

Compiler Technology for Parallel Scientific Computation

Can Özturan, Balam Sinharoy¹ and Boleslaw K. Szymanski

Department of Computer Science, Rensselaer Polytechnic Institute²

Troy, New York 12180-3590, USA

Abstract

There is a need for compiler technology that, given the source program, will generate efficient parallel codes for different architectures with minimal user involvement. Parallel computation is becoming indispensable in solving large-scale problems in science and engineering. Yet, the use of parallel computation is limited by the high costs of developing the needed software. To overcome this difficulty we advocate a comprehensive approach to the development of scalable architecture-independent software for scientific computation based on our experience with Equational Programming Language, EPL.

Our approach is based on a program decomposition, parallel code synthesis and run-time support for parallel scientific computations. The program decomposition is guided by the source program annotations provided by the user. The synthesis of parallel code is based on configurations that describe the overall computation as a set of interacting components. Run-time support is provided by the compiler-generated code that redistributes computation and data during object program execution. The generated parallel code is optimized using techniques of data alignment, operator placement, wavefront determination and memory optimization.

In this paper we discuss annotations, configurations, parallel code generation and run-time support suitable for parallel programs written in the functional parallel programming language EPL and in Fortran.

1 Introduction

With a constant need to solve scientific and engineering problems of ever-growing complexity, there is an increasing need for software tools that provide solutions with minimal user involvement. Parallel computation is becoming indispensable in the solution of the 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 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. Parallel program correctness requires the results to be independent of the number and speed of the processors. This requirement can be satisfied only if the parallel tasks are independent of each other or properly synchronized when a dependence exists. Designing proper synchronization is a major source of difficulty in ensuring parallel program correctness. 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. Poor portability of parallel programs has resulted in a duplication of efforts and has limited the use of developed systems.

Parallel computation can be viewed as an interwoven description of operations that are to be applied to data distributed over the processors, and of data mapping and synchronization that dictate the data movements and the computation order. The traditional programming languages, such as Fortran, C, or C++, cope well with the task of prescribing operations to be performed. However, the description of data mapping and synchronization in such languages is often introduced by ad

¹Currently with IBM Corporation, P.O. Box 950, Poughkeepsie, NY 12602.

²The authors can be reached through the following e-mail addresses: ozturanc@cs.rpi.edu, balaram@vnet.ibm.com and szymansk@cs.rpi.edu.

hoc architecture-dependent extensions. Examples are various synchronization constructs, like 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. To avoid such architecture-dependent language definitions, we propose to separate the description of operations to be performed on the data values from the definition of data mapping and synchronization needed to supply these data values to the proper processor at the proper instance of the program execution.

With this goal in mind, we developed tools [1, 2] that (i) decompose, at least partially, the parallel program into the two (nearly) orthogonal parts described above, (ii) translate the necessary data movements into optimal form customized for the target architecture, and (iii) synthesize an overall parallel computation. Using these tools the user can describe high-level features of a program and synthesize parallel computation from numerical algorithms, program fragments, and data structures that are implemented separately. The tools support (i) parallel task generation and their allocation to the processors, (ii) distribution of data to the processors, (iii) run-time optimization, and (iv) rapid prototyping of different parallel implementations.

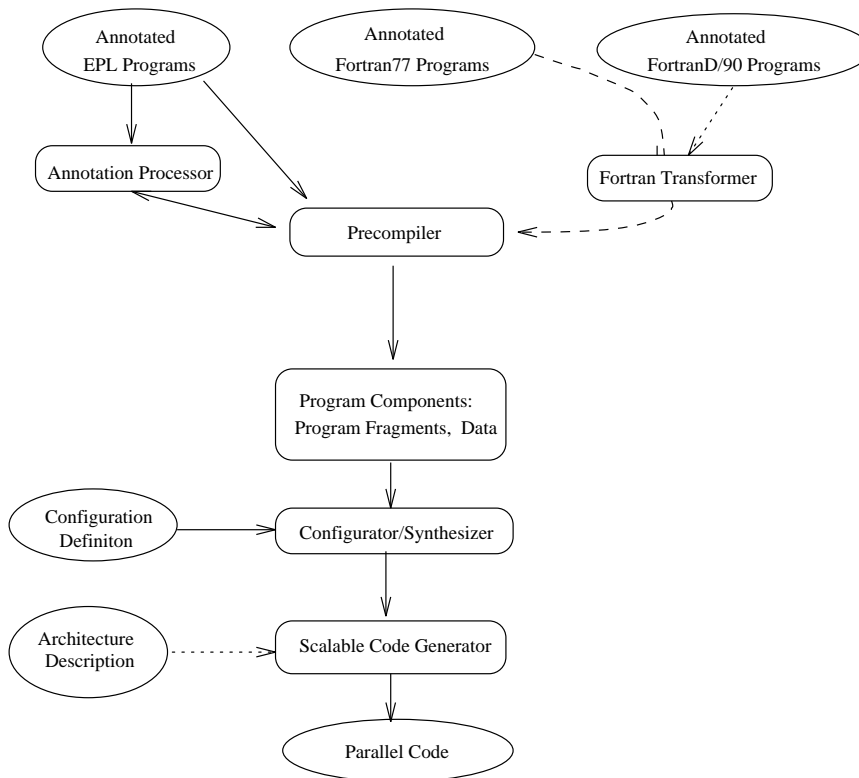


Figure 1: Software tools and their uses

Through the application of transformation techniques, different versions of the same program can be generated from decomposed components. The synthesized computation uses load assignment, data distribution, and synchronization appropriate to the size and type of target parallel architecture. The computation synthesis is guided by conditional dependence graphs that represent externally accessible information in each of the synthesized fragments. Usage of conditional information in data flow analysis and parallelization significantly increase efficiency of the generated parallel code.

The summary view of our approach is given in Figure 1. Program components are created by

annotating source programs in Fortran or in the functional parallel programming language EPL [3]. Fortran programs are transformed into an equational form before decomposition. The configuration definition guides the synthesis of the components into a parallel computation. The synthesized computation together with the architecture description is used by the code generator to produce an object code customized for the target architecture. In Figure 1, continuous lines describe system paths that have been implemented, broken lines represent paths currently under development, and dotted lines correspond to paths at an early stage of investigation.

This paper is intended as an overview of the research done towards implementing software tools as envisioned in Figure 1. More technical discussion can be found elsewhere [4, 5, 6, 3, 7, 8, 9, 10].

A brief description of the EPL language, its annotations and configurations is given in Section 2. The relationship of EPL constructs and tools to different levels of parallelism is discussed in Section 3. The EPL compiler is discussed in Section 4. Section 4.4 includes an overview of our approach to scalable parallel code generation. A dynamic load management strategy for adaptive scientific computation on SIMD architecture is the topic of Section 5. Finally, conclusions and comparison to other approaches is given in Section 6.

2 Overview of the EPL Language

EPL is a simple non-strict functional language with a type inference designed for scientific computation. Although computationally vast, scientific computations are typically quite regular both in terms of control flow patterns and employed data structures. The data structures used are usually 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 just a few constructs: generalized arrays and subscripts for data structures, recurrent equations for program specification, ports for process communication, and virtual processors to facilitate mapping of computation onto processors and load balancing.

A computation is viewed in EPL as a collection of cooperating processes. A process is described by an EPL program that consists of only *data declarations* and *annotated conditional equations*. The canonical data structure is a tree with nodes that can repeat and with leaves of primitive types. In its simplest form such a tree can be viewed as a multi-dimensional array; each level of a tree corresponding to a new dimension of the corresponding 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 multiple-assignment object code). Thus equations, though syntactically reminiscent of assignment statements, are best viewed as assertions of equality.

The EPL programmer also defines the process interconnection network (the graph obtained by representing processes as nodes and port interconnections as edges) in the *configuration* file. Processes along with the configuration files are provided by the user to facilitate the compiler in extracting the coarse grain parallelism in the computation by generating processes and inter-process communication constructs. Configurations also allow the programmer to reuse the same process in different computations.

2.1 Iterations

An iteration is a staple of scientific computing. In EPL, iterations are programmed 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 the *operational view*, they 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 indirect indexes, 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 index s defined over the subscript i is sublinear to this subscript if it satisfies the following property:

$$(0 \leq s[1] \leq 1) \text{ and } (s[i] \leq s[i+1] \leq s[i] + 1) \text{ for } i = 1, 2, \dots$$

It immediately follows from this definition that the sublinear subscript $s[i]$ starts with the value of either 1 or 0 and then, with each increase of i , it is either incremented by 1 or kept unchanged. Typically, there is a condition associated with each sublinear subscript. The condition dictates when the subscript increases. This is the way a sublinear subscript is defined in EPL. For example, a sparse matrix S that is a row-major representation of a matrix D can be defined in EPL using a sublinear subscript $col[j]$ as follows:

$$\begin{aligned} \text{subscript: } col \text{ is sublinear } j : D[i, j] \neq 0; \\ S[i, col] = D[i, j] \end{aligned}$$

Sublinear subscripts have 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 at run-time. An example of such a definition is given in Section 5.2.

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 results in a vector of partial results. For example, consider a matrix A multiplied by a vector X with the result placed in a vector r . This operation can be written in EPL as:

```
Temp[i,j] = if j==1 then A[i,j]*X[j] else Temp[i,j-1]+A[i,j]*X[j];
r[i] = Temp[i,range.j];
```

or, even shorter as

```
r[i]= scan(+,A[i,j]*X[j], j);
```

Such operations result in references of the form $V[...range.i,...]$, where *range.i* indicates the range of the reduced/scanned dimension of a multidimensional array V . (In general, the EPL *range* variable prefix denotes the size of its suffix.) The presence of such references in the program is explored by memory optimization and scheduling which is discussed later.

A more detailed description of the language is given in [3].

2.3 Configurations

In our approach a parallel computation is viewed as a collection of cooperating *processes*. Processes are defined as functional programs. 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. They are supplied by the user to direct integration of the processes into a parallel 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, except the parent process that can be arbitrary.

Consider as an example an iterative solver of linear equations $Ax = b$ which uses the following recursion:

$$\begin{aligned} r_k &= Ax_{k-1} \\ x_k[i] &= \frac{b - r_k}{A[i, i]} + x_{k-1}[i] \end{aligned}$$

The first part of the recursion is a matrix vector multiplication which may form a separate process, defined as:

```

process: mvm; in: inf; out: ouf;

file: inf,
  int n, double A[*,*],          /* first record with n*n matrix A */
  record iter[*, double x[*];    /* sequence of records with vector x */
file: ouf,
  record appr[*, double r[*];    /* sequence of result vectors r */
subscripts: i,j,k;

range.A=n; range.A[i]=n; range.x=n; range.r=n;
r[k,i]=scan(+,A[i,j]*x[k,j],j);

```

Figure 2: Matrix vector multiplication in EPL

Note that there are no explicit input/output statements and the order of equations is irrelevant because all variables are singly valued. If we assume that the separate process, let's call it XC, calculates the new approximation of the vector x and monitors convergence and the third process, MAIN, provides final input/output, then the corresponding configuration is shown in Figure 3. The textual definition lists data-flow paths that cover a configuration graph. The graphical definition is built from process boxes and file edges. It is augmented with file structure information provided by the EPL system (see Figure 3 b).

2.4 Program Decomposition through Annotations

Annotations provide an efficient way of introducing the user's directives that assist the compiler in program parallelization. Annotations have been proposed in many systems by various researchers

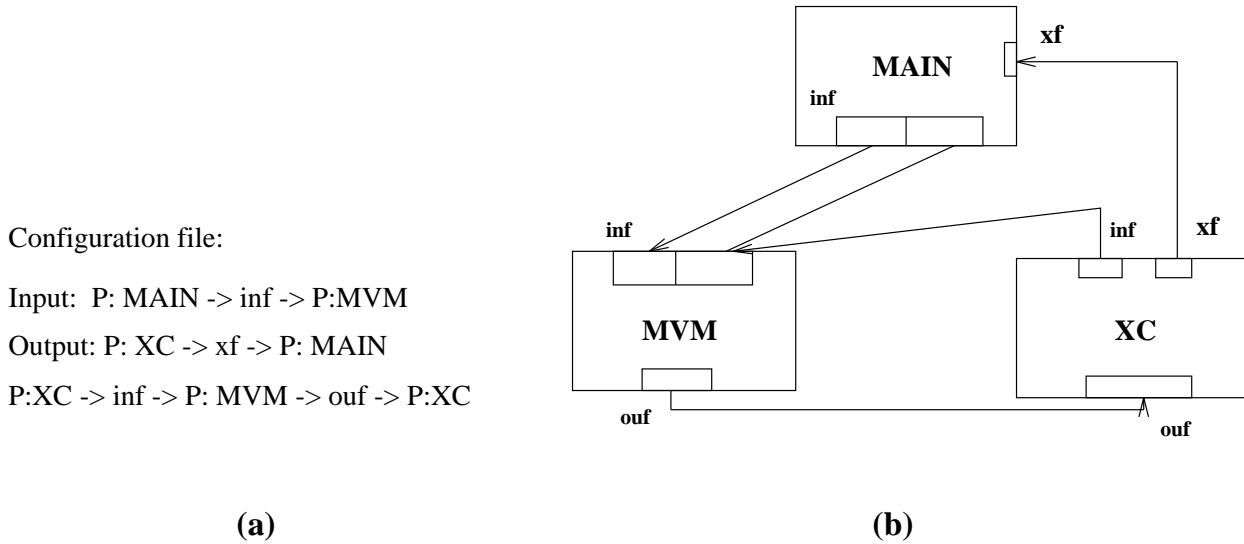


Figure 3: Configuration for an iterative solver in a) textual and b) graphical form

[11, 12, 13, 14, 15] and are used mainly as compiler directives. In our approach annotations limit the feasible mappings of computation onto the processors. Hence, they are used only during the decomposition of a process into smaller fragments. This kind of annotation is similar to `ON` clause as used in the Kali compiler [11], Fortran D [12] or Vienna Fortran [13].

Annotation does not have any effect on the result computed by a program. Consequently, sequential programs that have manifested their correctness over many years of usage are good candidates for parallelization through annotations. Being orthogonal to the program description, annotations support rapid prototyping of different parallel solutions for the same problem, which can be helpful in performance tuning.

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. For example, for the process MVM the following annotation:

$$P[i]: r[k,i]=scan(+,A[i,j]*x[k,j],j);$$

will cause the compiler to consider only the tasks that define a sequence of r vector elements. Each task will locally store one row of array A but the vectors $x[k, *]$ must be broadcast to all of those tasks.

The above partitioning allocates a slice of the equation defined by a single subscript value. The resultant granularity may be too fine for a target architecture. However, when an annotation is indexed by a sublinear subscript, then the corresponding sublinear expression dictates how the annotated equations are clustered onto the virtual processors. For example, let p be a sublinear subscript of i , then $range.p$ is the number of physical or virtual processors. (This number may be a system constant not even known explicitly to the user; it may depend on the architecture, system load, or it may be defined by the user or compiler directive.) Considering again the previous example of a matrix vector multiplication, we can use an annotation:

$$P[p]: r[k,i] = reduce(+,A[i,j]*x[k,j],j);$$

It will distribute (or partition) the last dimension of r and A over $range.p$ processors in a block fashion (each processor will hold $\lfloor \frac{n}{p} \rfloor$ or $\lceil \frac{n}{p} \rceil$ elements of r and rows of A). In Section 5.2 there is an example in which a different distribution is achieved using a sublinear subscript in an annotation. This distribution balances the load on the processors.

There are similarities as well as differences between the EPL annotations and the Fortran language extensions that have been introduced in many systems, e.g., Vienna Fortran [13, 16, 17], Fortran D [12, 18, 19] and SUPERB [20]. Vienna Fortran provides directives for array-like processor structure definition. The distribution of arrays can be specified at compile-time through the use of a **DIST** directive with **BLOCK** or **CYCLIC** options. **INDIRECT** directives can be added to indicate run-time distribution. Such a distribution may have a range of valid distributions defined in its declaration. It uses an explicit mapping array to assign a distribution by an executable statement. The assigned distribution can be part of the condition in the source program. In addition to direct distribution definition, an array in Vienna Fortran can inherit a distribution from the definition of its alignment relative to some other array (and vice versa). Directive **DIST** can be used with options like **=A**, **TRANPOSE(A)**, **PERMUTE(A, PERM)** to align an array with, respectively, another array AD , transposed array A or array A with indices permuted according to the given vector **PERM**.

Fortran D directives are similar to Vienna Fortran, however distribution is separated from alignment. In Fortran D, first the **DECOMPOSITION** statement is used to declare a problem domain for each computation. The **ALIGN** statement is then used to describe problem mapping that defines the alignment of arrays with respect to each other. Finally, the **DISTRIBUTE** statement is used to map the problem and its associated arrays to the physical machine.

In EPL, by subscripting the annotated virtual process names and defining the appropriate ranges for the subscripts, the user can distribute the arrays in blocks, columns or rows. The arrays can also be transposed by permuting the subscripts of annotated virtual processors. Unlike Vienna Fortran and Fortran D, EPL does not provide the user with directives to do manual alignment of data. Instead, data alignment algorithms have been developed to facilitate this task automatically (see Section 4.4.1). Hence embedding alignment directives in source programs is not necessary.

3 Parallelism Extraction in EPL

In EPL, compile-time parallelism is sought on three levels:

- **Coarse Grain** parallelism is sought by creating tasks that are primarily dictated by the user-defined processes and process interconnection network described in the configuration files. The process interconnection network is decomposed into parallelizable tasks by the compiler. Since the optimal decomposition is NP-hard for machines having more than three processors [21], the EPL compiler uses heuristics.
- **Medium Grain** parallelism is sought at the level of equation clusters. Several equations in a program can be clustered into a group. Separate tasks are generated for each of the clusters. A cluster can run concurrently with other clusters in the same program. Programmers can assist the compiler in determining such clusters by annotating each equation by a virtual processor name. To minimize interprocess communication the compiler uses an heuristic to impose a hierarchy among the generated processes (see Section 2.4).
- **Fine Grain** parallelism is explored at the level of individual instances of equations or their clusters. This source of parallelism is of the greatest importance in massively parallel SIMD architectures. Mapping 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 this respect is known as the *data alignment problem* which is discussed in Section 4.4.1. Another

problem arises in connection with mapping operators onto processors. The solution to the latter problem is discussed in Section 4.4.2. The order of evaluation of the array elements is important for SIMD code efficiency. A compile-time method of determining an efficient order, known as *wavefront determination*, is discussed in Section 4.4.3.

| | <i>Granularity</i> | Coarse Grain | Medium Grain | Fine Grain |
|---------------------------|----------------------------|---|---|-------------------------|
| Compile-Time | <i>Type of Parallelism</i> | Control Parallelism | Loop Parallelism | Data Parallelism |
| | <i>Developed Tools</i> | Configurator | Annotations | Data Alignment |
| | | Scheduler | Memory Optimization | Wavefront Determination |
| | | Partitioning Algorithms | | |
| <i>Problems Addressed</i> | Synchronization | Matrix Computations Direct Solvers | Recurrence Equations Iterative Solvers | |
| Run-Time | <i>Developed Tools</i> | Dynamic Load Balancing | | |
| | <i>Problems Addressed</i> | Adaptive Solution of Partial Differential Equations | | |

Figure 4: Developed tools and their relationships to issues in parallel scientific computation

Figure 4 shows the tools that have been developed and their correspondence to various models of parallel computations. The *control-parallel* model assumes that there are separate, relatively independent processes or functions that can be executed simultaneously. This model requires the user to handle the error-prone and difficult task of synchronizing these independent processes. The configurator eases the burden of programming for control parallelism by automating the definition of interprocess coordination.

Data parallelism, popular in massively parallel systems, assumes that there are large data structures to be processed and that each element of every structure can be assigned 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. Data alignment tools described in this paper can find sub-optimal solutions to this problem without user involvement.

Annotations, relevant mainly to loop-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 operations in a program.

A load-balancing problem surfaces at all three levels of parallelism. In Section 5 we describe how the partitioning tools developed for the presented compiler can be used to do either static or dynamic load balancing on linear or rectangular arrays of processors. The partitioning tool is applicable to irregular computations that result from using adaptive solvers of partial differential equations on either homogeneous or heterogeneous processors.

In EPL, the programmer can assist the compiler in extracting coarse- and medium-level parallelism. As described earlier, coarse-grain parallelism is obtained by creating tasks from the processes and their

interconnection network as specified in the configuration files. The programmer can help in determining the medium-grain parallelism by annotating the equations in the source program. After determining the coarse- and medium-grain parallelism, the parallel program is synthesized with the help of the configurator.

4 EPL Compiler

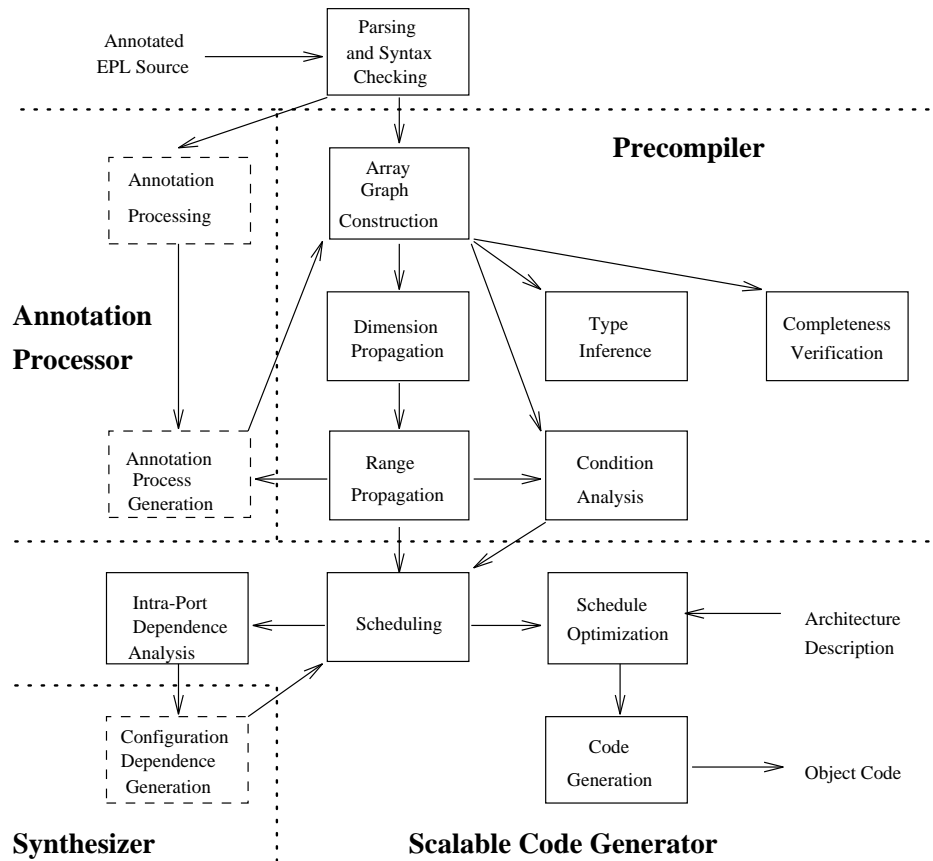


Figure 5: The structure of the EPL compiler

The basic techniques used in EPL compilation are data-dependence analysis and data-attribute propagation. In a single program, the dependences 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 dependences among processes of the computation and is used for scheduling processes and mapping them onto the processors. Figure 5 depicts the structure of the EPL compiler by showing part of Figure 1 in more detail. In particular, all components of **Annotation Processing**, **Precompiler** and **Scalable Code Generator** are explicitly shown. The major stages of the EPL compilation are:

1. *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.

2. *Dimension Propagation* that checks correctness and assigns dimensionality to each EPL variable.
3. *Type Checking* which verifies that all variables and expressions have or can be assigned consistent types.
4. *Completeness Verification* that performs various semantic checks and verifies that each variable is defined over its entire domain.
5. *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.
6. *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.
7. *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.
8. *Schedule Optimization* is an architecture-dependent step that customizes the generated schedule to the target architecture (see, for example, [10] for SIMD specific optimizations).
9. *Annotation Processing, Configuration Processing and Code Generation* are discussed in Sections 4.2, 4.3 and 4.4, respectively.

4.1 Single Assignment Fortran

Through extensions and annotations, imperative languages, particularly Fortran, have maintained their dominance in scientific computation over such non-traditional languages as dataflow, logic or functional. Nevertheless, languages based on the single assignment rule have proven to be a convenient basis for developing sophisticated program optimizations. EPL research has centered its program optimization techniques on the array graph representation of recurrence equations. We believe that by transforming the Fortran programs to array graph representation, a wider spectrum of program optimization and parallel code generation methods can be applied to the transformed programs than to their Fortran source.

An important step towards an efficient parallelization of Fortran programs with the help of the EPL compiler involves an equational transformation during which the equational equivalent of the program is generated [2]. The transformed programs obey the single assignment rule and do not contain any control statements [22]. The transformation is done in the following two steps:

1. Program expansion, during which the variables are expanded to enforce the single assignment rule. In particular, the **reassignments elimination** involves replacing the reassigned variables by:
 - vector (additional dimension) – inside loops,
 - variants – in “if” branches and basic blocks.
2. Program optimization, that consists of:

Condition Analysis: Conditions in the transformed program are analyzed using a Sup-Inf inequality prover [4] and the Kaufl variable elimination method [23] to find pairwise equivalent or exclusive conditions.

Variable’s Variants Elimination: Variants created in equivalent and exclusive conditions are merged into a single variable.

Additional Dimension Elimination: During scheduling and code generation for individual processes, memory optimization is performed to replace entire dimensions by windows of few elements for multidimensional variables [7]. This step restores the memory efficiency of the original program.

The transformed Fortran program is then compatible with the programs produced by annotating EPL programs.

4.2 Annotation Processing

Each virtual processor produces data, typically used by other virtual processors, and in turn consumes data produced by others. By performing data-dependence analysis in a style of PTRAN [24], the annotation processor can find the dependencies local to each virtual processor as well as data structures produced and consumed by this processor. All data produced by the processor become local to it and are placed in the its local memory. The created parallel tasks are supplied with communication statements needed to move non-local data. Parallel tasks associated with virtual processors at the bottom of the block hierarchy are the smallest components used in the program synthesis. Hence, 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 dependences 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.³ All used but not local data must be sent in from the other tasks. The annotation processor builds the task communication graph. Then, it augments 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.

Let $G(V, E)$ be a task communication graph with a set of nodes V representing tasks and a set of edges $E \subseteq V \times V$ representing intertask communication. Each edge $e_{i,j} \in E$ has the associated cost, $c(e_{i,j})$, that represents the volume of data being sent from the task i to the task j . In a spanning tree T , the distance $d^T(e_{i,j})$ defines the minimum length path from task i to task j . Using these definitions, the cost of the spanning tree T can be defined as:

$$C(T) = \sum_{e_{i,j} \in E} c(e_{i,j}) * d^T(e_{i,j})$$

To minimize the total communication cost, proper cut-tree must be found. It can be done in $O(|V|^4)$ steps [25] by solving $|V|$ maximal flow problems.

To embed the tree, we developed an heuristic which selects the embedding using the following criteria:

³We refer to this principle as *Executor Owns* rule, it is an inverse of the more commonly used *Owner Computes* rule. In [8] we have shown an example of computation for which neither of the two rules results in an optimal solution.

- **Dimension nesting:** If two tasks with different dimensionalities are connected in the task communication graph, the task with more dimensions should be located lower in the spanning tree.
- **Range nesting:** Whenever possible, tasks sharing the same range should be clustered together in the spanning tree. Variables that share ranges usually 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.

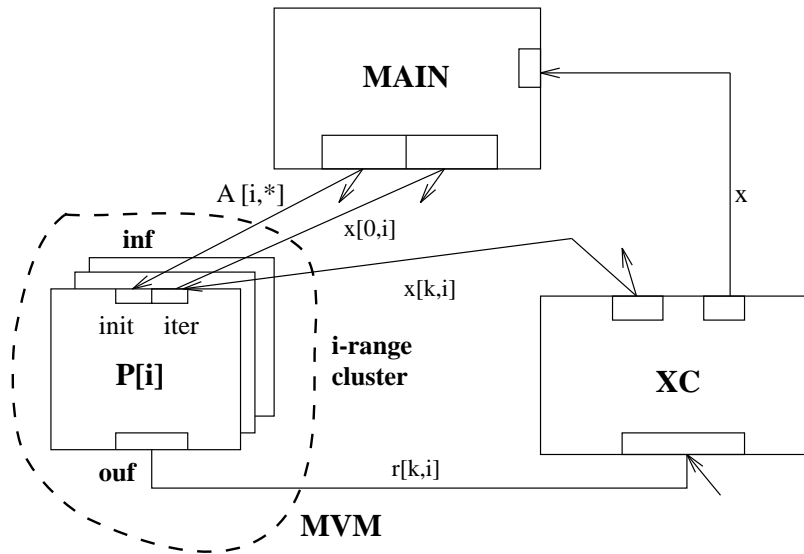


Figure 6: Communication tree for matrix vector multiplication

Trees created from an annotation of the matrix vector multiplication program are shown in Figure 6. The double outgoing arrows indicate scattering the data from a task to a group of tasks. The double incoming arrows represent an inverse operation of gathering the data. For example, process *MAIN* scatters the vector $x[0]$ among processors $P[i]$. On the other hand, process *XC* gathers the vector $r[k]$ by collecting individual elements $r[k, i]$ from processes $P[i]$.

4.3 Program Synthesis with the Configurator

The goal of configuration processing is to establish scheduling constraints for the overall computation. In the 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 dependences (in addition to dependences 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

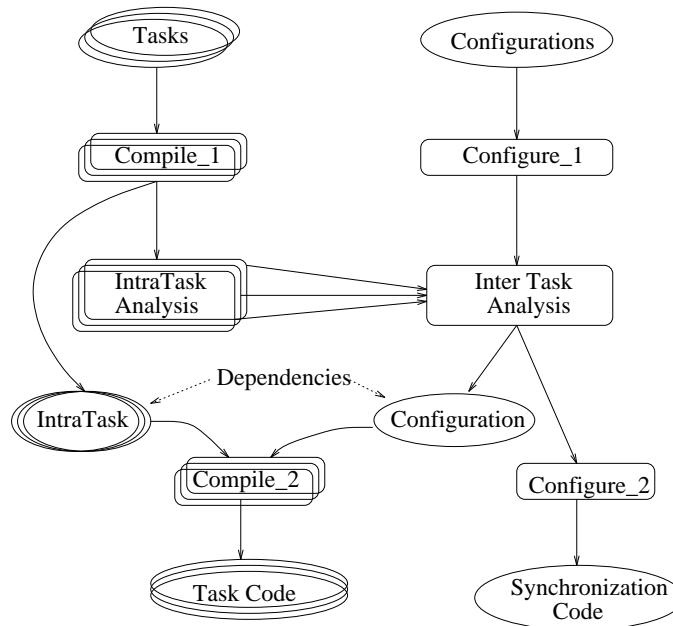


Figure 7: Two-stage dependence analysis

statement is executed in a separate loop from the matching receive statement, then all messages will be sent before any one can be received, and the successors of such nonatomic tasks cannot start until its predecessors in the task communication graph finish sending messages.

The algorithm for finding external data dependences has been presented in [6]. The analysis starts by inspecting all atomic processes and then propagates transitive dependences 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 dependences (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 this 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.

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

4.4 Code Generation and Optimization for Massively Parallel Architectures

Data structures used in scientific computation can be viewed as a function δ from an *index domain* I to a *value domain* V . An *index domain*, in general a set of tuples of integers $\langle i_1, i_2, \dots, i_n \rangle$, is often a subset of the Cartesian product of integer intervals, for regular n -dimensional arrays. For example, $I = I_1 \times I_2 \times \dots \times I_n$, 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 $\langle i_1, \dots, i_n \rangle$ as $v[i_1, \dots, i_n]$.

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 dependences that rarely impose the total order.

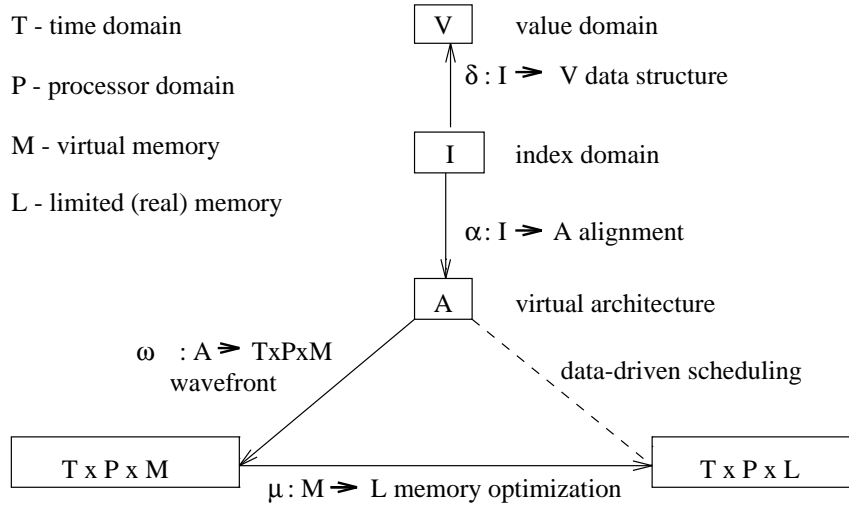


Figure 8: Functional view of code generation

Figure 8 shows the conceptual stages of mapping the index domain of a variable to the Cartesian product of the processor domain, their local memory domains and the time domain. The goal is to find a mapping that results in the minimum execution time. In Figure 8, A represents a *virtual architecture*. 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 $A = [1, n_1] \times [1, n_2] \times \dots \times [1, n_k]$, where $N = n_1 * n_2 * \dots * n_k$. The processor $p[l_1, l_2, \dots, l_k]$ is connected with processors $p[l_1, \dots, l_j \pm 1, \dots, l_k]$, $1 \leq j \leq k$ provided that processor $p[l_1, \dots, l_j \pm 1, \dots, l_k]$ 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 architecture A is compatible with the domain I . *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 memory domain* M is of the same structure as the domain L , except it has unlimited memory size. The execution time steps are represented by *time domain* $T = [1, t_{max}]$, where t_{max} is the total number of time steps needed to complete the computation.

In such a view, there are three major problems that need to be solved for generating optimized code for massively parallel architectures: *Data Alignment*, *Time Scheduling* and *Memory Optimization*.

Data alignment is discussed in some detail in the next section. Time scheduling of iterative computations is usually done either through *data-driven scheduling* or *wavefront determination*. 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 [26]. Such a scheduler has been implemented in the compiler for EPL language [7] and is not discussed here. Wavefront Scheduling is presented in Section 4.4.3.

Programs written in EPL or transformed from Fortran 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 value for 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 EPL compiler can often reduce the memory requirement of a program by replacing the entire dimension of an array by a few elements [7]. However, we have proven [27] that the problem of finding the optimum replacement is equivalent to the well-known NP-hard problem of determining the maximum weight clique problem. Consequently, the EPL compiler uses heuristics to determine a good loop arrangement for memory optimization.

4.4.1 Data Alignment

In a distributed-memory parallel computer, a significant speedup can be achieved by distributing (or mapping) data structures in a program onto the processors. One processor is allocated (at least conceptually) to each array element or composite data structure. Operations on elements of two data structures can be performed entirely locally if the elements are allocated to the same processor; otherwise, processor communication has to be involved. The cost of communication depends on the relative position of the two processors involved and the architecture under consideration. One of the major challenges in programming distributed-memory parallel computers is to distribute data structures among the processors so that the communication cost is minimized.

The problem is particularly acute when the communication is synchronous, such as in the case of SIMD machines. In addition, different alignments of multi-dimensional arrays on a grid-connected SIMD architecture result in different communication patterns during parallel program execution. The usual approach to this problem [28, 29] is to select the best alignment for each array in the program independently of other arrays. Hence, such an approach does not succeed when the independently found alignments conflict with each other. Similarly, the algorithm presented in [30] finds the minimum communication cost of evaluating an expression over a distributed processor array but only for a single expression. Given the initial allocation of data, the algorithm determines the processors at which the temporary variables should reside and a subexpression evaluation should take place to minimize the communication cost.

In [31], we have shown that the data alignment problem for an entire program is NP-hard for all communication cost metrics. In [8], we proposed an heuristic that starts with an integer approximation of the rational minimum of the cost function when the distance is defined by the second (Euclidean) norm. The initial solution is then iteratively improved by following the steepest decline direction of the cost function. Results of using this algorithm on random graphs are encouraging [8].

Here, we focus on the definition of the problem and its impact upon the code generation. Let's consider an equation $e_{\{I_1, \dots, I_k\}}$ defined over k subscripts I_1, \dots, I_k (such an equation corresponds to a statement nested in k iterations):

$$e_{\{I_1, \dots, I_k\}} : v_l[s_1, s_2, \dots, s_k] = \dots v_r[f_1, f_2, \dots, f_k] \dots$$

where each simple indexing expression s_j on the left side of the equation is an affine function of the corresponding subscript I_j , and each indexing expression f_j on the right side is a function over possibly many subscripts. A large class of parallel scientific computations can be expressed as Regular Iterative Algorithms (RIA) [32] in which all indexing expressions are of the form “ $I + c$ ”, where I is a subscript and c is an integer constant.

To generate efficient code for SIMD machines, one or two dimensions of a data array should be projected along the processor array [10]. For the i -th projected dimension of each array (each equation), we define an alignment function α_i that maps the index of that dimension into the position of the virtual

processor that stores (executes) its value. We consider the simplest but also the most useful form of the alignment function defined as a constant shift, e.g., for variable v_l ,

$$\alpha_l(I_i) = I_i + a_{li}$$

Hence, the equation e with alignment shifts can be written as:

$$e_{\{I_1, \dots, I_k\}} : \quad v_l[I_1, \dots, I_k] = \dots v_r[I_1 + c_1, \dots, I_k + c_k] \quad (1)$$

This equation incurs the communication cost:

$$C = \sum_{\text{for all } v \text{ in } e} \gamma * d(|a_{e1} - a_{v1} + c_{v1}|, \dots, |a_{ek} - a_{vk} + c_{vk}|)$$

where d is a distance metric, γ denotes the time needed for sending a unit message between two directly connected processors, and n is the dimensionality of the communication network. The distance metric is defined by the interconnection of the processors in the considered parallel architecture. Thus, the problem is to find alignment functions α 's for each of the variables and equations such that the communication cost C for the given set of assignments is minimal. Figure 9 shows the communication

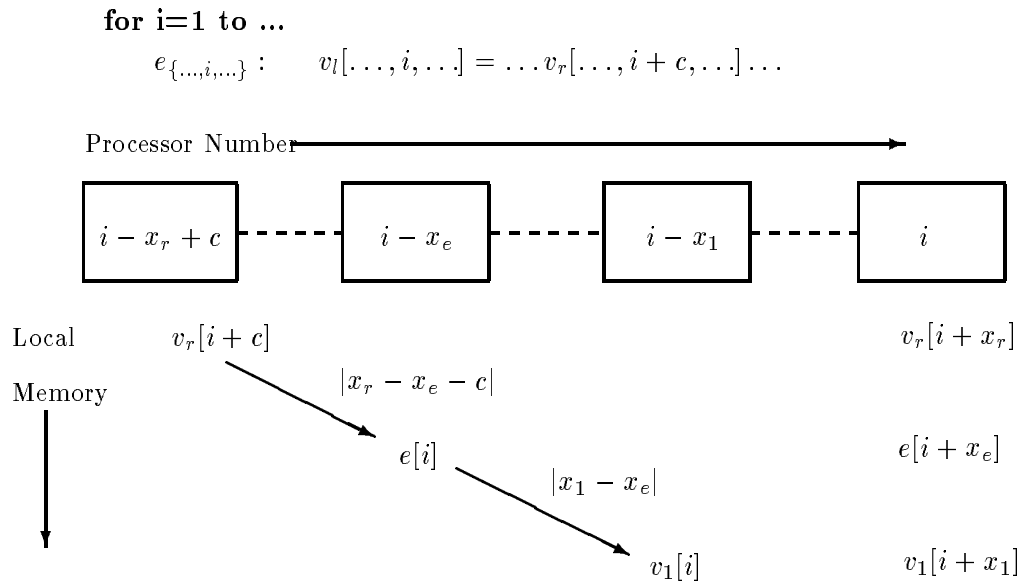


Figure 9: Communication cost of executing equation e

among the processors executing the i -th instance of Eq. 1 along a single dimension. Contrary to the well known *Owner Computes* rule, to minimize communication costs, the processor executing the i -th instance of the equation may be different from the processor that stores the i -th element of the array defined by this equation.

4.4.2 Array Operator Placement

Proper assignment of array operators to processors in large scientific computations executed on a distributed-memory machine can reduce total computation time significantly. For example, consider the following computation⁴ evaluated over the rectangular stencil. Let n_1, n_2 stand for the lengths of

⁴This example is based on the computation arising in modeling ecosystem on the MasPar [33].

the sides of the stencil and p_1, p_2 be the offsets (measured from the lower left corner of the stencil) of the desired position of the result. Let $s_{i,j}$ be a data structure distributed over the two-dimensional processor array and (m, q) be the coordinates of the processor that should receive the result. The computation is defined as:

$$result = \sum_{i=m-p_1}^{m+n_1-p_1} \sum_{j=q-p_2}^{q+n_2-p_2} f(s_{m,q}, s_{i,j})$$

The above computation is evaluated repeatedly for each rectangular stencil in the processor array. Hence, it is likely to dominate the total execution time. The above computation is an example of a reduction evaluated simultaneously over many overlapping continuous sections of an array. Other examples of usage of such operations are likely to be found in cluster recognition, fractal dimension computation in biological modeling [34], or in modeling physical phenomena (e.g., solvers of partial differential equations characterizing fluid flow).

Simultaneous reduction is evaluated over a one-dimensional consecutive section of an array, called here an *array interval*; each array element is used as an operand to many reductions evaluated simultaneously over different overlapping intervals. This is distinct from what is usually referred to as *Parallel Reduction*, which involves the parallel evaluation of a single reduction [35] or its variants. An algorithm for standard parallel reduction that uses a balanced binary tree implementation for mesh-connected architectures has been presented in [36]. Another standard parallel reduction algorithm has been introduced in [37] for tree topologies of arbitrary but bounded fan-in and arbitrary tree depth. The segmented prefix problem is a variant of parallel reduction that subdivides a single dimension of processors into non-overlapping contiguous regions of varying size. A multiple prefix algorithm that reduces non-contiguous regions simultaneously for this variant has been presented in [38]. None of the published algorithms cope with the overlapping of the regions being reduced.

Efficiency of the simultaneous reduction has been discussed in [39]. It can be expressed as a function of (i) operation count; i.e., the number of required reduction operation steps, (ii) communication cost; that is, a function of the number of messages sent (message count), the distances traveled by messages (hop count) and the length of the messages (message size), and (iii) memory count; i.e., the number of memory locations used to store intermediate results at each processor. The lower bounds for the above counts are: $\lceil \log_2 n \rceil$ for the operation, message and memory counts, $n - 1$ for the hop count, and 1 for the message size. For the interval of size $n = 2^k$ and an arbitrary offset p , a modification of the well known *Parallel Prefix* algorithm [35] achieves the above bounds. The modification defines the direction of the message transfer in each step by the corresponding bit of the binary representation of the offset p .

For an arbitrary interval size n and an arbitrary offset p we have designed an algorithm called *Intersect* that achieves the lower bound of communication and memory counts and is within a factor of 1.5 of the lower bound of operation count.

For an arbitrary interval size n and an arbitrary offset p , we have designed an algorithm called *Split* which produces the result with the memory, hop and message size equal to their lower bounds. The operation and message counts are at most twice the value of the corresponding lower bound. Depending on the relative cost of the increased message and operation counts versus the smaller hop count, this algorithm may or may not outperform *intersect* for the given interval and offset.

For an arbitrary interval size we have designed two algorithms that require asymptotically small operation and message counts: both counts are $\log_2 n + 2$ if the reduction's binary operator has an inverse and $\log_2 n + 2(\log_2 n)^c + o((\log_2 n)^c)$, where $c = \log_{12} 6 \approx 0.721057 \dots$, otherwise.

4.4.3 Wavefront Determination

One of the most common forms of parallelism available in a scientific computation is data parallelism, in which the same operation is performed on many elements in an n -dimensional data array. In computation over such an array, a wavefront of computation can be defined as an $(n - 1)$ -dimensional subarray whose elements are all evaluated simultaneously. Different wavefronts result in different performance, so the question arises how to determine the wavefront that results in the minimum computation time. Wavefront determination should also define which wavefront elements are to be computed by each processor at every execution step. This type of scheduling is appropriate for Single Program Multiple Data (SPMD) [40, 41] implementation on distributed-memory architecture or for data parallelism on SIMD architectures. SPMD implementation, in general, requires larger parallel granules than SIMD implementation; therefore, it is more efficient provided that the computations at each index point are fairly complex (i.e., involve computationally intensive function evaluation).

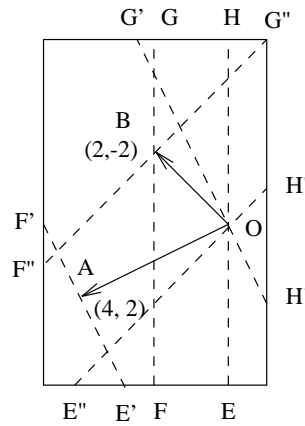


Figure 10: Different wavefronts to evaluate array E

Figure 10 illustrates how the choice of a particular wavefront can affect the performance of an algorithm. A two-dimensional array E is to be evaluated on a one-dimensional (logically) processor array. The elements are defined by the following equation (elements that are beyond the array boundary are considered to be zero):

$$E[x_1, x_2] = f(E[x_1 - 2, x_2 + 2], E[x_1 - 4, x_2 - 2]) \quad (2)$$

A *data dependence* vector of an equation is any vector that connects two index points. The end point of this vector is an index point at which the equation is executed and the starting point of the vector is an index point at which some value used in the definition is evaluated. For Regular Iterative Algorithms [32] expressed in EPL, the dependence vectors are defined by the difference between the corresponding subscript expressions used in the left and right side of the equation. In the above computation, there are just two dependence vectors: OA $([4,2])$ and OB $([2,-2])$.

In general, let $D = \{\bar{d}_1, \bar{d}_2, \dots, \bar{d}_k\}$ be the set of dependence vectors in a program (i.e., a set dependence vector for all equations in the EPL program). Variables can be evaluated simultaneously at all index points on a wavefront \bar{h} , if and only if $\bar{h} \cdot \bar{d}_i > 0$ for all dependence vectors \bar{d}_i . Intuitively, this condition requires that all index points reachable from a wavefront \bar{h} are known at the time of execution of this wavefront or, in other words, all array elements in an appropriate side of the wavefront have already been evaluated. In Figure 10 all dependence vectors are on one side of the lines EH , $E'H'$ and $E''H''$, so all of them are wavefronts. Evidently, any line between OB and OA (traversed

clockwise) in Figure 10 may be a wavefront, since for these and only these lines are the dependence vectors on one side of the line. However, such a wavefront does not always exist. For example, when data dependences are different at different regions of the index domain, there may be no single wavefront with the required property in the entire index domain.

Two parallel wavefronts form a strip of computation that can be divided among a number of processors for execution. The separation between the wavefronts can be made such that once all packets (containing array elements evaluated by other processors) reach their destination, no more communication is needed to complete the evaluation of all the array elements between the two wavefronts. In Figure 10, $EFGH$, $E'F'G'H'$ and $E''F''G''H''$ are three such strips. Since $EFGH$ covers a bigger area than $E''F''G''H''$, computation along this wavefront results in less frequent communication and synchronization. Wavefront EH can be preferred to $E''H''$ for another reason; namely, the smaller distance that data must travel (compare projection of OA on $E''H''$ with the projection of OA on EH). Wavefront EH can be partitioned into more sections than $E''H''$ with the similar computation-to-communication ratio, leading to a higher degree of parallelism.

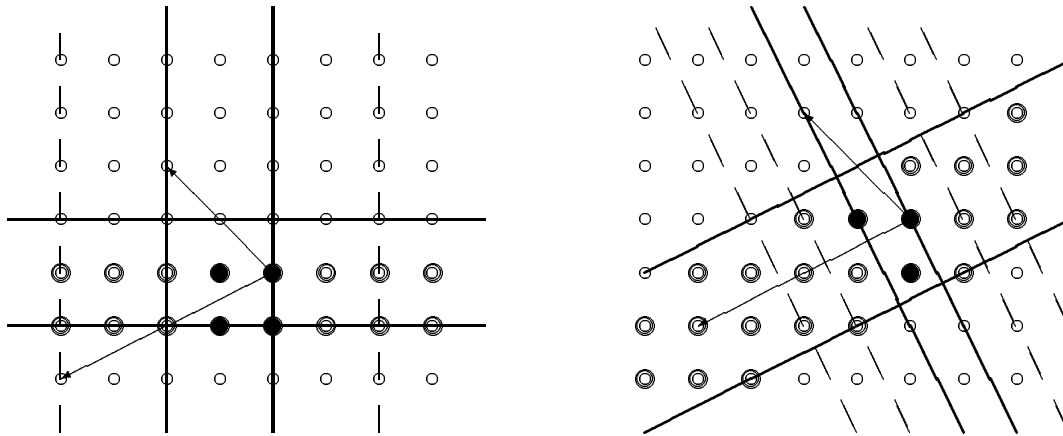
Even if there are no restrictions on the number of available processors, it is not straightforward to determine how the wavefronts should be optimally partitioned and mapped to the processors. A small partition increases communication time, because most of the input array elements needed to evaluate a particular index point may reside outside the evaluating processor's local memory. For certain dependence vectors and the sizes of the partitions, input array elements may be quite a few processors away. On the other hand, the processors may be underutilized, if a large partition of the wavefront is assigned to a single processor.

The wavefront approach to finding the set of index points at which evaluation can proceed simultaneously was originally proposed in [42]. However, to find the wavefront minimizing the total execution time, an NP-hard integer programming problem has to be solved. This original result has been extended by many researchers over the years [43, 44, 45, 46]; however, the proposed solutions still are NP-hard because they can be reduced to an instance of the integer-programming problem.

Assuming that the spacetime representation of an algorithm is a continuous domain, we can determine the wavefront \bar{h} with the minimum execution time with polynomial complexity. This result holds for two-dimensional arrays processed on a linear, arbitrary large array of processors. It is valid for two different models of communication: (i) individual element transfer, and (ii) packet transfer. In the first case, we have proven, under the above simplifying assumptions, that the only wavefronts which can be optimal are those that are either perpendicular to one of the dependence vectors or parallel to the y -axis. This property leads to a simple but efficient procedure for finding an optimal wavefront by just inspecting all potentially optimal wavefronts (complexity of this procedure is linear in the size of the input).

For the example in Figure 10, there are only three angles of a wavefront to consider: $\gamma_1 = \pi/2$, $\gamma_2 = \arctan(-2)$, $\gamma_3 = \pi/4$. The wavefronts with γ_1 and γ_2 are shown in Figure 11. In a single execution step with the wavefront defined by γ_1 , each processor evaluates four index points and needs to receive eight values from the neighboring processors. However, for γ_2 wavefront, the number of evaluated points and received messages is at most three. The number of steps needed is also different for these two wavefronts because they move in different directions. If we assume that the computation is defined over a rectangle with corners at the points $(0, 0)$, $(0, Y)$, $(X, 0)$, (X, Y) ; $X = 100$, $Y = 10$, then the number of steps made by the first wavefront is 50 and by the second one is 105. The corresponding total computation times for all three discussed wavefronts will be $T_1 = 315e + 315c$, $T_2 = 200e + 400c$, $T_3 = 630e + 630c$, where e is the cost of execution at each index point, and c is the cost of communicating one datum between neighboring processors. Depending on the value of c/e , the first or the second angle should be selected (see Figure 11).

Usually, array elements are not passed individually, but several of them are grouped together and sent in a single packet. This method is commonly used in the communication model known as *block*

Figure 11: Optimal wavefronts for array E

SIMD. In this model, off-processor values required to compute a designated block of parallel code are obtained immediately before the beginning of the block, and all off-processor values generated within the block are communicated immediately after the end of the block [47]. Typically, packets of values are formed for communication and transferred between non-neighboring processors by means of hopping.

The wavefront strip is partitioned among the processors and the width of each partition impacts the total computation time. With too small a width, processors spend less time computing and more time communicating, because less relevant information is available in the local memory. On the other hand, a large width enables processors to spend more time computing between data transfers, resulting in a smaller communication cost. Beyond a certain width, the communication cost does not decrease any further with an increase in the partition width. If the partitions are too large, the available parallelism may not be exploited fully.

As in the previous case, we have proved that the optimal wavefront can only be at certain angles to the dependence vectors (the number of possible angles is limited by the square of the number of dependence vectors). Once again the proof leads to an efficient enumeration procedure.

In our analysis we have assumed a continuum of data elements in an array. In reality, arrays are discrete, so the analysis is approximate. For example, in mapping a computation onto a linear array of processors, the algorithm provides a good wavefront when the longest projections (on each side) of the data dependence vectors on the selected wavefront are much larger than the length of packets sent along the wavefront.

The methods described here can be applied to any set of uncoupled recurrence equations. To decrease the communication cost, a good alignment of all arrays in the program should be determined first [8, 48]. Many methods described in the literature [44, 45, 46, 43] determine the actual mapping of the computation onto the processors, once the wavefront is determined by solving an integer programming optimization problem. These algorithms can be used for the wavefronts obtained by our method.

There are many open problems in this area. One major issue concerns finding an efficient algorithm to determine a good wavefront when a set of recurrence equations involving m -dimensional arrays are to be computed on an n -dimensional array of processors ($m \geq n$). Another important question is how to generate the packets of convenient sizes and shapes efficiently, once their size and orientation are known.

5 Run-Time Support

As discussed earlier, the main problem of efficient parallelization is to properly map addresses of values being computed onto the computer processors. Pure compiler techniques have been successful in cases when the data addresses are input-independent and can be established at compile time. However, many important applications involve sparse matrix computations, adaptive numerical algorithms or computations over irregular meshes and therefore do not belong to this category.

Traditionally supported compiler optimizations for parallel computation involves subscript analysis or directives for regular problem decompositions and distribution. Language and software tools for dealing with irregularity in parallel computation rely either on user-provided partitioning algorithms, e.g., dynamic distributions in Vienna FORTRAN [17] or the tracing of sample executions, e.g., Kali compiler [11, 49] and the PARTI library [50, 51]). Communication patterns of many advanced parallel computations are rarely known at compile time. However, transferring individual data is expensive because of the usually large latency of MIMD architecture communication. Fortunately, often communication patterns change with each input data but remain constant inside the loop at run-time. Therefore, both the Kali compiler and the PARTI library attempt to group messages. Entire blocks of data that must be sent to the single processor are assembled into a single message in loop preprocessing done at run-time [49, 50].

In adaptive computation, the run-time support is needed because the workload distribution among the subregions of the computational domain changes during run-time. Therefore, there is a need for run-time task reallocation of adaptive computation executed on massively parallel distributed-memory machines. Such task reallocation requires different methods than the large-grain, few-processor approaches discussed in the literature [52]. We have proposed a new type of so-called density workload problems appropriate for such environments [5].

5.1 Run-Time Task Distribution

One of the most challenging problems encountered while implementing adaptive scientific computations on distributed-memory machines is run-time mapping of a dynamically changing computational load onto the parallel processors. In [53], the following *Rectilinear Partitioning Problem* (RPP) has been proposed and solved:

Partition the given $n \times m$ workload matrix into $(N + 1) \times (M + 1)$ rectangles with $N + M$ rectilinear cuts in such a way that the maximum workload among rectangles is minimized.

Such optimization is appropriate for adaptive finite element computations on architectures with local communication that is faster than the global one. Since balanced partitions tend to increase the volume of local versus global communication, the overall communication cost can be decreased by using the optimum rectilinear partition.

In [5], we investigated the balancing of an adaptive scientific computation on SIMD machines: this is the problem with similar motivation and applications as the RPP problem. In RPP, the sum of the weights is taken as the cost of a rectangle, whereas in our problem the cost is expressed as the workload density, i.e., the ratio of the workload to the area with which this workload is associated. The area is proportional to the number of processors active in it. Such cost definition is motivated by the mesh refinement techniques used in adaptive numerical methods. Each entry in the workload matrix represents the solution error obtained by an error estimation procedure [54]. The high-error regions need recomputing and the needed work is proportional to the magnitude of the error. Hence, the number of processors reassigned to each solution region should be proportional to the refinement factor.

Consider a load-balancing problem as illustrated in Figure 12 for a one-dimensional problem. The uniform mesh yields the solution with a high error in the interval $b \leq x \leq c$ and within the required

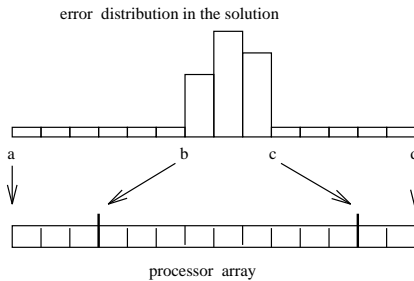


Figure 12: Example of one-dimensional partitioning

accuracy in intervals $a \leq x \leq b$ and $c \leq x \leq d$. Taking the magnitude of an error as an estimate of the work ω_i for each element $i = 1, \dots, n$, we assign a small weight $\epsilon \ll \max_i \{\omega_i\}$ to work the estimate in regions $a \leq x \leq b$ and $c \leq x \leq d$. To balance the workload, the majority of the processors should be assigned the interval $b \leq x \leq c$.

In adaptive solutions of partial differential equations, parallel tasks perform basically the same computation over different spatial subdomains (intervals for one-dimensional problems) and with a different discretization parameter Δx . Let K denote the number of such tasks. It is important to keep this number small for the following reasons. The subdomain interactions are proportional to the number of existing subdomains and in higher dimensions such interactions require expensive global communications. In each time step of the subdomain computation, a fraction of executed code is subdomain-specific (e.g., in hyperbolic equations the time step has to be set differently in each subdomain). For purely SIMD machines, execution of this code fraction has to be done in K consecutive stages. In each stage, processors in one subdomain are executing while processors belonging to the remaining $K - 1$ subdomains remain idle.⁵ Therefore, each subdomain associated with a parallel task should represent a localized structure in the solution domain.

Figure 13(a) shows an example of the more difficult two-dimensional case in which a coarse mesh is trivially mapped to the processor mesh. In regions A and B, the mesh must be refined due to the presence of high errors. Hence, we have to spread sub-domains A and B over bigger rectangular subsets of processors to improve load balancing as in Figures 13(b) and (c).

If *mesh-movement* or *static rezone* techniques are used, the mesh elements are moved into high-error regions. A *global* solution strategy will refine the high-error regions and repeat the entire step of the iteration. Consequently, a reassignment of processors is needed. A *local* solution strategy, on the other hand, repeats the iteration only where it is needed. Hence, local refinement results in less direct computation and enables more processors to be assigned to regions A and B. However, local refinement requires more interactions between the local and global solutions. Such interactions involve global communication that can outweigh the benefits of an adaptive procedure. Global solutions and mesh-movement techniques require less interactions of this kind. Careful buffering of the high-error regions can increase the number of iterations executed before re-gridding or mesh movement is needed. This will, in turn, decrease the frequency of the needed load balancing. It is these global mesh-refinement and mesh-movement techniques executed on a mesh-connected architecture that motivated us to develop density-type partitioning.

It should be noted that applying RPP partitioning to the example shown in Figure 13(d) results in assigning unnecessary processors to regions C and D. To avoid such a waste, partitioning methodology cannot be restricted to rectilinear cuts extending across the whole domain in both dimensions. Hence,

⁵For more general MIMD architectures that support coordinated parallelism (i.e., CM-5), all K subdomains can execute this fraction of code in parallel.

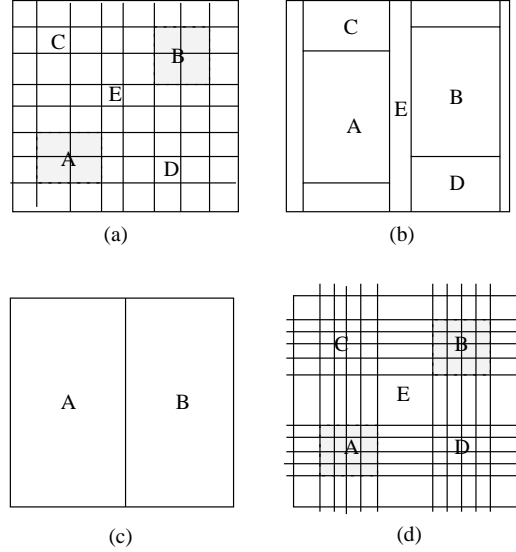


Figure 13: (a) Coarse mesh with high error regions A and B, (b) repartitioning with global refinement, (c) repartitioning with local refinement, (d) Nicol’s partitioning

in our problem definition and solution [5], we require that K selected rectangles cover the whole domain. The heuristics for the two-dimensional case projects the weights to one dimension and results in rectilinear cuts extending across the whole dimension in one direction. Figure 13(b) shows an example of this kind of partition.

To give a brief formal treatment of a one-dimensional case, let P_K be the set of all K -partitions of a one-dimensional workload array $\omega_i, i = 1, \dots, n$ into K subintervals (x_{k-1}, x_k) , where $1 \leq x_{k-1} \leq x_k \leq n, k = 1, \dots, K$. The one-dimensional workload partitioning problem can then be stated as:

$$\bigoplus_{(x_0, \dots, x_{n-1}, x_n) \in P_K} \left\{ \bigotimes_{k \in [1, K]} \left\{ \frac{\sum_{i=x_{k-1}}^{x_k} \omega_i}{f(k)} \right\} \right\} \quad (3)$$

As shown in Table 1, selecting a different meaning for operations \bigoplus and \bigotimes yields different optimization problems. For $\bigoplus \equiv \min$, $\bigotimes \equiv \max$ and $f(x_{1_k}, x_{2_k}) = 1$, an instance of RPP is obtained which can be solved in $O(Kn)$ or $O(n + (K \log n)^2)$ steps [53].

| Problem | \bigoplus | \bigotimes | $f(k)$ |
|---|-------------|--------------|-----------------------|
| One-dimensional partitioning [53] | \min | \max | 1 |
| Density type for PDEs | \min | \max | $(x_k - x_{k-1} + 1)$ |
| Shortest path with k arcs | \min | + | 1 |
| Partitioning for heterogeneous processors | \min | \max | s_k |

Table 1: Instances of problem represented by Eq.(1)

The problem of load balancing for adaptive PDE solvers on machines where the number of processors exceeds the number of tasks can be obtained by putting $\bigoplus \equiv \min$, $\bigotimes \equiv \max$ and $f(k) = (x_k - x_{k-1} + 1)$, i.e., when the sum of the workloads in each partition is divided by the interval length (i.e., the number

of processors). If there are K heterogeneous processors, each with a different speed s_k $k = 1, \dots, K$, then $f(k)$, \oplus and \otimes can be instantiated according to the fourth row of Table 1. The RPP algorithm can handle only the case with the monotonically increasing cost function $\sum_{i=x_{k-1}}^{x_k} \omega_i$. In contrast, our algorithm can solve more complicated problems with an arbitrary cost function $\sum_{i=x_{k-1}}^{x_k} \omega_i/f(k)$ in $O(Kn^3)$ steps.

There is a similarity between the weighted independent set for interval graphs and our problem [55]. The interval graph for our problem can be created as follows. Each possible subinterval (x_{k-1}, x_k) is represented by a node of the interval graph. The weight of the node representing (x_{k-1}, x_k) is set to $\sum_{i=x_{k-1}}^{x_k} \omega_i/f(k)$. In such a graph, the independent set of size K which covers the whole interval, $1, \dots, n$, gives the solution to the original problem. The interval graph can be converted to a directed acyclic graph (DAG). The shortest path algorithm applied to this DAG will find the minimum weight dominating set [56]. This approach results in the optimal algorithm for the one-dimensional case and also leads to an heuristic algorithm that can be easily generalized to two dimensions (by projecting the workloads to one dimension).

5.2 Run-Time Array Distribution in EPL

To illustrate the run-time support provided in the EPL compiler, consider the sparse matrix-vector multiplication. This operation lies at the heart of many numerical algorithms, such as the conjugate gradient algorithm for the solution of linear systems of equations. The corresponding computation is:

$$r = Ax$$

Let S be the row-major representation of the sparse $n \times n$ matrix A , $colend[i]$ be the number of non-zero entries in each row ($i \leq n$) and $colmap[i, j]$ be the column number for each non-zero entry. The variable-sized rows of S must be mapped onto P processors where $P < n$. The total execution time of such a computation is defined by the execution time on the processor with the largest number of non-zero elements (because processors synchronize after each multiplication step in an iterative solver). Hence, it is important that rows of S are distributed in such a way that processors are load-balanced—i.e., each has about the same number of non-zero elements to evaluate. The corresponding EPL program is shown in Figure 14.

The load-balancing scheme can be implemented solely on the basis of the ranges of rows in S . The scheduler implemented in the EPL compiler [3] detects that the ranges of the rows in S must be available before the matrix-vector multiplication loop starts. Hence, the last two statements in the above EPL program which explicitly implement a simple load-balancing algorithm will always be scheduled before the loop body. The rows of S are then distributed accordingly to the run-time defined sizes. If these load-balancing statements are not given explicitly, then the block distribution will result, with each processor having the same (or nearly the same) number of rows, independently of the number of nonzero elements in those rows.

The program for distributing arrays was run on several benchmarks including meshes originally used by Hammond [57] and test cases from the Harwell-Boeing Sparse Matrix Collection [58]. The characteristics of the tests are given in Table 2. The first test case is an unstructured triangular mesh around a three-component airfoil, while the second test is a portion of a larger mesh representing an unstructured tetrahedral mesh about a Lockheed S-3A Viking aircraft. The third test case arises from a mixed kinetics diffusion problem (specifically, the study of ionization in the stratosphere with 38 chemical species). The fourth mesh is derived from a model of a gas cooled nuclear reactor core, and the fifth test was generated using a package for reservoir modeling.

The most straightforward implementation of the sparse matrix vector multiplication used in IT-PACK library [59] is shown in Figure 15(a). It multiplies each nonzero element by the corresponding vector element that is fetched through communication, if necessary. The results of several runs of the

```

process:sparse_multiply; out: outfile; in : infile;

file: infile,
  int n, np, colend[*], colmap[*,*], /*number of rows, processors, column ends, non-zeroes in each row*/
  double x[*],S[*,*]                /* vector and sparse matrix in major-column format */

file: outfile, double r[*] ;          /* output vector */

subs: i,j;

range.S = n; range.colend = n; range.b = n;
range.S[i] = colend[i]; range.colmap[i] = colend[i];

P[p]: r[i] = reduce(+,S[i,j]*x[colmap[i,j]] , j);

/** optional user statements and declarations for run-time load balancing ***/
int tload, load[*];                  /* total load, cumulative load */
subs: p is sublinear i: load[i] != load[i-1];

tload = reduce(+,colend[i], i);
load[i] = (scan(+,colend[i], i)*np+tload-1)/tload);

```

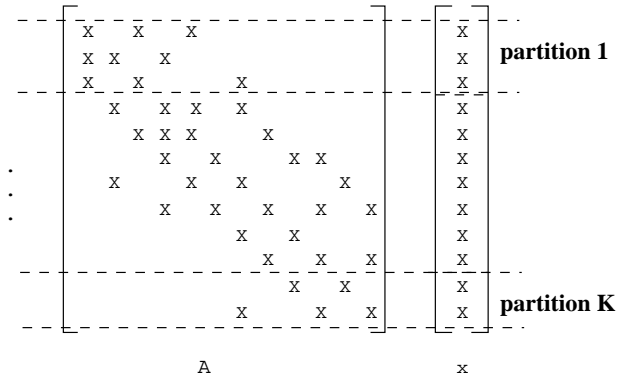
Figure 14: Sparse matrix-vector multiplication with dynamic load balancing

```

SUBROUTINE PMULT (...)
  ....
  DO 30 I = 1,N
    IBGN = IA(I)
    IEND = IA(I+1)-1
    SUM = 0.0D0
    IF (IBGN.GT.IEND) GO TO 20
    DO 10 J = IBGN,IEND
      JAJJ = JA(J)
      SUM = SUM+A(J)*X(JAJJ)
10    CONTINUE
20    W(I) = SUM
30    CONTINUE
  RETURN

```

(a)



(b)

Figure 15: (a) ITPACK matrix-vector multiplication code and (b) ordered array partitioning

sparse vector multiplication are given in Table 2. The rows labeled “block” and “load balance” give times for runs of the multiplication with a standard block distribution and with the block distribution adjusted by the load-balancing step, respectively. Results from executions presented in Table 2 showed up to a 21 percent cost reduction for the MP-1.

| Method/constant | 3elt | viking6 | fs_760_1 | nnc1374 | pores_2 |
|-------------------------|-------|---------|----------|---------|---------|
| Mesh Characteristics | | | | | |
| number of rows | 4720 | 6000 | 760 | 1374 | 1224 |
| number of nonzeroes | 27444 | 73734 | 5976 | 8606 | 9613 |
| Multiplication timings. | | | | | |
| | t (s) | t (s) | t (s) | t (s) | t (s) |
| block distribution | 33.6 | 92.8 | 63.8 | 59.6 | 64.5 |
| load balance | 28.5 | 79.8 | 50.5 | 52.8 | 55.3 |
| 100%*balanced/block | 85% | 86% | 79% | 88% | 86% |

Table 2: Mesh characteristics and execution times for test runs on the MasPar for 1000 iterations

6 Conclusion and Comparison with Other Approaches

In this section we characterize EPL in terms of criteria that identify important properties of parallel languages [60].

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 on-going work on C code generators for the CM-5 and SP1 architectures. Nevertheless, 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 parallel execution of EPL programs.

- specifying *data and program decomposition*

Only 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 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 yet, 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 [61], the Timer relies on a set of architecture measurements that can be established by running initiation programs 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's a need for a better tool. Timer drives transformations of equational programs into schedules and the stage of 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).

Program decomposition through annotations and computation synthesis through configurations can support efficient parallel code generation for domain-specific computation. Annotations support rapid prototyping and performance tuning of a parallel program. Adaptivity, with its associated error estimates and the shrewd use of computation resources only in regions where accuracy requirements are not satisfied, can provide the needed numerical reliability and efficiency to parallel computation. In the EPL system, adaptivity is supported through run-time task distribution.

There are several premises underpinning our approach, among the most important ones are:

- Annotations provide an easy and efficient way to parallelize existing codes.
- Large parallel programs consist of interconnected processes which represent logical partitions of the program.
- Absence of control statements simplifies program analysis and increases compiler's ability to produce an efficient parallel code.
- Most parallel code optimization problems are NP-hard; hence, development of proper heuristics is important.
- A hierarchical view of parallel computation is helpful in extracting functional parallelism.

Our research on scalable program synthesis has left many interesting issues unexplored. Future work on program synthesis that we intend to undertake includes parallelization of dynamic task distribution and run-time support for irregular computation. Efficiency of our methods will be measured for large applications, such as finite difference and finite element formulations for various scientific computations.

Acknowledgement

The authors wish to thank the anonymous reviewers and Prof. Hans Zima, Associate Editor, for their valuable comments on the earlier versions of this paper. Special thanks are also due to Louis Ziantz for assistance with running the EPL tests on the MasPar and to Joyce Brock for help in preparing the manuscript. This work was partially supported by the Office of Naval Research under grant N00014-93-1-0076, by the National Science Foundation under grants CCR-9216053 and ASC-9318184 and by the IBM Corp. Development Grant. The content of this entry does not necessarily reflect the position or policy of the U.S. Government—no official endorsements should be inferred or implied.

References

- [1] R. Govindraj and B. Szymanski, "Synthesizing scalable computations from sequential programs," in *Proc. Scalable High Performance Computing Conference 1992*, Williamsburg, IEEE Computer Science Press, Washington, DC, 1992, pp. 228–231.

- [2] B. Szymanski, "Scalable software tools for parallel computations," in *Software for Parallel Computation* (J. Kowlik and L. Grandinetti, eds.), NATO ASI Series F, Springer Verlag, Berlin, 1994, vol. 106, pp. 76–90.
- [3] B. Szymanski, "EPL - parallel programming with recurrent equations," in *Parallel Functional Languages and Environments* (B. Szymanski, ed.), ACM Press, New York, 1991, pp. 51–104.
- [4] J. Bruno and B. Szymanski, "Analyzing conditional data dependencies in an Equational Language Compiler," in *Proc. Third Supercomputing Conference 1988*, Boston, Supercomputing Institute, Tampa, 1988, pp. 358–365.
- [5] C. Özturan, B. Szymanski, and J. E. Flaherty, "Adaptive methods and rectangular partitioning problem," in *Proc. Scalable High Performance Computing Conference 1992*, Williamsburg, IEEE Computer Science Press, Washington, DC, 1992, pp. 409–415.
- [6] K. Spier and B. Szymanski, "Interprocess analysis and optimization in the Equational Language Compiler," in *CONPAR-90. Lecture Notes in Computer Science*, Springer, Berlin-Heidelberg-New York, 1990.
- [7] B. Szymanski and N. Prywes, "Efficient handling of data structures in definitional languages," *Science of Computer Programming*, 1988, vol. 10, pp. 221–245.
- [8] B. Sinharoy and B. Szymanski, "Data and task alignment in distributed memory architectures," *Journal of Parallel and Distributed Computing*, March 1994, vol. 21, pp. 61–74.
- [9] B. Sinharoy and B. Szymanski, "Finding optimal wavefront for parallel computation," *Journal of Parallel Algorithms and Applications*, 1994, vol. 2, pp. 1–22.
- [10] B. McKenney and B. Szymanski, "Generating parallel code for SIMD machines," *ACM Let. Programming Languages and Systems*, 1992, vol. 1, pp. 37–46.
- [11] P. Mehrotra and J. Van Rosendale, "Programming distributed memory architectures using Kali," in *Advances in Languages and Compilers for Parallel Processing* (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, eds.), MIT Press, Cambridge, MA, 1991, pp. 364–384.
- [12] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and W. Wu, "Fortran D language specification," Tech. Rep. COMP TR90079, Department of Computer Science, Rice University, Houston, March 1991.
- [13] B. M. Chapman, P. Mehrotra, and H. P. Zima, "Vienna Fortran - a Fortran language extension for distributed memory multiprocessors," in *Languages Compilers and Run-Time Environments for Distributed Memory Machines* (J. Saltz and P. Mehrotra, eds.), Elsevier, Amsterdam, 1992, pp. 39–62.
- [14] P. Hudak, "Para-Functional programming in Haskell," in *Parallel Functional Languages and Environments* (B. Szymanski, ed.), ACM Press, New York, 1991, pp. 159–196.
- [15] B. Chapman and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, August 1992, vol. 1, pp. 31–50.
- [16] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran - a language specification version 1.1," Tech. Rep. Interim 21, ICASE, NASA, Hampton, VA, March 1992.
- [17] S. Benkner, B. Chapman, and H. Zima, "Vienna Fortran 90," in *Proc. Scalable High Performance Computing Conference 1992*, Williamsburg, IEEE Computer Science Press, Washington, DC, 1992, pp. 51–59.
- [18] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiler support for machine-independent parallel programming in Fortan D," in *Languages, Compilers and Run-Time Environments for Distributed Memory Machines* (J. Saltz and P. Mehrotra, eds.), Elsevier, Amsterdam, 1992, pp. 139–176.
- [19] S. Hiranandani, C. Kennedy, K. Koelbel, U. Kremer, and C. Tseng, "An overview of the Fortran D programming system," in *Fourth Workshop on Languages and Compilers for Parallel Computing*, (Santa Clara, CA), August 1991.
- [20] M. Gerndt and H. P. Zima, "SUPERB: Experience and future research," in *Languages Compilers and Run-Time Environments for Distributed Memory Machines* (J. Saltz and P. Mehrotra, eds.), Elsevier, Amsterdam, 1992, pp. 1–15..

- [21] S. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system.," *IEEE Trans. Soft. Eng.*, November 1981, vol. SE-7.
- [22] X. Ge and N. Prywes, "Reverse software engineering of concurrent programs.," in *Proc. Fifth Jerusalem Conference on Information Technology 1990*, Jerusalem, IEEE Computer Science Press, Washington, DC, 1990, pp. 731-742.
- [23] T. Kaufl, "Reasoning about systems of linear inequalities.," in *Ninth International Conference on Automated Deduction*. Argonne. IL, Springer-Verlag, Heidelberg-Berlin, 1988, pp. 563-572.
- [24] V. Sarkar, "PTRAN - the IBM parallel translation system.," in *Parallel Functional Languages and Compilers* (B. Szymanski, ed.), ACM Press, New York, 1991, pp. 309-391.
- [25] R. Gomory and T. Hu, "Multi-terminal network flows," *SIAM J. of Appl. Math.*, 1961, vol. 9, pp. 551-570.
- [26] A. Pnueli, N. Prywes, and R. Zahri, "Scheduling equational specifications and nonprocedural programs," in *Automatic program construction techniques* (Biermann, Guiho, and Kondratoff, eds.), McMillan, New York, 1984, pp. 273-287.
- [27] B. Sinharoy and B. Szymanski, "Memory optimization for parallel functional programs," in *Abstracts of International Meeting on Vector and Parallel Processing*, Porto, CICA, Portugal, 1993. Submitted to *Computing Systems in Engineering*.
- [28] M. O'Boyle and G. Hedayat, "Data alignment: Transformation to reduce communication on distributed memory architectures," in *Proc. Scalable High Performance Computing Conference 1992*, Williamsburg, IEEE Computer Science Press, Washington, DC, 1992, pp. 366-371.
- [29] K. Knobe, J. Lukas, and G. Steele Jr., "Data optimization: Allocation of arrays to reduce communication on SIMD machines," *Journal of Parallel and Distributed Computing*, 1990, vol. 8, pp. 112-118.
- [30] J. Gilbert and R. Schreiber, "Optimal expression evaluation for data parallel architectures," *Journal of Parallel and Distributed Computing*, September 1991, vol. 13, pp. 58-64.
- [31] B. Szymanski and B. Sinharoy, "Complexity of the closest vector problem in a lattice generated by (0,1)-matrix," *Information Processing Letters*, 1992, vol. 42, pp. 141-146.
- [32] S. K. Rao, *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Department of Electrical Engineering, Stanford University, Stanford, 1985.
- [33] B. Maniatty, B. Szymanski, and B. Sinharoy, "Efficiency of data alignment on MasPar," *SIGPLAN Notices*, 1993, vol. 28, no. 1, pp. 48-51.
- [34] B. Szymanski and T. Caraco, "Spatial analysis of vector-borne disease: A four species model," *Evolutionary Ecology*, 1994, vol. 8. in press.
- [35] G. Andrews, *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1991.
- [36] A. Gibbons and R. Ziani, "The balanced binary tree technique on mesh-connected computers.," *Information Processing Letters*, 1991, vol. 37, no. 2, pp. 101-109.
- [37] S. Miguet and Y. Robert, "Reduction operators on a distributed memory machine with a reconfigurable interconnection.," *IEEE Trans. Parallel and Distributed Systems*, 1992, vol. 3, no. 4, pp. 501-512.
- [38] J. L. C. Sanz and R. Cypher, "Data reduction and fast routing: A strategy for efficient algorithms for message-passing parallel computers.," *Algorithmica*, 1992, vol. 7, no. 1, pp. 77-89.
- [39] B. Szymanski, B. Maniatty, and B. Sinharoy, "Simultaneous parallel reduction," Tech. Rep. CS 92-31, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, September 1992. Submitted to *Parallel Processing Letters*.
- [40] F. Darema-Rogers, V. Norton, and G. Pfister, "A VM parallel environment," Tech. Rep. RC 11225, IBM Corp., Yorktown Heights, NY, 1985.
- [41] H. Jordan, "Parallel computation with the Force," Tech. Rep. 84-45, ICASE, NASA, Hampton, VA, 1985.

- [42] L. Lamport, "The parallel execution of do loops," *Communications ACM*, 1974, vol. 17, pp. 83–93.
- [43] D. I. Moldovan, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Computers*, January 1986, vol. C-35, pp. 1–12.
- [44] J.-P. Sheu and T.-H. Tai, "Partitioning and mapping nested loops on multiprocessor systems," *IEEE Trans. on Parallel and Distributed Systems*, October 1991, vol. 2, pp. 430–439
- [45] P.-Z. Lee and Z. M. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Trans. on Computers*, December 1988, vol. 37, pp. 1578–1598.
- [46] P.-Z. Lee and Z. M. Kedem, "Mapping nested loop algorithms into multidimensional systolic arrays," *IEEE Transactions on Parallel and Distributed Processing*, January 1990, vol. 1, pp. 64–76.
- [47] M. Rosing, R. B. Schnabel, and R. P. Weaver, "Scientific programming languages for distributed memory multiprocessors: Paradigms and research issues," in *Languages, Compilers and Run-Time Environments for Distributed Memory Machines* (J. Saltz and P. Mehrotra, eds.), Elsevier, Amsterdam, 1992.
- [48] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," *Journal of Parallel and Distributed Computing*, October 1991, vol. 13, pp. 213–221.
- [49] C. Koelbel and P. Mehrotra, "Compiling global name-space parallel loops for distributed execution," *IEEE Trans. on Parallel and Distributed Systems*, October 1991, vol. 2, pp. 440–451.
- [50] S. Hiranandani, J. Saltz, M. Piyush, and H. Berryman, "Performance of hashed cache data migration schemes on multicomputers," *Journal of Parallel and Distributed Computing*, 1991, vol. 12, no. 3, pp. 315–422.
- [51] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani, "Distributed memory compiler design for sparse problems," Tech. Rep. TR91-13, ICASE, Hampton, VA, January 1991.
- [52] M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors.," *IEEE Trans. Computers*, 1987, vol. C-36, pp. 570–580.
- [53] D. M. Nicol, "Rectilinear partitioning of irregular data parallel computations," Tech. Rep. 91-55, ICASE, Hampton, VA, 1991.
- [54] J. E. Flaherty, P. J. Paslow, M. Shephard, and J. D. Vasilakis, eds., *Adaptive Methods for Partial Differential Equations*. SIAM, Philadelphia, 1989.
- [55] M. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
- [56] A. Bertossi and A. Gori, "Total domination and irredundance in weighted interval graphs," *SIAM J. Disc. Mathematics*, August 1988, vol. 1, pp. 317–327.
- [57] S. W. Hammond, *Mapping Unstructured Grid Computations to Massively Parallel Computers*. PhD thesis, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, 1991.
- [58] I. Duff, R. Grimes, and J. Lewis, *User's Guide for the Harwell-Boeing Sparse Matrix Collection*. CERFACS, Toulouse Cedex, France, 1992, first ed.
- [59] D. R. Kincaid, J. Respass, D. Young, and R. Grimes, "ITPACK 2C: A Fortran package for solving large sparse linear systems by adaptive accelerated iterative methods," Tech. Rep., University of Texas at Austin.
- [60] D. Skillicorn, *A Model for Practical Parallelism*. Cambridge, U.K.: Cambridge University Press, 1994. To appear.
- [61] T. Fahringer and H. Zima, "A static parameter based performance prediction tool for parallel programs," in *The Seventh ACM International Conference on Supercomputing*, (Tokyo, Japan), ACM Press, New York, July 1993.