms of speedup over the original sequential version, which we normalize to be 1. The speedup on a four processor Cray Y-MP was 3.3 (see figure 1 for more detail). As we expected, three processors perform at almost exactly the same rate as two. This is due to the fact that we have four roughly equivalent tasks to perform. Given three processors, one of them will do two of the tasks while the others do one each. The overall time is no improvement over two processors doing two of the tasks each.

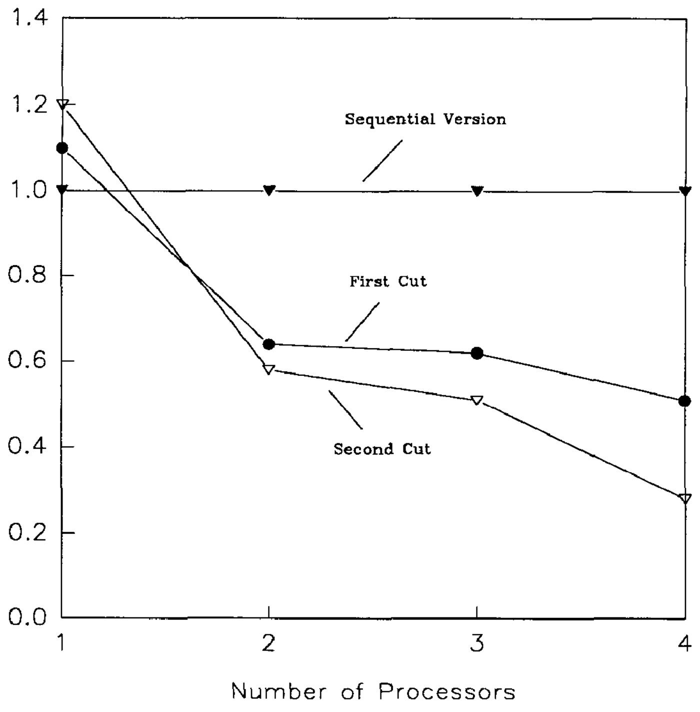

Figure 1: Retina Simulation on Cray Y-MP

We should mention that we have not devoted much effort to improving vectorization; we expect any such improvements to affect the sequential and parallel versions in roughly the same way, given the relative loads described above.

6 Case Study #2: Parallel Compilation

In most scientific codes, arrays are overwhelmingly the data structure of choice. We were curious how our ideas would transfer to other application domains which are often based on different structures. As we needed an optimizing compiler for Delirium, we decided to write it in itself [25]. We chose to use a fairly traditional implementation based on walking a parse tree; the challenge was to fit this strategy within our model and still get a reasonable speedup.

6.1 The Task

The compiler's task is to convert a Delirium program into a set of graphs, one per function. Unnecessary nodes in the graph translate into extra overhead at run-time, so the compiler uses a number of optimization techniques to improve the output. These include constant propagation, common sub-expression elimination, dead-code elimination, and in-line function expansion.

The basic algorithms for doing all these operations are well understood, but they depend on incremental update of a large data structure (a tree). This is a difficult task to perform in parallel. We examined each of the passes over the tree, and realized that with some work they can all be cast into one of three kinds of tree walk.

6.2 Parallel Tree Walking

The walking strategies are:

1. top-down update — walk the tree, updating each node as it is encountered. All updates can rely on an update having been completed beforehand for all of its ancestors.
2. inherited-attribute update — walk the tree, computing an inherited attribute as you move down. For each node, hand the operator an information package that represents all computations on the way down.
3. synthesized-attribute update — walk the tree from the bottom up, doing an update at a given node based on the information that has been computed for each of the children.

We used a simple strategy to parallelize these operations. Each walk is accomplished by traversing the crown of the tree, clipping off sub-trees. Sets of subtrees are allocated to each processor and handled independently. At the end, the pieces are merged back into a single tree. As before, "merging" is implicit and involves no actual work other than returning the pointer to the entire tree.

To ensure that the sets of subtrees allocated to each processor are roughly equivalent in weight, every tree node is annotated with the size of the subtree below it. We divide the total weight of the tree by the number of processors we will be using. The tree traversal runs until we find a subtree that is less than one-third of the desired weight.

After each of the sets of subtrees has been similarly handled, they are merged into a single tree again. In the case of the top-down update walk, there is no work left to perform so the merge simply returns a pointer to the entire tree. The synthesized attribute walk, on the other hand, must run over the crown of the tree finishing the pass now that the values for the subtrees have been computed.

6.3 Improving Load Balance

We ran the compiler with node timings enabled and discovered a number of problems. For example, the inherited attribute tree walk was poorly balanced – dividing the tree took as long as operating on one of the pieces. As the division is done sequentially, this limited speedup severely. We fixed this and a few other problems in less than a day using the Delirium profiling tools.

6.4 Results

The compiler is roughly 5500 lines of code. 5000 are the same in both the sequential and parallel versions. To switch to the parallel version, we remove a 100 line main module and replace it with 100 lines of Delirium and a 400 line auxiliary module that defines the operators. Most of the latter is made up of parallel tree-walking primitives.

After some tuning, we achieved a speedup of roughly 2.2 using three processors on a Sequent Symmetry; table 1 shows the amount of time required for each pass in the sequential version and on three processors in the parallel version. The speedup per pass ranges between two and three.

Time Per Compiler Pass (in msec)		
Pass	Sequential	Parallel (n=3)
Lexing	91	91
Parsing	200	78
Macro Expansion	117	50
Env Analysis	300	120
Optimization	350	160
Graph Conversion	380	160
Totals	1438	659

Table 1: The Parallel Compiler (on a Sequent)

7 The Run-Time System

Our research into efficient Delirium execution has focused on shared memory multiprocessors as a target architecture. We are also running a prototype Delirium implementation on a BBN Butterfly T2000.

The Delirium compiler outputs a *coordination graph* which expresses the data dependencies between operators. Coordination graphs have some of same semantics as traditional dataflow graphs [1]. In particular, when all the incoming arcs to an operator contain data, the operator is scheduled for execution.

Coordination graphs are a particularly flexible form of dataflow graph designed for efficient software implementation. In this form, subgraphs can be passed between operators. When a subgraph is passed to a special *call-closure* operator, the subgraph is expanded and evaluated. This extra flexibility makes it easy to express recursion, tail recursion, and closures using coordination graphs. Calls to functions (and closures) correspond directly to subgraph expansions.

The Delirium runtime system executes coordination graphs using a technique called *template activation*. The compiler converts functions into subgraphs called *templates*. The run time system executes small data structures called *template activations* which contain enough data buffer space to execute the given subgraph, and a pointer back to the template.

During the evaluation of a graph, the state of the computation is represented by a tree of template activations. Like Hudak's *program graphs* [15] these trees represent a parallel generalization of the stack data structure used to execute conventional sequential code.

The system maintains a *ready queue* of operators ready for execution. Whenever an operator has all its inputs, it is put in the ready queue. The ready queue has three levels of priority. In decreasing order of priority, they are: normal operators, non-recursive call-closure operators, and recursive call-closure operators. The priority scheme reduces the number of template activations required to evaluate a Delirium program, by making activations available for re-use as early as possible.

Operator scheduling is cheap because the template activation strategy supports the following simple assumptions about graph execution:

1. Each operator executes only once.
2. Once data is present on the input of an operator, it remains there until the operator executes and is never again present.

On the Cray Y-MP, Delirium runtime system overhead contributed less than one percent to the total execution time of the retina model (on four processors). This number is especially significant because the coordination graph for the retina model includes both closure creation and nested tail-recursive loop implementation.

A template activation contains a pointer to its template and enough memory to evaluate itself once. Since the templates do not change at runtime, they can be replicated in the local memory of each processor. As templates represent over 80% of the memory used by the runtime system at a given time, this organization reduces traffic on the Sequent and Cray busses and on the Butterfly network.

8 A Comparison of Coordination Models

Any model for coordination of sub-computations must incorporate a straightforward notion of how sub-computation scheduling interacts with memory access. A *coordination model* must specify a *data contention protocol* by which several sub-computations can agree on the modification of shared data.

The simplest of all coordination models is that of uniform, distributed shared memory. In this model, any sub-computation can access any part of shared memory. Sub-computations communicate through this memory. Higher-level coordination is done with locking (mutual exclusion) primitives embedded in a host language. On shared memory multi-processors such as the Sequent Symmetry, this model directly reflects the underlying architecture, and is a good low-level environment for programming [12].

The Ivy [19] system attempted to implement this coordination model on a message-passing multiprocessor. It had several drawbacks which led to poor performance. The refinements in the Tarmac [21] and Shared Data Object [4] systems partially address these drawbacks.

Linda coordinates sub-computations through Tuple Space, an abstraction which emulates shared memory. All data shared among sub-computations passes through Tuple Space via read, insert, and remove operations. The Tuple Space mechanism is novel in that it provides only associative primitives for finding tuples. A sub-computation requests a particular kind of tuple, and the system responds with a random selection from the set of tuples which match the request.

Like all the examples cited above, Linda is an *embedded* coordination language. Programmers embed uses of Linda primitives within a host language, usually C or FORTRAN [13]. The notation for these primitives resembles a procedure call in the host language.

RPC systems embed special procedure calls in a host language. Generally, uses of RPC are indistinguishable from regular procedure calls, but have special semantics (they may cross address-space boundaries). Similarly, message passing systems such as VORX [17], V [10], and Butterfly [18] embed calls to library functions with special data exchange semantics in a host language. These calls are expressed in the standard notation of the host language.

RPC and message passing systems have a simple coordination model — all shared data is passed in messages (or procedure arguments). This model supports a variety of higher level coordination models, among them replicated-worker [26] execution. To realize a higher-level model using message passing, one must devise a complex *communication protocol*. This is a difficult and error-prone task that most application programmers will not have time to undertake [2].

Object-oriented systems such as Emerald [7], Amber [9], and Sloop [20] provide an improvement over message-passing systems. They still use implicit RPC to implement complex higher-level protocols. However, these protocols are easier to understand because the abstract data types defined in such systems *encapsulate* shared data. That is, a particular RPC call on an abstract data type can only directly modify local data for the called instance of that type. This encapsulation has the additional benefit that one can improve performance by explicitly moving object instances about the network of processors [16]. Sloop, Emerald and Amber all provide an embedded primitive for specifying object locality.

The Ada coordination model is based on *task rendezvous*. The relative merits of this model are discussed in detail elsewhere [5]. Like the languages discussed above, one can think of Ada as containing an embedded notation for expressing operations in its coordination model.

Our coordination model is as follows:

1. All shared memory is explicitly passed between sub-computations.
2. A sub-computation may destructively modify a block of data only if it owns the sole reference to that data.
3. All sub-computations are *encapsulated*. That is, they have unique, well-defined entry and exit points. We call such encapsulated sub-computations *operators*.
4. The communication topology connecting individual operators does not change during execution. The topology itself supports conditional expression evaluation.

This model is somewhat restrictive (see table 2 for a comparison between coordination models). Iin the next section, we will explore some of the disadvantages of requiring a programmer to follow these constraints. However, the model provides several important advantages. As with other coordination languages, programmers can use existing code and program development tools to create sub-computations.

Most important, execution within the model is deterministic. If there is a bug in the program it will recur in exactly the same way every execution, greatly simplifying debugging. Development is simpler because a program that runs correctly on a uniprocessor will run correctly on a multiprocessor. Profiling is also straight-forward, as was demonstrated in the case studies.

Coordination Languages		
Language	Coordination Model	Notation
Delirium	restricted shared data	embedding
ADA	rendezvous	embedded
OCCAM	protocol	embedded
RPC	protocol	embedded
Linda	shared database	embedded
Concurrent Prolog	shared variables	radical
ALFL	shared data	radical
Enhanced Fortran/C	task-oriented	embedded
Emerald/Sloop	protocol	embedded

Table 2: Coordination Model Comparison

9 A Critique of the Model

We have shown how to implement some common types of applications while operating within these boundaries. Nevertheless, there are some inherent limitations to our approach which we would like to discuss.

9.1 Some Approaches Don't Fit

There are several methodologies for solving problems in parallel. One of them is the "replicated-worker model" [26] [8], in which tasks are generated and put on a queue (at least abstractly – the implementation may not explicitly create a queue). A group of identical workers reads from the queue, executing jobs as they appear and possibly adding more jobs to the queue. This is, somewhat ironically, precisely how our run-time system is written. Any implementation of the replicated-worker model within our system, however, would be clumsy and inefficient due to the unrestricted nature of interactions between the workers.

On the other hand, it is just those kinds of unrestricted interactions that make MIMD parallel programs very difficult to debug. Within our model, the computed result is deterministic regardless of the number of processors you are using and the order of execution. We are accustomed to debugging applications on a single processor and have grown to expect them to work immediately when executed on many. We have found this to be a great advantage which compensates, to some degree, for the loss in flexibility.

9.2 Parallelism is Hard-Wired

As we have shown above, one of the most common types of parallelism that is convenient to express for large data structures is the fork-join. In the examples above, the number of pieces into which a data structure is divided is chosen explicitly by the Delirium programmer. This is an awkward way to describe high degrees of parallelism and cannot take into account the load of the system. We have addressed this problem by generalizing the language with a notation that encompasses more complex coordination [22], but it is beyond the scope of this paper to discuss the details.

9.3 Locality

In our basic model, there is no notion of exploiting locality. Because we have been dealing with shared-memory machines that have uniform access times to memory, we are able to ignore the issue to some extent. Caching is an exception, however; once the cache of a processor has been filled with data from one part of a data structure, that processor may be quite a bit faster than another in performing the next operation on that data. On a machine like the Butterfly where access to remote memory locations is a great deal more expensive, it becomes very important to exploit locality wherever possible.

We have two preliminary approaches to this problem which we have just begun to experiment with, both based on the notion of *affinity*. The first idea is based on operators; once a given operator has executed on a processor, it prefers to run on that processor in the future. This preference is overridden if the desired processor is busy and another can't find any other operator to execute. The second version is a little more complex, attaching a processor preference to the header of each data block. When an operator is scheduled for execution, the run time system takes into account the size and cached locations of its inputs in an effort to exploit locality.

We expect affinity to be of some use on machines like the Cray, but to be particularly important on architectures like the Butterfly which have non-uniform access to memory. As remote access becomes more expensive, the "preference" will be correspondingly strengthened.

10 Conclusion

We have outlined a new strategy for expressing coordination of sequential sub-computations, realized in the embedding language Delirium. In contrast to existing embedded languages, the notation clearly expresses the coordination framework of the application. All the coordination required to execute the program is expressed in a unified Delirium program. The program contains the computational code in the form of embedded operators, written using conventional tools (and, usually, existing sequential code).

Our environment, which executes on a variety of shared memory multi-processors, provides tools which make it possible to develop parallel applications quickly. It supports a coordination model that can guarantee deterministic execution. Programmers who remain within the restrictions of the model can develop a program on a sequential machine and be certain that it will execute deterministically on a variety of parallel architectures as well.

Thus, Delirium incorporates a powerful coordination model within the framework of an embedding coordination language, providing considerable leverage to parallel programmers without requiring them to change any of their computational code.

We have provided two case studies to show how the system works in practice. These case studies demonstrate that the environment can support rapid development of efficient parallel programs in a variety of application domains.

11 Acknowledgements

We would like to gratefully acknowledge the help and support of Eugene Miya (at the NAS Systems Division, NASA Ames Research Center), Berni Alder (at Lawrence-Livermore National Laboratory) and particularly George Michael (also at LLNL). We also thank David Gelernter for pointing out the importance of coordination languages as a practical discipline.

References

[1] W.B. Ackerman. "Data Flow Languages,". *Computer*, 15(2), February 1982.

[2] David Anderson. "Automated Protocol Implementation with RTAG,". *IEEE Transactions on Software Engineering*, 14(3):291–300, March 1988.

[3] Gregory R. Andrews and Fred B. Schneider. "Concepts and Notations for Concurrent Programming,". *Computing Surveys*, 15(1), 1983.

[4] H. Bal and A. Tannenbaum. "Distributed Programming With Shared Data,". In *Proceedings of the International Conference on Computer Languages*, 1988.

[5] Henri Bal, Jennifer Steiner, and Andrew Tanenbaum. "Programming Languages for Distributed Computing Systems,". *Computing Surveys*, 21(3):261–322, September 1989.

[6] A. Birrell and B. Nelson. "Implementing Remote Procedure Calls,". *Transactions on Computer Systems*, 2(1):39–59, February 1984.

[7] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald,". *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[8] N. Carriero, D. Gelernter, and J. Leichter. "Distributed Data Structures in Linda,". In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1986.

[9] Jeffery S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. "The Amber System: Parallel Programming on a Network of Multiprocessors,". *Proc. Twelfth ACM Symposium on Operating Systems, Operating Systems Review*, 23(5):147–158, December 1989.

[10] D. R. Cheriton. "The V Distributed System,". *Communications of the ACM*, 31(3):314–333, March 1988.

[11] Frank H. Eeckman, Michael E. Colvin, and Timothy S. Axelrod. "A Retina-Like Model for Motion Detection,". In *IJCNN International Conference on Neural Networks*, pages 247–249, Washington, D.C., 1989.

[12] Gary Fielland. "The Balance Multiprocessor System,". *IEEE Micro*, 1(8):57–69, February 1988.

[13] David Gelernter. "Parallel Programming in Linda,". In *Proceedings of the International Conference on Parallel Processing*, pages 255–263, August 1985.

[14] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages,". *Computing Surveys*, 21(3):359–411, September 1989.

[15] Paul Hudak and Ben Goldberg. "Distributed Execution of Functional Programs Using Serial Combinators,". *IEEE Transactions on on Computers*, C-34(10), October 1985.

[16] E. Jul, H. Levy, N. Hutchinson, and A. Black. "Fine-Grained Mobility in the Emerald System,". *Transactions on Computer Systems*, 6(1):109–133, February 1988.

[17] H.P. Katseff, R.D. Gaglianello, and B.S. Robinson. "The Evolution of HPC/VORX,". In *ACM Symposium on Principles and Practice of Parallel Programming*, 1990.

[18] T. J. LeBlanc, M. L. Scott, and C.M. Brown. "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor,". In *Proceedings of the ACM SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 161–172, July 1988.

[19] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems,". In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distribued Computing*, pages 229–239, 1986.

[20] Steven Lucco. "Parallel Programming in a Virtual Object Space,". In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987.

[21] Steven Lucco and David Anderson. "Tarmac: A Language System Substrate Based on Mobile Memory,". In *to appear in: Internation Conference on Distributed Computing Systems*, 1990.

[22] Steven Lucco and Oliver Sharp. "Parallel Programming With Coordination Structures,". submitted for publication.

[23] J. Andrew McCammon and Stephen C. Harvey. *Dynamics of Proteins and Nucleic Acids.* Cambridge University Press, 1987.

[24] James R. McGraw. "The VAL Language: Description and Analysis,". *ACM Transactions on on Programming Languages and Systems*, 4(1):44–82, January 1982.

[25] Oliver Sharp. "Pythia: An Optimizing Parallel Compiler,". Master's thesis, University of California/Berkeley, 1990.

[26] Mark Sullivan and David P. Anderson. "Marionette: a System for Parallel Distributed Programming using a Master/Slave Model,". In *International Conference on Distributed Computing Systems*, 1989.

[27] A. Takeuchi and K. Furukawa. "Parallel Logic Programming Languages,". In *Proceedings of the 3rd International Conference on Logic Programming*, pages 242–254, July 1986.

[28] S. Ulam and N. Metropolis. "The Monte Carlo Method,". *Journal of the American Statistics Association*, 44:335ff, 1949.

[29] S. Yemini. "On the Suitability of ADA Multi-Tasking for Expressing Parallel Algorithms,". In *ADA TEC Conference on ADA*, pages 91–97, October 1982.