Structured Parallel Programming: How Informatics Can Help Overcome the Software Dilemma

HELMAR BURKHART, ROBERT FRANK, GUIDO HÄCHLER

Institut für Informatik, Universität Basel, Mittlere Strasse 142, CH-4056 Basel, Switzerland;
e-mail: {burkhart,frank,haechler}@ifi.unibas.ch

ABSTRACT

The state-of-the-art programming of parallel computers is far from being successful. The main challenge today is, therefore, the development of techniques and tools that improve programmers' productivity. Programmability, portability, and reusability are key issues to be solved. In this article we shall report about our ongoing efforts in this direction. After a short discussion of the software dilemma found today, we shall present the Basel approach. We shall summarize our algorithm description methodology and discuss the basic concepts of the proposed skeleton language. An algorithmic example and comments on implementation aspects will explain our work in more detail. We shall summarize the current state of the implementation and conclude with a discussion of related work.

1 SOFTWARE IS THE PROBLEM

Computer power is a precondition to tackle scientific problems. The term *Grand Challenges* has been coined for the variety of applications that need to be solved, and the discussion has sometimes been reduced to the question of who will have the first teraflop computer. However, this approach has already been critized [2]. We believe that *the real challenge is to develop concepts that enable programmers to use parallel systems in a more productive way*. Today, software development for parallel systems is still in its infancy. Programs are written by using low-level concepts, resulting in software that is often neither portable, reusable, nor maintainable [3].

To overcome this software dilemma, sound software engineering techniques need to be applied. *Structured programming* was introduced by informaticians two decades ago to overcome the software crisis of sequential systems. Other techniques have since followed. It is high time to transfer successful software production techniques to parallel processing!

What we need is progress toward the solution of the "big P" challenges: *programmability - portability - performance*. We like to have programmability at a high level of abstraction to get correct and maintainable software. However, this influences the performance because high-level constructs are always in danger of performance loss. We also prefer to have portable software to minimize software changes if the computer environment changes. But a very high level of portability is again a source of inefficient solutions. In a

Received November 1994
Revised September 1995

Scientific Programming, Vol. 5, pp. 33–45 (1996)
CCC 1058-9244/96/010033-13

word: These three attributes can only be optimized together. It is one of the main challenges today to develop high-level abstractions that still preserve the performance that users expect of parallel systems.

2 THE BASEL APPROACH

What basic elements are necessary for further progress in programming parallel systems? Below, we shall list the problem areas we consider to be priority topics.

2.1 Developing a Taxonomy Describing the Essential Elements of Parallel Algorithms

Many research projects emphasize the need for more "programming tools." There is no doubt that tools are needed to develop parallel processor software (we, too, are building tools within our project as you will see). However, tools can easily become obsolete as soon as new generations of parallel systems are announced. Programming methodology, on the other hand, has to remain stable over a much longer period. This is the reason why the first step will have to be the investigation of methodological aspects, such as the development of a common terminology, algorithmic descriptions and classifications, as well as concepts on how to address software problems. As parallel processing projects are usually interdisciplinary, a common conceptual base used by people with different backgrounds is essential.

2.2 Supporting High-Level Abstractions for Parallel Programming

Programming parallel systems at a mainly system-oriented level is a major weakness of today's environments. Having developed a methodology of the type mentioned above, the next step will have to consist of transferring these concepts into languages and programming tools. Instead of integrating these-concepts within a single language (e.g., High-Performance Fortran) we propose a separation into two language levels:

1. The core of a parallel application, we call it a skeleton, should be written in a language that as much as possible enforces programmability and correctness. This statement certainly applies to programming in general. For parallel systems it is, however, even more important because parallel programs are always more complicated than sequential programs. Carriero and Gelernter [7] already introduced the notion of coordination languages for this level. The coordination of processes is certainly one aspect that is important. However, there are other aspects that are equally important: management of data, support of compositional programming using basic software building blocks, etc. This is the reason why we call it the layer of a *skeleton programming language*.
2. On the basic building block level traditional languages should be used for programming. Today, Fortran and C dominate science and engineering applications. The investment in existing software is tremendous, and informaticians should not ignore these economical aspects.

2.3 Make Parallel Software Reusable

One challenge of software development is the software factory, i.e., the availability of building blocks that may be reused. We cannot expect complete application programs to be reusable because there are always slight differences, even in the same problem domain, e.g., different calculation sequences and different input/output formats. We can, though, expect to get reusable components on two levels of granularity.

1. *Reuse-in-the-large* targets for reusable program parts that are building blocks for application programs. This is possible for regular problems using only a small number of coordination schemes. On massively parallel systems, this trend is emphasized because hundreds or thousands of different processes cannot possibly be managed individually. Libraries of algorithmic skeletons will play a central role.
2. If such a library offers no solution there is still a possibility to apply the *reuse-in-the-small* concept. Reusable process topology and data distribution patterns will always be necessary for writing parallel programs. Such a support can be given either on the language or library level.

2.4 Make Parallel Software Portable

The software dilemma mentioned above has been tackled from the system side. High-level interfaces,

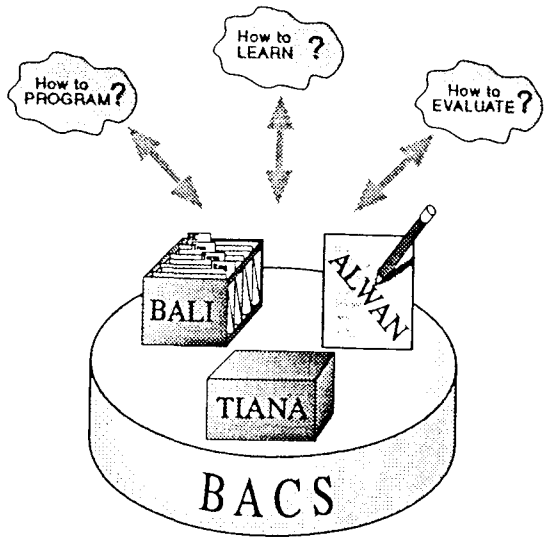


FIGURE 1 The Basel approach.

so-called *parallel virtual machines*, have been defined to hide the existence of the different operating systems and architectures. PARMACS, EXPRESS, PICL, PVM, MPI, and p4 are a subset of the models mentioned in the respective literature ([22] presents an up-to-date list). Most of these models are very similar because they address the same type of parallel computers, so-called message-passing systems. In the near future, standardization (e.g., MPI) will provide a technical basis. It is crucial that software researchers benefit from these developments and offer support in the form of libraries and tools.

Our project provides answers to these four topics (Fig. 1 presents the core items):

1. BACS (Basel Algorithm Classification Scheme), the vocabulary and methodology for describing parallel algorithms and programs.
2. ALWAN, the language used for writing algorithmic skeletons offering reuse-in-the-small constructs.
3. BALI, a library of reuse-in-the-large constructs: algorithmic skeletons that contain ALWAN modules to be modified and extended by programmers.
4. TIANA, the program generator that produces portable source code sksletons for different target systems and programming languages.

This integrated approach provides additional benefits. Different user groups may interact and profit from synergies by using common system elements: programmers (*how to program?*), benchmarkers (*how to evaluate?*), and students (*how to learn?*) who share common system components. Both programmers and benchmarkers benefit from the skeleton approach because skeletons can be completed either into application programs or synthetic benchmark suites by adding artificial loads. Finally, teaching and research nicely interact because the library can offer both courseware and production software.

In this article, we shall summarize BACS and concentrate on ALWAN and TIANA.

3 BACS: A FRAMEWORK FOR PARALLELISM

BACS [4] is a framework for the description and classification of parallel algorithms. As will be shown in Section 4, it also provides a terminological platform on which structured programming of parallel systems can be built.

3.1 BACS Summary

Parallel algorithms can be classified regarding process properties (administration and binding), interaction properties (coordination of processes), and data properties (partitioning and placement).

In BACS, we are refining this list and end up with a generic description tuple that fully characterizes a parallel algorithm (Fig. 2).

Process Properties

The *process topology* defines the geometric structure and connectivity of the process set. We concentrate on regular topologies such as grids, hypercubes, trees, farms, etc. And we distinguish between static and dynamic *process structures*. A process structure is called static if a process topology remains unchanged during the execution of the actual algorithm. As of today, static algorithms are dominating the field of numerical applications. The *execution structure* defines the compositing order of calculation and interaction building blocks. This can be expressed using control structures such as sequences, conditions, and iterations. We created a formula-like notation to provide the algorithm with a kind of signature.

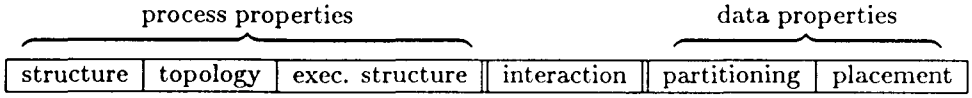


FIGURE 2 Tuple for the classification of parallel algorithms.

Interaction

Interactions define the coordination of the processes at run-time. Coordination operators are used for data exchange, the signaling of events, and for consistency purposes. BACS identifies *direct* interactions that link exactly two processes and *global* interactions involving more than two processes. Global interactions are called *total* if all processes of a topology are interacting. They are called *partial* if only a subset of the processes is involved.

Data Properties

Parallel algorithms normally work with distributed data. The data distribution consists of the *data partitioning* and the *data placement*, i.e., mapping the partitioned data to processes. Arrays are typically partitioned in a blockwise or cyclic manner. Each dimension of an array may be partitioned separately, e.g., a two-dimensional array may be split in rows, columns, or subblocks.

3.2 Example: Systolic Matrix Multiplication

Systolic algorithms are well-known candidates for massively parallel execution. Our sample multiplication algorithm makes use of blockwise distributions of two input matrices A and B, and an output matrix C. The process topology used is a static torus. All processes operate on their matrix blocks and compute an intermediate C block. The regular execution pattern, where interactions (here in two dimensions) are followed by local computations, is typical for systolic design. Thus, the execution structure consists of several parts: a prerotation of the A matrix, a prerotation of the B matrix, and the systolic part, a fixed loop consisting of the calculation and the rotation of one position both for A and B. The BACS tuple is a compact description of these algorithmic properties (Fig. 3). We shall elaborate this example in Section 4.6.

3.3 TINA: The Skeleton Generator Prototype

The tuple information provides a first, coarse-grained view of a parallel algorithm. However, it is too informal to be usable as input for a program generator tool. In his dissertation, Stephan Gutzwiller [6, 16] elaborated the BACS terminology into a script-like, C-based language that can be used to specify all parallel aspects of a program. The script contains entries for the specification of process topologies, data partitions and distributions, and the overall execution structure. Within the script, the programmer also specifies the parallel virtual machine and the programming language for which the code is used.

The program generator, called TINA, reads the input script and produces a source code output. TINA is a kind of text merger. Predefined templates (stored in supporting libraries) are filled with the relevant parameter information. The TINA prototype supports PVM and EXPRESS in a C language environment.

TINA was a rapid prototype used for first portability studies. The script language also bridged the gap between our environment and the more problem-oriented descriptions of an SPP partnership project [10]. Although the prototype was quite useful, we decided to redesign the skeleton language and to put even more emphasis on enhanced programmability. While knowledge collected in the support libraries was reused, the language itself changed completely. The next section will introduce the basic language aspects and an example.

4 ALWAN: A PROGRAMMING LANGUAGE FOR SKELETON PARALLELISM

Below, part of a language called ALWAN is described, with which it is possible to implement an algorithm, starting with a BACS tuple, in a most platform-independent way [5].

ALWAN is based on MODULA-2, a structured high-level language. As ALWAN is a skeleton pro-

Process properties

structure	topology	execution structure	interactions
static	Torus	$I_{Rot}, I_{Rol}, Fix_r(C^t, I_{Rot}, I_{Rol})$	{Rot, Rol}

Data properties

partitioning	placement
A,B,C [Block,Block]	default

FIGURE 3 Classification tuple of systolic matrix multiplication.

gramming language, only a subset of MODULA-2 is used to which a few new concepts had to be introduced reflecting the necessities of parallel programming.

Furthermore, ALWAN allows calling external procedures for the actual calculation parts (which may already be present) or input and output.

4.1 Topologies

Topologies are the core of programming with ALWAN on a parallel machine. A topology specifies the geometric structure of a number of processes as well as the neighborhoods and possible communication paths. A topology is declared very much like a procedure and, on the parallel machine, it can in fact be viewed as the procedure running on each parallel process. All statements within a topology are executed in parallel, all other statements are executed only by one process (the *controller*).

It would prevent reusability if the programmer needed to specify all topology properties each time a slightly different topology is declared. Thus ALWAN introduces a mechanism (`INHERIT`) allowing it to inherit properties from a given topology. We provide a library of frequently used topologies, so the programmer simply needs to inherit one of these topologies and expand the inherited definitions with the data definitions necessary for the given algorithm.

```
TOPOLOGY SystolicMult(p:CARDINAL);
  INHERIT Torus(p,p);
  VAR a: ...
BEGIN
  ...
END SystolicMult;
```

This program sequence defines a new topology inheriting properties from the predefined torus, such as `col_id`, `row_id`, `west`, and `north`. These properties will be used throughout the following examples.

4.2 Communication

Communication is essential on any parallel machine. Shared memory systems define implicit communication while message-passing systems define an explicit one. ALWAN also requires explicit communication, but while message-passing mechanisms quite often require a well-paired send and receive function, ALWAN requires only a special assignment construct that includes both the send and the receive function. Communication in ALWAN is initiated by a simple assignment with the location of the communication data being specified by the `variable@location` construct. Depending on whether the location specifier is on the left side of the assignment or on the right side, the data will be sent or fetched from the view of the initiating process. The directions in which communication takes place (e.g., `west`) depend on topology and may either be inherited from a predefined topology or specified by the `DIRECTION` construct not explained here. The following example describes a communication where processes store the contents of their local variables `'a'` from their western neighbors in their local variables `'a'`. Occasionally, not all processes have to participate in an interaction. The set of processes that initiate this communication may be specified by the `ACTIVE ... DO` construct which acts as a selector.

```
ACTIVE row_id <= i DO
  a:=a@west;
END
```

This statement defines three groups of processes: the initiating processes that will start a com-

munication, the passive processes that are the communication partners of the former, and the processes not participating in the communication. Obviously, the active and the passive groups are not necessarily disjunct.

4.3 Data Partitioning and Distribution

To use a parallel machine efficiently, it is necessary to distribute the usually large amount of data to the different processes. To distribute data, they must first be partitioned. With a new construct (PARTITIONED AS), each index of an array can be indicated as split blockwise (BLOCK), split in a cyclic manner (CYCLE), or not partitioned (NONE). The following array declaration shows how to partition a matrix into rectangular data blocks:

```
ARRAY [0..n-1],[0-m-1] OF LONGREAL
    PARTITIONED AS
    BLOCK(n CDIV p),BLOCK(m CDIV p);
```

The mapping of partitioned data onto the processes depends on the process topology. Once the data are distributed, they can be viewed in two ways: globally, with each process able to access its part using the global index (e.g., considering the array declared above: the first process can access [0, 0] . . . [0, m CDIV p −1], the next one [0, m CDIV p] . . . [0, 2 * (m CDIV p) −1], and so on), or locally when declared as PART OF, with all processes able to access their parts with each index starting at 0.

In the example below, PART OF refers to a distributed variable A, making it an alias to the corresponding local part of A:

```
VAR A : PMatrix;
...
TOPOLOGY ...
  VAR
    a : PART OF A;
```

4.4 Dynamic Types

ALWAN provides the possibility to declare "data templates," called dynamic types. The syntax is very similar to that of a normal type declaration except that the identifier has a parameter list just like a procedure and that the range indexes and the arguments of the partitioning functions are composed of the variables listed in the parameter list as well as any constant expressions. Dynamic variables are dimensioned at run-time with the DIM(...) function.

```
TYPE
  PMatrix(n,m,p:CARDINAL) =
    ARRAY [0..n-1],[0..m-1] OF LONGREAL
    PARTITIONED AS
    BLOCK(n CDIV p),BLOCK(m CDIV p);

VAR
  A : PMatrix;
...
DIM(A,p.nA,p.mA,p.proc);
...
```

4.5 Input and Output

Parallel input and output are nontrivial. In general, this is highly specific to hardware and has to consider several issues such as byte sex for heterogeneous systems, host–node communication, true parallel I/O, etc. To provide a certain level of portability, ALWAN defines two constructs (INPUT and OUTPUT) that are completely independent of the platform and allow the input or output of distributed variables.

Instead of building a huge library of special low-level input–output routines, ALWAN requires the programmer to supply a procedure to write to or read from media using a buffer. The ALWAN input and output statements will then take care of distributing or collecting the data.

```
INPUT A USING readFileA;

PROCEDURE readFileA(
   VAR elem: ARRAY OF LONGREAL;
   coord: ARRAY OF INTEGER,
   length:CARDINAL):INTEGER; EXTERNAL;
```

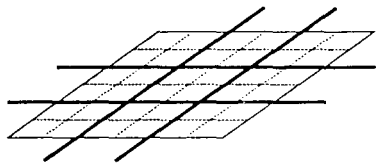
Any user-specific I/O routines have to be declared. If declared as EXTERNAL, they are written in another language such as C. The corresponding function prototype will be generated by the skeleton generator and may look like this:

```
short int readElement(float *e,
                      short int *coord,
          unsigned short int length);
```

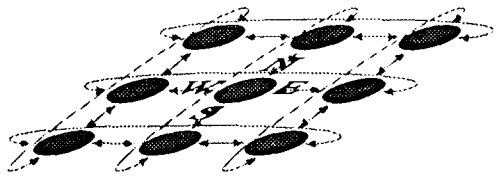
4.6 Example Skeleton—Systolic Matrix Multiplication

Figure 4 shows the ALWAN program for the systolic matrix multiplication algorithm described in

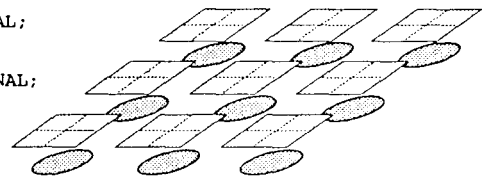
```
1: MODULE Sys;   (* 14.10.94 *)
2:
3:   FROM TLib IMPORT Torus;
4:
5:   TYPE
6:     Parameter! = RECORD
7:                    proc,nA,mA,nB,mB: CARDINAL;
8:                  END;
9:
10:    PMatrix(n,m,p:INTEGER) =
11:      ARRAY [0..n-1],[0..m-1] OF LONGREAL
12:        PARTITIONED AS
13:        BLOCK(n CDIV p), BLOCK(m CDIV p);
14:                                    (* cf figure a*)
15:  VAR
16:    A,B,C: PMatrix;
17:    p: Parameter;
18:
19: PROCEDURE readParameter(VAR p:Parameter);
20:                          EXTERNAL;
21: PROCEDURE readFileA(VAR elem:ARRAY OF LONGREAL;
22:                      coord : ARRAY OF INTEGER,
23:                      length:CARDINAL):INTEGER; EXTERNAL;
24: PROCEDURE readFileB(VAR elem:ARRAY OF LONGREAL;
25:                      coord : ARRAY OF INTEGER,
26:                      length:CARDINAL):INTEGER; EXTERNAL;
27: PROCEDURE writeFileC(VAR elem:ARRAY OF LONGREAL;
28:                       coord : ARRAY OF INTEGER
29:                       length:CARDINAL):INTEGER; EXTERNAL;
30:
31: PROCEDURE initialize(VAR mat: ARRAY OF LONGREAL;
32:                       n,m: INTEGER); EXTERNAL;
33: PROCEDURE multiply(VAR mat: ARRAY OF LONGREAL;
34:                    nA,mA,mB: INTEGER); EXTERNAL;
35:
36: TOPOLOGY SystolicMult(p:CARDINAL);
37:   INHERIT Torus(p,p); (* cf figure b *)
38:   VAR
39:     a : PART OF A; (* cf figure c *)
40:     b : PART OF B;
41:     c : PART OF C;
42:     i : INTEGER;
43: BEGIN
44:   initialize(c,HIGH(c,1),HIGH(c,2));
45:   FOR i := 0 TO p-2 DO
46:     ACTIVE row_id <= i DO (* cf figure e *)
47:       a:= a@west
48:     END;
49:     ACTIVE col_id <= i DO
50:       b := b@north
51:    END
52:   END;
53:
54:   FOR i := 0 TO p-1 DO
55:     multiply(c,a,b,HIGH(c,2),
56:              HIGH(c,1),HIGH(a,1));
57:     a := a@west; (* cf figure f *)
58:     b := b@north
59:   END
60: END SystolicMult;
61:
62:
63:BEGIN
64:  INPUT p USING readParameter;
65:  IF p.mA = p.nB THEN
66:    DIM(A,p.nA,p.mA,p.proc);
67:    DIM(B,p.nB,p.mB,p.proc);
68:    DIM(C,p.nA,p.mB,p.proc);
69:    INPUT A USING readFileA; (* cf figure d *)
70:    INPUT B USING readFileB;
71:    SystolicMult(p.proc);
72:    OUTPUT C USING writeFileC
73:  END
74:END Sys.
```



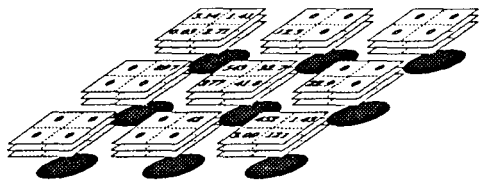
a) Data partitioning



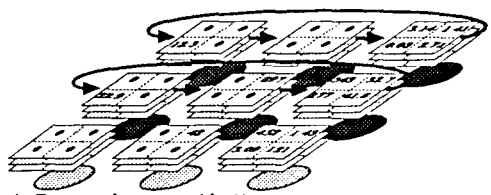
b) Inherit process properties



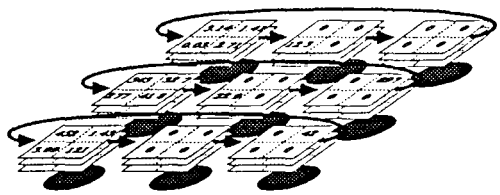
c) Data mapping



d) Data input



e) Prerotation step (i=1)



f) Rotation after multiplication step

FIGURE 4 The ALWAN program: systolic matrix multiplication.

Section 3.2. The pictures on the right illustrate the basic components and intermediate states during program execution.

The distribution pattern of the matrices is declared on lines 10 to 13 and the corresponding memory blocks are allocated on lines 66 to 68. Local views of the matrices are defined on lines 39 to 41. The parallel processes (TOPOLOGY) are declared on lines 36 to 60 and activated on line 71 by the controller. The matrices A and B are prerotated on lines 45 to 52, during which not all processes participate at all times (ACTIVE). On lines 54 to 59 the local computation (line 55) and the rotation (lines 57 and 58) alternate in an iteration loop. The two matrices A and B are input on lines 69 and 70 and the resulting C is output on line 72.

5 TIANA: THE PORTABILITY PLATFORM

Algorithms specified in ALWAN can be transformed into programs for various parallel architectures. This section provides a brief survey of how to transform an algorithm description into an executable program suitable for running on a parallel machine. This transformation process is outlined in Figure 5 to which the roman numerals refer.

First, one has to describe the algorithm skeleton (I) using the ALWAN notation, or even better, retrieve a similar description from the skeleton library, BALI (II), and change it according to needs. The BACS methodology can assist in finding appropriate descriptions.

From this description a skeleton source code (III) is generated using the skeleton generator TIANA (IV). As ALWAN supports the module concept, each ALWAN program can use predefined modules, where frequently used topologies or routines are collected in libraries (V).

The gaps in the generated source code skeleton are exactly the procedures declared as EXTERNAL; a well-defined interface (function prototype) is generated for each of them. These procedures have to be implemented (VI) but may often be extracted from an existing sequential program requiring few changes. These routines normally form the dominant part of the entire program code.

Finally, all code parts are compiled on a specific target machine (VII) and linked with the TIANA library (VIII) to form an executable program (IX). The TIANA library is implemented for various machines. It is the interface to the virtual machine available on the given platform.

Porting an application to a different platform only requires a recompilation on the target machine, replacing the TIANA library with the appropriate new one. Changes in the (external) computation code segment also only require a recompilation on the target system, provided that the interfaces did not change. A complete recompilation is only required after modifying the ALWAN source, e.g., when changing interfaces or distribution patterns.

6 THE TIANA PROGRAMMING SYSTEM

The compiler translates ALWAN programs to C source code which makes calls to the TIANA library to connect to a virtual parallel machine. The translation of the sequential ALWAN parts (a subset of MODULA-2) is straightforward. The translation scheme of the parallel extensions and the dynamic arrays is explained below.

TIANA is designed as a two-pass compiler in which the first pass builds symbol tables and syntax trees of a program and does syntactical and semantical analysis, whereas the second pass generates the actual target source code.

ALWAN enforces a strong typing concept allowing the compiler to catch many possible errors during the first pass. Run-time checks can be enabled which verify the integrity of index ranges and assignments. By specifying appropriate switches, references can be included into the target C code, which allow tracing errors back to the ALWAN code for debugging, as well as instructions which allow recording tracing or timing information.

As ALWAN supports the module concept, information of the imported modules is required. The compiler uses the output of the first pass when importing a module, preventing subsequent recompilations. Thus, the output of the first pass is stored to an intermediate file (the reference file).

6.1 Topologies

Parallel programs consist of parallel code, which is executed on different nodes in parallel, and sequential code, which is executed only by a special node (controller). Such a program can be implemented in at least two ways:

1. Controller and parallel parts are written in separate programs (host–node paradigm).
2. Only one program exists (SPMD paradigm).

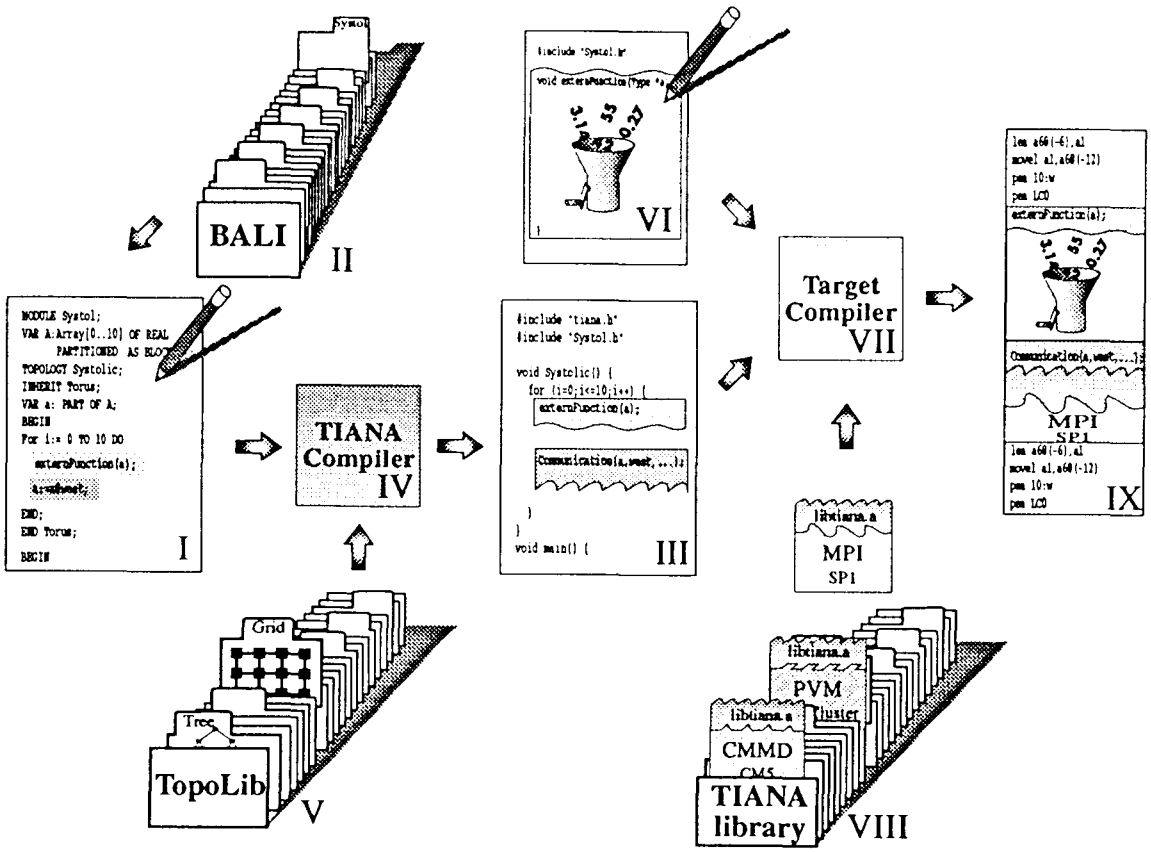


FIGURE 5 System overview.

TIANA will support both paradigms. In the following, we will refer to the SPMD paradigm.

In an ALWAN program it is easy to distinguish between parallel and controller code: All statements within a topology belong to the parallel part, all other statements to the controller. The implementation is more complex. Whenever shifting from controller to parallel mode, modified data have to be updated on all nodes. Two schemes seem to be feasible:

1. Only one process executes the controller parts and marks the modified data to be broadcast on transition to parallel execution.
2. All nodes execute the controller parts simultaneously preventing broadcasts but causing redundant computations.

Both schemes have their advantages and disadvantages depending on the context in which they are implemented.

Each topology is translated into a C function. The inherited topology's function is called as the first statement of the inheriting topology function. Exported topology variables cannot be translated into local function variables as they have to be accessible by both the inherited and the inheriting function. In the current implementation they are thus translated into variables in the global name space. To prevent naming conflicts, these variables are prefixed by the topology's name.

The system provides a set of topologies for convenience. These are written in ALWAN and collected in a library called TopoLib. Currently defined are farm, pipe, ring, mesh (2 and 3 dimensions), torus (2 and 3 dimensions), hypercube, and tree.

6.2 Communication

In ALWAN, communication is described by an ACTIVE statement and an assignment statement with a direction or group specification. This simple

description must be transformed into the usually complex procedure required by the target system.

Three communication patterns are possible when a group constructor is given.

```
ACTIVE TRUE DO
 dst: =src@group
END
```

is mapped to an all-to-all communication, where *dst* has to be an array the size of the number of members in the group. (In this case, the ACTIVE TRUE DO and END may be omitted.)

```
ACTIVE <condition> DO
 dst: =src@group
END
```

is mapped to a many-to-one communication, where *dst* again is an array as above.

```
ACTIVE <condition> DO
 dst@group; =src
END
```

is mapped to a one-to-many communication. The contents of *dst* are not determined if the condition evaluates to TRUE on more than one process per group.

TIANA maps the ACTIVE condition to code specifying whether a process will participate in a communication and whether it will send, receive, or send and receive data. Other parameters such as subrange descriptors and those of partner processes (DIRECTION, GROUP) are produced and passed to a library call. This library must be implemented for each virtual machine. Again, the high level of abstraction of these library calls facilitates an efficient implementation.

6.3 Dynamic and Partitioned Arrays

In Section 4 we introduced the concept of dynamic and partitioned data. Some parameters describing the shape of these data will only be available at run-time. Memory to hold these parameters as well as the actual data needs to be allocated. It is possible to describe the shape of any partitioned array in a finite set of parameters as shown in Figure 6.

6.4 Subrange Assignment

Subrange assignments allow moving parts of arrays, e.g., when exchanging borders. These assignments are translated into a code section defining appropriate descriptors and a library call, which performs the actual data movement, using the previously constructed descriptors. This implementation works equally well for subranges in local assignments, communication, and I/O. This very high level of describing data movement allows optimized adaptations to the target environments.

6.5 Input and Output

TIANA generates descriptors and library calls for the diverse input and output functions. Three kinds of I/O are distinguished:

1. I/O on global data, declared outside a topology—handled only by the controller process, possibly requiring a broadcast.
2. I/O on local data, declared within a topology—may be done independently (where a special scheme has to be implemented for target systems not supporting parallel I/O).
3. I/O on partitioned data—is handled either independently or by the controller, using the parameters describing the partitioned data as explained in Section 6.3.

7 CURRENT STATE OF THE IMPLEMENTATION

A first version of the TIANA compiler is implemented and produced both ANSI and K&R C code. Compile-time errors are detected but currently no code for run-time checks is generated. Both passes of the compiler are written in a recursive-descent manner. The result of the first pass is stored in an intermediate reference file on which the second pass of the compiler is based. The first pass consists of approximately 7,000 lines of C code including the scanner and routines for storing and loading the reference file. The implemented parts of the second pass add up to approximately 3,500 lines.

The example in Figure 4, which is 74 lines of ALWAN code, is translated to 134 lines of C code. The TIANA libraries containing communication

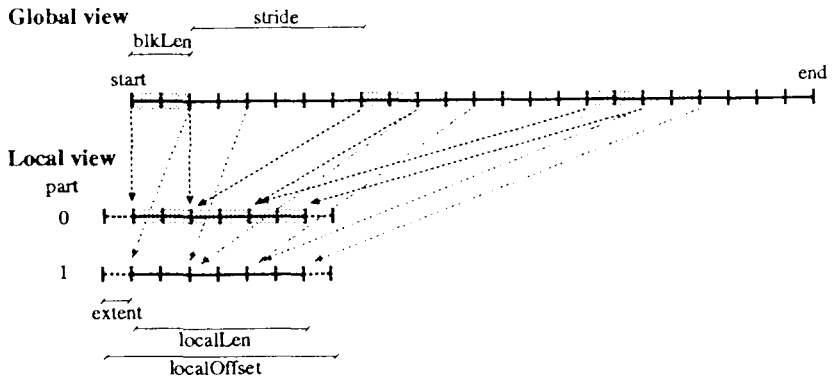


FIGURE 6 Parameters describing partitioned data.

and I/O routines are split into a part independent of the underlying virtual machine (186 lines) and a part depending on the virtual machine (PVM: 392 lines; MPI: 311 lines; CMMD: 250 lines; NX: 248 lines).

The generated code was successfully compiled without any changes on a CM5, SP1, Paragon, and a workstation cluster containing NeXTs and Suns.

Other algorithms, such as the LU-decomposition, the computation of a transitive closure of a graph, and stencil computations have been proven to work. We also use the system for teaching parallel programming on the undergraduate level.

Detail analysis of the performance is necessary and one of our goals (see Section 1). Other short-term goals are further virtual machine interfaces and mixed language support.

8 RELATED WORK

Other projects at our Parallel Processing Laboratory have similar goals. PEMPI is a programming environment based on MPI that uses the BACS terminology to increase programmability by providing higher abstracts compared to MPI [12]. The ALPSTONE project uses ALWAN and TIANA for performance prediction and portable benchmark generation [20]. The BALI project targets for software reuse by collecting ALWAN programs together with descriptive information [21].

Our research is, of course, influenced by developments at other sites:

1. PCN and Strand are *coordination languages* that provide compositionality of parallel programs [13, 14]. Like PCN, ALWAN will support mixed-language computations and *compositional programming*. Similar developments have been reported for the CAPER programming environment [24]. [23] is a recent summary of innovative parallel languages that have been proposed.
2. *High-level abstractions* similar to ALWAN constructs are reported in the literature. For instance, the scientific modeling language DPML [15] has a similar interaction construct, the C-HELP language [11] includes process topologies, and high-performance Fortran has similar data distribution primitives [19].
3. *Software engineering aspects* have been emphasized in many projects. For instance, *portability* has been exploited in several Esprit projects, such as PPPE, GM-MIMD, GENESIS, and PUMA. See also [17] for a collection of papers addressing both portability and performance aspects. *Reusability* is, for instance, emphasized within the Archetype project where a program library similar to BALI is built [8].
4. *Skeleton-oriented programming* was introduced by Cole [9] within a functional programming context. For procedural language environments, the P4 methodology (P3L language and P3M machine model) developed at the University of Pisa addresses portability and abstract machine issues [1].

9 CONCLUSIONS

Software engineering for parallel systems is a new field as yet. Because productivity of parallel pro-

cessing has to increase dramatically, many of the concepts applied within "sequential" environments need to be revised. Portability and reusability are two of the key issues to be solved. Application platforms representing system designs that are extended by application- and organization-specific code are well known in business software environments (e.g., financial application architecture, insurance application architecture, frequent-flyer applications template). Skeleton-oriented parallel programming is a technique based on similar ideas. We proposed a methodology that guarantees reuse-in-the-small (e.g., reusable process topology and data distribution patterns) and targets toward reuse-in-the-large (reusable program blocks that are composed and parameterized toward complete application programs). Our approach addresses the P-P-P challenge:

1. Programmability improves because a basic set of concepts (BACS) serves as the basis of a language design (ALWAN) and a library design (BALI). Structured parallel programming is particularly emphasized by language extensions that reflect well-accepted design principles.
2. Portability is enhanced because TIANA, our program generator, acts as a portability platform.
3. Of course, performance is the ultimate measure for our approach and initial results are very promising.

Ken Kennedy [18] claims that programming massively parallel systems today shows most of the disadvantages of programming in an assembly language. We agree, but hope that the Basel approach is a step in the right direction.

ACKNOWLEDGMENTS

Niandong Fang, Walter Kuhn, Edgar Lederer, and Gérald Prétôt are working on additional subprojects not primarily discussed in this article. We thank them for their countless suggestions. We also acknowledge former laboratory members Carlos Falcó Korn, Stephan Gutzwiller, Peter Ohnacker, and Stephan Waser for their contributions to earlier project phases. This research is sponsored by SPP IF Grant 5003-034357.

REFERENCES

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "P3L: A structured high-level parallel language, and its structured support," Department of Informatics, University of Pisa, Tech. Rep. TR-36/93, Dec. 1993.

[2] G. Bell, "Ultracomputers: A teraflop before its Time," *Communications ACM*, vol. 35, no. 8, pp. 27–47, 1992.

[3] J. E. Boillat, H. Burkhart, K. M. Decker, and P. G. Kropf, "Parallel programming in the 1990's: Attacking the software problem," Phys. Lett., vol. 207, pp. 141–165, 1991.

[4] H. Burkhart, C. Falco Korn, S. Gutzwiller, P. Ohnacker, and S. Waser, "BACS: Basel Algorithm Classification Scheme," Institut für Informatik, University of Basel, Basel, Switzerland, Tech. Rep. 93-3, March 1993.

[5] H. Burkhart, R. Frank, G. Hächler, P. Ohnacker, and Gérald Prétôt, "ALWAN programmer's manual," Institut für Informatik, University of Basel, Basel, Switzerland, Tech. Rep. 94-4, Nov. 1994.

[6] H. Burkhart and S. Gutzwiller, "Steps towards reusability and portability in parallel programming," in *Proc. IFIP Working Conference WG10.3 on Programming Environments for Massively Parallel Distributed Systems*, Burkhäuser, 1994, p. 147.

[7] N. Carriero and D. Gelernter, "Coordination languages and their significance," *Communications ACM*, vol. 35, pp. 97–107, 1992.

[8] M. Chandi, "Concurrent program archtetypes," in *Proc. Scalable Parallel Libraries Conference*, 1994, pp. 1–9.

[9] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA: MIT Press, 1989.

[10] K. Decker, J. Dvorak, and R. Rehmann, "A knowledge-based scientific parallel programming environment," in *Proc. IFIP Working Conference WG10.3 on Programming Environments for Massively Parallel Distributed Systems*, Burkhäuser, 1994, p. 127.

[11] J. L. Dekeyser, D. Lazure, and Ph. Marquet, "A geometrical data-parallel language," *ACM SIGPLAN Notices*, vol. 29, no. 4, pp. 31–40, April 1994.

[12] N. Fang and H. Burkhart, "PEMPI—From MPI standard to programming environment, in *Proc. Scalable Parallel Libraries Conference*. IEEE Computer Press, 1994, pp. 31–38.

[13] I. Foster and C. Kesselmann, "Language constructs and runtime systems for compositional parallel programming, in *Proc. CONPAR94—VAPP VI*, (B. Buchberger and J. Volkert, Eds. New York: Springer, 1994, LNCS 854, pp. 5–16.

[14] I. Foster, R. Olson, and S. Tuecke, "Productive parallel programming: The PCN approach," *Sci. Prog.*, vol. 1, pp. 51–66, 1992.

[15] R. Francis, I. Mathieson, P. Whiting, M. Dix, H. Davies, and L. Rotstayn," A data parallel scientific modelling language, *J. Parallel and Distrib. Comput.*, vol. 21, pp. 46–60, 1994.

[16] S. Gutzwiller, "Methoden und Werkzeuge des skelettorientierten Programmierens," PhD Thesis, University of Basel, 1994 (in German).

[17] T. Hey and J. Ferrante (Eds.), *Portability and Performance for Parallel Processing*, New York: John Wiley and Sons, 1994.

[18] K. Kennedy, "Software for supercomputers of the future," *J. Supercomput.*, pp. 251–262, 1992.

[19] Ch. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, *The High Performance Fortran Handbook*, Cambridge, MA: MIT Press, 1994.

[20] W. Kuhn," *The ALPSTONE Project: An Overview of a Performance Modelling Environment*, Basel: Institut für Informatik, University of Basel, Basel: 1995.

[21] W. Kuhn, P. Ohnacker, and H. Burkhart, "Support for software reuse: The Basel Algorithm Library BALI," in *ParCo95 Conference*, Ghent (Belgium), September 1995.

[22] Parallel Computing, Special Issue on Message Interfaces, vol. 20, no. 4, April 1994.

[23] D. Skillicorn and D. Talia (Eds.), *Programming Languages for Parallel Processing*, New York: IEEE Computer Press, 1995.

[24] B. Sugla, "Parallel application programming," in *Proc. Int. Conf. on Computers and Education*, New York: McGraw-Hill, 1994, pp. 174–187.