Languages, Compilers and Run-Time Systems for Scalable Computers,
Kluwer Academic Publishers, Reading, MA, 1996, pp. 169-184

13

INTEGRATING DATA AND TASK PARALLELISM IN SCIENTIFIC PROGRAMS[†]

Ewa Deelman, Wesley K. Kaplow,
Boleslaw K. Szymanski, Peter Tannenbaum, Louis Ziantz

Department of Computer Science,
Rensselaer Polytechnic Institute,
Troy, NY 12180 USA

ABSTRACT

Functional languages attract the attention of developers of parallelizing compilers because of the implicit parallelism of functional programs and the simplified data dependence analysis of functional statements. A major drawback of functional languages is that naive translation of functional programs results in code that requires excessive memory. In this paper we explore the connection between the memory optimization and communication optimization of parallel codes generated from functional languages. We also show how a functional language can be used as an intermediate form in the translation from FORTRAN to customized, architecture-specific parallel code.

1 INTRODUCTION

FORTRAN is a common language used in engineering and scientific computing. Recent versions of the language (*e.g.*, Fortran90 and High Performance Fortran) allow programmers to embed special directives for running their programs in parallel. However, many currently used programs are written in older dialects of FORTRAN and cannot benefit from faster parallel computers; these programs run strictly sequentially.

We are developing a system that automatically transforms serial FORTRAN into parallel C, performing memory optimization and introducing data and task parallelism (see Figure 1). The core of the system is a new version of the EPL compiler (the original version of EPL is described in [12]) with a FORTRAN front-end. EPL is a functional language and therefore obeys the single-assignment rule. As such, data dependencies in EPL are readily visible. We exploit this characteristic in the construction of a detailed data-dependency graph called an *array graph*. The array graph represents

[†]This work was partially supported by ONR Grant N00014-93-1-0076 and NSF Grant CCR-9216053.

both data and control flow in a single structure. From the array graph we generate a *schedule graph* that represents the minimal constraints on the execution order of a computation.

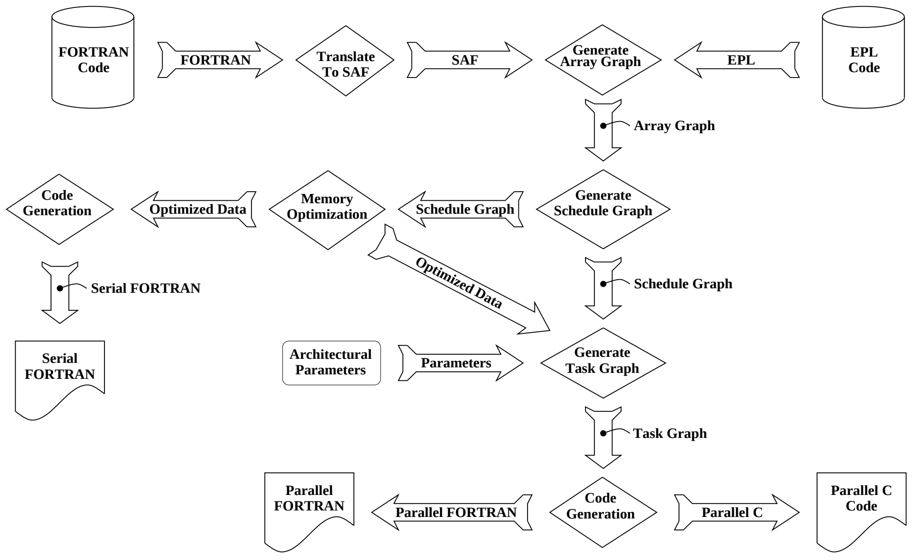

Figure 1 Overview of EPL System

The scheduled program is then optimized for memory usage and communication by finding optimal loop arrangements and variable representations. The schedule graph defines an initial task parallelism. Architectural parameters of the target machine are used both to tune the task parallelism and to extract data parallelism from the program. The new parallel schedule, called a *task graph*, is finally sent to the *code generator*.

Many scientific computations include iterative solvers, where an approximate result is repeatedly refined. In FORTRAN, each iteration writes the new values over the previous result. When such a solver is converted to a single assignment form, each successive result must be written to a new data location. This is one example of the excessive memory requirements inherent in functional languages.

Fortunately, analysis of array assignments and references can often show that each iteration step through a loop uses only a fixed number of adjacent elements of a data structure. This allows the object code to allocate a “window” large enough for only this fixed number of elements, not the entire dimension of the structure[13]. As with the original FORTRAN, the elements in the window are overwritten in each iteration.*

Programmers use variable windows intuitively, often with the intension of maintaining code readability. This approach may discount several more efficient (but counter-intuitive) windowing opportunities. Likewise, loops are the main source of task

*There is no reason for the optimized code to obey the single-assignment rule.

partitioning that enables data or pipeline parallelism. The scope of loops and their arrangements are crucial for parallelization of a program. We employ a systematic approach to windowing and parallelization in order to find an efficient solution. In general, new windows are formed by merging two or more loops found in the schedule graph. Large mergers, incorporating many loops, can potentially provide several windowing and pipelining opportunities. However, special care must be taken to preserve data dependencies.

In this paper we briefly present the major steps of FORTRAN to parallel C transformation and focus on loop transformations for both memory optimization and data and pipeline parallelization for distributed memory machines.

2 FORTRAN FRONT-END

The FORTRAN front-end is a procedural to functional language converter. Its purpose is to rename variables such that each variable is assigned exactly once. The result of this transformation is Single Assignment Fortran (SAF). SAF is still valid FORTRAN, but, as with EPL, the single-assignment rule exposes data dependencies. (Each assignment statement in SAF uniquely defines a variable, and so we will refer to EPL and SAF statements as equations.) A correct translation from FORTRAN into SAF centers on two major issues. First, each translated assignment must be guaranteed to bind a value to a unique variable. This requires a renaming scheme for variables that are multiply-assigned. The second issue is to ensure that every reference actually refers to the proper (renamed) variable. These two problems are intimately related, and are best addressed simultaneously.

As a first step in renaming, the line number of each assignment is appended to each assigned variable. FORTRAN allows only one assignment per line, and so the appended numbers are guaranteed to be unique. To find and rename variable references, we use def-use chaining: for each assignment, we find all uses of the renamed variable and rename the references accordingly. As a result of this first step, each basic block in the program, when considered independently, is in SAF. Any variable that was assigned repeatedly inside a basic block is now represented by new assignments, differentiated by line number.

The second step in renaming translates basic blocks from loops and subroutines; these blocks are executed iteratively. Each multiply-assigned variable assigned in these blocks is promoted in dimension: scalars become one dimensional arrays; one dimensional arrays become two dimensional arrays, *etc*. This new dimension, sometimes called a "temporal" dimension, is guaranteed to have a unique value for each iteration through the basic block. An iteration through a basic block that would have reassigned a variable in the original code will now make an assignment to a unique element in that variable's array. Note that assignments to two distinct array elements are valid in SAF; repeated assignments to the same array element are not valid. To determine whether assignments are made to the same array elements, we use various dependence tests (cf. [6, 7]). If the results of the dependence tests are negative within a loop, then no two elements are reassigned in the loop, and no temporal dimension is needed.

The temporal dimension is so named because it represents a time-stamp for each iteration. In the case of loops, its value can be simply the loop control variable. In subroutines, it can be an activation counter, automatically incremented each time the subroutine is called.[†] This additional dimension means that SAF programs require significantly more memory to execute than the original FORTRAN. Fortunately, memory optimization can reclaim this added space.

3 INTERNAL DATA STRUCTURES

Array Graph The array graph is one of the fundamental structures used by the EPL compiler. It represents control-flow and data dependence constraints on program execution in the form of dependencies between data elements in the program. The basic components of the array graph are nodes, edges, and edge attributes. The nodes of the graph represent different activities associated with the data elements and equations of a program. Data nodes indicate that a variable should receive its value, and equation nodes imply that an equation should be evaluated. The array graph has one node for each variable that is not a subscript and one for each equation.[‡] The edges in the graph indicate dependencies between nodes, and paths through the graph show an implied partial ordering of the computations associated with the equation nodes. Edge attributes list subscripting expressions and other information related to a particular dependency.

Formally, the array graph is defined as: $G_A = (N, E, M)$ where $N = \{n_1, \dots, n_p\}$ is a set of nodes, $E \subseteq N \times N$ is a set of edges, and $M : L = \{l_1, \dots, l_m\} \rightarrow N \cup E$ is a labeling function. E^* represents the standard transitive closure of the edge relation E. The labeling function M defines a label for each node and edge in the graph. The node label $l_i = M(n_i)$ includes information on the type and class of the node, its dimensionality, ranges of the dimensions, and subscripts associated with these dimensions. The edge label $l_{i,k} = M(e_{i,k})$ includes the type of the dependence and subscripting information.

A list of trees is created representing the equations in a program, and from this list the subgraphs for the equations are built. Each equation node is augmented with a list of the subscripts used in the corresponding equation — this is referred to as the node subscript list and is used in code generation. A dimension attribute node list is a sequence of indexing expressions associated with a dependency. One such list is attached to each edge based upon the array access that generated the edge. These lists are used in scheduling the computation.

[†]Strictly speaking, these counters do not obey the Single Assignment rule, and therefore they behave like subscripts in EPL. However, all other variables in the program are in SAF.

[‡]A variable of type subscript in EPL would correspond to a loop-control variable in an SAF program.

```
interface input {
  [integer A[150], B[250], C[250]]
};
interface output {
  [integer E[150]]
};
subscript i,j,k;
integer D[150];

a3:C[k] = C[k-1] + 1;

a1:D[i] = if (i == 2*j/3) then
          A[i] + B[j] endif;

a2:E[i] = if (i == 2*k/3) then
          D[i] + C[k] endif;
```

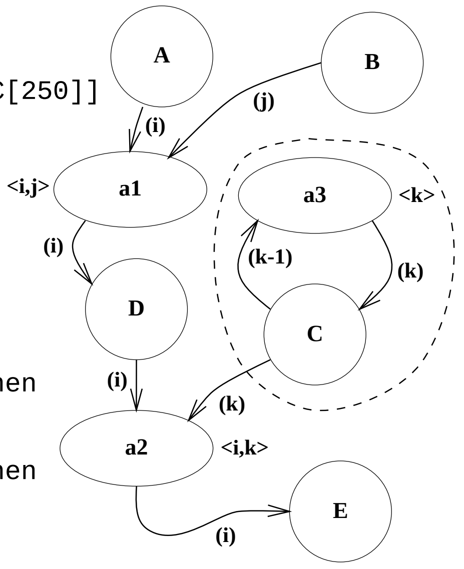

Figure 2 Example EPL code and corresponding Array Graph

Figure 2 shows an example array graph. The circles represent data nodes, and equation nodes are represented by ovals. The bracketed variables correspond to node subscript lists, and the variables enclosed by parentheses are dimension attribute node lists. Note that equation one (**a1**) represents the assignment of values to `D`, equation two (**a2**) represents the assignment to `E`, and equation three (**a3**) represents the assignment to `C`. The dotted line in Figure 2 encloses a strongly connected component (SCC).[§]

Schedule Graph The scheduler recursively traverses the array graph, storing a feasible program ordering in the (directed, acyclic) schedule graph. Nodes in the schedule graph represent loops, assignments, and I/O. One edge class represents temporal dependence: the action at the source of the edge must complete before the action at the destination begins execution. The other edge class, called a *zoom* edge, represents a loop nesting: the source of a zoom edge is always a loop node, and the subgraph at the destination is nested inside the loop. This subgraph can itself contain loop nodes and zoom edges, representing multiply nested loops.

Formally, the schedule graph is a refinement of the array graph in which nodes represent disjoint subsets of array graph nodes and edges include the additional class of zoom edges.

A major function of the scheduler is to find and break SCCs in the array graph. (Cycles in the array graph occur when variables are defined recursively.) If a component

[§] An SCC is a maximal subgraph of the array graph with the property that for any pair of its vertices v, w there are directed paths in the SCC from v to w and from w to v.

contains more than one node, it will be scheduled inside a loop. The components are then interconnected based on the dependencies present in the array graph. Each SCC is searched for cycles formed by edges containing the same subscript. Under certain conditions, the scheduler can form a loop over that subscript and mark the edges forming a subscript as resolved. For example, if a cycle contains edges labeled with indexing expressions k and $k-1$, then scheduling a loop over k will enforce a dependency represented by the edge labeled $(k-1)$.¶ This edge is redundant and can be removed, thus breaking the cycle and decomposing the SCC.

Each newly formed loop has associated with it a direction: ascending, descending, or either. In the case above, the dependencies dictate that the loop over k must be ascending. A loop that calculates $B[k] = A[k]$ could proceed in either direction.

Code Generation The C code generator traverses the task graph in a topological order. For each loop node, a corresponding C *for* loop is formed. Then all the zoom edges emerging from the loop node are traversed, and the code generation is performed on the subgraph. As the zoom edge is traversed, the loop subscript is remembered by placing it into an *active subscript* list. The code for an equation node is constructed as follows: first the loops for the subscripts present in the equation and not present in the *active subscript* list are formed, and then the code for the equation itself is emitted. After the code for all the subgraphs reachable via the zoom edges has been generated, other nodes dependent on the loop node are generated. If an equation is not reached via some zoom edge, loops will be emitted for all the subscripts present in the equation's node subscript list.

4 MEMORY OPTIMIZATION

The scheduler produces loops with the smallest possible scope for a correct schedule. This means that each loop defines a minimal number of variables. Loop merging expands the scope of loops to define several variables at once. As a simplification, we only consider merging loops that iterate over equivalent ranges. (An alternative would be to consider merging a loop over range n with a loop over range $m \subset n$; this would require special guards to avoid defining or referencing invalid values. For a special class of subscripts, called sublinear subscripts in EPL [12], this is a straightforward extension and it will be included in the next version of the EPL compiler.) Variable definitions are characterized by their *range-sets*.

Definition 4.1 *A range-set defines the iteration space for a single dimension of a variable's defining equation. An n-dimensional variable will thus have n (possibly distinct) range-sets. Two range-sets are equivalent if they define the same iteration space.*

To illustrate our algorithm for merging loops, we use a refined version of the schedule graph, called a *variable graph* (see Figure 3): $G_V = (N, E, M)$ where $N =$

¶The value of the $(k-1)^{st}$ iteration will be preserved for the k^{th} iteration.

$\{n_1, \ldots, n_p\}$ is a set of nodes, $E \subseteq N \times N$ is a set of edges, and $M : L = \{l_1, \ldots, l_m\} \rightarrow N \cup E$ is a labeling function. Here, the defining node for each variable (called the equation node) is shown explicitly in the graph only when the defined variable has a lower dimensionality than any of its constituent variables. If the defined variable has at least as many dimensions as its constituents, then the equation node is merged with the data node. This distinction is useful in defining loops as described below. Edges in the graph point to nodes in which variables are referenced. The edge label $l_{i,k} = M(e_{i,k})$ includes a *dependence bound*.

```
B = A[rnd * A.range];
C[i] = B + A[i];
```

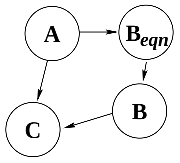

Figure 3 Example of EPL equations and resulting variable graph. Note that *B* is a scalar, but its assignment requires iterating through potentially all elements of *A*.

The dependence bound gives an indication of how many new elements of a variable are created before the current element is used. It is calculated for each variable graph edge as follows:

Let an edge $e \in E$ correspond to an equation of the form,

$$B[\ldots, se_0, \ldots] = f(A[\ldots, se_1, \ldots], \ldots, A[\ldots, se_k, \ldots])$$

where each se_j is a sub-expression on i. The bound on the range-set i for this edge is given by:

$$\max_{1 \leq i \leq i.range} \max_{1 \leq m \leq k} se_0 - se_m$$

For the current version of the EPL compiler, we assume that the bound is the full range of i except when $se_0 = i$ and all se_j have the form $i \pm c$, where c is a constant. If the bound is a compile-time constant, then i is written as $i_<$ for an ascending loop and $i_>$ for a descending loop. If the bound is not known at compile time, or is known to be the full range of i, then i is written as $i_\perp$. Future versions will include deeper subscript analysis.

Two sets are easily built from the variable graph: the set of windowable variables and the set of distinct range-sets. From these, four more sets are also easily created: the set of variables referenced and/or defined in each node, the set of nodes in which each variable appears, the range-sets associated with each node, and the nodes associated with each given range-set.

Definition 4.2 *Let $V = \{v_1, v_2, \ldots, v_q\}$ be the set of all variables in the variable graph that are potentially windowable. A variable is potentially windowable iff:*

1. *For at least one dimension, its dependence bound is defined at compile-time and is known to be less than the full range of that dimension.*
2. *All its successors in the variable graph have at least one range-set in common.*

Definition 4.3 *Let $R = \{r_1, r_2, \ldots, r_s\}$ be the set of all range-sets in the variable graph.*

Definition 4.4 *For each $n_i \in N$ let $W_{n_i} = \{w_1, w_2, \ldots, w_p\}$ be the set of variables associated with n_i, and $J_{n_i} = \{j_1, j_2, \ldots, j_r\}$ be the set of range-sets associated with n_i. For each $v_i \in V$ let $M_{v_i} = \{n_j \in N : v_i \in W_{n_j}\}$. M_{v_i} is the set of all nodes in the variable graph that are associated with variable v_i. For each range-set $r \in R$, let $P_r = \{n_k \in N : r \in J_{n_k}\}$. P_r is the set of all nodes that are associated with range-set r.*

The objective of merging loops is to create valid windowing opportunities. A variable window is valid only when the definition and all references to the variable are enclosed in the same loop. To see this, consider a one-dimensional variable $A[n]$. If A is windowed, then in the i^{th} iteration of its loop $A[i]$ will be defined and referenced. In a later iteration, the same memory location will be re-defined. If a reference to $A[i]$ appears outside the loop, the resulting value will be undetermined.

Another concern in merging loops is the preservation of data dependencies. Consider the EPL equation $B[i] = A[i]$. If A is windowed, then each iteration through the loop will assign an element[||] of A and then will reference that element in the assignment to the corresponding element of B. However, consider the slightly modified equation $B[i] = A[i] + A.last$ (where $A.last$ refers to the final element of the array A). The value of $A.last$ depends on the array A. If A is windowed, then the first iteration through its loop will define an element of A and will try to assign it, plus $A.last$, to the corresponding element of B. But, $A.last$ will not be defined until the final iteration of the loop. This is an instance where merging loops violates dependencies.

4.1 Well-Formed Loops

To address these three considerations: existence of a common range-set, presence of completely enclosed variables, and preservation of data-dependencies, we have developed the concept of *well-formed loops*. A well-formed loop is a subset of variable graph nodes with certain properties. Informally, nodes in the loop must all have at least one range-set in common (*i.e.*, they must all share at least one range-set in the source program). For at least one variable, the loop must contain all nodes associated with that variable. If any node inside the well-formed loop is dependent upon a node outside the loop, then the outside node cannot be dependent upon any nodes inside the loop. Finally, the loop cannot contain nodes that are not needed to satisfy the above constraints.

[||]This element can come from an equation or be read from an input port.

These properties can be defined in terms of functions over subsets of nodes in the variable graph.

Definition 4.5 *Let the* *iterators* *of a set of nodes in a variable graph, denoted* $I(X)$*, be defined as:* $I(X) = \prod_{n \in X} J_n$

Definition 4.6 *Let the* *core* *of* X *over* r*, denoted* $C(X, r)$*, where* $X \subseteq N$*, and* $r \in R$*, be defined as:* $C(X, r) = (\bigcup_{v \in V : M_v \subseteq (X \cap P_r)} M_v) \cap (\{n \in X : \forall n_1 \in X, (n, n_1) \in E \rightarrow r \in J_{n_1}$ *and the bound of* $(n, n_1) < r.range\})$

Definition 4.7 *Let the* *closure* *of a set of nodes in a variable graph, denoted* $Cl(X)$*, be defined as:*
$Cl(X) = X \cup \{n \in N : (\exists p_0 \in X : (p_0, n) \in E^*) \wedge (\exists p_1 \in X : (n, p_1) \in E^*)\}$

Iterators are similar to loop control variables in FORTRAN, with the associated loop direction; the core represents the minimal subset of variable graph nodes needed to completely enclose a windowable variable; the closure enforces preservation of dependencies. The following definition formalizes the concept of a well-formed loop:

Definition 4.8 *A non-empty subset* $L \subseteq N$ *of nodes in a variable graph is called a* *well-formed loop* *iff:*

1. *The set of iterators over* L *is non-empty.* $(I(L) \neq \emptyset)$
2. *The closure of the core of* L *is equal to* L*.* $(Cl(C(L)) = L)$

The algorithm for finding well-formed loops uses a slightly different definition:[**]

Definition 4.9 *A non-empty subset* $L \subseteq N$ *of nodes in a schedule graph is called a* *well-formed loop* *iff:* $I(L) \neq \emptyset$ *,* $Cl(L) = L$ *, and* $Cl(C(L)) \supseteq L$*.*

4.2 Generating and Merging Well-Formed Loops

Well-formed loops are constructed and merged one range-set at a time. The first step in the algorithm creates the lowest rank well-formed loops for each variable in the range-set; rank refers to the number of variables enclosed in a loop. Then, where possible, the loops are iteratively merged. For n variables in a given range-set, there are potentially 2^n loop mergers. Fortunately, certain loops will be incompatible with other loops; this feature can greatly reduce the algorithm's search space. A loop is incompatible with another loop, for example, if it is only partially enclosed in the second loop. (A similar rule applies to loops in procedural languages.)

[**]The proof that the two definitions are equivalent is straightforward but somewhat lengthy, and is not included here.

As the algorithm merges loops, it stores its results in a *loop graph*. Nodes in this graph represent well-formed loops, and edges (called exclusion edges) connect pair-wise incompatible loops. In addition to the rule above, two loops are marked incompatible if they are mutually dominated by a third loop. A loop dominates two other loops if it has a higher rank than the two smaller loops and its set of exclusion edges is the union of the exclusion edges of the smaller loops. When two loops are found to be dominated by a third, the two loops are removed from the list of loops to use in future mergers; only the dominator is retained. In this way, each domination reduces the search space by one.

Once the loop graph is fully constructed, the final step is to choose the most memory efficient, feasible (*i.e.*, compatible) set of well-formed loops. Memory efficiency is based on the window sizes for each loop as calculated in [9]. The results of the memory optimization are either read by the code generator for serial output, or parallelized as described in the next section.

Definition 4.10 *For the pair consisting of the subset of nodes $U \subseteq N$ and a range-set $r \in I(U)$, the loop over r generated by U, designated $L(U,r)$, is defined as $L(U,r) = Cl(C(U))$ if $Cl(C(U)) \subseteq P_r$ and $L(U,r) = \emptyset$ otherwise.*

Definition 4.11 *The rank of a loop $L(U,r)$ is defined as:*
$|\{v_j \in V : M_{v_j} \subseteq L(U,r)\}|$.

The construction incorporates the three rules in the definition of well-formed loops. Recall the distinction between the schedule graph and the variable graph. In the example shown in Figure 3, a well-formed loop containing the three nodes A, B_{eqn}, and C will be augmented to contain B by the closure requirement. However, B does not share a range-set with the other three nodes, and therefore the loop containing all four nodes is not a well-formed loop. If B did share a common range-set with the other nodes, then this would not be an issue, and there would be no reason to treat B's data node separately from its equation node. The general algorithm for generating the loop graph is then:

1. $\forall r \in R$:
   (a) $\forall v_j \in V : M_{v_j} \subseteq P_r$, create $L(\{ev_j\}, r)$, where ev_j is the equation node that defines v_j. This creates the initial set of seed loops.
   (b) Loops are iteratively joined as follows:
   For any two seed loops $L(U_j, r)$ and $L(U_k, r)$, if $Cl(C(u_j \cup u_k)) \neq \emptyset$ then create $L(u_j \cup u_k, r)$. If $L(u_j \cup u_k, r)$ dominates $L(U_j, r)$ and $L(U_k, r)$, draw an exclusion edge from $L(U_j, r)$ to $L(U_k, r)$, remove the two dominated loops from the set of seeds, and replace them with the dominator.
2. For all pairs of loops, $L(U_j, i)$ and $L(U_k, l)$, if $L(U_j, i) \cap L(U_k, l) \neq \emptyset$ and $L(U_j, i) \not\supseteq L(U_k, l)$ and $L(U_j, i) \not\subseteq L(U_k, l)$ then draw an exclusion edge

from $L(U_j, i)$ to $L(U_k, l)$. In other words, draw an exclusion between any two overlapping loops where one loop is not completely enclosed in the other.[††]

5 PARALLELIZATION

The semantics of EPL allow for a trivial parallelization of an EPL program on a dataflow machine. Every instance of each equation could be a separate thread enabled for execution when the data it requires becomes available [14, 1, 4]. Such synchronization would enforce a valid order of EPL program execution. However, this form of parallelization is not efficient on current dataflow machines, such as the Monsoon[5], because all synchronization is done at run-time, increasing the overhead incurred by token communication and matching.

A more efficient approach is to recognize at compile time which threads must execute sequentially relative to each other and then to merge them. This is the motivation behind creating the EPL schedule graph, introduced earlier, in which equations that are cyclically dependent on each other are clustered together. The nodes of the schedule graph constitute the smallest unbreakable tasks of computation in EPL parallelization.

The schedule graph nodes that are data dependent are clustered further at the time of memory optimization when the well-formed loops (WFL) are created. As discussed below, to explore pipeline parallelism such clustering can be extended beyond the boundaries of the well formed loops. An essential observation motivating this clustering is that the bodies of the pipeline loops can be parallelized in two ways. If the input to the pipeline loop can be provided in parallel, a data parallelization can be used in which an instance (or a range of instances) of the loop body is assigned to a separate processor. Regardless of the way the input arrives, the loop body can be pipelined, *i.e.*, several processors can be assigned to different parts of the loop body. The second method, pipelining of EPL programs, is discussed below. It should be noted that the object code created by pipelining invokes different parallel tasks executing together, and therefore such parallelization reaches beyond the SPMD model.

The goal of pipeline loop optimization is to enable the compiler to adjust the *granularity* [3] of the resulting computation to match the architectural parameters of a target architecture. This can be done by first adjusting the size of the pipeline stages to balance the computational load of each resulting cluster. Then, the pipeline communication message size can be selected to reduce the overhead incurred by inter-cluster communication based on the communications performance of the target.

5.1 Creation of Pipeline Loops

The objective of this stage of EPL parallelization is to create the largest clusters of schedule graph nodes that are amenable to pipelining. We refer to those clusters as Pipeline Loops (PLs). As in the case of well-formed loops (WFLs), nodes in such

[††]Note that these exclusion edges go both within and across range-sets.

clusters must share the same or equivalent range set[‡‡] and the closure of the cluster must be equal to itself. Unlike the well-formed loops, there is no need for a PL to cover all references to variables associated with the nodes. As a result, for each PL there is a WFL such that PL $\supseteq$ WFL, so WFLs are a good starting point for building PLs. Because of this looser restriction on the variable coverage, several well-formed loops can be clustered together into a pipeline loop.

Definition 5.1 *A non-empty subset $L \subseteq N$ of nodes in a variable graph is called a pipeline loop iff:*

1. *The set of iterators over L is non-empty.* $(I(L) \neq \emptyset)$
2. *The closure of L is equal to L.* $(Cl(L) = L)$

Any clustering of the schedule graph nodes defines an equivalence relation between those nodes. Therefore, there exists a corresponding refinement graph which we will refer to as a Clustered Schedule Graph (CSG). To determine the largest pipeline loops of a computation, we will consider only maximal clusterings and their corresponding graphs. Initially, each node of the CSG is a well-formed loop, and each edge describes the communication requirements between each pair of well-formed loops. Using this information we are able to enumerate all of the pipelines of a computation. Figure 4 shows an example CSG, illustrating the two possible maximal pipeline loop arrangements.

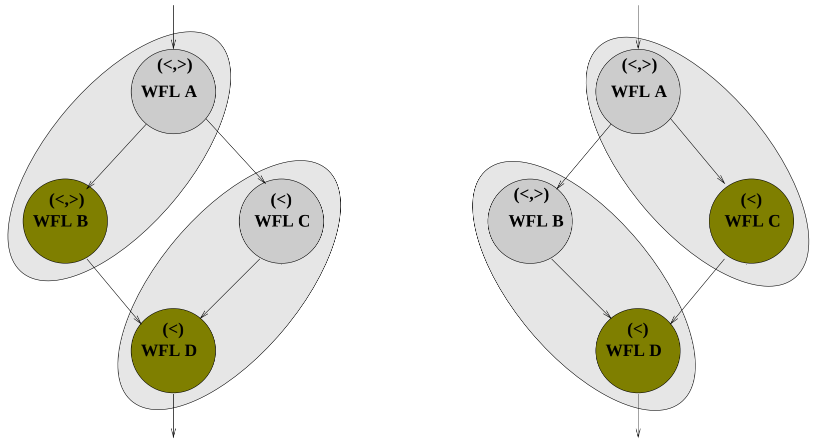

Figure 4 Two Candidate Dominant Pipeline Loop CSGs

In a CSG, each node is labeled with its directed iterators. The edges are labeled with the minimum delay.

[‡‡]Equivalence is defined over the relation in terms of subscript sublinearity (see [12]).

The graph in Figure 5 is a CSG where each node represents a well-formed loop, except for WFL C which also exposes its component schedule graph nodes. For example, consider WFL C and WFL E; the latter depends on values created by T ordered by the range set i in ascending order. We can easily see why a component of WFL E could not be merged into WFL C. This is due to the fact that WFL G uses the same values from T, but in a different order (descending). Thus, the windowing could not be extended. However, this does not mean that a pipeline loop could not be extended. In particular the pipeline loop could be extended to include both WFL E and WFL F. In the backward direction, we could extend the pipeline loop to include WFL A as well.

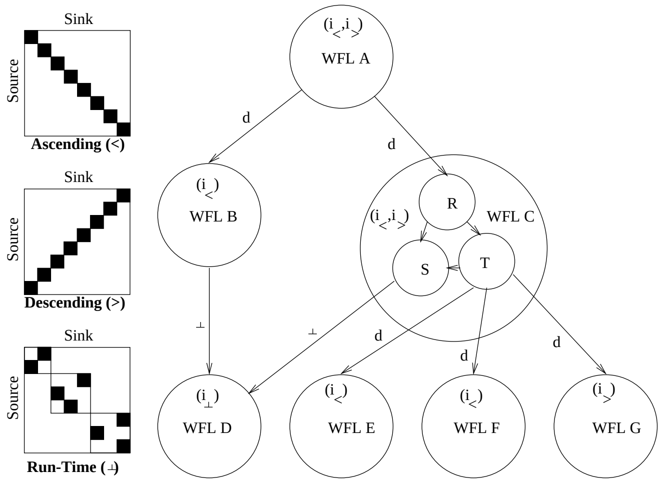

Figure 5 Example of a Clustered Schedule Graph

In our example CSG, WFL A and WFL C are labeled with $i_<, i_>$, indicating that the nodes are pipelineable in either direction. Therefore, another pipeline can be formed, one that includes WFL A, WFL C and WFL G. However, this is an exclusive-or in that it excludes the previously formed pipeline loop (*i.e.*, WFL A, WFL C, WFL E, and WFL F). To determine the compatibility for each node in the CSG we will use an algorithm described in [10], for *internal* data dependency propagation.

5.2 Optimizing Pipelines for Target Architectures

Pipeline Optimization Process The following steps are used to optimize the execution of pipeline loops on a given target architecture:

1. Apply algorithm in Section 4.2, replacing Definition 4.8 with Definition 5.1. This creates an exclusion graph of the *dominant* pipeline loops.

2. For each candidate dominant pipeline loop CSG, optimize for a given architecture:

   (a) Determine the number of processors that can be effectively used for each dominant pipeline loop.

   (b) Determine the optimal pipeline communication size.

3. Cluster dependent pipeline loops.

The first step is performed by the process described in the previous section. Figure 4 shows a CSG with the two possible pipeline loop configurations where optimization can be explored.

For parallelization and pipeline optimization, only the boundaries of the pipeline loops are significant. Thus, each schedule graph node in a pipeline loop can potentially be scheduled on a separate processor as a pipeline stage. However, as shown in [8, 3] this may lead to a parallelization with too fine a granularity. Moreover, a pipeline's throughput is limited by its *worst* stage. Because of the regular communication patterns within a pipeline loop, Step 2a in the process is free to *cluster*, or *internalize* the communication between stages in the pipeline to increase the granularity to match target architecture parameters, while not reducing the potential speedup due to pipeline execution. Thus, Steps 2a and 2b are executed iteratively. Starting with n processors, Step 2a partitions the pipeline into n equal pipeline stages. Once this is done, the optimal block size (discussed in the next subsection) is chosen. If the resultant efficiency is lower than a serial execution, then the number of processors is decreased, and the pipeline is again repartitioned and communication optimized. If a binary search method is used, then the execution time is $O(Nlog(P_n))$ where N is the number of schedule graph nodes in the pipeline loop, and P_n is the maximum number of processors that can be assigned.

For the final step, we are investigating how known clustering techniques, *e.g.*, reducing the makespan [2], or increasing throughput [11], apply to this environment.

Determining the Pipeline Parameters The following is a simple analytical model that is used to illustrate how optimal parameters of a pipeline can be determined. The base case of the model assumes that we have a pipeline loop which consists of a sending node E_1 with C_1 instructions executing on processor with S_1 instruction rate, and a receiving node, E_2, with C_2 and S_2 defined similarly. For simplicity, we assume a linear cost model for inter-processor communications, $C = \alpha + n \times \beta$, where α is the startup message overhead, β is the time to transfer one element over the communication channel, and n is the number of elements to transfer. The minimum number of elements required for a pipeline stage is given by d.

The cost of executing the pipeline loop with blocking factor k is:

$$kS_1C_1 + (n/k - 1)max(kS_1C_1, kS_2C_2, \alpha + kd\beta) + \alpha + kd\beta + kS_2C_2$$

If we assume that the computation of the pipeline can be balanced, that is: $A = S_1C_1 = S_2C_2$, then to find k that minimizes the cost, consider two cases: $k \geq \alpha/(A - d\beta)$, so $A \geq d\beta + \alpha/k$, and therefore, $k = \alpha/(A - d\beta)$, is the minimum. In the opposite case: $k < \alpha/(A - d\beta)$, and therefore $A < d\beta + \alpha/k$, so the derivative of the cost is: $2A - (n\alpha)/k^2$, and $k = \sqrt{n\alpha/2A}$, hence,

$$k_{opt} \quad = \quad min\left(\sqrt{\frac{n\alpha}{2A}}, \frac{\alpha}{A - d\beta}\right)$$

We can further observe that,

$$k_\infty \equiv \lim_{n\to\infty} k_{opt} = \frac{\alpha}{A - d\beta}$$

In fact, if

$$n \quad \geq \quad \frac{2A\alpha}{(A - d\beta)^2} \approx 2\frac{\alpha}{A}\text{ , then } k_{opt} = k_\infty$$

This formula helps to define the proper granularity for the pipeline for a given architecture. As long as the pipeline executes more than n repetitions, then it achieves optimum performance. Also, for a given n and latency α, the granularity of A can be adjusted by clustering.

6 CONCLUSIONS

This work shows how a uniform representation of fine-grain parallelism can be used for both the memory optimization and parallelization processes. We have presented a common formal basis for memory optimization and task creation. This method can be incorporated into a compiler to determine optimum pipelines based on a model of target architecture parameters. Much research remains, and we intend to incorporate these ideas into the EPL compilation system.

REFERENCES

[1] J. L. Gaudiot and L. Bic. *Advanced topics in dataflow computing*. Prentice Hall, New Jersey, 1991.

[2] A. Gerasoulis and T. Yang. Comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, (16):276–291, 1992.

[3] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, 1993.

[4] B. Lee and A. R. Hurston. Dataflow architectures and multithreading. *IEEE Computer*, pages 27–39, August 1994.

[5] G. M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[6] K. Psarris, X. Kong, and D. Klappholz. The direction vector I test. *IEEE Transactions on Parallel and Distributed Systems*, 11(4), November 1993.

[7] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. In B. K. Szymanski and B. Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 1–14. Kluwer Academic Publishers, Boston, 1995.

[8] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. Research monographs in parallel and distributed computing. MIT Press, Cambridge, MA., 1989.

[9] B. Sinharoy and B. K. Szymanski. Memory optimization for parallel functional programs. *Computer Systems in Engineering*. To appear; abstract published in "Abstracts: International Meeting on Vector and Parallel Processing," CICA, Porto, Portugal, 1993, p. 36.).

[10] K. L. Spier and B. K. Szymanski. Interprocess analysis and optimization in the equational language compiler. In Burkhart, editor, *CONPAR 90-VAPP, Joint International Conference on Vector and Parallel Processing, Zurich, Switzerland*, Lecture Notes In Computer Science. Springer-Verlag, September 1990.

[11] J. Subhlok, D. R. O'Hallaron, T. Gross, P. A. Dinda, and J. Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proceedings of SuperComputing 1994*. ACM, 1994.

[12] B. K. Szymanski. EPL–parallel programming with recurrent equations. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press/Addison Wesley, New York, 1991.

[13] B. K. Szymanski and N. S. Prywes. Efficient handling of data structures in definitional languages. *Science of Computer Programming*, 10(3):221–245, 1988.

[14] A. H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4), 1986.