RIACS

Research Institute for Advanced Computer Science
NASA Ames Research Center

THE PARADIGM COMPILER; MAPPING A FUNCTIONAL LANGUAGE FOR THE CONNECTION MACHINE

JACK B. DENNIS

April, 1989

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.15

NASA Cooperative Agreement Number NCC 2-387

THE PARADIGM COMPILER: MAPPING A FUNCTIONAL LANGUAGE FOR THE CONNECTION MACHINE

JACK B. DENNIS
Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, California 94035, USA

Abstract

The Paradigm Compiler implements a new approach to compiling programs written in high level languages for execution on highly parallel computers. The general approach is to identify the principal data structures constructed by the program and to map these structures onto the processing elements of the target machine. The mapping is chosen to maximize performance as determined through compile time global analysis of the source program. In the work reported here, the source language is Sisal, a functional language designed for scientific computations, and the target language is Paris, the published low-level interface to the Connection Machine. The data structures considered are multidimensional arrays whose dimensions are known at compile time. Computations that build such arrays usually offer opportunities for highly parallel execution; they are *data parallel*. The Connection Machine is an attractive target for these computations, and the parallel `for` construct of the Sisal Language is a convenient high level notation for data parallel algorithms. The paper discusses the principles and organization of the Paradigm compiler.

Keywords: Connection Machine, Functional Programming, Sisal, Compiler, Mapping Data Structures

Introduction

The advent of highly parallel computers presents new challenges for the designers of compilers to support high performance scientific computation. Current programming methodology for parallel computers usually requires the explicit use of message-passing commands or other synchronizing commands by the author of the high level language program. This methodology yields source programs that are needlessly large, often involve much overhead in their execution, and are relatively difficult to make correct and maintain. An important goal toward making parallel computers more useable for practical computations is to provide compiling technology that is able to convert algorithms expressed directly and simply in a high level language into efficient programs for parallel

computers. The parallelism is implicit in the expression of the algorithm and must be identified and exploited by the compiler.

The Paradigm Compiler implements a new approach to compiling programs for highly parallel computers. Its operation is based on identifying the principal data structures constructed in the course of a computation through global compile-time anzlysis, and mapping these structures onto the processing elements of the target machine[1,2]. For the work reported here the source language is Sisal[3], a functional programming language designed for scientific computation, and the target language is Paris, the published low-level interface to the Connection Machine[4].

The Sisal language and the Connection Machine[5,6] form a particularly attractive combination for evaluating this approach. The absense of global variables and the clear differentiation of arguments and results of function modules make it easy for a compiler to analyze source programs and identify the parts of the code that define the major data structures of the computation. We call these parts of the source language program *code blocks*.

The data structures appropriate for scientific computaion on the Connection Machine are large multi-dimensional arrays of numerical data. Each code block of a Sisal Program represents a computation that may be spread over the processing elements of the machine according to a chosen assignment (or mapping) of data structure elements to processing elements of the Connection Machine. This is the essence of *data parallel* computation[7,8]. The parallel `for` expression of the Sisal language provides a convenient high level notation for writing data parallel algorithms.

The Paradigm compiler will compile programs written in the Sisal functional programming language into the Paris code for the Connection Machine. Because the parallel `for` construct of Sisal is a natural match to the capabilities of the Connection Machine, the design of the Paradigm Compiler emphasizes efficient treatment of nested parallel `for` expressions in Sisal programs. This report presents an overview of the Paradigm Compiler.

Our goal and approach require some departure from usual practice in the structure of programming language support systems. Efficient machine code programs can be generated only if the compiler is able to consider the entire collection of program modules involved in a job in making decisions regarding how the computation should be mapped onto the target machine. This implies that the linking of program modules should be accomplished prior to the compiler's analysis and optimization decisions. A second change is more fundamental: instead of carrying out optimization as a sequence of independent steps, each of which supposedly leads to an "improvement" of the code, we perform an analysis of the given code, determine the best mapping strategy, then synthesize machine code according to the specified mapping.

Our interest in basing code synthesis on the data structures of the computation is reinforced by the experience of users of hypercube multiprocessors. Many papers have now been published in which users of large-scale multiprocessor computers have successfully exploited parallelism through manual analysis of the data structures involved in a computation[9]. Our long term goal is to automate this process and to incorporate it into compilers for parallel computers. The reported work even suggests that such nonnumeric computations as combinatorial search problems can be handled efficiently by

certain paradigms of parallel processing[10]. Eventually it may be possible for an intelligent compiler to recognize these patterns and construct efficient machine code.

1. The Source Language

For the Paradigm Project the source language is Sisal, a functional programming language developed at the Lawrence Livermore Laboratory for use in high performance scientific applications[3]. The design of Sisal derives from the functional language Val developed by the Computation Structures Group of the MIT Laboratory for Computer Science[11]. A Sisal compiler has been developed at the Livermore Laboratory which converts programs into an intermediate form called IF1[12]. This compiler serves as the front end for the Paradigm Compiler.

Functional programming languages are attractive for parallel programming for two main reasons: First, the parallelism is implicit, eliminating any need for process sychronizing facilities, and providing a guarantee of determinate execution. This makes programs written in functional languages more compact than parallel programs written in conventional languages with features for explicit parallel programming. Second, compile time analysis of functional language programs is easier than for conventional languages because there are no side effects: each use of any data definition is readily identified.

The parallel `for` expression. The most important feature of Sisal for expressing data parallel computation is the parallel `for` expression. This form may be used to define an array value of arbitrary rank. For example, the following expression defines a vector value Z, each element of which is the average of the two adjacent elements of a given vector X.

```
Z: array [ real ] :=
   for i in 1, n
      Y :=
         if i = 1|i = n
         then X[i]
         else 0.5 * ( X[i-1] + X[i+1] )
         end if
   returns array of Y
   end for
```

In this example the index variable `i` is specified to range over the domain `[1, n]`. The expression as a whole defines a one-dimensional array having an element for each value of the the index `i`. The body is a conditional expression having two arms, one that gives the end elements (boundary conditions) and one that gives the rule for computing the internal elements.

Nested parallel `for` expressions may be used to define arrays of two dimensions (matrices) or higher. Our experience with typical programs for large-scale scientific computation indicates that most loops in these programs can easily be expressed as parallel `for` expressions in Sisal. The Paradigm Compiler is designed to construct

efficient Connection Machine code from array definitons (code blocks) written as parallel `for` expressions.

Source language limitations. A typical numerical computation consists of a main loop, the body of which includes several code blocks that define array values. The code blocks form an acyclic graph in which each link represents a producer/consumer relationship for array values passed between two code blocks. This observation suggests the following strategy for the initial version of the Paradigm Compiler: efficient Paris code will be constructed for Sisal programs that conform to the following general structure. All (multidimensional) array values are defined by nested parallel `for` expressions; these parallel `for` code blocks may occur within an outer program structure composed of arbitrary conditional and iteration expressions. However, the program may not contain any array definitions other than the parallel `for` expressions. In particular, Sisal iterations of the form

```
for init
....
....
returns value of ...
end for
```

may be used, but those with `array of` instead of `value of` may not be used. The array fill operation

```
array_fill (a, b, v)
```

is treated as a short hand notation for

```
for i in a, b
returns array of v
end for
```

The following features of Sisal are not supported:

1. Records and Streams: These types are not essential to numerical computation. We expect that a future version of the compiler will support Sisal records, but the extension to stream types is not envisioned at present.

2. Complex numbers: The Sisal front end converts the complex data type into a record type in IF1, which is not supported by Version 1 of Paradigm. Users should represent complex numbers by two-element arrays.

3. Arrays: The only array constructor supported is the parallel `for` expression. Other operations of Sisal that define new array values (the Replace and Catenate operations, and operations `array_adjust`, `array_addh`, `array_addl`, `array_remh`, and `array_reml`) are not supported. The dimensions of arrays (the

index ranges of parallel `for` expressions) must be known at compile time (but may be defined after module linking).

4. Parallel `for` expression: Only the cross product form of the parallel `for` expression is supported, not the dot product form. Each parallel `for` expression must have just one result value. The concatenate reduction opeator is not supported.

5. Union Types: Union types and the case expression are not supported in Version 1 of Paradigm.

6. Basic data types: The types **null** and **character** are not supported.

7. Function Application: All function definitions for a computation job must be in a single source program file. Recursive application of functions is not supported.

The bodies of the parallel `for` code blocks will be executed on the CM virtual processors. The initial version of Paradigm will not support array definitions within the bodies of these code blocks. We hope to remove this restriction in later versions of the compiler.

2. Structure of the Paris Code

The output of the Paradigm Compiler is a text file of C language code with embeded Connection Machine (CM) instructions in the form of Paris macro invocations. All source code outside the parallel `for` code blocks is converted into normal C language code. Variables and intermediate values defined by operations within the bodies of parallel `for` code blocks are represented by CM variables allocated in a virual processor set appropriate to the index domain of the code block in which they occur. The bodies of parallel `for` code blocks are implemented on the virtual processors.

Static allocation of CM memory is used in Version 1 of the Paradigm Compiler. This means that all CM variables for all parallel `for` code blocks of the computaton are allocated to CM memory locations at the begining of program execution, and the allocation is fixed throughout program execution. A future version of Paradigm will utilize the dynamic memory allocation facilities provide by the Paris interface.

The structure of the Paris code generated by the Paradigm Compiler is determined once mappings of arrays defined by parallel code blocks have been fixed. For each array, most attributes of the mapping are determined by the rank and index ranges of the corresponding Sisal parallel `for` expression. In particular the rank of the CM geometry is fixed and the number of address bits assigned to each axis is determined by the specified subscript ranges. This fixes the size of the virtual processor set. The remaining information required to determine a geometry is the assignment of coordinate address bits to CM "send address" bits, that is, the bit masks for each axis. The choice of this assignment will have a significant effect on performance in many cases. In addition, the choice of offset value (difference) between a Sisal subscript value and the corresponding CM coordinate index may influence performance significantly. The choice of these attributes of data structure mapping is not made by the Paradigm Compiler but is left to the user. The Specify Mapping module provides the user with means to indicate these choices to the compiler. We hope to automate this process in the future.

It is intended that the compiler choose the best Paris implementation of references to array variables possible. The CM `send_to_news` and `get_from_news` instructions are used to implement these references whenever array element selections have the form `[i + a]` where `i` is a subscript variable of a parallel `for` expression and `a` is a constant. There are two cases depending on whether the a is known at compile time or is fixed only immediately prior to execution of the nested parallel `for` expression. In the first case the compiler can determine the necessary sequence of `CM_news` instructions before program execution. In the second case, Paradigm must produce C code that selects the appropriate `CM_news` instructions during program exectution. This will probably result in slower execution due to host computer speed limitations. For other forms of array element selection, the compiler will generate appropriate `CM_get` instructions to implement communication using the general router.

In the case of Sisal parallel `for` expressions that specify reduction operators Paradigm generates the result array using appropriate `CM_scan` instructions, or `CM_global` reduce operations if the result is a single value. If an argument array of a parllel `for` is of lower dimension that the defined array, then `CM_multi_spread` instructions are used to broadcast elements of the argument array to the virtual processor where they are used.

3. Structure of the Compiler

The top level structure of the compiler is shown in Fig. 1. The Front End performs syntax analysis and does static semantic checks on the Sisal function definitions in a source language file. It converts the definitions into dataflow graphs and creates data structures in the Program Description Tree (PDT) format to represent the hierarchy of dataflow graphs for the constituent expressions of Sisal function definitions. The Transform module of the compiler recognizes nests of parallel `for` expressions and expresses them as *partition* constructs in the PDT format. The Specify Mapping module of the compiler interacts with the user to create complete mapping specifications for the output array of each code block. The Code Constructor produces the Paris file from the PDT based on the mapping specifications supplied by the user. A Print PDT program is provided for examining programs in the PDT representation. An Analyze module is planned that will provide the user with information about the program to help with making the best choice of mapping. This information includes attributes of code blocks such as operation counts and critical path lengths. The Evaluate module will use the mapping specifications and the measures provided by the Analyze module to calculate an estimate of the running time of the job on the Connection Machine.

4. The Program Description Tree Format

The principal data structure used is the *Program Description Tree* (PDT), which is designed to serve the needs of the several parts of the compiler. The front end modules use the PDT as a common format for program modules processed and linked by the compiler. Not all components of the PDT data structure are used in all phases of the compilation process. By accepting this, the input and output of the Transform module can share one representation, leading to a degree of simplicity.

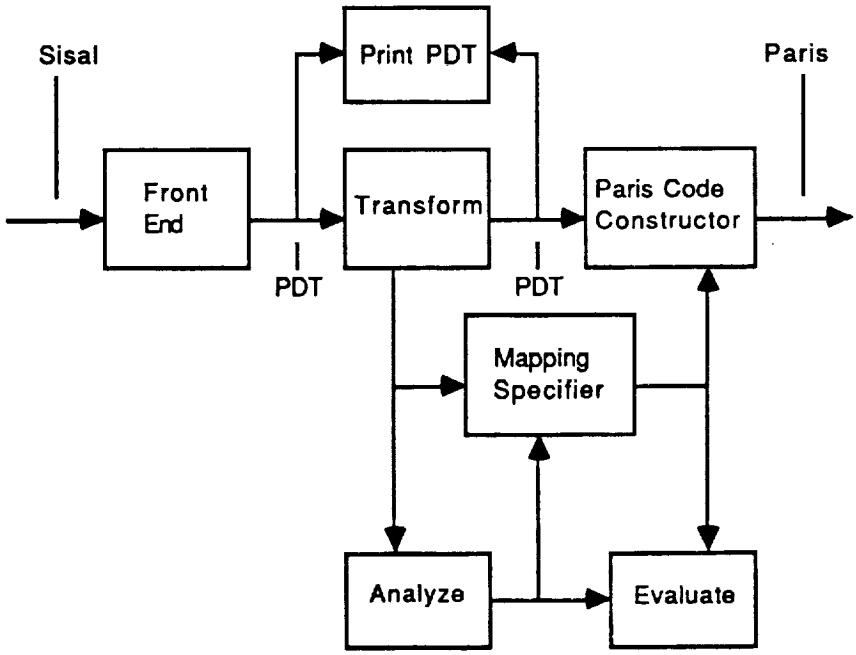

Figure 1. Top level structure of the Paradigm Compiler.

In the PDT format the several major constructs of the Sisal language are represented using graphs for their components. For example, the familiar **`if .. then .. else .. endif`** form is represented using two graphs, one for each alternative. At the outer level the PDT is a tree because its constituent graphs form a hierarchy. Nodes of the graphs represent both *simple nodes* and *compound nodes*. The compound nodes refer to PDT structures that represent major constructs such as conditional expressions, function applications or parallel **`for`** expressions.

To illustrate the PDT format let us consider how dataflow graphs are represented. The PDT for a graph is a data structure with its root being of the structure type

```
struct PD_Graph
{
   int node_cnt;
   int in_cnt;
   int out_cnt;
   int *out_sce_node;
   int *out_sce_port;
   struct PD_Vertex **vertex;
}
```

The graph as a whole contains a number of nodes given by `node_cnt`, and has `in_cnt` input ports and `out_cnt` output ports. As we shall see, each node of the graph may have any number of input and output ports. Connections in the graph are specified by giving the source of the link that terminates on each output port of the graph, and each input port of each node of the graph. The connections to the output ports of the graph are specified by the two tables of integers included in the `PD_Graph` structure as fields `out_sce_node` and `out_sce_port`.

The `vertex` field of the root structure points to a table of `PD_Vertex` structures that represent the nodes of the graph:

```
struct PD_Vertex
{
   enum PD_NodeKindTag kind;
   enum PD_Opcode opcode;
   union PD_NodeKind node;
}
```

The `kind` field distinguishes unary operators, binary operators, literal nodes, and compound nodes, and the various types of operators are further distinguished by the `opcode` field. The first two kinds of nodes represent most of the simple operators of the Sisal language. For compound nodes the `node` field refers to a structure:

```
struct PD_CompNode
{
   int in_cnt;
   int out_cnt;
   int *in_sce_node;
   int *out_sce_node;
   union PD_CompBody body
}
```

This gives the counts of input and output ports of the compound node and specifies the

sources of its input links. The body field points to one of the several possible compound node types, for example

```
struct PD_Forall
{
    int low;
    int high;
    enum PD_RedOp reduce;
    struct PD_Graph *body
}
```

This structure represents a parallel `for` expression in Sisal that defines a one-dimensional array. Parallel `for` expressions defining arrays of higher rank are represented as nests of compound nodes. Fields `low` and `high` of this structure specify which input ports of the compound node receive the start and finish values that define the index range The reduce field specifies the reduction operator to be used if the programmer has specified **value of** in the returns clause of the Sisal parallel `for` expression. The body field points to the `PD_Graph` structure that represents the body expression of the parallel **`for`**.

An important operation done by the Transform module of the compiler is to recognize nests of `PD_Forall` structures and reformulate them as *partition* nodes:

```
struct PD_Partition
{
    int dim_cnt;
    struct PD_ParamExp *low;
    struct PD_ParamExp *high;
    enum PD_RedOp *reduce;
    int arm_cnt;
    struct PD_ArmSpec **arms
}
```

In this structure, the `dim_cnt` field gives the depth of the nest of `PD_Forall` nodes from which it was formed. In the absence of reduction operators this is the rank of the array defined by the represented code block. The `low` and `high` fields point to structures that specify how the index ranges depend on input parameters of the complete Sisal program. The values of these expressions must be known before mapping choices are made. Futher aspects of partition nodes and the Transform module are discussed below.

In the Paradigm Compiler, a PDT presented to the Paris Code Generator will contain `PD_Partition` nodes, but no `PD_Forall` nodes.

5. The Sisal Front End

The components of the Sisal Front End are shown in Fig. 2. This arrangement has been chosen to make use of the Sisal parser/checker developed at the Lawrence Livermore Laboratory which translates source language programs into the IF1 intermediate format[12]. This compiler performs the parsing and compile time semantic checks, so its IF1 output file is free of syntactic errors and is type-correct. An IF1 file is loaded and formatted as a PDT structure by programs Load IF1 and Convert. The issues in the design of these programs concern the intricacies of the IF1 format. For example, the numbers of nodes and edges in a dataflow graph are not given explicitly in IF1, so these parameters must be determined for each graph by counting items in the input file before storage can be allocated for the tables used in the PDT format. The Load IF1 program is based on a version written by the dataflow research group at McGill University[13].

The Link module shown in Fig. 1 provides for combining Sisal function definitions contained in several separate files. Implementation of the linker requires routines that merge the data type definitions contained in the separate IF1 files produced by the Sisal Parser.

It is envisioned that future versions of the Paradigm Compiler will use a new parser/checker that produces the PDT format directly.

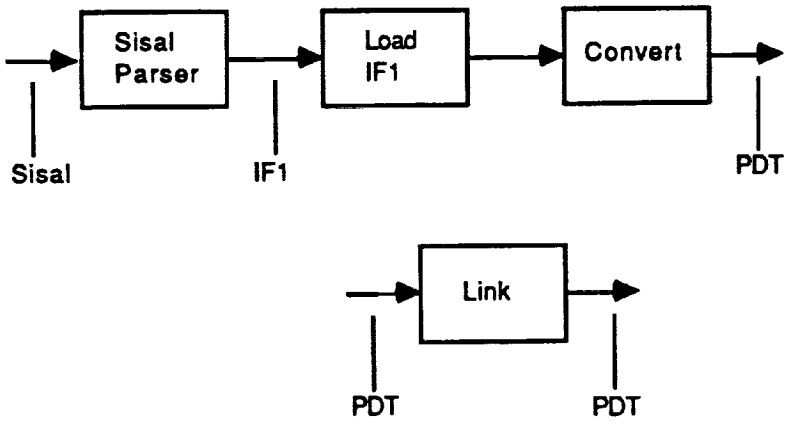

Figure 2. Structure of the Sisal Front End.

6. The Transform Module

In addition to combining nested `PD_Forall` nodes, the Transform module puts the body of the code block into a form better suited to the generation of efficient machine code for the Connection Machine or other highly parallel computers. Two transformations are performed where possible: Firstly, array element selection operations are separated from the arithmetic operations of the code block body. This permits the generation of code to access nonlocal data with knowledge of all access requirements of the code block.

Secondly, conditional expressions in the code block body are identified that perform simple tests on the index variables of the constituent `PD_Forall` constructs. If these tests define simple subarrays of the array defined by the code block, more efficient machine code is possible than a direct coding of the conditional expressions. Specifically, if such conditional expressions define rectangular subarrays, then the body of the node is represented by a set of `PD_ArmSpec` structures, each of which contains the PDT of the code to be executed for the union of a set of specified subarrays. The corresponding code for the Connection Machine will set the context flag for each arm by executing a series of tests on coordinate values in the code block geometry.

7. The Specify Mapping Module

The Specify Mapping compiler module interacts with a user to complete the definition of the mappings to virtual processors for each array defined by a code block of the program. (Note that mappings for any array-valued inputs of the program must also be completed by the user.) The initial version of Paradigm will be concerned only with the `PD_Partition` nodes of the PDT, which correspond to nests of parallel `for` expressions in the Sisal source program. Because we assume that each array element will be mapped to a unique virtual processor, the appropriate CM virtual processor configuration is determined by the dimensions of the array value defined by the code block. The rank of the geometry will equal the number of dimensions (depth of nesting) of the parallel `for`. Since we assume that the index lower/upper bounds for each dimension are manifest the choices open to the user are the following:

1. Specifiy an offset (for each dimension) of the index origin to position the index range within the 2^n length of the CM coordinate axis (where n is the smallest integer such that 2^n is at least as great as the index range).

2. Specify the "mask" for each axis that selects the virtual hypercube address bits to be used for accessing array elements in the corresponding dimension. These choices for the various code blocks will determine the consistency (or inconsistency) of the mappings for different code blocks, and hence the efficiency of communication that is possible. The mask bits must be mutually exclusive, and will usually account for all bits in the send address format for the code block's virtual processor configuration.

Another function of the Mapping Specifier program is to set the values of any job parameters that are undetermined in the outermost function definition of the PDT.

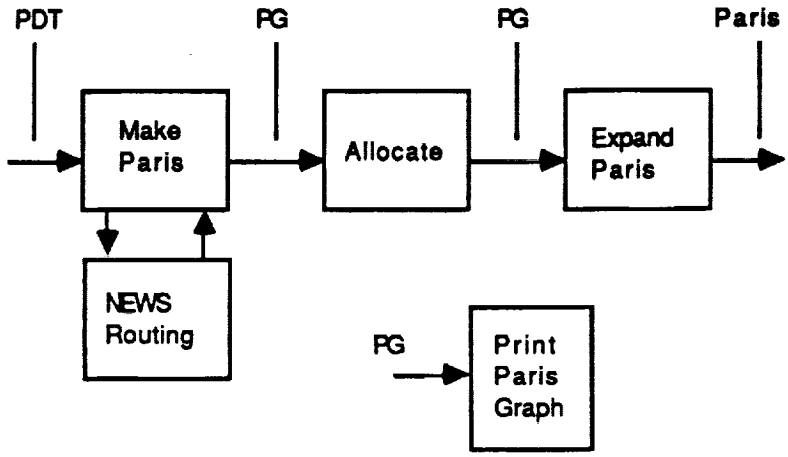

Figure 3. Structure of the Paris Code Constructor.

8. The Paris Code Constructor

The organization of the Paris Code Constructor is shown in figure three. These compiler modules start with a PDT and produce an ASCII file containing Paris macros embedded in C code for processing by the Connection Machine host computer. These modules are described in the following paragraphs. First we introduce the Paris Graph structure used by the code constructor.

The Paris Graph. The principal data structures used by the modules of the Paris Code Constructor are the PDT, and the Paris Graph (PG). The PG format is used to represent the instructions of the target Paris program, initially without commitment regarding storage allocation or sequencing. The format of a PG is similar to that used for dataflow graphs within a PDT. In the PG format, links represent data dependencies through the source/destination fields of Paris instructions.

For at least two reasons the Paris instructions are represented as values of an "enum" type rather than the character strings that will appear in the Paris output file: (1) Parts of the Code Constructor need to test for specific Paris instructions, which is most efficiently done by comparing integers; and (2) The character strings can be put in one memory area, which will lessen their impact on paging performance during compiler execution. The instruction names (codes) in the PG are "generic" names without the type-indicating prefixes or the postfix codes that indicate numbers of source/dest fields and field length specifiers. These are appended to the instruction names by the Expand Paris module.

The Make Paris Graph Module. The `PG_Graph` data structure is a hierarchical graph structure in which the root node is always associated with a `PD_Partition` code block of the PDT. The Paris Graph structure has as many root nodes as there are partition nodes in the description tree. The outer code (that not contained within any partition node) is represented in the Paris Graph structure by the corresponding parts of the PDT. The Paris Graph for a partition node contains Paris communication instructions generated from the specified array element selections, and Paris instructions for the PDT of each arm of the partition node. Information about the virtual processor set, geometries, and parallel variables for the partition code block are recorded in the Paris Graph format for use by the Allocate and Expand Paris modules of the Code Constructor. Make Paris Graph operates by making a traverse of the PDT, generating the Paris Graph for each partition node. To construct the nesessary `CM_get` instructions, the program determines the origin of each input array value of the code block and consults its mapping specifications. Connection Machine spread instructions are placed in the Paris Graph to implement broadcast requirements, and Connection Machine scan instructions are used to implement reduction operators specified in the PDT. Make Paris Graph translates the opcodes in the PDT format into coded Paris operation codes. This translation also provides type and size information for inclusion in the instruction nodes of the Paris Graph.

This program must be able to generate code for partition bodies that contain conditionals or loops (sequential or parallel). Conditionals will be supported in the initial version; code generation for inner loops will be implemented later.

NEWS Routing Module. This module is used by Make Paris to generate Paris communication instructions that implement efficient communication among CM virtual processors using the hypercube geometry of the Connection Machine. As of this report, the principles of operation for this module have been developed, but work on its implementation has not begun. The recent work of Levit[14] provides a basis for choosing optimal Paris code for the partition nodes of programs in PDT format.

Print Paris Graph. This program provides a listing of the hierarchical Paris Graph for testing purposes.

Paris Allocator. This program performs two functions: (1) Generate the Paris instructions required to allocate fields for all sources and destinations of instructions in the given Paris Graph; and (2) Place the instructions in a suitable order. (The order of instructions in the Paris code matters only if compact allocation is performed. An exception to this occurs with some Paris instructions for which non-overwriting forms are not provided.) This function is done by constructing tables of host and CM variables that are used by Expand Paris to generate variable nanes, C language declarations, and allocation statements for the C compiler of the Paris file.

Expand Paris. This program generates an ASSCI file of Paris code from the allocated hierarchical Paris Graph. The principal operations are: (1) generate the C language code and Paris instructions that set up virual processor sets and geometries, set up host computer variables, and provide for transmission of arguments and results of the Sisal program; (2) generate C code for the outer portion of the job (the portion of the PG_Graph that refers to PDT parts); and (3) translate instruction nodes of the Paris graph into their character string values.

Acknowledgements

Sisal is a functional programming language developed at the Lawrence Livermore Laboratory for use in high performance scientific applications[3]. We appreciate the availability of the Sisal parser/checker developed by the Livermore group because this has made our task considerably easier. The initial version of the Paradigm Compiler has been developed by Jack B. Dennis during his appointment as Visiting Scientist at RIACS from May 1988 through April 1989. The Load IF1 module of this compiler is based on software supplied by the dataflow research group at McGill University, Montreal[13]. The support of the Defense Advanced Research Projects Agency through the Center for Advanced Architecture of RIACS is acknowledged with appreciation. The author wishes to thank Eric Barszcz for his suggestions from a careful reading of the paper.

References

1. J. B. Dennis, "Mapping array computations for a dataflow multiprocessor," *Proceedings of Mapcon IV: Multiprocessor and Array Processor Conference*, Society for Computer Simulation, 1988, pp 71-76.

2. J. B. Dennis, "Dataflow Computation: A Case Study," Chapter 9 of *Computer Architecture: Concepts and Systems*, V. Milutinovic, Ed., New York: North-Holland 1987.

3. J. McGraw, et al., "SISAL: Streams and Iteration in a Single Assignment Language," Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

4. Thinking Machines Corporation, "The Connection Machine System, Volume II: Paris Reference Manual," Cambridge, MA, June 1988.

5. W. D. Hillis, *The Connection Machine*, Cambridge, Massachusetts: MIT Press, 1985.

6. Thinking Machines Corporation, "Connection Machine Model CM-2 Technical Summary," Cambridge, MA, April 1987.

7. W. D. Hillis, and G. L. Steele, Jr., "Data parallel algorithms," *Communications of the ACM 29*, 12 (December 1986), 1170-1183.

8. K. Knobe, J. D. Lukas, and G. L. Steele, Jr., "Massively parallel data optimization," to be published.

9. M. T. Heath, ed., *Hypercube Multiprocessors 1987*, Philadelphia: Society for Industrial and Applied Mathematics, 1987.

10. J. B. Dennis, "Dataflow Computation for Artificial Intelligence," Chapter 14 of *Parallel Processing for Supercomputers and Artificial Intelligence*, K. Hwang and D. DeGroot, Eds., New York: McGraw-Hill, 1989.

11. W. B. Ackerman and J. B. Dennis, "Val – A Value-oriented Algorithmic Language: Preliminary Reference Manual," Technical Report MIT/LCS/TR-218, MIT Laboratory for Computer Science, Cambridge, MA 02139, 1978.

12. S. Skedzielewski and J. Glauert, "IF1: An Intermediate Form for Applicative Languages," Technical Report M-170, Lawrence Livermore National Laboratory, July 1985.

13. W.-K. Wong, IF-1 Parser for HDDG, Technical Note 01, Advanced Computer Architecture and Program Structure Group, School of Computer Science, McGill University, Montreal, Canada, June 1988.

14. C. Levit, "Grid Communication on the Connection Machine: Analysis, Performance, and Improvements," these proceedings.