A Domain-Specific Parallel Programming System

I: Design and Application to Ecological Modelling

Elaine Wenderholm *
Sch. of Computer and Info. Science
Syracuse University, New York
wender@top.cis.syr.edu

Micah Beck
Dept. of Computer Science
University of Tennessee at Knoxville
beck@cs.utk.edu

September 1994
SCCS-640

Abstract

The goal of the $\mathcal{E}m$ project is to make parallel programming easily accessible to a broad community of scientists. Previous approaches such as the use of general parallel programming languages and parallelizing compilers for sequential languages have fallen short in this respect. The approach is to design a special purpose programming language which is oriented towards a specific area of application. The result is a specialized and effective scientific tool.

$\mathcal{E}m$ is a high-level programming system which puts parallelism into the hands of scientists who are not sophisticated programmers. By restricting and simplifying the programming interface, $\mathcal{E}m$ eases both the conceptual task of the programmer and the analytical task of the compiler. The model of success is the financial spreadsheet, a specialized tool which makes programmers out of relatively naive end-users and makes computer technology broadly accessible to business. Here the initial prototype is described, motivated by practical ecological modelling problems.

*Supported by a Patricia Roberts Harris Fellowship

1 Introduction

Parallel computing hardware now is affordable to a broad range of scientific users. Current parallel programming efforts have focused on application areas where specialist programmers can extract the maximum possible performance from the hardware. The decreasing cost of hardware will allow a much larger class of users to exploit parallelism if the programming model can be simplified. The main obstacle to the widespread use of parallel computing hardware is the difficulty of the programming model.

The goal of the $\mathcal{E}m$ [1] project is to put into the hands of knowledgeable scientists the ability to program parallel systems. The approach is to design a special purpose programming language which is oriented towards a specific area of application. By restricting the problem domain the complexity for the programmer is reduced, and at the same time compilation is simplified. The result is a specialized and effective scientific tool.

Two examples of specialized programming systems are financial spreadsheets such as Lotus and symbolic computation systems such as Mathematica. Each of these tools has allowed a community of users to write applications that previously required specialist programmers. Many users simply would be unable to develop such applications without the use of these specialized software systems.

The successes of these tools share three principles:

1. Each addresses a restricted and well-defined problem domain.
2. The interface to each tool is designed to be intuitive to the target user community.
3. Features from declarative and functional programming are incorporated into the language, thereby freeing the user from programming details.

   the need to manage storage and other machine resources.

The design of $\mathcal{E}m$ follows these same three principles:

1. $\mathcal{E}m$'s problem domain centers on the class of simulation problems which is statically decomposed, has communication localized to a fixed neighborhood, and has time incremented synchronously after all cells are updated.
2. $\mathcal{E}m$ provides a high-level interface with a domain-specific library. The library can be customized to a specific area of scientific investigation.

[1] pronounced *ĕm*

3. $\mathcal{E}m$ programs are almost purely functional. This relieves programmer of the need to manage storage and other machine resources, a most difficult task when writing parallel programs.

The rest of this paper is organized as follows. Section 2 details the advantages of a domain-restricted language. Section 3 gives an overview of $\mathcal{E}m$: first the major components of the system are explained; next a wetland ecosystem example is defined, along with it's implementation in the $\mathcal{E}m$ language. In Section 4 the $\mathcal{E}m$ programming language is compared to existing parallel program systems, and the specific goals of the project are related to design of the language. To better support parallelism, $\mathcal{E}m$ enforces a set of array access rules. These rules are presented in Section 5. The unique features supported by the $\mathcal{E}m$ language are explained in detail in Section 6 using the wetland example. Section 7 justifies the design decisions made in light of conflicting requirements. Section 8 shows how the major paradigms for describing ecosystem models map into $\mathcal{E}m$ programs. In Section 9 $\mathcal{E}m$ is compared and contrasted with related work, and finally, plans for future work are discussed.

2 Domain-Specific Parallel Programming

> *In the long run, high-level programming environments incorporating domain knowledge may well supersede current low level programming tools.* [2]

$\mathcal{E}m$ is designed to provide scientists with a powerful, convenient, and easily understood tool for expressing, understanding, and modifying domain-specific programs. The class of problems which $\mathcal{E}m$ addresses have been called *loosely synchronous* [FJL+88], and the type of loop required to solve this class of problems has been called a *sequentially-iterated parallel loop* [HA90]. The iterative structure of such problems occurs commonly in in computational science [FJL+88, HA90].

The loop nest required for this domain-restricted problem is one in which the outer temporal DO loop is sequential, synchronizing after each time step. The inner spatial loops are those which may run in parallel, depending on data dependences, and update large data sets.

The data for the inner spatial loops are partitioned across processors. Each processor executes the same code for different portions of the data, and communication is required for processors which must share neighboring data. Such problems are computationally intensive, highly structured and amenable to a declarative style of programming.

[2] from [ZC91], p. 4

The strengths of $\mathcal{E}m$ as a parallel programming language for scientists who are not programming specialists derive from the features of the restricted domain.

1. An $\mathcal{E}m$ program is *data parallel*: the same computational kernel is performed at every cell of the grid by different processors at each instant in simulated time. The $\mathcal{E}m$ language reflects the structure of a specific problem domain, and so is architecture-independent.

2. The computational kernel is specified in a declarative style, enabling the programmer to focus on *what* the model is to compute, and not *how* to compute it. The ordering of statements, the reuse of memory, and the transformation of loops to maximize parallelism are the responsibility of the compiler.

3. The domain-specific library used by $\mathcal{E}m$ is intuitive for scientific programmers and is also an efficient implementation of much of the kernel code. Each procedure in the library includes with it a *procedure summary* which reflects an analysis of data accesses made by that procedure.

3 Overview of $\mathcal{E}m$

3.1 Components

A block diagram of the $\mathcal{E}m$ programming system is shown in Figure 1. Programming using $\mathcal{E}m$ is divided between two components: *Model Description* and *Domain-specific Programming*. The first component contains the code for the model description, i.e., code for problems which are loosely sunchronous. It is written in the $\mathcal{E}m$ language, and consists of a high-level, declarative description of the model: space and time bounds, state variables, and the procedures which the model calls.

All procedures which are called in the $\mathcal{E}m$ code are contained in the *Domain Library*, DS-lib. These procedures are written in a source language, e.g. Fortran or C, for which there is a compiler on the target architecture. Procedures are written to conform to simple rules established by the $\mathcal{E}m$ programming system: the procedure's calling sequence is standardized; each procedure in the DS-lib has a *procedure summary*: a description of the data accesses performed by that procedure.

Both the *Model Description* and the *Domain Library* are user-written. In order for $\mathcal{E}m$ to tailor the application program to a specific architecture, $\mathcal{E}m$ is bundled with an *$\mathcal{E}m$ Library*, $\mathcal{E}m$-lib, which contains data access routines. These routines are used by the the programmer to read and write data.

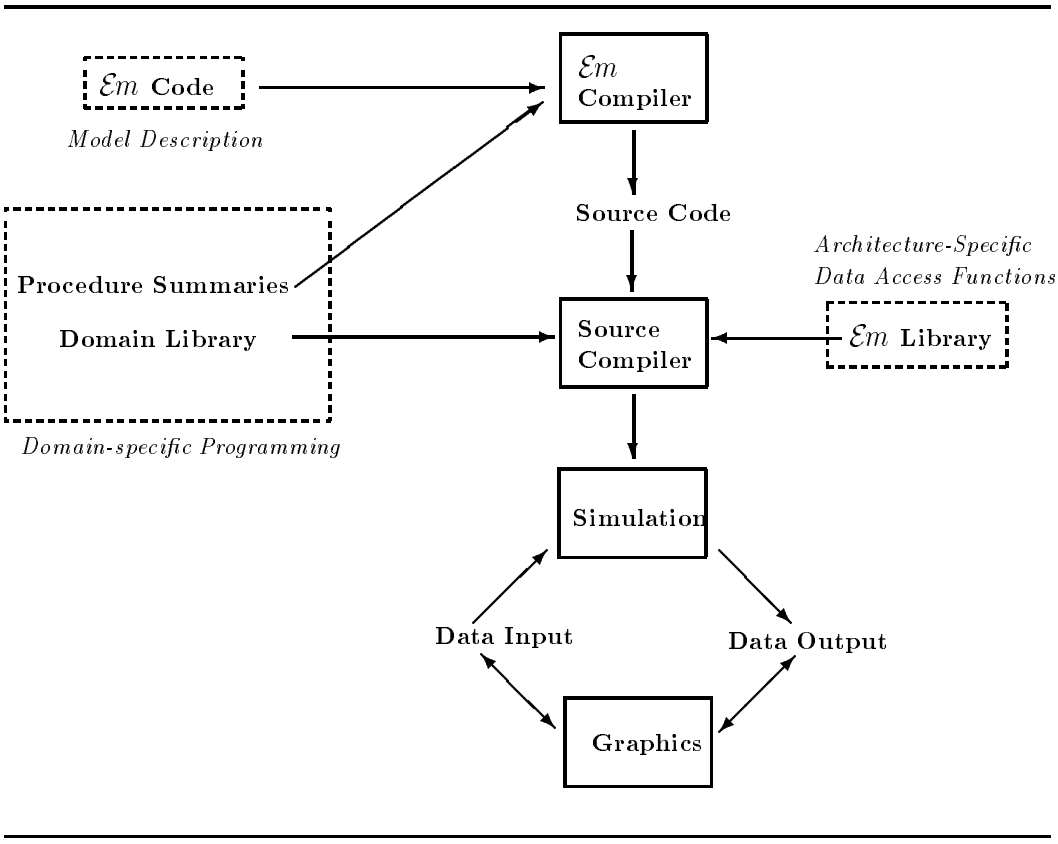

Figure 1: Block Diagram of $\mathcal{E}m$ Programming System

Generating executable code for the target architecture is a two-step process. The $\mathcal{E}m$ compiler generates the main source code for the target architecture. Next, the compiler/linker on the target architecture generates the final executable code, using the main source code, the procedure source contained in the DS-lib, and data access routines contained in the $\mathcal{E}m$-lib.

Finally, the simulation code is run within a standardized I/O interface.

The unique features of models written using $\mathcal{E}m$ are detailed below.

3.2 An Example

$\mathcal{E}m$ is explained through the use of an example from ecology.

The Las Tablas de Daimiel National Park in the La Mancha region of Spain is a wetland, and is home to many species of ducks. The region, hot and dry in the summer, is known for its wine and cheese. Rainfall in the area has been relatively

low for the past 15 years, causing farmers in the surrounding countryside to drill deep wells to irrigate their crops. Compounding the lack of inadequate rainfall, the low price of grapes has caused many farmers to switch to other crops, such as corn, which require more water.

The pumping of water causes the water table in the park to drop, often to the point where the wetland dries up and the ducks are forced to leave. In addition, the risk of fire in the dry marshes is high. In order to develop policy for the rationing of water to farmers, it is important to model the effect of such policies on the duck populations and on the risk of fire.

This ecosystem is described easily using $\mathcal{E}m$. Each cell in the simulation is defined by five state variables:

1. the fixed geography of the region,
2. the concentration of water,
3. the amount of water being pumped,
4. habitation by ducks, and
5. the risk of fire.

The simulated ecosystem has five processes:

1. randomized determination of rainfall based on historical data,
2. flow of water at and below the surface,
3. determination of pumping levels according to the policy being simulated,
4. the movement of ducks, and
5. computation of the risk of fire.

The geographical area is modeled as a three-dimensional grid with the x coordinate (corresponding to latitude) ranging from 1 to 100 and the y coordinate (corresponding to longitude) ranging from 1 to 400. The z coordinate (corresponding to depth) ranges from 1 at the surface to 10.

In this example, an $\mathcal{E}m$-lib is assumed to include procedures which implement the basic processes of the simulation. An $\mathcal{E}m$ program which describes this model is shown in Figure 2.

```
model wetland over time: 1..20000;

space
   x: 1..100;
   y: 1..400;
   z: 1..10;
end space

variables
   int geography;
   int water;
   int pumping;
   int ducks;
   float risk_of_fire;
end variables

simulation

initialize(geography)
initialize(water)
initialize(pumping)
initialize(ducks)
loop
   flow(water,geography, water)
   extract(water, pumping, water)
   move(ducks, water, ducks)
   estimate_risk(water, risk_of_fire)
   rain(water,water)
end loop
save_results(ducks)
save_results(risk_of_fire)

end simulation
```

Figure 2: $\mathcal{E}m$ Wetland Model

4 Design Features

4.1 Language Design Philosophies

The design philosophy for parallel programming systems often includes the structure of the target architecture and requirements for stringent performance. Languages which stress message passing and data parallelism were developed to meet the immediate needs of the programmers of new MIMD and SIMD architectures respectively [FJL+88, Hil85]. Highly parallel functional languages were embraced by designers of dynamic dataflow architectures [NA89, Nik90]. Each of these approaches has added to the understanding of parallel programming, achieving high performance on problems which match the structure of the corresponding architectures.

These architecture-independent parallel programming have had limited success:

1. Data parallel programming model has found the widest application, being applied to vector, SIMD, and MIMD architectures [CFR+92, Thi89, CG89]. However not all algorithms have a natural and efficient expression in a data parallel language [FJL+88].
2. Functional languages support increaced parallelism, but have not achieved high performance on any parallel execution platform [VB90, CFD90] because of their reliance on dynamic data mechanisms and on compilers to map the language to the target architecture.
3. Message passing extensions to sequential programming languages have been developed which are portable between a wide variety of MIMD architectures and which achieve high performance [GBD+93]. The parallel aspect of this programming model is an irregular pattern of communication, and usually is too difficult for most non-specialist programmers. Additionally these extensions limit the portability of the code.

In the design of $\mathcal{E}m$, an element is borrowed from each of these successes:

1. Data parallelism is a natural programming paradigm for many scientific problems, and it allows a parallelizing compiler to *reduce the problem of process decomposition to that of data distribution.*
2. Functional programming languages reduce data dependences by enforcing a restricted storage model, thereby increasing parallelism.
3. Most parallel algorithms can be expressed in terms of sequential primitives which are most naturally and efficiently expressed in a conventional programming language.

4.2 Expression and Analysis of Parallelism

Data parallel and functional programming paradigms require the programmer to express algorithms so that parallelism is *implicit* in the program. In contrast, sequential languages augmented with message passing require the programmer to express *explicitly* the parallelism in algorithms.

The programming model for explicit parallel systems directly reflects the placement of data and communication between computing elements as specified by the programmer. But in order for the program to achieve high performance, the programmer must perform a sophisticated ad hoc analysis of the program, and also must have a clear understanding of the execution model. More implicit systems rely on a compiler or some other tool to introduce parallelism automatically. The programmer thus has less control over the use of parallelism and the allocation of resources.

In most current systems, resource allocation is shared between the system and the programmer: the programmer specifies the data placement and the system automatically inserts the communication implied by the partition [ZBG88, CFR+92]. The scientific community continues to use Fortran and its variants as a programming language, and the compiling systems rely on the fact that the programmer provides the data decomposition. Given such a standard imperative language, the compiler must perform program analysis in order to transform the sequential program into equivalent concurrent code for execution on the parallel machine. In $\mathcal{E}m$, the goal is to capture the power of implicit systems without sacrificing the performance of explicit parallel programming.

Compiler technology for vector and parallel computers has advanced both in language analysis and in the mapping to the target machine. Parallelizing compilers for imperative languages have concentrated on nested DO loops, which are of primary importance for scientific programming. The techniques developed for nested DO loop optimization rely on subscript analysis, which characterizes the necessary order between operations in different iterations of a loop. When dependence analysis can be expressed as an integer programming problem, its exact solution is exponential [Pug92]; in the general case it is undecidable.

Optimizing compilers also must make optimum use of the specialized hardware features of the target machine. Loop transformations [Wol89, ZC91] performed by the compiler must take into account not only the data dependence constraints but also the machine architecture.

Two factors limit the effectiveness of these compilers: the structure of most programs is not sufficiently regular, and low level code is hard to analyze. Even a program which might be written in a regular style might not be parallelizable by the compiler, simply because a general purpose language is used.

The design of the $\mathcal{E}m$ language addresses the current state of compiler technology in several ways:

1. Programs expressed in $\mathcal{E}m$ have regular communication patterns.
2. $\mathcal{E}m$ programs are data parallel and functional, and are expressed in terms of calls to a domain-specific library, DS-lib.
3. The efficient implementation of procedures in the DS-lib is accomplished using conventional imperative languages and ad hoc analysis techniques.

4.3 Domain-Specific Compilation

$\mathcal{E}m$ not only simplifies programming, but also simplifies the task for the compiler. The problem of compiling models is divided between two portions: the compilation of regular programs consisting of pre-analyzed procedures; and the efficient implementation and accurate analysis of those procedures. The scheduling of regular, fully analyzed programs can be performed well using current compiler technology. The use of a domain-specific library eliminates the need for the compiler to analyze irregular code.

Because $\mathcal{E}m$ leaves the specification of most resources to the compiler, it is a highly implicit parallel programming system. The use of conventional programming languages and ad hoc analysis techniques in the development of the DS-lib will provide performance close to that of an explicitly parallel programming system.

The $\mathcal{E}m$ language does not deal with resources explicitly. The $\mathcal{E}m$ compiler generates the decomposition, which is used by the parallelizing compiler on the target machine. The user selects the parallel communication package to be used, and the $\mathcal{E}m$ compiler inserts the communication messages. The parallelizing compiler distributes the execution of the procedures across processors.

In the wetland example, as in most $\mathcal{E}m$ programs, the structure of the $\mathcal{E}m$ program is trivial. The kernel of the computation takes place in a sequence of calls to procedures in the domain-specific library. This reduces the problem of program analysis to that of interprocedural analysis, a problem which is generally more difficult than global program analysis.

The compilation strategy of the $\mathcal{E}m$ compiler lies in the fact that data access characteristics for the $\mathcal{E}m$ procedures are known to the compiler without any compile-time analysis. The data access patterns of each DS-lib procedure are described in a *procedure summary* which is entered into the library along with the code which implements the procedure.

The procedure summary specifies exactly the information needed by the $\mathcal{E}m$ compiler to perform resource allocation and parallelization. A range of strategies are available for deriving the procedure summary: automatic analysis, programmer-aided analysis, or explicit specification by the programmer.

In the current implementation of $\mathcal{E}m$, the information is explicitly provided by the programmer. Clearly it is preferable to automate the derivation of program summaries, but this is an open area of research. From an engineering point of view, it is important that the project does not depend on such solutions.

The procedure summary is used for two tasks performed by the $\mathcal{E}m$ compiler: generating the target source code and generating the data partition. Such information has already been used successfully for generating data partitions [HA90].

The $\mathcal{E}m$ compiler does not generate assembly language, but instead generates source code for some standard imperative language supported by the target architecture. Currently $\mathcal{E}m$'s high level language may be mapped to Fortran or C(++), thereby giving the programmer flexibility in porting application code to other machines. The target language must be able to call the procedures in the DS-lib, so the choice of a target language is not arbitrary.

Additionally, the programmer selects a serial model or a parallel model. This facilitates program debugging. Since parallel programs are difficult to debug, $\mathcal{E}m$ will generate serial code so that the model can be debugged easily. $\mathcal{E}m$ generates parallel code for the parallel communication system that resides on the target machines.

5 $\mathcal{E}m$ Array Access Rules

$\mathcal{E}m$ restricts memory reads and writes in order to promote parallelism. These rules serve to eliminate resource dependences.

The following simple rules are enforced:

1. A read may be to any previous time step.
2. Any read to the current time step causes textual order of procedure invocation.
3. All writes must be to the current time step.
4. Writes may overlap between procedures (causing an output dependence) provided the overlap is at the current cell. In this case the compiler must preserve the textual order of procedure invocation.

5. If a write is declared a *reduction* at *any* location, then procedure invocation order is not important. This rule may be superceded by Rule 4.

6. Any other write is an error.

6 $\mathcal{E}m$ Language Features

The unique language features of $\mathcal{E}m$ are described through the use of the wetland program shown in Figure 2.

$\mathcal{E}m$ has two unique data types: *coordinate variable* and *abstract variable*. An $\mathcal{E}m$ program has three main components; coordinate declaration, abstract variable declaration, and commands. All computations are performed within procedures from the domain-specific library. Procedures are defined implicitly by use.

6.1 $\mathcal{E}m$ Data Types

6.1.1 Coordinate Variable

A set of coordinate variables are used to define the $\mathcal{E}m$ model space. The *model space* is composed of the spatial coordinates, textually ordered, and the temporal coordinate, `time`, which is the last component of the space. By default coordinate variables are incremented by 1.

`time` has two properties: a lower bound and an upper bound. A lower bound and upper bound specify the minimum and maximum values, respectively, for the coordinate in the model space. A spatial coordinate has three properties: lower bound, upper bound, and wrapped.

The keyword *wrapped* specifies a non-linear coordinate space. If not declared as wrapped, the coordinate is in linear space. If it were wished to, say, define the `x` coordinate as wrapped, then `x` would be defined as `wrapped x: 1..1000`. Because of the difficulty in handling boundary conditions in theoretical models such as forest growth, spatial coordinates typically are wrapped [Bel]. But in the wetland example a geographic area is being modelled, and wrapped is inappropriate.

The wetland model has three spatial coordinates, `x`, `y` and `z`, and the temporal coordinate `time`. The compiler generated DO loop nest will consist of the variables `x`, `y`, `z`, and `time`.

Coordinate variables are *read only*, and may be referenced (read) in order to influence

the computation using a conditional expression.

6.1.2 Abstract Variable

The second $\mathcal{E}m$ data type is the *abstract variable*. Abstract variables are declared within the delimiters `variables ... end variables`. The variables `water` and `ducks` are two of the abstract variables in this program.

Notice how these variables look as though they are scalars. In the $\mathcal{E}m$ language, programmers use *abstract variables* instead of explicitly-defined arrays for state variables. An abstract variable is an application-specific data type which allows the programmer to describe *what* will be computed, but not *how* it will be computed.

Viewed functionally, a simulation calculates an iterated expression for a set of variables which are defined incrementally. Variables in a simulation have spatial components and a temporal component, and are typically implemented as an array data structure in an imperative language.

Without the ability to avoid implementation details, a programmer must use prevailing languages. The syntax and associated semantics of array declaration and array use in imperative languages vary significantly. Being tied to a specific source language hinders software portability. In addition, increasing the capability of the program invariably requires the redefinition of the data structures and those routines which access them – a time consuming and error-prone task. $\mathcal{E}m$'s abstract variables allow the programmer to *abstract* away implementation details.

A *concrete variable* is a finite and bounded data structure which implements an abstract variable using a finite amount of memory. It is the job of the $\mathcal{E}m$ compiler to generate a concrete variable for each abstract variable. Program summaries, discussed below, provide the means to deduce the concrete representation for an abstract variable.

Abstract variables have two attributes: primitive data type and distribution. The primitive data types are integer and real (float). The distribution may be defined as either *dense* or *sparse*. Distribution directly reflects implementation: state variables which are sparsely distributed may be more efficiently implemented as a linked list. The exact implementation is determined by the compiler. The default distribution is *dense*. All variables in the example have dense distribution. Should it be desired to model `ducks` as sparse, the declaration would be `int ducks of sparse distribution;`.

Reads and writes to abstract variables are performed solely through the $\mathcal{E}m$ data interface routines, comtained in the $\mathcal{E}m$-lib. These routines are packaged with the $\mathcal{E}m$ system, and are tailored to the target source language and target machine.

6.2 $\mathcal{E}m$ Loop Command

Notice how the loop in the example is defined declaratively: a loop command is simply `loop ... end loop`. The loop nest level of the variables and the loop bounds are left unspecified by the programmer. As before, $\mathcal{E}m$ allows the implementation details of the loop nest to be abstracted away. The $\mathcal{E}m$ compiler determines the optimal loop nest. The maximum loop bounds for each level (in this case for `x`, `y`, `z`, and `time`) are also defined declaratively. The actual loop bounds are deduced using the declared model space and data access information in the procedure summaries. An optimized loop nest is determined based on the target source language and machine architecture.

Typically optimizing compilers perform loops transformations by rewriting the original loop nest written by the programmer. $\mathcal{E}m$ takes a different approach: since the $\mathcal{E}m$ compiler has facts about the loop variables, it can determine the optimal loop nest. The programmer is not burdened with the task of specifying the loop in the first place.

The loop command has restricted use: a loop command appears only once in an $\mathcal{E}m$ program; a loop commands may not be part of a conditional command. In the event that the use of loop commands requires expansion, the language can be modified appropriately.

6.3 $\mathcal{E}m$ Procedure Summary

Each $\mathcal{E}m$ library routine has a *procedure summary*, similar in spirit to [Cal88]. As stated in Section 3, there is a standardized calling sequence for procedures which bears a strong resemblance to procedure invocation in functional languages. Specifically, a procedure uses value-result semantics for procedure paramaters, and has no access to non-local variables. In essence, then, the procedures appear to act as functions, returning one or more values, with no side-effects. Conventional programming languages, and Fortran in particular, require extensive interprocedural analysis due to the effects of aliasing, and equivalence statements. The restrictions placed on $\mathcal{E}m$ procedures makes analysis much simpler.

At the statement level, $\mathcal{E}m$ uses standard imperative semantics: variables have values which can be modified by assignment statements. This programming paradigm is quite natural to programmers familiar with conventional programming languages, and makes programming in $\mathcal{E}m$ an easy task.

Figure 3 shows the procedure summaries for all procedures in the example. The procedure summary contains data access information as read, write and +reduce

```
procedure flow(oldwater, soilmap, newwater)

read oldwater [0,0,0,-1] [0,0,-1,-1]
read soilmap [0,0,0,na] [0,0,-1,na]
write newwater [0,0,0,0]
+reduce newwater[0,0,-1,0]

procedure extract(oldwater, pumping, newwater)

read oldwater [0,0,ub,-1]
read pumping  [0,0,na,na]
write newwater [0,0,ub,0]

procedure move(oldducks, water, newducks)

read oldducks [1,0,na,-1] [1,1,na,-1] [0,1,na,-1] [-1,1,na,-1]
read oldducks [-1,0,na,-1] [-1,-1,na,-1] [0,-1,na,-1] [0,0,na,-1]
read water [0,0,lb,-1]
write newducks [0,0,na,0]
+reduce newducks [1,0,na,0] [1,1,na,0] [0,1,na,0] [-1,1,na,0]
+reduce newducks [-1,0,na,0] [-1,-1,na,0] [0,-1,na,0] [0,0,na,0]

procedure estimate_risk(water, fire)

read water [0,0,lb,-1]
write fire [0,0,na,0]

procedure rain(oldwater, newwater)
read oldwater [0,0,0,-1]
write newwater [0,0,0,0]
```

Figure 3: Procedure Summaries for Wetland Model

(sum reduction) and *reduce (product reduction).

A relative address specifies a cell address in the model space relative to the current cell. The textual definition of coordinates in the $\mathcal{E}m$ model space specify the order of relative address coordinates. In a three-dimensional space, the current cell has relative address `[0,0,0]` and absolute address `[x,y,time]`; "`[1,-1,-1]`" refers to the the current cell's southeastern neighbor, in the previous time step. Thus the absolute address of the southeastern neighbor is `(x+1, y-1, time-1)` .

An absolute address is used to isolate a *slice* of the iteration space. An expression "[lb|ub] [[+|-] INT]", where lb and ub denote, respectively, the declared lower bound and upper bound of the coordinate, and INT denotes an integer. The lower and

upper bounds are specified symbolically in the program summary; the actual values are determined by the $\mathcal{E}m$ compiler. Finally, the string, "`na`", denotes the coordinate is inapplicable or unnecessary.

All coordinates must be specified, that is, if the $\mathcal{E}m$ model defines a 4-dimensional space, then a 4-tuple must be specified for each variable.

The data access patterns are used by the $\mathcal{E}m$ compiler to

1. generate data dependences,
2. generate the data structures for the concrete variables,
3. define DO loop bounds,
4. and generate, if necessary, conditional commands within the DO loop.

Let's examine a little further just what these data access patterns *mean* to the programming model.

If all variables are read only from previous time steps, then there are no read dependences in the current time step. In this case, all data reads can be arbitrarily ordered within the current time step. As evidenced in the wetland example, this cannot be done.

Notice that both routines `flow` and `extract` write to `water` at the current cell address. This constitutes an output dependence. This dependence imposes sequentiality on the order of execution of the two procedures.

The execution of procedure `move` is restricted in the `x`, `y`, and `z` coordinates: the restriction on `x` and `y` are due to non-local reads and writes; the restriction on `z` is due to execution of the procedure only at the "surface" of the grid, as evidenced by the `lb` declaration in the procedure summary.

Notice that a dependence graph for this program can be generated even when the scheduling of the program into loops has not been done. Because of the features of the language, $\mathcal{E}m$ does not need to do loop normalization. The important aspect to realize here is this: *the nesting order of the loop is really a scheduling decision rather than an inherent dependence relation.*

6.4 Output Code

For ease of explanation, Figure 4 shows the generated serial Fortran code optimized only for Fortran's column-major array access ordering. Figure 5 shows generated serial C code. The procedure summary information induces both the bounds on the spatial variables, and the concrete data structures. Array storage is allocated only for needed space.

Procedures `move`, `estimate_risk` and `extract` operate on "slices" of the `z`-component of the iteration space: `move` and `estimate_risk` at the surface, and `extract` at the lowest level. Hence the generated conditional execution of these procedures contingent on `z`.

The conditional execution of `flow` contingent on `z` enforces `flow` to execute within bounds. The conditional execution of `move` and `estimate_risk` contingent on `x` and `y` are for similar reasons. `rain` is the only procedure which has unconditional spatial execution.

The procedure summaries contain reduction statements for variables `ducks` and `water`. The variables `water` and `ducks` both require that two states are saved for the simulation: the current time, and one time-step back. The generated code calls an $\mathcal{E}m$-lib function to initialize the reductions for `ducks` and `water` in the next time step. Because they are declared as sum reductions, the initialization values are integer zero. Initialization for reduction is applied to these two variables at the end of the spacial loop nest iteration. Local variables which are required are denoted `zem`, concatenated with a number.

```
program wetland

integer time
integer x
integer y
integer z
integer geography(100,400,10)
integer water(100,400,10,2)
integer pumping(100,400,1,1)
integer ducks(100,400,2,1)
real risk_of_fire(100,400,1,1)
integer zem1

call initialize(geography,100,400,10,1)
call initialize(water,100,400,10,1)
call initialize(pumping,100,400,1,1)
call initialize(ducks,100,400,1,1)

do time = 1,20000
 do z = 1,10
  do y = 1, 400
   do x = 1, 100
    if (z .gt. 1) then
     call flow(water, geography, water, x, y, z, time)
    endif
    if (z .eq. 10) then
     call extract(water, pumping, water, x, y, z, time)
    endif
    if (z.eq.1 .and. x.gt.1 .and. x.lt.100 .and.
&       y.gt.1 .and. y.lt.100) then
     call move(ducks, water, ducks, x, y, z, time)
    endif
    if (z .eq. 1) then
     call estimate_risk(water, risk_of_fire, x, y, z, time)
    endif
    rain(water, x, y, z, time)
   enddo
  enddo
 enddo
 zem1=emcurtime(time)
 do x = 1,100
   call emwrite(water,x,1,1,zem1,0)
 enddo
 do y=1,400
  do x=1,100
   call emwrite(ducks,x,y,1,zem1,0)
  enddo
 enddo
enddo

call save_results(ducks,100,400,1,1)
call save_results(risk_of_fire,100,400,1,1)
end
```

Figure 4: Serial Fortran Code Generated by $\mathcal{E}m$ for Weland Example

```
#include "emlibc.h"
#include "wetland.h"
#include <malloc.h>

int     time, x, y, z;
int     *geography;
int     *water;
int     *pumping;
int     *ducks;
float   *risk_of_fire;
int     zem1;

main() {

   geography = (int*)malloc(100*400*10* sizeof(int));
   water = (int*)malloc(100*400*10*2* sizeof(int));
   pumping = (int*)malloc(100*400*1*1* sizeof(int));
   ducks = (int*)malloc(100*400*2*1* sizeof(int));
   risk_of_fire = (float*)malloc(100*400*1*1* sizeof(float));

   initialize(geography,100,400,10,1);
   initialize(water,100,400,10,1);
   initialize(pumping,100,400,10,1);
   initialize(ducks,100,400,1,1);
   initialize(risk_of_fire,100,400,1,1);

   for (time=0; time < 20000; time++) {
      for (x=0; x < 100; x++) {
         for (y=0; y < 400; y++) {
            for (z=0; z <10; z++) {
               if (z > 0) flow(water,geography,water,x,y,z,time);
               if (z == 9) extract(water,pumping,water,x,y,z,time);
               if (z==0 && x>0 && x < 99 && y>0 && y < 99) {
                  move(ducks,water,ducks,x,y,z,time);
               }
               if (z==0) estimate_risk(water,risk_of_fire,x,y,z,time);
               rain(water,x,y,z,time);
            }
         }
      }
      zem1 = emcurtime(time);
      for (x=0; x<100; x++) {
         emwrite(water,x,1,1,zem1,0);
      }
      for (x=0; x<100; x++) {
         for (y=0; y<400; y++) {
            emwrite(ducks,x,y,1,zem1,0);
         }
      }
   }
   save_results(ducks,100,400,1,1);
   save_results(risk_of_fire,100,400,1,1);
}
```

Figure 5: Serial C Code Generated by $\mathcal{E}m$ for Weland Example

7 Design Issues

- $\mathcal{E}m$ is a highly implicit system which relies heavily on automatic parallelization, mapping, and resource allocation in the compiler.

  This approach is known to generate non-optimal code in many cases. The design of $\mathcal{E}m$ restricts the problem domain and requires that much program analysis be performed by the implementer of the intrinsic library. The performance of the compiler on the remaining regular program structure will be close to that of hand-coded programs. For example, parallelizing compilers do not generate code to do red-black tiling communication [FJL+88], but due to the restricted computational problem domain, this type of communication pattern could be generated by $\mathcal{E}m$, thereby increasing performance.

- The large body of domain-specific code in the DS-lib may not be portable to other architectures.

  While programs written in the $\mathcal{E}m$ language are architecture-independent, porting the library may be a daunting task. Since the library is implemented in a standard language, the portability of the library should be no less than the portability of those languages.

- The $\mathcal{E}m$ programming language is designed as a restriction on imperative programming languages.

  Goto statements, concealable side effects, and procedures as parameters are not allowed. The result is that $\mathcal{E}m$ is a constrained language, which greatly simplifies program analysis and enhances the ability to perform program reasoning. As will be discussed in Section 6, the most restrictive aspect of functional programming is the treatment of aggregate data structures such as arrays. In order to eliminate program dependences wherever possible, the model of storage update must be constrained. These constraints have been weakened as far as is possible without introducing program dependences. as is detailed in Section 8.3.

  The minimization of program dependences allows $\mathcal{E}m$ programs to be mapped to a wide variety of sequential and parallel architectures, without the need for program rewriting. The strongest storage update model is write-once which allows synchronization to be combined with reading and writing, and eliminates resource dependences. Weaker models preserve this desirable property while providing a more convenient programming model.

- $\mathcal{E}m$ is a new programming language, and most new languages have a problem with user acceptance.

  Mulitprocessor architectures are becoming increasingly complex, and so too is the programming of these archtitectures with explicit message passing languages. If current trends continue, multiprocessing environments will become

commonplace, and some form of high level programming environment will be essential. The form of this environment is not yet established. Compilers for sequential languages to parallel architectures do exist, but compilers cannot do everything: programmers must help. Such compilers are used for "dusty decks", but as the field matures, more and more code will be written, and rewritten, in the newer languages.

$\mathcal{E}m$ will meet with acceptance provided there are benefits to its use. $\mathcal{E}m$'s compiler optimization is greatly facilitated because of having a restricted language and a library with procedure summaries: most compilers do not perform interprocedural analysis, not even for sequential machines. Interprocedural optimizations that are missed by such compilers are handled in $\mathcal{E}m$. This, coupled with using a restricted language to achieve automatic scheduling will provide the base on which to rate performance.

8 Ecosystem Modelling

In this section, the computational problem domain and the programming models used to implement these problems are discussed. Based on the programming models, the required memory models supported by $\mathcal{E}m$are presented.

Ecosystem models typically specify the simulation space as a 2- or 3-dimensional space in time. Each cell in the space has local, internal attributes, such as root density, tree type, animal species. The entire space may also have global, external factors, such as oxygen, sunlight, rain, climate, pollutants. External factors (also called forcing functions, or exogenous variables) usually drive the simulation, but are not affected by the simulation. Models may either be theoretical, or based on geographical areas.

Plant growth models are distinguished by the fact that the cell's attributes are fixed in their location throughout the simulation. That is to say, trees grow and die, but do not move to another cell; tree growth, by definition, never extends beyond a cell boundary. This does not preclude the introduction of seeds for growth: in this case, seeds are disbursed over the terrain, yet once planted, they remain fixed.

Population studies may describe migration, reproduction, and/or competition between species in a particular region. Typically the space is sparsely populated with the species, usually one per cell. These models are distinguished by the fact that migration of entities across cell boundaries occurs. Population models may or may not involve external factors. For models which do not involve external factors, only those cells which are occupied require simulation. Models which incorporate both are more complex, and require all cells to be updated.

The calculation performed in each cell, then, is a function of the local and/or global

attributes to which it has been assigned, and whether or not entities are densely or sparsely distributed across the space.

In general the models have synchronous time steps, and each cell is updated at the current state based on attributes (variables) contained in a defined neighborhood, from previous states.

8.1 Ecosystem Paradigms

System Dynamics modelling is used extensively in ecosystem modelling [HJ77] and may be considered a simplified version of differential equation modelling. The change of each state variable in a cell over time is determined by the input and output flows. The change of a state variable in spacial simulations is determined for a cell by flows between the cell and its neighboring cells. The rate of transfer of each flow is determined by an equation, which expresses the rate of flow as a function of the values in neighboring cells.

Ecological Field Theory (EFT) [WSPW89] is an extension of classic neighborhood models. In the models just discussed, each cell reads data from neighborhood cells, writes data only in it's own location. EFT models spatial influences between cells as *influence domains*: a cell calculates it's influence(s) on each neighboring cell and updates each neighbor's influence domains. In general the spatial composition of the influence zone changes dynamically, e.g., with the growth or death of a plant. In turn, plant growth calculated by a cell is a function of state variables within its neighborhood, and of the accumulated influences written by neighboring cells.

Ecologists have begun to formalize theories [AN90] using the formal specification language **z**. The reasons stated for using such formalism are ease and clarity of description, and to provide a mathematical basis for comparison between theories.

8.2 Programming Models

Programming models are used to implement ecosystem paradigms. In general, ecosystem models solve recurrence equations. The programming model that is selected to solve these equations is determined by the programmer. Some choices of programming model are better than others; this is a matter of software design and not relevant to this discussion.

The $\mathcal{E}m$ language supports three programming models: declarative cellular, imperative cellular, and influence. These programming models appear sufficient to handle the types of spatial simulations required for ecosystems.

A cellular automaton is a simple programming paradigm. For each time t, each cell computes the same data, which is based upon simple rules involving variables in neighboring cells at time t-1. Cellular automata are inherently parallel. The simplest cellular automaton has only one variable to update.

$\mathcal{E}m$ divides cellular automata into two classes: *declarative cellular* and *imperative cellular*. *Declarative cellular* computes a value for a variable only once in the cell at time t. *Imperative cellular* computes an accumulated value for a variable in each cell at time t. In other words, imperative cellular has successive *updates* to a variable in each cell at time t. Cellang [Eck92] is an example of a system which supports a imperative cellular automata.

A declarative cellular programming model exhibits the most data independence and is inherently functional in nature. A variable value may be computed as soon as all input values become available. Thus the ordering of statements in the inner loop is unimportant because each variable is written at most once.

An imperative cellular model is restricted in that the order of variable update, especially between different procedure calls, cannot be assumed to be commutative. Still, this places a restriction only upon the current loop iteration, and keeps the important property of no order between iterations due to reuse.

The EFT paradigm may be supported with an *influence* model. In an influence model, the current cell at (x, y, z, t) writes to variables outside it's current location. This paradigm is inherently different from cellular because these are writes to variables outside the current cell location.

8.3 Memory Models

Memory Model	*Programming Model*
write-once	declarative cellular
locally imperative	imperative cellular
reduction	influence

Figure 6: Correspondence Between Memory Models and Programming Models

There is a memory, or aggregate, model in $\mathcal{E}m$ which supports each of the programming models.

8.3.1 Write-Once Memory

In the *write-once* memory model, a memory location is written at most once. After the memory location is written, any number of reads my be performed. This type of memory model offers the most amount of parallelism [ANP89]. The only type of data dependence which can occur is flow dependence. Write-once memory supports a declarative cellular programming model.

8.3.2 Locally Imperative

The locally imperative memory model is a relaxation of write-once memory. Consider the following assignment statement, which might appear in a $\mathcal{E}m$ library procedure. The state variable Q is a formal parameter of the procedure.

$$Q(x, y, z, t) = Q(x, y, z, t) + (flow1 + flow2 + \cdots + flowk)$$

This assignment is a simple example of a difference equation. It is assumed that the *flows* in this equation are not array values.

If the memory model were to adhere to the strictly write-once paradigm, this type of assignment statement would require not only a new formal parameter name on the left-hand side of the assignment statement, e.g.,

$$R(x, y, z, t) = Q(x, y, z, t) + (flow1 + flow2 + \cdots + flowk)$$

but also a "new" variable name (as an actual parameter of the procedure) to be the target of the assignment. Such an assignment maintains referential transparency.

The advantage of allowing only the write-once model is that the $\mathcal{E}m$ program is declarative: the textual sequence of statements is irrelevant to state variable update. The disadvantage is variable name explosion.

At the end of each iteration, all temporary variable assignments must be assigned to the permanent state variable. Variable renaming can eliminate imperative cellular references. However, the cost to the user, presumably an ecologist, with this type of restriction is: the need to create several temporary variables; and, the need to name the temporary variables in such a way that the compiler generates the correct update sequence. It was considered that this programming style is too burdensome, even with the potentially increased parallelism. This especially becomes apparent when temporary variables are needed for several state variables.

By permitting locally imperative assignments, i.e., assignment updates which are local to the cell, potential parallelism between iterations is not sacrificed. The main cost in this case is declarative style: locally imperative variable updates requires that the compiler use the textual order of procedure invocation.

The end result, from the viewpoint of compilation, is that the order of variable update for write-once is determined by the compiler. For locally imperative the order of variable update is the same order as the textual order. In one case, the programmer defines it indirectly through temporary variable use, and in the other case, the programmer specifies it directly through textual order.

It should be stressed that $\mathcal{E}m$ does not require locally imperative update over write-once. Locally imperative update issupported to providea more convenient programming model. For example, the wetland procedure summaries, Figure 3, are written to be general: the input and output formal parameter names are different. However, the $\mathcal{E}m$ wetland code uses the same names for the for the actual parameters. Another programmer may decide to use these procedures more functionally.

8.3.3 Reduction

A reduction operator applies a sum or product operator to elements of an array, generating a scalar value. On machines which do not support reduction, this is implemented in a loop, with imperative update to a scalar variable. An example is the sum of an array of values:

```
SUM = 0
DO I = 1, N
   SUM = SUM + A(I)
ENDDO
```

Arithmetic operators may or may not be associative in the application; non-arithmetic functions, such as MAX, are associative.

$\mathcal{E}m$ supports declaration of reductions in the procedure summary. If a variable is declared as a reduction, $\mathcal{E}m$ assumes that the operation is associative. By declaring reduction, $\mathcal{E}m$ can implement the reduction efficiently, and it is not necessary to save all the data before the reduction, because the $\mathcal{E}m$ compiler has access to write-many memory. Use of the reduction operator allows for more parallelism because procedure invocation is order independent. The influence model is best supported using reduction.

9 Conclusions

$\mathcal{E}m$ is distinctive in that it incorporates a restricted language and a library with procedure summaries in order to achieve automatic scheduling with high parallel performance. Related research in language development and compilation techniques will now be addressed.

Several parallel programming languages have been proposed to support general purpose parallel programming. OCCAM [Cok91] is designed for the transputer. Languages such as Fortran [Thi89, CFR+92, CMZ92] and C [RS87], have been extended, include new compiler directives, and have library routines to manage processes and communication. Typically these languages are difficult to analyze. Pointer code in C makes analysis difficult. In both languages code may be written so that subscript analysis is difficult, if not impossible, for the compiler. Equivalence statements and common blocks in Fortran add to the difficulty of interprocedural analysis. Because of the inherent difficulty, interprocedural analysis typically is not performed.

Languages like $\mathcal{E}m$ simplify both compiler transformations needed to extract parallelism and interprocedural analysis because of value-result semantics for procedure parameters and disallowal of access to non-local variables [MVR85, HC88]. An additional benefit of $\mathcal{E}m$ is the existence of procedure summaries for interprocedural analysis, because only the interface to the procedures requires analysis.

With the exception of FIDIL [HC88] and Cellang [Eck92], $\mathcal{E}m$ appears to be the only language which is domain-specific. FIDIL has a strongly applicative style and requires many users to learn a new style of programming. $\mathcal{E}m$ is designed for users accustomed to programming imperative languages. Cellang is a declarative and easy to learn language, but is limited to integer data and lacks procedure invocation.

Languages such as BLAZE [MVR85] for shared memory machines, and Kali [MVR90], Vienna Fortran [CMZ92], DINO [RSW90], Booster [PvGS90], Fortran D [FHK+91], Fortran 90D [WF91], Adaptor [Bra93], SUPERB [ZBG88] for distributed memory machines, permit the user to specify the distribution and alignment of data. Adaptor has a back end which maps to several communication packages. SUPERB maps to SIMD and MIMD machines.

Crystal [Che86] and ASPAR [IFKF90] perform automatic data distribution based on symbolic pattern matching. Crystal requires a loop nest similar to $\mathcal{E}m$, and more than one sequential loop may be defined. Neither performs interprocedural analysis.

Ecological modelling is the application area used herein to present the features of $\mathcal{E}m$. As stated before, $\mathcal{E}m$ is not restricted to this domain. Any statically decomposable domain which requires a sequentially iterated parallel loop nest can be modelled using this system. Cellular automata, popularized as the game of Life [Gar70], are used to model physical systems [Eck92, BH92, Dew84]. Other applications include

solving partial differential equations using the finite difference method, including relaxation [FJL+88, HA90].

The ease of developing models using $\mathcal{E}m$ as a programming language is due largely to its declarative nature. Consider, for example, the automatic deduction of array bounds. Suppose a set of domain-specific procedures induces a certain array bound. Should the programmer modify the program to incorporate an additional (or merely modified) set of procedures which access different portions of the array, the compiler automatically adjusts the array declaration, thereby eliminating the need for the programmer to modify the program.

$\mathcal{E}m$ also has several characteristics of literate programming [Knu84, Ben86], a style of programming which promotes readability and comprehension. Programming details are removed in $\mathcal{E}m$, and variable declarations alone specify their semantic meaning in the model. Modularity is supported through the use of procedures to perform all computations. The standardized procedure interface supports model sharing. Because of the declarative nature of $\mathcal{E}m$, it is easily adaptable to graphical programming.

A further consideration for $\mathcal{E}m$ is it's input/output format. It is desriable to have the format for both to be identical [Eck92] . This would allow models to be pipelined. For ecosystems, however, the existence of various GIS databases does not directly support an identical format. It is an area which should be addressed in the future.

A forthcoming report will discuss compilation of $\mathcal{E}m$ programs to a MIMD environment. There, the data partitioning and scheduling strategy is discussed, along with the use of PVM [GBD+93] as the parallel communication package.

References

[AN90] D. E. Abel and B. S. Niven. Application of a formal specification language to animal ecology. I. environment. *Ecological Modelling*, 50:205–212, 1990.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[Bel] Ronald Beloin. *personal communication.* Boyce Thompson Institute, Cornell University, Ithaca NY.

[Ben86] Jon Bentley. Literate programming. *Communications of the ACM*, 29(5):364–369, May 1986.

[BH92] Per Brinch Hansen. Parallel cellular automata: A model program for computational science. Technical Report SU-CIS-92-18, Syracuse University, September 1992.

[Bra93] Thomas Brandes. Compiling data parallel programs to message passing programs for massively parallel MIMD systems. In *Programming Models for Massively Parallel Computers.* IEEE Computer Science Press, Berlin, September 1993.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, Atlanta, Georgia, June 22–24, 1988. Published as ACM SIGPLAN Notices 23(7).

[CFD90] David C. Cann, John T. Feo, and Thomas M. DeBoni. SISAL 1.2: High-performance applicative computing. Technical Report UCRL-JC-103980, Lawrence Livermore National Laboratory, May 1990.

[CFR+92] Alok Choudhary, Geoffrey Fox, Sanjay Ranka, Seema Hiranadani, Ken Kennedy, Charles Koelbel, and Chau-Wen Tseng. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. Technical Report SCCS-251 (CRPC Report CRPC-TR92203), Syracuse Center for Computational Science, Syracuse University, March 1992.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[Che86] Marina C. Chen. A parallel language and its compilation to multiprocessor machines for VLSI. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 131–139, St. Petersburg Beach, Florida, January 13–15, 1986.

[CMZ92] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. Technical Report ICASE 92-9, NASA Langley Research Center, Hampton, VA, March 1992.

[Cok91] R.Š. Cok. *Parallel Programs for the Transputer*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Dew84] A. K. Dewdney. Computer recreations. *Scientific American*, 251(6):14–22, December 1984.

[Eck92] J. Dana Eckart. A cellular automata simulation system. *ACM SIGPLAN Notices*, 27(8):80–106, August 1992.

[FHK+91] Geoffrey Fox, Seema Hiranadani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report SCCS-42c (Rice COMP TR90-141), Syracuse Center for Computational Science, Syracuse University (Center for Research on Parallel Computation, Rice University), April 1991.

[FJL+88] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors, Volume I.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Gar70] Martin Gardner. Mathematical games. *Scientific American*, 223(10):120–123, April 1970.

[GBD+93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[HA90] David E. Hudak and Santosh G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 187–200, Amsterdam, The Netherlands, June 11–15, 1990. Published as ACM SIGARCH Computer Architecture News 18(3).

[HC88] Paul N. Hilfinger and Phillip Colella. FIDIL: A language for scientific computing. Technical Report UCRL-98057, Lawrence Livermore National Laboratory, January 1988.

[Hil85] W. D. Hillis. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

[HJ77] Charles A. S. Hall and John W. Day Jr. *Ecosystem Modeling in Theory and Practice: An Introduction with Case Histories*. John Wiley and Sons, New York, NY, 1977.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the Fifth Distributed Memory Conputing Converence*, pages 1105–1114, Charleston SC, April 1990.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, May 1984.

[MVR85] Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. Technical Report ICASE 85-29, NASA Langley Research Center, Hampton, VA, May 1985.

[MVR90] Piyush Mehrotra and John Van Rosendale. Programming distributed memory architectures using Kali. Technical Report ICASE 90-69, NASA Langley Research Center, Hampton, VA, October 1990.

[NA89] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, Jerusalem, Israel, May 28–June 1, 1989.

[Nik90] Rishiyur S. Nikhil. Id version 90.0 reference manual. Technical Report CSG Memo 284-1, M.I.T. Laboratory for Computer Science, July 1990. Revised September 1990.

[Pug92] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[PvGS90] E. Paalvast, A. van Gemund, and H. Sips. A method of parallel program generation with an application to booster language. In *Proceedings of the 1990 International Conference on Supercomputing*, page ??, Amsterdam, The Netherlands, June 11–15, 1990. Published as ACM SIGARCH Computer Architecture News 18(3).

[RS87] J. Rose and G. Steele. C*: an extended C language for data parallel programming. Technical Report PL87-5, Thinking Machines, Inc., 1987.

[RSW90] M. Rosing, R.W̃. Schnabel, and R.P̃. Weaver. The dino parallel programming language. Technical Report CU-CS-457-90, University of Colorado, Boulder, CO, April 1990.

[Thi89] Thinking Machines, Inc., Cambridge, MA. *CM Fortran Reference Manual*, version 5.2 edition, 1989.

[VB90] A. H. Veen and R. van den Born. The RC Compiler for the DTN Dataflow Computer. *Journal of Parallel and Distributed Computing*, 10:319–332, 1990.

[WF91] Min-You Wu and Geoffrey C. Fox. Fortran 90D compiler for distributed memory MIMD parallel computers. Technical Report SCCS-88b, Syracuse Center for Computational Science, Syracuse University, 1991.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, Boston, MA, 1989.

[WSPW89] J. Walker, P. J. H. Sharpe, L. K. Penridge, and H. Wu. Ecological Field Theory: the concept and field tests. *Vegetatio*, 83:81–95, 1989.

[ZBG88] H. P. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, NY, 1991.