Specification of Parallel Algorithms, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 18, American Mathematical Society, Providence, RI, 1994, pp. 201-223.

Specifying Parallel Programs in a Functional Language: the EPL Experience

BOLESLAW K. SZYMANSKI

ABSTRACT. This paper describes the research and development associated with the parallel functional language called EPL – Equational Programming Language – and its compiler. The emphasis is on opportunities and challenges arising from the use of a functional paradigm for specifying parallel programs. The EPL approach is based on a two-level computation description: the specification of the individual processes in EPL and the description of their interactions in a macro data flow language, called configuration. The EPL process specification targets loop and data parallelism, while the interaction description is oriented towards task parallelism. The EPL compiler performs an extensive program analysis and customizes the generated parallel code to specific architectures. The parallel computation decomposition and synchronization is based on the source program annotations provided by the user and on the user's defined configurations.

The paper provides a general description of the features of the EPL system, the detailed technical results have been published elsewhere. It concludes with a discussion of the general properties required of higher level parallel programming languages and with an assessment of EPL from that point of view.

1. Why Use a Functional Language for Parallel Programming

Parallel computation has become indispensable in the solution of large-scale problems that arise in science and engineering. While the use of parallel computation has been increasing, its widespread application has been hampered by

1991 *Mathematics Subject Classification.* Primary 68N15, 68N20; Secondary 68Q60, 68Q10.

The author was supported in part by ONR grant N00014-93-1-0076, by NSF grants CCR-9216053 and ASC-9318184, and by a grant from IBM Corporation. The content of this entry does not necessarily reflect the position or policy of the U.S. Government—no official endorsement should be inferred or implied.

The paper is in final form and no version of it will be submitted for publication elsewhere.

the level of effort required to develop and implement the needed software. Parallel software often must be tuned to a particular parallel architecture to execute efficiently; thus, it often requires costly redesign when ported to new machines. Different categories of parallel architectures have led to a proliferation of dialects of standard computer languages. Varying parallel programming primitives for different parallel language dialects greatly limit parallel software portability.

Parallel computation can be viewed as an interwoven description of operations that are to be applied to data values, and of data movement and synchronization that dictate the form of data access and the computation order. The traditional programming languages, like Fortran, C, or C++, provide a description of data movements and synchronization through ad hoc architecture-dependent extensions. Examples are various synchronization constructs, such as busy-wait, locks or barriers, used in programs for shared-memory machines, send and receive with different semantics employed by programs for message-passing architectures, and dimension projection and data broadcast popular in programs for SIMD computers.

Traditional imperative languages over-specify programs and as a result such programs are hard to write and, more importantly, difficult to understand and debug. The over-specification falls into one of the two classes:

(i) *Data storage related:* unnecessary details describing storage representation or management. For instance, the program may state that two names must be mapped to the same storage instead of a separate space, or a particular integer variable may be treated as a list of bits and not as an encoded number. Another example is an explicit deallocation by the program of part of its memory at a given point of the program execution.

(ii) *Flow of control related:* unnecessary details that specify the exact order of program execution or allocation of the program's resources. For instance, the program may direct the compiler to execute two particular loops in parallel or to execute a specific code on a particular processor, etc.

Functional programming paradigm has been one of the main directions in developing new languages that directly addresses the challenge of parallel programming. Functional languages encourage the use of functional abstractions in place of control abstractions. They employ type inferencing for consistency checking in place of type declarations. The storage related details have no place in functional languages because of their fundamental premise of value transformations. In addition, functional languages can easily separate the description of operations to be performed on data values from the definition of data movements and the synchronization needed to supply these data values to the proper processor at the proper execution instance. Hence, a well-designed functional language can shield the programmer from designing a detailed implementation, a flow of control, or a synchronization, while automatically exploiting parallelism that exists in a program.

In functional languages, computational abstractions are expressed through functions. A first-order function takes data objects as arguments and produces new data objects as results. What is abstracted by the function is the method used to produce the new objects from the arguments. Much of the elegance of functional languages stems from such semantics and from the absence of operational or machine-dependent details. It can be further advanced if the referential transparency is supported by enforcing the single assignment rule. Under this rule, each variable can be assigned a value only once. Thus, all references to a variable yield the same value, no matter where the reference is placed in the program. Because arguments of a function could be evaluated in any order, functional programming languages exhibit significant amounts of implicit parallelism. The above-discussed properties are also important in sequential programming. However, compilers of conventional languages for sequential machines are less complex and produce more efficient code than the compilers of functional languages. This advantage of conventional languages does not carry to parallel programming.

Many large codes were written long before parallel processing was easily accessible, and their methods are based on efficient, sequential algorithms. Turning such algorithms into algorithms that can be easily and efficiently parallelized is not merely a job for an "optimizing" compiler, but requires reprogramming at a fairly high level of the algorithm and data structure design. In general, the task of automatically reprogramming computation into parallel programs is extremely difficult and costly except for some very restricted classes of problems. Lower level details of parallel implementation, such as identifying independent threads of control flow, spawning parallel tasks, and deciding low-level details of data representation, can be automated. Functional languages are a good basis for building compilers capable of such automation, thanks to their simple semantics and lack of notion of the execution state.

2. What is EPL

This paper describes Equational Programming Language (EPL). It is a simple, array-manipulation functional language designed for programming scientific parallel computations. Scientific computations are particularly well suited for parallel processing. It can be claimed that, so far, they have driven parallel applications most aggressively. Although computationally vast, scientific computations are typically quite regular both in terms of control flow patterns and employed data structures. Often such computations are comprised of iterative applications of numerical algorithms to all (or the majority of) the parts of the data structures. We refer to such computations as *iterative computations*. Typically, the data structures used in scientific computations are some variations of multidimensional arrays (sparse matrices, grids, jagged-edge arrays, and even some hierarchical structures can be viewed as such). Correspondingly, the EPL language is defined in terms of a few constructs: generalized, jagged-edge arrays and subscripts for data structures, recurrent equations for data value definitions, ports for process interactions, and virtual processors for execution directives. The canonical data structure is an irregular tree which, in its simplest form, can be viewed as a multi-dimensional array. Structured files are provided for communication with an external environment (in *records*) and with other processes (through *ports*). EPL enforces a *single-assignment* rule, i.e., each data element should be defined exactly once (the EPL compiler, however, is free to produce imperative code including reassignment of variables). Equations, though syntactically reminiscent of assignment statements, are best viewed as assertions of equality. In EPL, a computation is represented by a collection of processes that interact through ports. Such a representation is particularly convenient for expressing task parallelism. Each process is specified with equations and data declarations. Equations may be annotated by virtual processors on which they are to be executed. A process definition is used by the EPL compiler to uncover loop and data parallelism. An integration of different level parallelism in a single computation is one of the design goals of the EPL system.

2.1. Iterations. One of the most important constructs in any functional language is that of iteration. In EPL, iterations are constructed using *subscripts*. A subscript assumes a range of integers as its value. Subscripts give EPL a dual flavor. In the *definitional view*, they may be treated as universal quantifiers and equations can be viewed as logical predicates. In such a view, the compiler must produce a program that for the given input will produce such EPL variable values that all predicates become satisfied. In the *operational view*, subscripts can be seen as loop control variables and each equation can be seen as a statement nested in loops implied by its subscripts.

There is a special class of subscripts, called *sublinear subscripts*, that are used in scientific applications so often that a special construct devoted to them has been introduced in EPL. Formally, an indirect subscript s defined over the subscript i is an array $s[i]$ associated with a condition $cond[i]$ such that:

$$s[i] = \text{if } cond[1] \text{ then } s[i-1]+1 \text{ else } s[i-1] \text{ and } s[0]=0.$$

It immediately follows from this definition that the sublinear subscript $s[i]$ starts with either the value of 1 or 0 and, then with each increase of i, it is incremented by either 1 or 0. A sublinear subscript has an implicit range determined by the number of times the defining condition yields true.

The sublinear subscripts are convenient in expressing such operations as creating a list of selected elements, operating on sparse matrices, or defining a subset of the given set. Even more important is the fact that in the implementation of a process no new iteration has to be created for computation associated with the sublinear subscripts. Instead, all necessary computation can be nested in the iterations created for subscripts in terms of which the considered sublinear subscript has been defined. Sublinear subscripts are also useful in defining dynamic distribution of data to processors.

2.2. Reduction. A computation that frequently occurs in scientific applications is to apply a binary operation over an entire vector and store the result in the last element of the vector. For example, in scientific computation there is often a need to apply an associative operator (such as $+, *, -, max, min$, etc.) selectively on the elements of an array. **Scan** and **Reduce** are language constructs in EPL and other parallel languages that allow such operations to be succinctly written. Reduce applied to a vector of values produces a scalar result, whereas scan creates a vector of partial results.

For example, let's consider the Jacobi iterative solver of a system of linear equations

$$Ax = y$$

where A is a nonsingular square matrix. If $x^{(k)}$ is the vector of solution at the k-th iteration and $x^{(k+1)}$ at the $(k+1)$-th iteration, the Jacobi iteration can be written as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(-\sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + y_i \right) \tag{2.1}$$

for $i = 1, \ldots, n$, where $x_j^{(k)}$ is the j-th element in the solution vector at the k-th iteration. In EPL, this iteration can be written as:

$Temp[i, j, k] = \text{if}(j == 0) \text{ then } 0$
    $\text{else if } (j \neq i) \text{ then } Temp[i, j-1, k] + A[i, j] * X[j, k] \text{ else } Temp[i, j-1, k];$
$X[i, k] = (Y[i] - Temp[i, range.j, k-1])/A[i, i];$

or even shorter as
$X(i, k) = (Y(i) - Reduce(+, A[i, j] * X[j, k-1], i : i \neq j)/A[i, i];$

Scan and reduce operations generate references of the form $B[\ldots range.i, \ldots]$, where $range.i$ indicates the range of the reduced/scanned dimension of a multidimensional array B (in general, the EPL *range* variable prefix denotes the size of its suffix). Such references are important in the memory allocation and scheduling of EPL programs.

For a more detailed description of the language, see [**20**].

2.3. Configurations. In EPL approach, a parallel computation is viewed as a collection of cooperating *processes*. Processes are defined as functional programs (either by the user or by a program decomposition incurred by the user's annotations, see Section 2.4). Process cooperation is described by a simple macro dataflow specification, called a *configuration*. Configurations support programming-in-the-large. The user can experiment with various configurations to find the one that results in the most efficient code.

The configurator uses the dependence graph created during configuration analysis to generate an architecture-independent parallel description which is fed to the code generator. Configurations define processes (and their aggregates) and ports. Statements of the configuration represent relations between ports in different processes. Some of these statements are generated during annotation processing (at the subprogram level, see Subsection 4.2); others must be supplied by the user (to direct how the processes are to be integrated into a computation).

Processes created dynamically can communicate with ports located at parent, child, and sibling processes (each of those processes is just a copy of the same program or program fragment, except the parent process that can be arbitrary).

2.4. Annotations. Annotations provide an efficient way of introducing the user's directives to assist the compiler in program parallelization. To be effective, annotations have to be carefully limited to a few constructs. They also should preserve semantics of the original program. Annotations have been proposed in many systems [**4, 5, 9**] and are used mainly as compiler directives. In contrast, in our approach annotations are used in decomposing a program into fragments. Annotations in the EPL system are introduced solely to limit the feasible allocations of parallel tasks to processors. Hence, programs decorated with annotations produce the same results as unannotated programs.

```
int: n; /* array size */
real: Ain[*,*], U[*,*], L[*,*];
subscript: i, j;

range.Ain = n;  range(2).Ain = n;
range.U[k] = n - k + 1;
range.L[j] = j - 1;

T[i,j] :  A[k,i,j] = if k == 1 then Ain[i,j]
              elsif i == Piv[k] then A[k - 1,k,j] - L[k,k - 1] * U[k - 1,j - k + 1]
                elsif i == k then A[k - 1,Piv[k],j] - L[Piv[k],k - 1] * U[k - 1,j - k + 1]
                else A[k - 1,i,j] - L[i,k - 1] * U[k - 1,j - k];
D[j] :    L[j,k] = if j == k then 1
              else A[k,j,k]/U[k,1];
D[j] :    U[k,j - k] = A[k,Piv[k],j];
D[j] :    Piv[k] = scan(max, abs(A[k,j,k]), j : j >= k);
```

FIGURE 1. LU decomposition of a matrix A in EPL

In EPL, each equation can be annotated with the name of an array of virtual processors on which it is to be mapped. Virtual processors can be indexed by the equation's subscripts to identify instances of equations assigned to individual virtual processors. Such instances constitute the smallest granule of parallel computation. An example of the use of EPL annotations in a program for the LU decomposition of a matrix is shown in Figure 1. The program follows directly the standard definition of LU decomposition with pivoting [**10**]. The input matrix Ain is decomposed into two matrices: lower triangular L and upper triangular U.

The program operates on the sequence of matrices $A[i, *, *]$ starting with the input matrix Ain. As discussed in [**10**], the computation can be partitioned into the following tasks:

- **diagonal:** The n tasks $D[j]$ that operate on the diagonal of the matrix A.
- **subarray:** Tasks $T[i, j]$ that operate on submatrices of the matrix A.

The presented annotation imposes this partitioning.

3. EPL Compiler

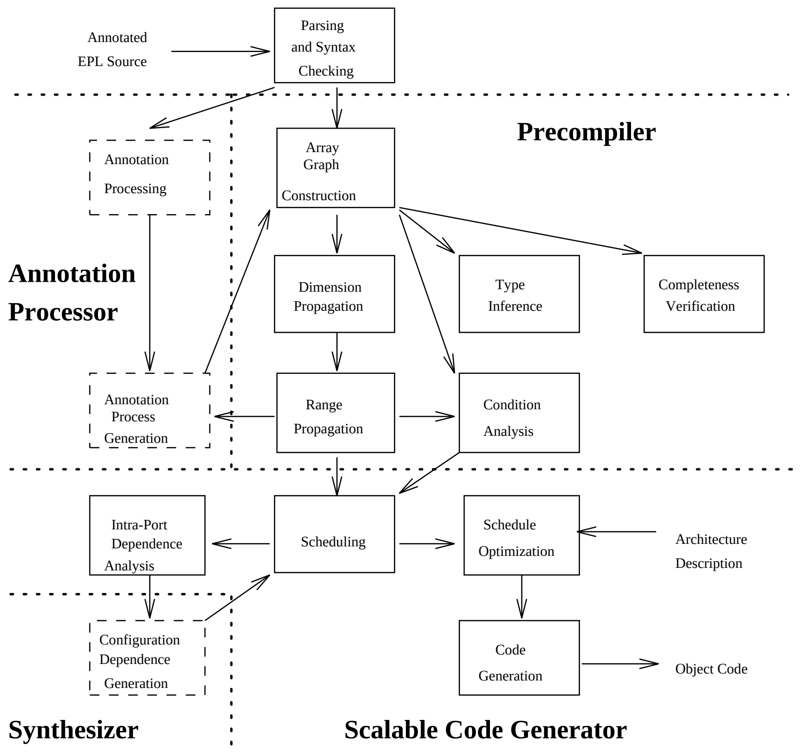

FIGURE 2. The structure of the EPL compiler

The basic techniques used in EPL compilation are data-dependence analysis and data-attribute propagation. In a single program, the dependencies are represented in the compact form by the *conditional array graph*. A similar dependence graph is also created for a configuration. It shows the data dependencies among processes of the computation and is used for scheduling processes and mapping them onto the processors. Figure 2 depicts the structure of the EPL compiler, grouping the components into the following parts (discussed later): Precompiler, Annotation Processor, Synthesizer, and Scalable Code Generator. The stages of the EPL compilation are:

(i) *Array Graph Construction* which transforms the source code into its intermediate form. The main components of this form are the array graph and the symbol table. The array graph nodes represent the variables and the equations. Each array graph edge represents the dependence between the nodes and is labeled by its attributes such as the associated subscript expressions, dependence type, and conditions under which the dependence holds.

(ii) *Dimension Propagation* that checks correctness and assigns dimensionality to each EPL variable.

(iii) *Type Checking* which verifies that all variables and expressions have or can be assigned consistent types.

(iv) *Completeness Verification* that performs various semantic checks and verifies that each variable is defined over its entire domain.

(v) *Range Propagation* that finds equivalences between ranges of variables and equations. The EPL compiler uses the concept of a range set as an object to which all equivalent ranges are linked. Range propagation links all dimensions which share a common bound into a range set.

(vi) *Condition Analysis* which establishes equivalence and/or exclusiveness of predicates used in conditional equations. The found relations of predicates are used in scheduling and verification.

(vii) *Scheduler* that finds an array graph evaluation order which is minimal among all orders preserving the program semantics. Scheduler also defines the scopes and nesting of the loops in the object program. The output generated by the scheduler is used by the schedule optimizer and the code generator.

(viii) *Schedule Optimization* is an architecture-dependent step that customizes the generated schedule to the target architecture (see, for example, **[11]** for SIMD specific optimizations).

The remaining stages are discussed below: *Annotation Processing* in Section 4.2, *Configuration Processing* in Section 4.3 and *Code Generation* in Section 3.4.

3.1. Conditional Array Graph. The conditional array graph is defined as

$$G = (V_G, E_G, M_G)$$

where $V_G = \{v_1, \ldots, v_n\}$ is a set of nodes, $E_G \subseteq V_G \times V_G$ is a set of edges, and $M_G : L_G = \{l_1, \ldots, l_m\} \rightarrow V_G \cup E_G$ is a labeling function. Each node v_i represents one of the following:

- **Equation:** Each equation from the program creates one node.
- **Variable:** Any variable that appears in equations is represented by a node in the graph (i.e., primitive data structures, sublinear subscripts

and implied data structures created by prefixing variable names with keywords like `range`).

- **Interface structure:** A structure declared in the program as a part of an input or output file.
- **File:** Each file creates two nodes: representing respectively an opening and closing of the file.

Generally a node in the array graph represents a program activity, e.g., record nodes indicate a read or write operation, equation nodes represent the evaluation of the equation. Nodes that do not trigger an activity (data nodes and sublinear subscripts) generally indicate a stage in the computation. For example, data nodes indicate that a field has been defined and is available for access.

An edge $e_{i,k}$ drawn between nodes i,k can be of one of the following three types:

- **Hierarchical:** A dependence that is implied via declared structures. For example, in input structures, a record node is implied to be dependent on the file node from that structure. This dependence documents the fact that a record cannot be read until the file has been opened.
- **Data:** A dependence that is derived from equations. The node for the data element being defined is made to be dependent on the equation's node and the equation node is, in turn, made dependent on the data nodes for the data elements used in its definition (appearing on the right side of the equation or in a subscripting expression on the left side).
- **Parameter:** A dependence implied by the presence of a range definition for the EPL variable.

A labeling function M defines a label for each node and edge in the graph. The node label $l_i = M(v_i)$ 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 conditions under which the dependence associated with the edge holds, the type of the dependence, and subscripting information.

Typically, data flow analysis in procedural compilers ignores conditions guarding the control flow paths; however, recognizing equivalent or exclusive guarding conditions improves the checking and optimization capabilities of the compiler. In the functional programming environment, such recognition can be done efficiently (at least for some common cases) because of the simple semantics of the underlying language. The analysis of the conditional dependencies involves traversing various computational paths in the equational program and collecting a conjunctive set of inequalities for each path. When the conjunction of the conditions is not satisfiable, the dependencies labeled by these conditions never holds and therefore can be ignored in further analysis. We have designed and implemented algorithms for conditional dependence analysis that are polynomial in the size of the source program [**2**]. This is a significant improvement over methods applied in conventional languages that are exponential in the input code size. The advantages of performing condition analysis are threefold:

- More efficient code is generated. Parallelism can be exposed to a greater degree if some dependencies are eliminated. In addition, the equivalent conditions are tested only once and the code dependent on them is consolidated into one block.
- Storage use is optimized. The EPL compiler uses the windowing technique for representing the large dimensions through a small-sized window in memory even in cases involving fairly complex subscript expressions.
- Program verification is improved. The single assignment rule can be enforced by checking satisfiability of the conjunction and disjunction of conditions in several equations defining the same variable. Verification of noncircularity of variable definitions is also refined in a similar manner.

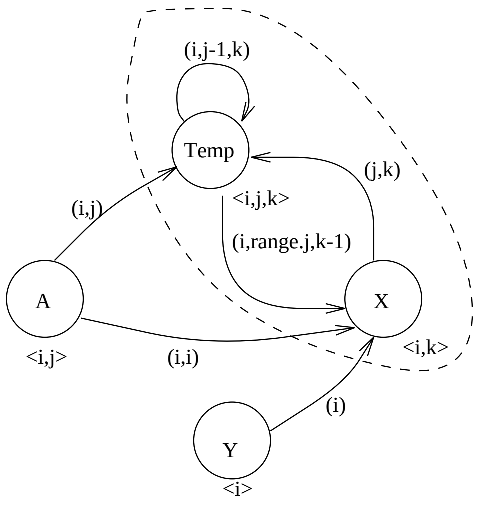

FIGURE 3. Array graph for Jacobi iteration

3.2. Memory Optimization and Scheduling. Programs written in EPL obey the single assignment rule. A variable which is reassigned in a procedural language is seen as a vector of values with a different subscript in each assignment. This extra temporal dimension allows the program to be specified without any reassignments but, unless optimized, may require an exorbitant amount of memory. The problem is important for any functional language and the solution presented here is a general one, not restricted to EPL. The essence of optimization is to replace some of the dimensions of a variable by *windows* which hold only a few elements of each windowed dimension at a time. Returning to the example of Jacobi's linear solver defined by Equation (2.1), it is easy to notice that a procedural program (for example in Fortran) will require memory proportional to the problem size n. However, a naively implemented EPL program using variable $Temp$ would require memory proportional to n^2, quite unacceptable overhead. Figure 3 shows the array graph for the Jacobi iteration. The dotted line in Figure 3 encloses a strongly connected component, SCC, i.e., a maximal subgraph of the array graph such that for any pair of vertices v, w in SCC there are directed paths from v to w and from w to v. Consider the SCC consisting of nodes $Temp$ and X. Elements of $Temp$ are used to define elements of X and vice versa. If the loop on k proceeds in the forward direction, the edge with label $(i, range.j, k-1)$ is redundant, because all elements computed at $(k-1)$-th iteration are available at the k-th iteration. Removal of this edge renders the array graph schedulable. From the array graph, it is apparent that X and $Temp$ should be scheduled within the same loop on k. Moreover, in any loop arrangement for the Jacobi iteration, we have to assure that the loop on j surrounding the equation that defines $Temp$ is completed before the equation defining X uses those values of $Temp$ (as implied by the edge $(i, range.j, k-1)$). All these restrictions allow only five loop arrangements for the Jacobi iteration as shown in Figure 4.

Careful analysis of the array references reveals that for a problem of size $n = 500$, the optimized code will require about 1000 location for a window of size 1 for array $Temp$ and a window of size 500×2 for array X. Even more revealing are the communication requirements of different solutions. In terms of local memory requirements, the first, third and fourth loop arrangements are the best and are all equivalent. However, if arrays X and $Temp$ are evaluated on separate processors, the third solution is preferable since only a single element of Temp is needed to evaluate a new value in the vector of X. This kind of consideration can easily be included in compiler optimization by assigning greater penalty to global communication delay than to allocated memory.

The execution ordering of the statements in a functional program is determined by a recursive procedure called the *scheduler* [**20**]. At each level of recursion, the scheduler linearly orders all SCCs in the array graph of the program to satisfy the existing data dependencies. It then attempts to schedule each SCC by enclosing it in such loops that dependencies represented by some of the edges inside the SCC are enforced by the loop execution. If successful, the scheduler removes the loop-enforced edges from the SCC thereby creating a cascade of new SCCs (components of the parent SCC) that can be ordered linearly. Removal of the redundant edges at each level defines the loops that should surround all statements represented by nodes within the parent SCC.

Each level of recursion produces one or more of such loops. Loops obtained for an SCC M are always nested within those obtained in a previous recursion for the SCC that contains M. By analyzing all SCCs at each level of recursion, it is possible to obtain the loops that should surround each statement and the partial order of the loop nests. In [**14**] we have proven that the loop arrangement problem is NP-hard by reducing it to the weighted clique problem. The EPL compiler uses an heuristic to find a suboptimal solution. Once a loop cluster is selected, a simple linear algorithm can be used to find the best loop arrangements

```
Loop on $k$
    Loop on $i$
        $X[i] = (Y[i] - Temp[i])/A[i,i];$
    Loop on $i$
        Loop on $j$
            $Temp[i] = Temp[i] + A[i,j] * X[j];$
            /* Window on $X$: [500, 1] on $Temp$: [500, 1, 1] */
```

```
Loop on $k$
    Loop on $i$
        $X[i] = (Y[i] - Temp[i, range.j])/A[i,i];$
    Loop on $j$
        Loop on $i$
            $Temp[i,j] = Temp[i,j-1] + A[i,j] * X[j,k];$
            /* Window on $X$: [500, 1] on $Temp$: [500, 500, 1] */
```

```
Loop on $k$
    Loop on $i$
        Loop on $j$
            $Temp = Temp + A[i,j] * X[j,k];$
        $X[i,k+1] = (Y[i] - Temp])/A[i,i];$
            /* Window on $X$: [500, 2] on $Temp$: [1, 1, 1] */
```

```
Loop on $k$
    Loop on $i$
        Loop on $j$
            $Temp[i] = Temp[i] + A[i,j] * X[j];$
    Loop on $i$
        $X[i] = (Y[i] - Temp[i])/A[i,i];$
            /* Window on $X$: [500, 1] on $Temp$: [500, 1, 1] */
```

```
Loop on $k$
    Loop on $j$
        Loop on $i$
            $Temp[i,j] = Temp[i,j-1] + A[i,j] * X[j];$
    Loop on $i$
        $X[i] = (Y[i] - Temp[i, range.j])/A[i,i];$
            /* Window on $X$: [500, 1] on $Temp$: [500, 500, 1] */
```

FIGURE 4. Different implementations of Jacobi iteration in a procedural language

for the selected loop cluster and the best windows for variables enclosed in it [**14**].

3.3. SCC Scheduling in Presence of Conditional Dependencies. As discussed above, the computation represented by an SCC is enclosed in a set of loops. The scheduler can ignore any dependencies that are labeled with the subscript expression of i-k (i is a subscript or sublinear subscript, and k$\geq$ 0 is a constant) for loops with an increasing loop control variable, and i+k for loops with the decreasing loop control variable. Such an elimination of certain dependencies may decompose an SCC into parts that can again be enclosed into loops and decomposed further. In the case of SCCs with both types of subscript expressions present in an SCC, more subtle analysis is needed. An SCC with diverging subscripts (henceforth referred to as diverging SCC) recursively defines some variable(s) in different evaluation directions for those subscripts. Part of the SCC requires an ascending loop in certain dimensions, while another part requires a descending loop in the same dimensions. The EPL compiler uses the condition analysis to verify that the computation can be split into loops in different directions and that these loops can be scheduled in such a way that the recursion refers to instances of the variable(s) that have already been calculated.

Such scheduling requires that the diverging SCC be split into separate SCCs that share common data. All subscript expressions for the dimension being scheduled have to be either i and i-k or i and i+k in each split SCC. For each variable defined in the diverging SCCs, the EPL compiler builds a *defined area* structure. This structure is a linked list of upper and lower bound expressions. The existence of a defined area structure indicates that part of the dimension between the upper and lower bound expression in the structure (including the upper and lower bound) is defined.

The compiler first identifies initialization areas of the dimension that are not defined recursively and then enters them into the defined area structure. Next, it repetitively looks for diverging SCCs with recursive equations that border on the defined areas (those are identified by their condition sets). These SCCs are scheduled and the sections of the dimension that they define are added to the defined area structure.

Each diverging SCC defines a subsection of an array variable that is an *entry point* node for this SCC. The expressions for the bounds of the area being defined are derived directly from the equation conditions within the SCC. In each diverging SCC, there is a conditional equation defining the entry point node, called an *alternate equation*. Part of the condition in the alternate equation must specify bounds for the subscript, called the *bounding predicate*. The condition may be a disjunction of the Boolean terms, but it must have a common subscript bound expression across the disjunct terms. The bounding predicates are extracted from the equation's condition and create the bounds for the diverging SCC. The bounds for the initial defined areas are derived from the nonrecursive equation (not appearing in the SCC but attached to an entry point node) in the same manner.

The compiler also identifies the depth of the recurrence. Part of the diverging SCC with a negative (positive, respectively) depth is considered schedulable when its minimum (maximum, respectively) bound, offset by its depth, falls within a defined area.

The diverging SCCs arise naturally in programs that describe phenomena which spread outward from the defined initial region. Definitions of this type can be found in simulations of the protein folding [**2**], fractals, growth of crystals, shock waves, etc. A natural way to describe such behavior is to define the outward spread of the values in terms of conditional relations between discrete structures or between the elements of multidimensional structures. The conditions of the relations dictate the direction of the spread. If the conditions for the dependencies were ignored, the dependencies modeling such behavior would appear to be cyclic (due to equations that would define the values as spreading in conflicting directions).

3.4. Code Generation for Massively Parallel Computers. Data structures used in scientific computation can be viewed as a function δ from an *index domain* ID to a *value domain* VD. An *index domain*, in general a set of tuples of integers $< i_1, i_2, \ldots, i_n >$, is often a subset of the Cartesian product of integer intervals, for regular n-dimensional arrays. For example, ID $= I_1 \times I_2 \times \ldots \times I_n$, where $I_j = [1, I_{max,j}]$. Often an inverse function δ^{-1} does not exist. Following the standard higher-level programming language notation, we denote the value of the function δ at point $< i_1, \ldots, i_n >$ as $v[i_1, \ldots, i_n]$.

Program execution can be seen as an evaluation of the arrays at various index points (elements of the index domain). The order of execution is restricted only by data dependencies that rarely impose the total order.

Figure 5 shows the conceptual stages of mapping the index domain of a variable to the processor domain, the memory domain and the time domain. The goal is to find a mapping that results in the minimum execution time. In Figure 5, VPD represents a *virtual processor domain*. It is defined by the computer interconnection network. For example, in a k-dimensional mesh-connected architecture of size N, processors can be thought of as arranged in a k-dimensional array, with $VPD = [1, n_1] \times [1, n_2] \times \ldots \times [1, n_k]$, where $N = n_1 * n_2 * \ldots * n_k$. The processor $p[l_1, l_2, \ldots, l_k]$ is connected with processors p$[l_1, \ldots, l_j \pm 1, \ldots, l_k]$, $1 \le j \le k$ provided that processor p$[l_1, \ldots, l_j \pm 1, \ldots, l_k]$ exists ($l_j \pm 1 \bmod n_j$, in the case of torus-connected architecture). To facilitate data alignment and time scheduling, we assume that a virtual processor domain VPD is compatible with the index domain ID. *Local memory domain* L can be viewed as a multidimensional cube with the volume equal to the actual local memory available on each processor. Virtual processors in VPD has local memory of the same structure as the domain L, except each is of unlimited size. The execution time steps are represented by *time domain* $T = [1, t_{max}]$, where t_{max} is the total number of

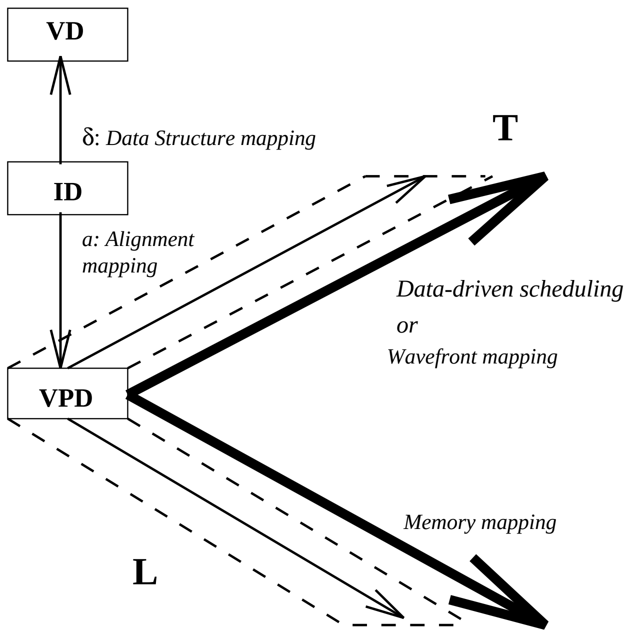

FIGURE 5. Functional view of code generation

time steps needed to complete the computation.

In such a view, there are three major mappings that need to be found to generate optimized code for massively parallel architectures: *Alignment Mapping, Time Mapping* and *Memory Mapping.*

Data alignment challenge is to select such a mapping α of index domains onto the virtual processor domain that communication of array operation arguments is minimized. Data alignment performed by the EPL compiler was presented in [**16**] and is not discussed here due to the space limitation. Time mapping of iterative computations is usually done either through *data-driven scheduling* or *wavefront mapping* [**15**]. Both methods explore the fact that iterative computations often allow the simultaneous evaluation of many array elements. Data-driven scheduling starts the execution of an index point as soon as all data that this point is dependent on becomes available. However, data dependencies often hold under conditions that involve input data and therefore can be resolved only in run-time. Consequently, data-driven scheduling typically relies on run-time distributed synchronization. In the case of functional programs with single assignment and recurrent relations, the compile-time data-driven scheduling is decidable [**13**]. Such a scheduler has been implemented in the compiler for EPL language and is discussed in Section 3.3. As discussed in Section 3.2, the EPL scheduler determines also a memory mapping. Wavefront scheduling for EPL programs was presented in [**15**] and is not discussed here due to the space limitation.

4. Parallelism in EPL

4.1. Levels of Parallelism. In EPL, compile-time parallelism is sought on three levels:

(i) **Task Parallelism** is dictated by the process definitions and their interconnections into a configuration. The configuration graph (the graph obtained by representing processes as nodes and port interconnections as edges) is decomposed into parallel tasks by the EPL compiler. **Send** and **Receive** operations necessary for process coordination are also generated. Since the optimal decomposition is NP-hard for multiprocessors [**1**], the EPL compiler uses an heuristic described in [**12**].

(ii) **Loop Parallelism** is sought at the level of equation clusters. Each recurrence equation can be annotated by a process name to assist the compiler in parallelizing computation across the subscripts. This is very similar to the loop parallelism in imperative languages like Fortran. Within the EPL framework, separate processes are generated by the compiler for each equation cluster. To minimize interprocess dataflow, the compiler uses an heuristic to impose an hierarchy among the generated processes. The details of these heuristics are given in Section 4.2.

(iii) **Data Parallelism** is explored at the level of instances of equations or their clusters [**11**]. This source of parallelism is of the greatest importance in SIMD architectures. Mapping of arrays onto the processors dictates communication costs of fetching the arguments and storing the results of operations. The problem of finding the mapping optimal in that respect is known as the *data alignment problem* and is discussed in relation to EPL programs in [**16**]. The execution order of array element evaluation is important for SIMD code efficiency. A compile-time method of defining such an order, known as *wavefront determination*, is discussed in [**15**].

Task parallelism relies on the existence of separate, relatively independent processes or functions that can be executed simultaneously. In the traditional approach, the user is required to handle the error-prone and difficult task of synchronizing these independent processes. The configurator eases the burden of harnessing task parallelism by automating the definition of interprocess coordination.

Data parallelism, popular in massively parallel systems, relies on large data structures to be processed and assigns individual elements of such structures to a single processor (either virtual or real). The same sequence of instructions is applied simultaneously to all elements of the processed structures. It is also necessary to decide which elements of the different structures should be placed on the same processor in order to minimize the cost of fetching arguments for operations involving those elements.

Annotations, relevant to both task and data parallelism, provide the user with the means of rapid-prototyping alternative parallelizations of the program. For example, supplying proper annotations, the user can experiment with various combinations of column- and row-wise parallelizations of the matrix programs.

4.2. Annotation Processing. Annotation processing includes:

- creating parallel tasks defined by annotated fragments of the original program,
- declaring ports needed to interconnect created tasks into a network,
- interconnecting ports according to the task communication graph to preserve data dependencies between created tasks.

Each annotated fragment of the source program becomes a separate task. All data elements defined in the task are local to it[1]. All needed non-local data have to be received from the other tasks. The annotation processor analyzes the flow of data incurred by the decomposition of the source program and builds a corresponding representation of this flow in the form of the task communication graph. Then, the annotation processor supplements the code of each task by port declarations and send and receive statements that are needed to implement the required intertask data flow. To minimize the communication generated by the added statements, the annotation processor embeds a tree in the task communication graph.

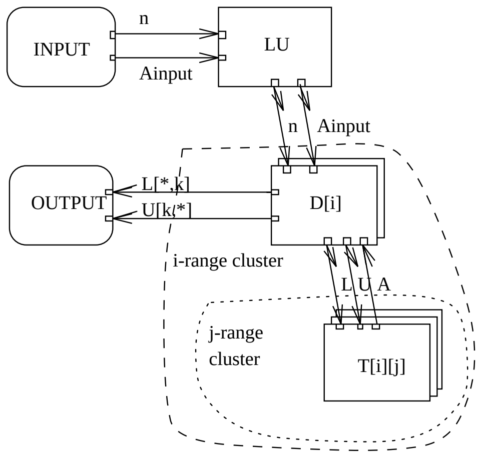

FIGURE 6. Communication tree for EPL program

We developed heuristics [**12**] which select the embedding that satisfies the following criteria:

[1]We refer to this principle as *Executor Owns* rule. It is an inverse of the more commonly used *Owner Computes* rule. In [**16**] we have shown an example of computation for which neither of the rules yields an optimal solution.

- **Dimension nesting:** If two tasks with different number of dimensions are connected in the task communication graph, the task with more dimensions should be located lower in the spanning tree. If, for example, tasks $T[i, j]$ in Figure 6 were located above the tasks $D[j]$ in the spanning tree, the addressing and creation of child tasks in T would involve executing an **if–then** statement in all $i * j$ T tasks.
- **Range nesting:** Whenever possible, tasks sharing the same range should be clustered together in the spanning tree. Variables that share ranges tend to appear in the same equations. Thus, clustering such variables together decreases the number of cross-process references to distributed variables.
- **Data flow:** The total communication cost of the selected spanning tree should be the smallest among all spanning trees satisfying the above two criteria.

A tree created from annotations of the LU decomposition program from Figure 1 is shown in Figure 6. The double-outgoing arrows indicate a broadcast of messages (from a task to a group of tasks) and double-incoming arrows correspond to the inverse operation of gathering the data.

4.3. Configuration Processing. The goal of configuration processing is to establish scheduling constraints for the overall computation and to synthesize a parallel computation from the decomposed parts. The latter task is performed by the EPL compiler part called *synthesizer* which generates codes for invoking and synchronizing parallel tasks.

In a parallel computation, individual process correctness is a necessary but not sufficient condition for the correctness of the entire computation. If a task has input and output ports that belong to a cycle in the configuration graph, then this task's input messages are dependent on the output messages. Such dependencies (in addition to dependencies imposed by the statements of a task) have to be taken into account in generating the object program for individual tasks; otherwise, loss of messages, process blocking, or even a deadlock can arise.

Tasks that belong to a cycle in the task communication graph can execute concurrently only if they are all enclosed in the same loop including the respective send and receive statements. Such tasks are called *atomic*, since they cannot be broken into parts without splitting the loop. For example, if a send statement is executed in a separate loop from the corresponding receive, then all messages must be sent before any one can be received. The successors of a such nonatomic task cannot start until its predecessors in the task communication graph finish sending all messages.

The algorithm for finding external data dependencies has been presented in [**19**]. The analysis starts by inspecting all atomic processes and then propagates transitive dependencies along the paths of the task communication graph restricted to atomic processes. As a result, a *configuration dependence* file is created and later used by the synthesizer and the code generator. This file contains a list of the additional externally-imposed data dependencies (edges and their dimension types) that need to be added to the task array graph. One task may have several such files, each associated with a different configuration in which the task participates.

Each edge in the configuration dependence file may have the following effects on the program generated from the array graph:

- an additional constraint is imposed by an edge if there is no equal or stronger internal dependency between the considered nodes, or
- an error is discovered when there are internal dependencies incompatible with the edge.

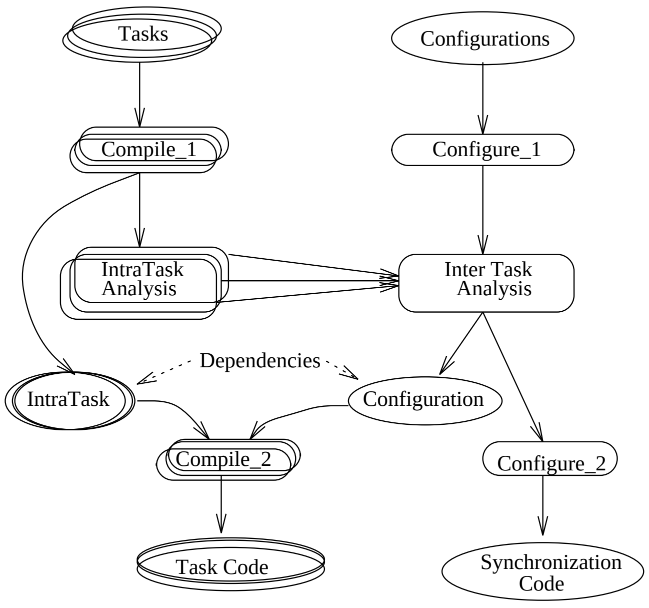

FIGURE 7. Two stages of dependence analysis

Hence, as shown in Figure 7, the dependence analysis for the synthesized computation has to be done in two stages.

5. Conclusion and Comparison with Other Approaches

In this section we characterize EPL in terms of criteria that identify important properties of parallel languages [**18**].

Architecture Independence. The same source code is used by the EPL compiler to produce different parallel executables for different architectures. Currently, EPL compiler includes code generators for MPL and C* languages for SIMD architectures (MasPar and CM-200), Dynix C for the shared-memory Sequent Balance, and message-passing C for the Stardent computer. There is ongoing work on C code generators for the CM-5 and SP1 architectures. Netherthe-less, the user may still prefer to use different annotations or even different EPL programs for different architectures to achieve the optimal performance.

Parallelism Specification. A high-level language should shield the user from having to specify each and every detail of parallel execution. Below we discuss the level of user involvement in defining the parallel execution of EPL programs.

- specifying *data and program decomposition*
  Only a partial specification is expected from the user. An EPL computation consists of cooperating functional processes that define an initial decomposition of the program. Parallel tasks are created by the EPL system through merging and splitting EPL processes based on the communication-to-communication ratio on the target architecture. The programmer can use explicit annotations to define the part of an EPL process that is to be assigned to a single virtual processor. The annotations define the lower limit on the granularity of decomposed tasks to improve the efficiency of generating program decomposition. If, during the process decomposition, a task is created that includes all computation designated to some virtual process, this task will not be further divided by the EPL system.
- specifying *mapping*
  Mapping of the parallel task (created from processes by the EPL system) to the physical processors is done entirely by the EPL system. However, the quality of the mapping is decided by the quality of the decomposition which, in turn (see point above), is partially defined by the user who defines the EPL processes.
- defining *communication*
  At each process description there is no difference between communication and regular input/output; both are seen as externally provided input to the process. The necessary communication code is generated by the EPL compiler.
- defining *synchronization*
  Again, the user is shielded from this aspect of parallel programming. The synchronization generated by the EPL compiler is derived from the data dependency imposed by the EPL processes.

Software Development Methodology. EPL relies on functional decomposition of the computation into processes. Processes are described in an equational language and their cooperation is described as a configuration. Programs describing processes are compiled by the EPL compiler and a configuration is processed by the configurator, i.e., the compiler for the configuration language. Hence, there is a separation of programming-in-the-large from programming-in-the-small. The process written as a functional program may be refined by user-supplied annotations. The parallel code is generated through a series of transformations. First, the flow of control is established and minimum synchronization necessary for preserving program's correctness is found (in EPL terms, a schedule of a process is created), which is still architecture-independent. Then, the decomposition and mapping takes place (creating another equivalent form of the source program). Finally, input/output and communication statements specific to the target architecture are generated and the final parallel code is produced.

- structure of the development process
  In EPL, the equational program for a process is written very independently from the programs of other processes. Only clearly defined interfaces (data structures exchange with the environment) are of concern for the process program writer.
- exposition of the decision points
  Preparing a configuration for the overall computation forces the user to decide on the method of writing the program at the global level without considering low-level details.
- record of constructs
  Thanks to their conciseness and lack of implementation details (i.e., input/output, communication, flow of control), computation configuration and equational programs for its processes form a good basis for program documentation.
- preservation of correctness
  The parallel code is produced in three major transformations that were designed to be correctness-preserving.
- limit of proofs to derivation system
  Proof of the correctness-preserving properties of the EPL transformation has not been made formally, however these properties strongly influence their design and implementation.

Cost Measures. There is a part of the system, called the Timer, that provides the user with the execution time estimates for equational programs. As in [**8**], the Timer relies on a set of simple architecture measurements that can be established by running benchmarks of the Timer on the given architectures. However, we do not have a mechanism for determining the overall computation execution cost (i.e., execution cost at the level of a configuration) at this time. For SPMD models, Timer is sufficient; however, in a more general setting, there is a need for a better tool. Timer results are used in transformations of equational programs into schedules and during program decomposition and mapping.

No Preferred Scale of Granularity. There is no upper or lower limit on the grain size in EPL with the exception of the statement instance; i.e., EPL does not explore parallelism on the level of expressions and below.

Efficiently Implementable. Our experience with the current EPL implementation indicates that the EPL generated code is no more than 20%-50% slower than the equivalent hand-written code. However, we have not yet measured the efficiency of larger applications (or even a large number of smaller ones).

Usually a large parallel computation can be efficiently designed as a set of interacting processes which represent logical partition of the problem. Each process is typically further functionally decomposed into procedures and subroutines. A hierarchical view of a parallel computation supported by the macro-data flow EPL configuration is helpful in extracting task parallelism. Program decomposition based on annotations and computation synthesis based on configurations can support efficient parallel code generation for this kind of parallelism. In addition, annotations support rapid prototyping and performance tuning of a parallel computation.

Using functional paradigm for EPL process specification results in absence of control statements in EPL programs. Hence, functional paradigm simplifies program analysis and enhances compiler ability to produce an efficient code. However, majority of parallel code optimization problems are NP-hard; hence, development of proper heuristics is important.

References

1. S.H. Bokhari, *A shortest tree algorithm for optimal assignments across space and time in a distributed processor system*, IEEE Trans. Software Engineering, **SE-7** (1981).
2. J. Bruno, *Analyzing Conditional Data Dependencies in an Equational Language Compiler*, Ph.D. Thesis, Rensselaer Polytechnic Institute-Troy, 1989.
3. M. Chen, Y. Choo and J. Li, *Crystal: theory and pragmatics of generating efficient parallel code*, Parallel Functional Languages and Environments (B. K. Szymanski, ed.), ACM Press, New York, 1991, pp. 255–308.
4. B. M. Chapman, P. Mehrotra, and H. P. Zima, *Vienna Fortran - a Fortran language extension for distributed memory multiprocessors*, Languages, Compilers and Run-Time Environments for Distributed Memory Machines (J. Saltz and P. Mehrotra, eds.), Elsevier, Amsterdam, 1992, pp. 39–62.
5. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng and W. Wu, *Fortran D language specification*, Technical Report COMP TR90079, Rice University-Houston, 1991.
6. A. Gerasoulis and T. Yang, *On the granularity and clustering of directed acyclic task graphs*, IEEE Trans. Parallel and Distributed Systems, **4** (1993).
7. R.E. Gomory and T.C. Hu, *Multi-terminal network flows*, SIAM J. Applied Math., **9** (1961) 551–570.
8. T. Fahringer and H.P. Zima, *A static parameter based performance prediction tool for parallel programs*, Proc. Seventh ACM International Conference on Supercomputing (Tokyo, July 1993), ACM Press, New York, 1993.
9. P. Hudak, *Para-functional programming in Haskell*, Parallel Functional Languages and Environments (B. K. Szymanski, ed.), ACM Press, New York, 1991, pp. 159–196.
10. R. E. Lord, J. S. Kowalik, and S. P. Kumar. *Solving linear algebraic equations on an*

MIMD computer, J. ACM, **30** (1983) 103–117.

11. B. McKenney and B. K. Szymanski, *Generating parallel code for SIMD machines*, ACM L. Programming Languages and Systems, **1**, (1992) 59–71.
12. C. Özturan. *Expressing parallelism in EPL*, Technical Report CS90-29, Rensselaer Polytechnic Institute-Troy, 1990.
13. A. Pnueli, N. S. Prywes, and R. Zahri, *Scheduling equational specifications and nonprocedural programs*, Automatic Program Construction Techniques (Biermann, Guiho, and Kondratoff, eds.), McMillan, New York, 1984, pp. 273–287.
14. B. Sinharoy and B. K. Szymanski. *Memory optimization for parallel functional programs*, Computer Systems in Engineering (to appear).
15. ______, *Finding optimum wavefront for iterative algorithms*, J. Parallel Algorithms and Applications, **2** (1994).
16. ______, *Data and task alignment in distributed memory architectures*, J. Parallel and Distributed Computing, **21** (1994) 61–74.
17. S. K. Skedzielewski, *Sisal*, Parallel Functional Languages and Environments (B. K. Szymanski, ed.), ACM Press, New York, 1991, pp. 105–159.
18. D. Skillicorn, *A Model for Practical Parallelism*, Cambridge University Press, Cambridge, U.K. (to appear).
19. K. Spier and B. K. Szymanski, *Interprocess analysis and optimization in the Equational Language Compiler*, Proc. CONPAR-90, Lecture Notes in Computer Science, Springer-Verlag, Berlin and New York, 1990.
20. B. K. Szymanski, *EPL - parallel programming with recurrent equations*, Parallel Functional Languages and Environments (B. K. Szymanski, ed.), ACM Press, New York, 1991, pp. 51–104.
21. B. K. Szymanski and N. S. Prywes, *Efficient handling of data structures in definitional languages*, Science of Computer Programming **10** (1988) 221–245.

Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180

E-mail address: szymansk@cs.rpi.edu