A Compiler Approach to Scalable Concurrent Program Design[1]

Ian Foster
Argonne National Laboratory
and
Stephen Taylor
California Institute of Technology

Abstract

The programmer's most powerful tool for controlling complexity in program design is *abstraction*. We seek to use abstraction in the design of concurrent programs, so as to separate design decisions concerned with decomposition, communication, synchronization, mapping, granularity, and load balancing. This paper describes programming and compiler techniques intended to facilitate this design strategy. The programming techniques are based on a *core programming notation* with two important properties: the ability to separate concurrent programming concerns, and extensibility with reusable *programmer-defined abstractions*. The compiler techniques are based on a simple *transformation system* together with a set of *compilation transformations* and *portable run-time support*. The transformation system allows programmer-defined abstractions to be defined as source-to-source transformations that convert abstractions into the core notation. The same transformation system is used to apply compilation transformations that incrementally transform the core notation toward an abstract concurrent machine. This machine can be implemented on a variety of concurrent architectures using simple run-time support.

The transformation, compilation, and run-time system techniques have been implemented and are incorporated in a public-domain program development toolkit. This toolkit operates on a wide variety of networked workstations, multicomputers, and shared-memory multiprocessors. It includes a program transformer, concurrent compiler, syntax checker, debugger, performance analyzer, and execution animator. A variety of substantial applications have been developed using the toolkit, in areas such as climate modeling and fluid dynamics.

[1]This research is sponsored by the Defense Advanced Research Projects Agency, DARPA Order 8176, monitored by the Office of Naval Research under contract N00014-91-J-1986, and by the National Science Foundation under Contract NSF CCR-8809615.

1 The Approach

This paper describes a compiler-based approach to the design of scalable concurrent programs. The approach is motivated by the view that significant advances in concurrent programming will not be achieved through compiler strategies that accept existing sequential programs. The design and implementation of new concurrent programming strategies and algorithms are our primary concerns; we seek simple, flexible tools to support this activity.

1.1 Abstraction

The programmer's most powerful tool is *abstraction*, the ability to neglect unimportant details until the appropriate time. Modern computer science has given us two basic methods by which to use abstraction in program design: *information hiding* [34] and *stepwise refinement* [41]. Both of these development methodologies attempt to separate concerns and place implementation details in unique components of a program. These strategies improve program clarity, localize change thus improving maintainability, and isolate system dependencies, thus improving portability. These concepts are the foundation upon which we strive to design large, correct, maintainable computer programs.

These basic program development methodologies are in principle directly applicable to concurrent program design. However, this requires the ability to delay and to separate design decisions specific to concurrent programming. At the lowest level these decisions concern the techniques used to achieve communication and synchronization and the definition of architectural specifics, such as connection topology and number of computers. During the design process there are other concerns: program decomposition, the granularity of the components, the mapping of components to computers, and load-balancing strategies. It should be possible to consider these concerns separately, isolate them in unique areas of a program, reason about alternatives, and reuse common strategies.

Unfortunately, concurrent programming systems often force a premature commitment to important design decisions or entangle unrelated aspects of a design. For example, designs expressed in terms of a small number of heavyweight processes necessarily encapsulate decisions concerning granularity; these decisions are difficult to change as a program scales to larger numbers of computers. An early commitment to a globally shared data structure, as an means of communication between subprograms, may hinder subsequent partitionings for execution on multicomputers. Many first-generation message-passing systems equate a process with its location, immediately entangling the unrelated concepts of mapping, communication, topology, and number of computers.

1.2 Basic Concepts

Early commitments in program design can be avoided by adopting an abstract, architecturally independent view of communication, synchronization, and concurrent execution. This architectural independence can be achieved by using a programming model based on four simple concepts: monotonicity, concurrent composition, choice between alternatives,

and separation of sequential code [19]. The notion of *monotonicity* provides an abstract model of communication and synchronization. *Concurrent composition* is used to specify opportunities for parallel execution. *Choice* is used to select between alternative program actions. Finally, *separation of sequential code* simplifies the use of state change and sequencing.

These concepts are language independent and have been incorporated into a commercially available programming system, Strand [21]. In this paper, we work with a second-generation system in which programs are expressed in a program composition notation (PCN) [8]. This notation provides a uniform treatment of concurrent composition, non-deterministic choice, and sequential programming. In addition, a simple syntax and the use of recursively-defined data structures allows PCN programs to be represented concisely as data structures. These data structures can in turn be manipulated by PCN programs that implement source-to-source transformations.

PCN programs may operate either *concurrently*, with communication and synchronization, or *sequentially*, by modifying memory. Yet they have the beautiful compositional qualities and declarative semantics that are generally associated with only functional and logic programs. Furthermore, PCN programs may incorporate pre-existing components written in sequential languages such as C, C++ or Fortran, thus supporting migration from sequential to concurrent programming.

1.3 Programmer-defined Abstractions

Although concurrent programming introduces additional concerns that are not present in sequential programming, these concerns are frequently application-independent. For example, when applying domain decomposition to problems of static structure, we must address the issues of partitioning, communication, mapping, and granularity. However, these issues are for the most part associated with the technique of domain decomposition rather than the problems to be decomposed. Similarly, although irregular computations typically require load-balancing strategies, the strategy can usually be specified in application-independent terms.

This independence between problems and generic solution strategies can be exploited by the use of domain-specific, but problem-independent, abstractions. These capture, in a reusable form, application-independent aspects of program design such as scalability constraints, partitioning, mapping, and granularity. The implementation of an abstraction is combined with problem-specific information to form a complete application. In previous work, we have explored these ideas in the context of mapping [39], self-scheduling computations [18], and tree reduction problems [20]. In this paper, we show how the specification and implementation of such abstractions can be incorporated into the compilation process.

1.4 Compiler Techniques

We seek techniques that permit efficient implementation of concurrent programs, expressed using the concepts described in previous sections, on a wide range of parallel

architectures. These techniques must permit applications expressed using high-level abstractions to attain both the communication and the computational performance of the underlying hardware. In particular, we wish to ensure that communication and synchronization overheads are directly transferred to the application, without multiple levels of system overhead, thus allowing *hardware message performance levels to be attained at the application level*. Similarly, we seek to minimize the impact of synchronization overhead on sequential code, allowing *sequential compiler performance to be achieved in sequential code fragments*.

The approach we have developed to meet these goals is based on the use of source-to-source transformation techniques. Successive transformations incrementally convert concurrent programs expressed in terms of programmer-defined abstractions into low-level executable parallel code. These transformations are applied by a simple programmable *transformation system* that allows complex transformations to be specified as concurrent programs.

As shown in Figure 1, the compilation pipeline involves four main stages. The first stage transforms application programs expressed in terms of predefined or programmer-defined abstractions into PCN. The result of this process is a collection of equivalent programs that implement the abstractions in terms of our four basic concepts (c.f. Section 1.2). The second stage applies a set of compilation transformations to the entire program produced by the first stage. These transformations incrementally transform PCN programs toward a simple canonical form called Core PCN [22]. This canonical form is a high-level representation of a fine-grain, concurrent programming model in which processes receive messages, make simple decisions, perform atomic actions to modify memory, and spawn additional processes.

The third stage translates Core PCN programs into the instruction set of an abstract, fine-grain, concurrent machine. This machine provides basic services such as process scheduling, message-passing communication, synchronization, data structure manipulation, and memory management. The abstract machine incorporates atomic operations that modify data structures and integrates the ability for concurrent programs to invoke pre-existing sequential routines written in C, C++, and Fortran. These routines are compiled with standard native-code compilers; the object code is linked into executable images by a fourth linking and assembly stage.

The abstract machine can be implemented in a variety of ways that trade off efficiency and portability. A general-purpose run-time system, or *emulator*, has been produced that executes the instruction set of the abstract machine directly [22]. This emulator is written in a portable subset of C that allows it to operate on a wide class of architectures; it typically compiles to a binary image of less than 100 Kbytes. Currently, the emulator operates on Sun, Next, IBM, DEC, SGI, and HP workstations, on Intel iPSC 386/860/Delta and Symult S2010 multicomputers, and on Sequent Symmetry and Sun shared-memory multiprocessors. The resulting programs have impressive and predictable performance characteristics across a variety of architectures [10, 27].

An alternative abstract machine implementation technique further compiles the encoded abstract machine instructions to make use of specific architectural features. For example, most machines provide high-performance floating point accelerators. The Mosaic

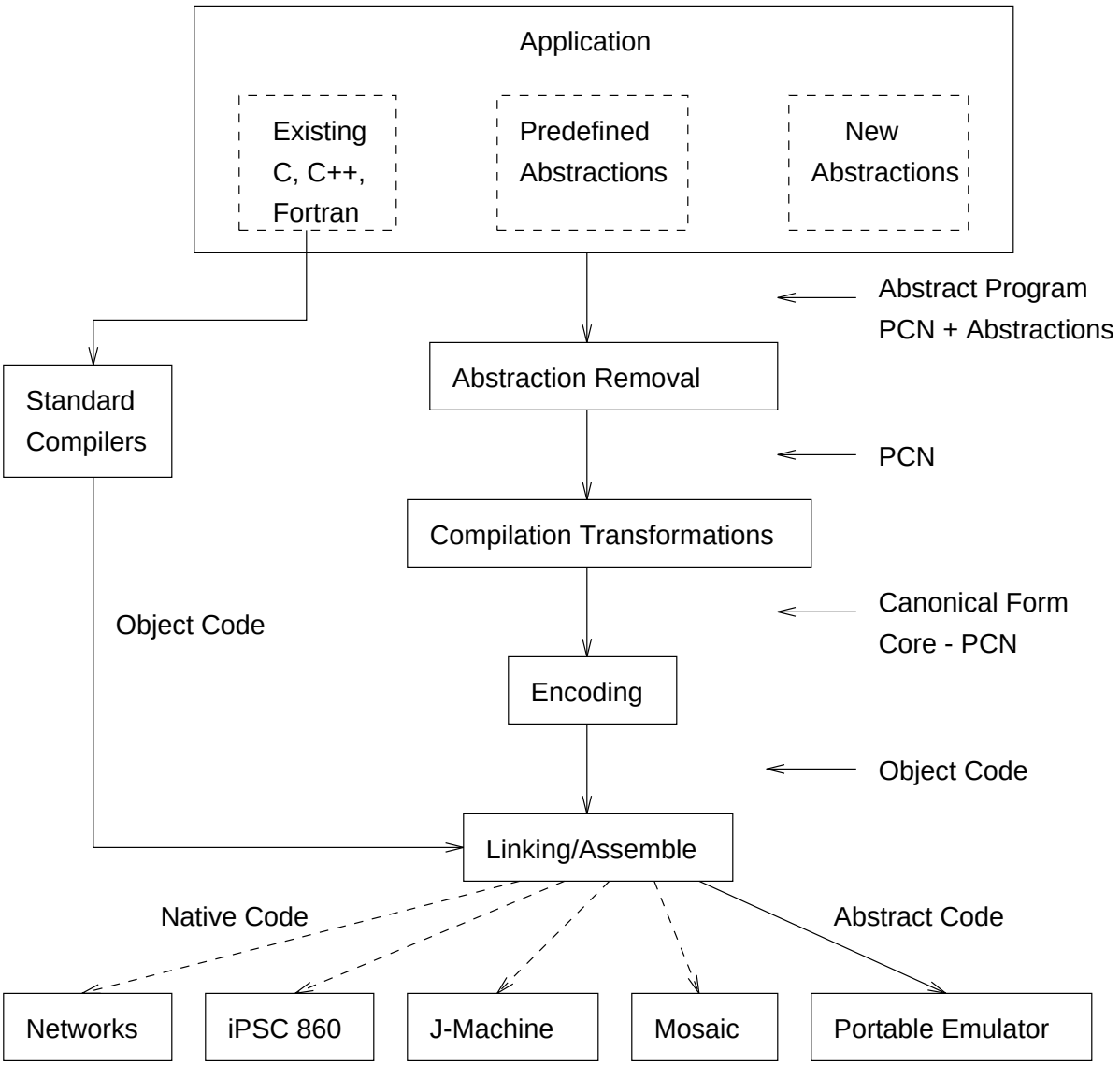

Figure 1: Compilation Strategy

architecture provides high-performance message-handling and fine-grain process scheduling [36]. The J-machine also provides high performance variable and code-manipulation hardware [15]. All of these features may be used to replace unique components of the emulator design, providing high-performance, *native-code* versions of the system. Implementations of this type are currently under construction.

1.5 Summary

The important characteristics of this approach are as follows. We employ a core programming notation based on the four concepts of monotonicity, concurrent composition, choice between alternatives, and separation of sequential code. This allows us to apply standard program development methodologies to cope with typical parallel computing problems. Common abstractions can be isolated in a reusable form and implemented by using source-to-source transformations. Both these transformations and the rest of the compiler are implemented as concurrent programs. A highly portable run-time system can be used to execute programs on a wide variety of architectures. Alternatively, specialized versions of the system can be developed for architectures of particular interest, by retargeting the final stage of the compiler.

2 Related Work

The benefits of an architecturally independent model of parallel computation have been widely recognized in the computer science community [29, 28, 25, 1, 7]. The notion of monotonicity is at the heart of several such programming models, notably concurrent logic programming [11, 24], functional programming [28, 26, 9], and object-oriented programming [1]. Similarly, concurrent composition underlies such diverse approaches as CSP [29], concurrent logic programming, functional programming, and Unity [7]. Unfortunately, these models either do not support concurrent source-to-source transformations or embed the basic ideas in complex language designs and programming paradigms that have little to do with concurrent programming. Furthermore, few approaches are developed to the point where they can be used to develop large-scale applications. We consider the basic ideas to be sufficient in and of themselves and have worked to develop them as a practical basis for concurrent programming [19].

The integration of sequential and concurrent programs has been the focus of a number of other systems, notably large-grain dataflow and Linda [2, 6]. However, we insist upon a clear separation of sequential and concurrent components in order to conveniently apply source-to-source transformation techniques and build programming abstractions. Previous work on reusable abstractions in parallel program design include the Argonne monitor macros [4] and Schedule package [17], and Cole's algorithmic skeletons [14]. However, in none of these approaches is support for abstractions incorporated into a compiler.

An alternative to our compiler techniques is to use run-time techniques such as higher-order functions [28, 31]. However, we prefer to use compile-time methods based on source-to-source transformations so as to avoid run-time overheads and achieve our goals of

efficient communication, synchronization, and sequential execution. The use of "meta-programs" to specify program transformations is common in declarative programming [3, 28, 38, 12, 5, 42]. Novel features of our approach include the integration of a programmable transformer into the compilation pipeline, linguistic support for invocation of transformations, and the use of set-oriented abstractions for specifying transformations. An alternative approach to the implementation of compile-time transformation uses meta-interpreters to specify transformations and partial evaluators to compile away the overhead of interpretation [35]. However, we find the complexity of this approach unnecessary and prefer to implement transformations directly.

The abstract machine design that we employ builds on our previous work in run-time support for concurrent programming [19, 39]. Unlike our previous designs and other uniprocessor systems [25, 30, 40], the PCN abstract machine emphasizes mutable data structures and the integration of sequential procedures, written in languages such as C, C++, and Fortran, into concurrent programs. In addition, we have focused on minimality in order to achieve a higher degree of portability and maintainability.

3 Programming Notations

Recall from Section 1.2 that PCN provides a uniform and convenient notation for the use of four programming concepts: monotonicity, concurrent composition, choice between alternatives, and separation of sequential code. The syntax of PCN is similar to that of the programming language C. Every procedure has the following form (k≥0):

procedure_name(Arg$_1$,Arg$_2$,...,Arg$_k$)
variable_declarations;
composition

where a composition has the form { operator P$_1$,P$_2$,...,P$_n$ } ($n \geq 0$) and operator defines how to execute the component procedures P$_i$. Each component P$_i$ is an assignment, procedure call, or nested composition.

An operator can be one of three basic operators or a programmer-defined operator. The basic operators signify *concurrent* execution (||), *choice* between alternatives (?), or *sequential* execution (;). Concurrent execution specifies that the components P$_1$, ..., P$_n$ are executed in any order or at the same time. Choice specifies that only one component is executed; the determination of which to execute is based on a simple Boolean condition. Sequencing specifies that the components are executed in textual order. A programmer-defined operator is enclosed in angle brackets (e.g., <op>) and signifies the use of an abstraction defined by some transformation. In this case, the appropriate transformation is applied to the procedure at compile time to yield a new procedure employing only the basic operators.

The following simple example illustrates the central PCN concurrent programming concepts and computes the minimum of four numbers.

```
min4(a,b,c,d,result)                    /* 1 */
{|| minimum(a,b,min1),                  /* 2 */
    minimum(c,d,min2),                  /* 3 */
    minimum(min1,min2,result)           /* 4 */
}

minimum(x,y,result)                     /* 5 */
{ ? x >= y -> result = y,               /* 6 */
    x <= y -> result = x                /* 7 */
}
```

The min4 procedure is a *concurrent composition* of three components (1). The first computes the minimum of a and b, producing result min1 (2). The second computes the minimum of c and d, producing min2 (3). Finally, the third computes the minimum of min1 and min2 to produce the final result (4). The minimum procedure uses *choice* to compute the minimum of two numbers (5). If x >= y, then the result is y (6). If x <= y, then the result is x (7). If x and y are equal, then either choice gives the correct result.

Monotonicity. PCN uses an architecturally independent method of specifying communication and synchronization: Components of a parallel composition may exchange information via shared monotone variables. A monotone variable is initially undefined; it can be assigned at most a single value and subsequently does not change. A procedure that requires the value of a variable waits until the variable is defined.

A shared monotone variable can be used to both communicate values and synchronize actions. Notice how the first call to minimum (2) communicates the value min1 to the last call (4) by variable sharing; similarly, the second call to minimum (3) communicates the value min2 to the last call (4).

Consider the effect of the third minimum procedure executing first. In this case the values of min1 and min2 have not yet been produced, and so the procedure call must wait, or *suspend*, until both values are available. This simple data availability test provides a powerful mechanism for program synchronization.

Monotonicity is valuable for two reasons. First, a program can be understood in isolation: choices made on the basis of monotone variables cannot change. This attribute eases the understanding of concurrent programs and avoids errors caused by time-dependent interactions. Second, the concept is trivial to implement efficiently: it maps directly to pointers within a single computer and to message passing between computers. Once available, the value of a variable can be propagated throughout a parallel machine without concern for consistency of copies [39]. Hence, programs can operate on distributed shared data without locking protocols or complex synchronization schemes.

Concurrent Execution. Procedure calls in concurrent compositions are able to execute when their data is available; if data is available, a procedure is guaranteed to execute eventually. The order in which procedures execute is not otherwise constrained. In particular, procedures can be executed in any order or in parallel.

A consequence of monotonicity and concurrent execution is that it is not important where and when procedures execute. Hence, decisions concerning partitioning, mapping,

and granularity can be isolated from the rest of the program design process [8].

Choice. Programs must inevitably choose between alternative actions; this choice is based on the values of variables. We adopt a simple method of specifying program actions that makes such choices explicit and avoids overspecification [16]. This is illustrated in the minimum procedure. Informally, the two rules in this program specify two alternative actions, each with an associated condition. The program can be understood in terms of pre- and postconditions: if x>y holds, z=x will hold eventually, while x<y leads to the postcondition z=y and x=y to the postcondition x=y=z.

This intuitive understanding of the program is valid because of monotonicity and concurrent execution. The monotonicity of x and y ensures that the preconditions are also monotone. For example, once x≥y, this condition holds forever and cannot be affected by actions performed by other programs. Concurrent execution ensures that once a precondition is satisfied, a valid postcondition will eventually be reached.

Separation of Sequential Code. State change and sequencing are familiar concepts from sequential programming. State change permits efficient management of memory via destructive operations to storage locations; sequencing permits state changes to be organized without the overhead of explicit synchronization operations on each access to data [23]. Although these concepts are valuable from a programming perspective, they are dangerous in parallel programs if used in an unrestricted manner, because of the possibility of race conditions. We employ these concepts under the constraint that shared variables are constant, or monotone, during concurrent execution. This constraint can be enforced by the programmer [21] or by the compiler using copying [8].

In this context, a procedure expressed in a conventional language such as C, C++, or Fortran can be viewed as an atomic black box. This box simply computes an input–output relation. Hence, it can be characterized in terms of pre- and postconditions in the same way as parallel program components. This integration of sequential languages into a parallel programming context has a number of benefits. It achieves a clean separation of concerns between sequential and parallel programming, provides a familiar notation for sequential concepts, and enables existing sequential code to be reused in parallel programs.

Mapping. Each invocation of minimum in the min4 procedure can be viewed as a separate locus of control, or *process*. Annotations of the form @location(...) can be added to the min4 procedure to specify how processes are mapped to computers, for example:

```
min4(a,b,c,d,result)                                  /* 1 */
{|| minimum(a,b,min1),                                /* 2 */
    minimum(c,d,min2) @ location(...),                /* 3 */
    minimum(min1,min2,result) @ location(...)         /* 4 */
}
```

In the absence of the annotations, all calls to minimum operate at the *same* computer. This interleaving at a single computer allows *overlapping of communication and computation*. If the location annotations are present, they indicate that a process should execute at an alternative computer within some *virtual machine* [33]. Virtual machines play two primary roles in program design: to reshape the physical machine to a form more

convenient for programming, and to provide scalability by expanding and contracting the physical machine to employ any arbitrary number of computers. Virtual machines may also be used to decompose a physical machine into a collection of submachines, each of which may be allocated a different computation. The combination of location annotations and virtual machines allows concurrent programs to be written that recursively unravel over a parallel architecture [39].

Programming Techniques. Extensive use of these programming ideas has convinced us that they are sufficient for all practical purposes. In particular, it has proved possible to develop a small set of concurrent programming techniques that address the vast majority of issues that arise in concurrent programming. These techniques support the organization of arbitrary communication protocols, termination detection in distributed computations, the construction of distributed data structures, and the implementation of atomic transactions [21, 8].

4 Example Programming Problem

Throughout the rest of this paper, we will repeatedly return to a single example program to demonstrate our programming, compilation, and run-time techniques. This program is a simplied implementation of an application developed to simulate the atmospheric circulation over the globe [10]. The actual code comprises approximately 750 lines of PCN code, 1,400 lines of Fortran, and 870 lines of C. It executes at 2.5 Gflops on the 528-computer Intel Delta and is portable across a wide range of architectures with predictable performance characteristics [10]. The code is typical of other application codes developed at Argonne National Laboratory and Caltech (e.g., [27]). These codes involve both substantial computational components, requiring efficient uniprocessor computation, and complex communication protocols, requiring efficient communication and synchronization.

The application involves the parallel implementation of a control volume method for solving partial differential equations on a sphere. This method is developed by using the icosahedral-hexagonal discretization of a sphere shown in Figure 2(a). This provides greater uniformity than the latitude-longitude grid commonly used for the same purpose. The icosahedral discretization can be structured as ten rhombi, each containing an $N \times N$ mesh, and two polar points. This organization is illustrated in Figure 2(b).

A parallel algorithm is obtained by the application of domain decomposition techniques. Each rhombus is decomposed into a number (say C^2) of *subdomains*, giving a total of $10C^2 + 2$ subdomains, two containing a single polar point and the others each containing $(N/C)^2$ points, where N^2 is the total number of points in a rhombus. The control volume method computes the new value of each grid point at each time step as a function of the previous value of that grid point and a small number of neighbors.

Our implementation of this algorithm is separated into two parts: a reusable abstraction and the application code. The abstraction encapsulates the concurrent programming concepts, defining spherical decomposition, communication structure, and mapping to computers. The application code implements the numerical method for a single subdomain. An operator icosahedron(c) is used to combine the abstraction with the application

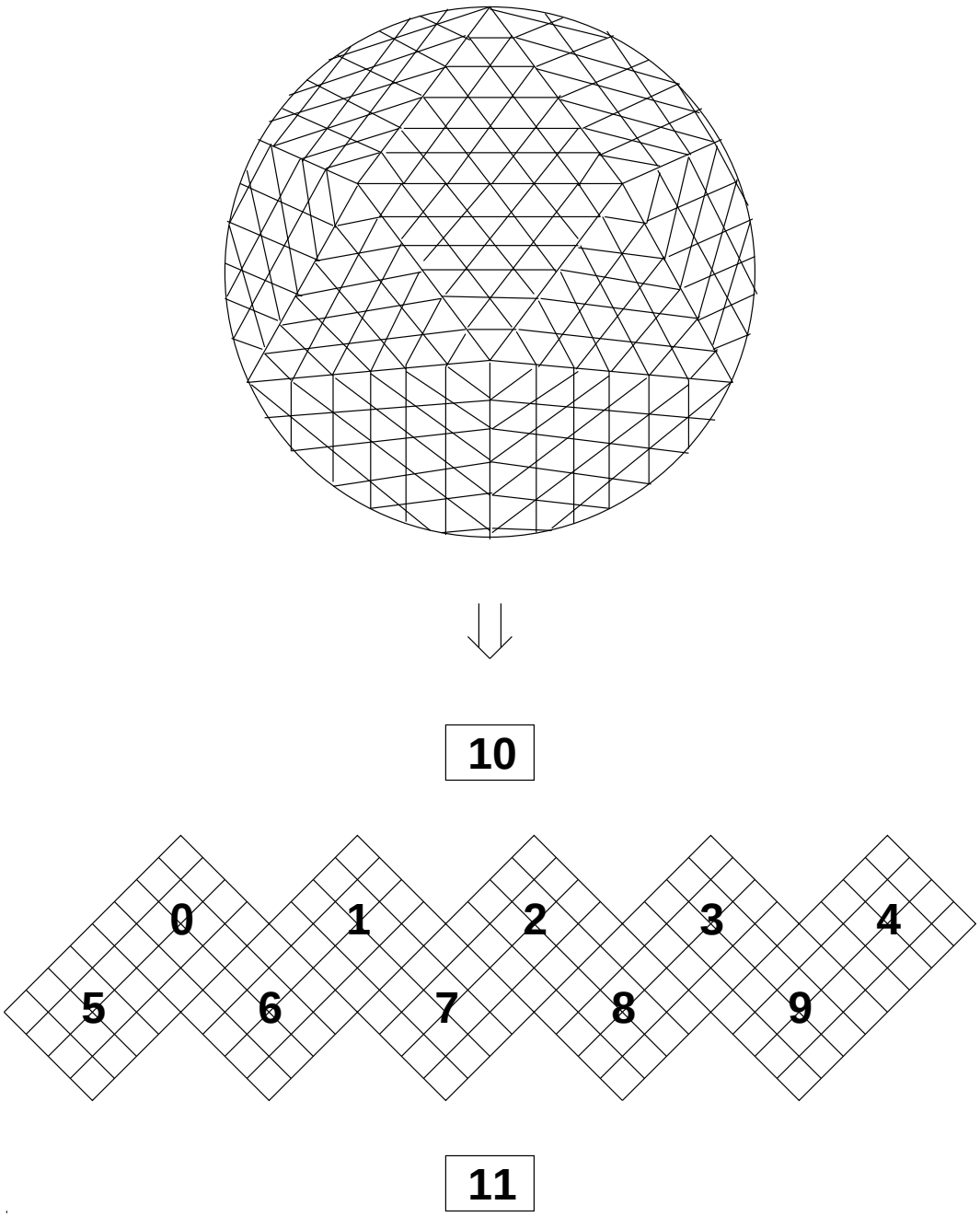

Figure 2: Icosahedral Structure

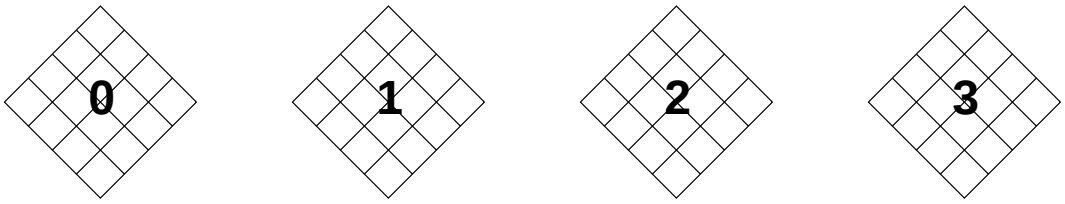

Figure 3: Octahedral Mesh Structure

code, so as to form a complete program. This operator takes as arguments the names of the procedures to be executed at polar and nonpolar subdomains. It triggers application of a source-to-source transformation that generates the necessary concurrent program. For example, the following procedure composes the procedures controlvolume and pole to implement a control volume method on the icosahedral grid.

```
main(c)
{ <icosahedron(c)>
      controlvolume(),
      pole()
}
```

For brevity, we work throughout this paper with the simpler octahedral grid illustrated in Figure 3. This grid has only four rhombi and no polar points. In addition, a five-point stencil is used throughout, meaning that each subdomain requires values from four neighbors. This artificial problem is considerably more homogeneous than the icosahedral grid, which has a mixed seven/six-point stencil with asymmetries at the poles. These complications lead to a more complex communication structure than considered here, but do not change the basic structure of the code or the principles involved in its design.

We show in Program 1 the application code developed for this problem. An octahedron abstraction is used in a manner analogous to the icosahedron abstraction, and the procedure controlvolume() is provided as the application-specific code to be executed in each subdomain. As a consequence of the five-point stencil, this procedure is invoked with eight arguments, representing input and output streams to four neighboring subdomains. When first invoked, it allocates an array to hold the local subdomain, calls the C language procedure c_initialize to initialize this array, and then calls the procedure compute to perform computation. The latter procedure is defined recursively. It repeatedly checks for termination (step<MAX_STEP), extracts and sends boundary values to its four neighbors, receives boundary values from four neighbors, and calls the C language procedure c_update to compute a single step. The syntax no=[edge | no1] denotes the sending of a message edge on a communication stream no; no1 represents the remainder of the stream. The syntax ni ? = [n | ni1] denotes the receiving of a message n on a stream ni; ni1 denotes the remainder of the stream.

```
#define SUBDOMAIN_SIZE 3600
#define EDGE_SIZE 16
#define MAX_STEP 1000
#define NORTH 0
#define EAST 1
#define SOUTH 2
#define WEST 3

main(c)                                          /* Application main program */
{ <octahedron(c)>                                /* Name abstraction */
    controlvolume()                              /* Application-specific code */
}

controlvolume(ni,ei,so,wi,no,eo,so,wo)           /* Application-specific code */
double mesh[SUBDOMAIN_SIZE];                     /* Allocate mesh */
{ ; c_initialize(mesh),                          /* Initialize mesh */
    compute(0,mesh,ni,ei,si,wi,no,eo,so,wo)      /* Execute numerical scheme */
}

compute(step,mesh,ni,ei,si,wi,no,eo,so,wo)
double mesh[], edge[EDGE_SIZE];
{ ? step < MAX_STEP  ->                          /* Until done ... */
        { ; c_get_edge(NORTH,edge,mesh),         /* Get north edge */
            no=[edge | no1],                     /* Send edge north */
            c_get_edge(EAST, edge,mesh),         /* Ditto for east */
            eo=[edge | eo1],
            c_get_edge(SOUTH,edge,mesh),         /* Ditto for south */
            so=[edge | so1],
            c_get_edge(WEST, edge,mesh),         /* Ditto for west */
            wo=[edge | wo1],
            { ? ni ?= [n | ni1], ei ?= [e | ei1],  /* Recv from N and E*/
                si ?= [s | si1], wi ?= [w | wi1]  ->  /* Recv from S and W*/
                { ; c_update(mesh,n,e,s,w),      /* Update mesh */
                    step(step+1,mesh,ni1,ei1,si1,wi1,no1,eo1,so1,wo1)
                }
            }
        },
    default  -> c_dump(mesh)                     /* All done: dump */
}
```

Program 1: Octahedral Application Code

5 Transformation System

Recall that the simple structure of PCN programs allows a concise representation as data structures. These data structures can in turn be manipulated by PCN programs, allowing source-to-source transformations to be specified as concurrent programs that operate on concurrent programs.

5.1 Defining Transformations

To simplify the specification of transformations, we define an abstract data type that implements a *set*. The elements of the set may be programs or program components such as blocks and procedure calls. We provide operations that transform each element of a set, split a set into subsets on the basis of a condition, compute a parallel prefix operation over a set, and form the union of two sets.

```
transform(set,trans_op,newset)
split(set,condition,set1,set2)
combine(set,combine_op,result)
union(set1,set2,newset)
```

Two additional operations support sets of programs. These operations compute unique procedure and variable names.

```
unique_id(set,newid)
unique_var(set,newvar)
```

When extended with the set data type, PCN becomes a powerful tool for implementing arbitrary source-to-source transformations. The basic operations listed above provide building blocks that can be used to implement more sophisticated operations. Libraries of such operations have been constructed and form the basis for the implementation of both the PCN compiler and abstractions such as icosahedron and octahedron. For example, Program 2 implements a useful operation map_over that applies a specified transformation (op) to every procedure call in a program component. This can be invoked in a call of the form

```
transform(set,map_over(op),newset)
```

to produce a newset in which the transformation op has been applied to every procedure call in set. Program 2 uses choice composition and the match operator ?= to distinguish program components representing procedures, blocks, lists of blocks, implications, and calls. The recursive calls to map_over incrementally break down the program structure to isolate program calls. Finally, when a call is isolated, the supplied operator ‘op‘ is applied at the end of the procedure.

Program 3 shows an example transformation defined in terms of map_over. This somewhat artificial example produces a newset, identical to an input set except that all procedures, other than those named procname, have calls to oldname renamed to

```
map_over(op,item,newitem)
{ ? item ?= procedure(id,args,decls,block) ->            /* Body of procedure */
        {|| map_over(op,block,newblock),
            newitem = procedure(id,args,decls,newblock)
        },
    item ?= block(blockop,bs) ->                         /* Blocks in composition */
        {|| map_over(op,bs,newbs),
            newitem = block(blockop,newbs)
        },
    item ?= [b|items] ->                                 /* Blocks in list */
        {|| map_over(op,b,newb),
            map_over(op,items,newitems),
            newitem = [newb|newitems]
        },
    item ?= {" -> ",guard,body} ->                       /* Body of implication */
        {|| map_over(op,body,newbody),
            newitem = {" -> ",guard,newbody}
        },
    default ->                                           /* Apply operator */
        `op`(item,newitem)
}
```

Program 2: Example Transformation Operation

be calls to newname. Note the use of the primitive operations split, transform and union. The split operation calls named to decompose the input set into a set1 containing procedures with the name procname and another set2 containing all other procedures. The transform operation calls map_over to apply the rename transformation to each program call in set2, producing set3. Finally, the union operation is used to combine set1 and set3 to form newset.

```
rename_procedure_calls(set,procname,oldname,newname,newset)
{|| split(set,named(procname),set1,set2),
    transform(set2,map_over(rename(oldname,newname)),set3),
    union(set1,set3,newset)
}

named(name,object,result)
{ ? object ?= procedure(id,args,decls,block) ->
        { ? name == id -> result = "true",
            name != id -> result = "false"
        }
}

rename(oldname,newname,oldcall,newcall)
{ ? oldcall ?= call(id,args,mapping) ->
        { ? id == oldname -> newcall = call(newname,args,mapping),
            default -> newcall = oldcall
        },
    default -> newcall = oldcall                    /* Primitive (e.g., =) */
}
```

Program 3: Example Program Transformation

The conciseness of expression permitted by this approach is evidenced by a recent programming experiment involving the remainder of the PCN compiler. This was originally developed without the use of the transformation system. A new version written with the transformation system implemented many additional optimizations and was nevertheless only one third the size of the original code.

5.2 Transforming the Octahedron Example

We illustrate the use of the transformation system by implementing the octahedron abstraction. This implementation consists of two parts: an *abstraction definition* and *mapping definition*. The abstraction definition is responsible for generating a process and

communication structure required to represent the octahedral mesh. This yields a PCN program in which mapping decisions are specified with respect to a virtual machine, by means of abstract annotations on procedure calls. The mapping definition deals with embedding the virtual machine into a particular physical machine. This separation of concerns allows physical machine dependencies to be isolated in a unique transformation. Typically, these dependencies can be encapsulated in a single procedure or library for a given machine.

5.2.1 Abstraction Definition

The abstraction definition is implemented by a transformation that combines a *library* with the application code given in Program 1. The library, given in Program 4, incorporates solutions to three distinct problems: the partitioning of the data domain into disjoint subdomains, the organization of communication between subdomains to exchange boundary values, and the mapping of subdomains to processors in a parallel computer. As described in [10], this code is developed by a series of refinement steps, each introducing a solution to one of these problems.

The library code creates a process structure comprising $4c^2$ subdomain processes. Each call to rhombus from within sphere creates c^2 processes by calling the row procedure c times, once per rhombus row; each call to row creates c subdomain processes.

Monotone variables are used to define the communication structure required for the use of a five-point stencil. This structure, illustrated in Figure 4, allocates each subdomain communication streams to four neighbors. The procedure sphere establishes the initial connections between the various rhombi, as shown in Figure 4 (a). These initial connections are used to establish connections between the meshes created within each rhombus. Each rhombus produces a list of communication streams on its north (nn) and east (ee) sides and consumes a list of streams on its south (ss) and west (ww) sides; these streams are used for communication between meshes in different rhombi, as in Figure 4 (b). Additional streams are created within the rhombus and row procedures for communication between subdomains within the same rhombus. Notice that the rhombus procedure eventually reduces to a concurrent composition of c^2 start_subdomain processes, at which point each subdomain has four communication streams to its north, east, south, and west neighbors (n, e, s, w). Finally, each of these neighboring streams is converted into a pair of input/output streams, as in Figure 4 (c).

Transformation. The octahedron abstraction requires only a trivial transformation. Recall the following block from Program 1 that uses the octahedron operator:

```
{ <octahedron(c)>
      controlvolume()
}
```

This block is transformed into a call sphere(c) that invokes the sphere procedure of the library in Program 4. In addition, the call to the subdomain procedure, in the library, is renamed to call the subdomain procedure supplied by the abstraction i.e. controlvolumn. This transformation can be specified by the following procedure, that is applied

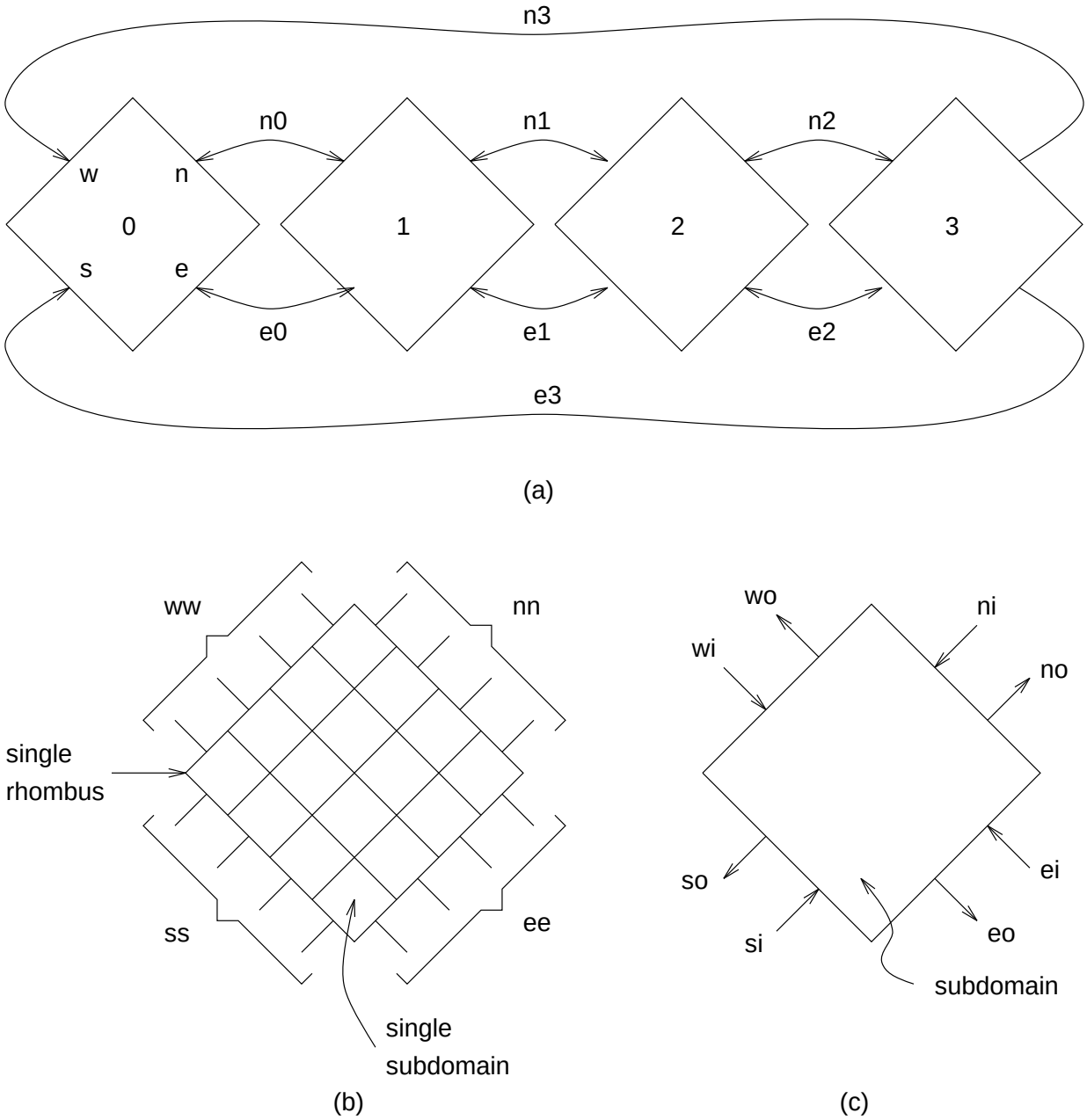

Figure 4: Octahedral Grid Communication Structure

```
sphere(c)
{|| rhombus(0,c,c,n0,e0,e3,n3),                    /* Rhombus 0 */
    rhombus(1,c,c,n1,e1,e0,n0),                    /* Rhombus 1 */
    rhombus(2,c,c,n2,e2,e1,n1),                    /* Rhombus 2 */
    rhombus(3,c,c,n3,e3,e2,n2)                     /* Rhombus 3 */
}

rhombus(r,i,j,nn,ee,ss,ww)                         /* Create a rhombus */
{ ? i > 0 ->
        {|| ee = [e | ee1],                        /* Produce E stream */
            ww ?= [wa | ww1a] ->                   /* Consume W */
                {|| ww1 = ww1a, w = wa}
            row(j,r,i,j,nn,ssm,w,e),               /* Create a row */
            rhombus(r,i-1,j,ssm,ee1,ss,ww1)        /* Recurse for more rows */
        },
    i == 0 -> {|| nn = ss, ee = []}                 /* Done with rhombus */
}

row(c,r,i,j,nn,ss,w,e)                             /* Create a single row */
{ ? j > 0 ->
        {|| nn = [n | nn1],                        /* Produce N stream */
            ss ?= [sa | ss1a] ->                   /* Consume S */
                {|| ss1 = ss1a, s = sa}
            map(c,r,i,j,locn),                     /* Compute mapping location */
            start_subdomain(n,em,s,w) @ locn,      /* Map single subdomain */
            row(c,r,i,j-1,nn1,ss1,em,e)            /* Recurse: more subdomains */
        },
    j == 0 -> {|| e = w, nn = []}                   /* Done with row */
}

start_subdomain(n,e,s,w)
{|| n = {no,ni}, e = {eo,ei},                      /* Make 2 streams */
    { ? s ?= {si,so}, w ?= {wi,wo} ->              /* Get 2 streams */
        subdomain(ni,ei,so,wi,no,eo,so,wo)
    }
}
```

Program 4: Octahedron Abstraction: Library Code

by the compiler to any program block containing the operator octahedron(c); it yields a newblock and a set of new procedures.

```
octahedron(c,block,newblock,set)
{ ? block ?= block(octahedron(c),[proc]) ->
        {|| load("octahedron_library",set1),                          /* 1 */
            transform(set1,map_over(rename("subdomain",proc)),set),   /* 2 */
            newblock = call("sphere",[c],[])                          /* 3 */
        }
}
```

Notice the reuse of the operations map_over and rename specified in Section 5.1. The primitive operation load is used to load the octahedron abstraction library into a new set, set1 (1). Then, the map_over and rename operations are used to rename all calls to "subdomain" (2). Finally, the original block is transformed to be a simple call to the procedure sphere (3).

5.2.2 Mapping Definition

The library code shown in Program 4 uses the notation @locn to signify process mapping. The mapping of the octahedral process structure to a parallel computer is encapsulated in the procedure map, which is called to compute the location of each subdomain process. One simple approach places one subdomain on each processor; this provides scalability at the expense of some non-nearest-neighbor communication. This may be specified as follows.

```
map(c,r,i,j,locn)
{|| locn = r*c*c + i*c + j }
```

An alternative approach is to fold the octahedral mesh so as to ensure nearest-neighbor communications [37]. In this approach, each processor is allocated four subdomains. This constrains scalability, but is useful when remote communication is expensive. The alternative can be implemented simply by redefining the map procedure. If the program is to execute on a $c \times c$ mesh, with processors numbered 0 to c^2-1, then the new definition is as follows.

```
map(c,r,i,j,locn)
{ ? r%2 == 1  -> locn = i*c + j,
    r%2 == 0  -> locn = (c-j)*c - (i+1)
}
```

5.2.3 Developing an Alternative Mapping Strategy

The library and transformation presented in the preceding section succeed in isolating mapping decisions in a separate map procedure. However, many details of the mapping remain in the abstraction library, making it difficult to reuse this library in other circumstances or to apply mappings with a different structure.

To simplify the exploration of alternative mapping strategies, we have developed tools that allow mappings to be specified with respect to a *virtual machine*. Recall that a virtual machine is an abstract architecture that is convenient for solving a programming problem. This approach can be generalized to allow the composition of multiple virtual machines in a hierarchy. This allows elements of the virtual machine structure to be isolated for reuse as shown in Program 5.

```
sphere(c)
{|| rhombus(c,c,n0,e0,e3,n3) @ mesh(0),                /* Map mesh 0 */
    rhombus(c,c,n1,e1,e0,n0) @ mesh(1),                /* Map mesh 1 */
    rhombus(c,c,n2,e2,e1,n1) @ mesh(2),                /* Map mesh 2 */
    rhombus(c,c,n3,e3,e2,n2) @ mesh(3)                 /* Map mesh 3 */
}

rhombus(i,j,nn,ee,ss,ww)
{ ? i > 0 ->
        {|| ...,
            row(j,nn,ssm,w,e),
            rhombus(i-1,j,ssm,ee1,ss,ww1) @ south      /* Map south */
        },
    i == 0 -> {|| nn = ss, ee = []}
}

row(j,nn,ss,w,e)
{ ? j > 0 ->
        {|| ...,
            mesh(n,em,s,w),
            row(j-1,nn1,ss1,em,e) @ east               /* Map east */
        },
    j == 0 -> {|| e = w, nn = []}
}
```

Program 5: Virtual Machine Mapping

For example, an octahedral virtual machine can be constructed by composing four mesh submachines, with each submachine containing c^2 virtual processors. The octahedral virtual machine supports a mapping annotation @mesh(n) that allow us to address the individual mesh machines. Within a mesh virtual machine, we address individual virtual processors using mapping annotations @south, @east, etc., that specify relative locations. This approach simplifies the specification of mapping within an application. For example, by combining the octahedral and mesh virtual machines, we may specify

the mapping as shown in Program 5.

Mapping constructs such as @mesh(i) and @east are themselves abstractions implemented by a combination of source transformations and mapping libraries. We have developed libraries of transformations that allow new virtual machines to be defined by the programmer and combined hierarchically to fit complex application and machine structures.

6 Compilation Transformations

In Section 5, we showed how the transformation system is used to convert programs expressed in terms of abstractions into PCN. We now move to the techniques used to compile PCN programs into executable code. The same transformation system is now used to specify compilation transformations that are used to compile PCN programs. Hence, the entire PCN compiler is a concurrent program that may be executed on multiple computers.

The compilation transformations incrementally transform programs into a canonical form that can be directly encoded into machine instructions. We term this canonical form Core PCN since it reflects the core ideas of the underlying implementation strategy, namely, fine-grain concurrent processes that communicate and synchronize through message passing [22]. These processes execute simple atomic actions that may modify memory.

6.1 Core PCN

All Core-PCN programs have the following form (k^i, l^i, $n \geq 0$):

```
program_name(Args)
declarations
{ ? G1 ->
        { ; A1,...,Ak1 , {| | p1(...),..., pl1(...) }},
            :
    Gn ->
        { ; A1,...Akn , {| | p1(...),..., pln(...) }},
    default ->
        { ; A1,...Akn+1 , {| | p1(...),..., pln+1(...) }}
}
```

In this form, G_i is a PCN guard action, A_i is an atomic action, and p_i is a process invocation. An atomic action is either an assignment or a call to a sequential procedure written in C, C++, or Fortran. Notice that this canonical form contains neither nested composition nor sequential compositions of PCN procedures. Core PCN programs simply receive messages in the guard, modify local state and/or spawn more processes; process synchronization occurs *only* in the guard components of a program.

The operational semantic of a Core PCN program consists of a subset of the semantic for PCN programs [8]; it is identical to that of Strand [21] except that atomic actions may modify data structures. If any guard G_i is *true*, the associated atomic actions are executed, and concurrent processes are then spawned. If all guards are *false*, then the default action is executed. Guard evaluation completes only when sufficient information is available for one of these conditions to be satisfied.

6.2 The Transformations

PCN programs are transformed into Core PCN by a pipeline of five principal transformations. Each transformation is developed using the transformation system described in Section 5, and hence can be specified, understood, and maintained independently. The transformations are described in the sections that follow. Although these descriptions ignore numerous optimizations that are performed in the PCN compiler, they convey the basic structure of the compiler.

Expression Removal. This transformation ensures that concurrent processes may be spawned immediately without waiting for their arguments to be evaluated. It extracts expressions from various locations in a program text and creates assignment statements to evaluate the original expressions. In the following examples, the original code is shown on the left and the transformed code on the right.

```
p(...)                          p(...)
{|| :                           {|| :
    f(...,X+Y,...)      ==>         {|| NewVariable=X+Y,
    :                                   f(...,NewVariable,...)
}                                   },
                                    :
                                }
```

Example Expression Removal

Atomic Action Generation. This transformation moves synchronization operations out of sequential and parallel blocks and into guards. This allows separate optimization of synchronization operations when compiling choice blocks. It also simplifies compilation of arithmetic, memory operations, and sequential subroutines. In particular, they can be compiled directly to sequential code so as to attain the performance of the underlying machine language.

The transformation considers statements such as V=M+V which contain monotone variables for which synchronization is required. For example, if M is monotone, evaluation must delay until M has a value. The transformation achieves this behavior by generating a choice block that performs a data check on the variable M. This ensures that the assignment does not execute until M has a value, at which time it executes as an atomic action and terminates.

Calls to sequential subroutines expressed in C, C++, or Fortran are handled in a similar manner. By ensuring that their data is available prior to subroutine entry, these routines may be treated as atomic actions that terminate immediately.

```
p(...,M,...)                          p(...,M,...)
int V;                                int V;
{ ; V=M+V,                            { ? data(M) ->
    C_program(...,M,V,...)   ==>              { ; V=M+V,
}                                                 C_program(...,M,V,...)
                                              }
                                      }
```

Example Atomic Action Generation

Nested Choice Removal. This transformation allows the underlying abstract machine to use a trivial process suspension mechanism that need not deal with suspension in the middle of procedure execution: Suspension may occur only during guard evaluation. A nested choice block is replaced with a call to a new procedure. This new procedure contains the original nested block. Its arguments are the variables shared by the original block and the enclosing procedure.

```
p(...)                                p(...)
{ ? :                                 { ? :
    { ? x > y -> f(...),      ==>         p.1(x,y,...)
        x < y -> g(...)                   :
    }                                 }
    :
}                                     p.1(x,y,...)
                                      { ? x > y -> f(...),
                                          x < y -> g(...)
                                      }
```

Example Choice Removal

Sequencing Removal. This transformation allows all PCN procedures to be executed as fine-grain concurrent processes. The essence of the idea is to translate sequential blocks into concurrent blocks with some added synchronization. Sequential semantics are retained by passing a token from one concurrent process to another in the order specified by the original program sequencing. Receipt of this token enables process execution.

The transformation achieves this behavior by transforming all sequential and concurrent programs into equivalent programs that wait to be enabled (e.g., data(L)), execute, and then forward the token through an appropriate argument (e.g., R).

```
p(...)                      p(...,L,R)
{ ; f(...),                 { ? data(L) ->
    g(...),        ==>              {|| f(...,L,M1),
    h(...)                              g(...,M1,M2),
}                                       h(...,M2,R)
                                    }
                            }
```

Example Sequencing Removal

Canonical Form Generation. This transformation translates procedures generated by the preceding transformations into the Core PCN canonical form. This involves activities such as combining nested parallel blocks, ensuring that every choice composition has a default implication, and wrapping single procedure calls with parallel composition.

```
p(...)                                  p(...)
{|| {|| f1(...), f2(...) },             {|| f1(...),
    g(...),                   ==>           f2(...),
    {|| h1(...), h2(...) }                  g(...),
}                                           h1(...),
                                            h2(...)
                                        }
```

Example Canonical Form Generation

6.3 Compiling the Octahedral Example

We illustrate the application of the compilation transformations by showing the code produced when they are applied to the compute procedure (Program 1). Notice that this procedure contains both sequential operators and nested choice blocks. The Core PCN generated for this procedure is presented in Program 6. The following aspects of the transformed procedure are important:

- The auxiliary procedure compute.1 is introduced to replace the nested choice block. Notice that the variables used by the nested choice block are passed to compute.1 as arguments and that an argument declaration for the mesh array is inserted.
- A synchronization variable _DE is introduced, to permit other programs to detect termination of compute. This variable is defined only after execution of compute is complete.
- Synchronization operations (data(nb), etc.) are inserted in compute.1 to ensure that calls to the sequential procedure c_update do not suspend.

```
compute(step,mesh,ni,ei,si,wi,no,eo,so,wo,_DE)
double mesh[], edge[EDGE_SIZE];
{ ? step < MAX_STEP  ->
        { ;  c_get_edge(NORTH,edge,mesh), no=[edge | no1],
             c_get_edge(EAST, edge,mesh), eo=[edge | eo1],
             c_get_edge(SOUTH,edge,mesh), so=[edge | so1],
             c_get_edge(WEST, edge,mesh), wo=[edge | wo1],
             {| | compute.1(step,mesh,ni,ei,si,wi,no1,eo1,so1,wo1,_DE) }
        },
    default  -> { ; c_dump(mesh), _DE = [] }
}

compute.1(step,mesh,ni,ei,si,wi,no1,eo1,so1,wo1,_DE)
double mesh[];
{ ? ni ?= [n | ni1], ei ?= [e | ei1], si ?= [s | si1], wi ?= [w | wi1],
    data(n), data(e), data(s), data(w)  ->
        { ;  c_update(mesh,n,e,s,w),
             {| | step(step+1,mesh,ni1,ei1,si1,wi1,no1,eo1,so1,wo1,_DE) }
        },
    default  -> { ; _DE = []}
}
```

Program 6: Core PCN Octahedral Code

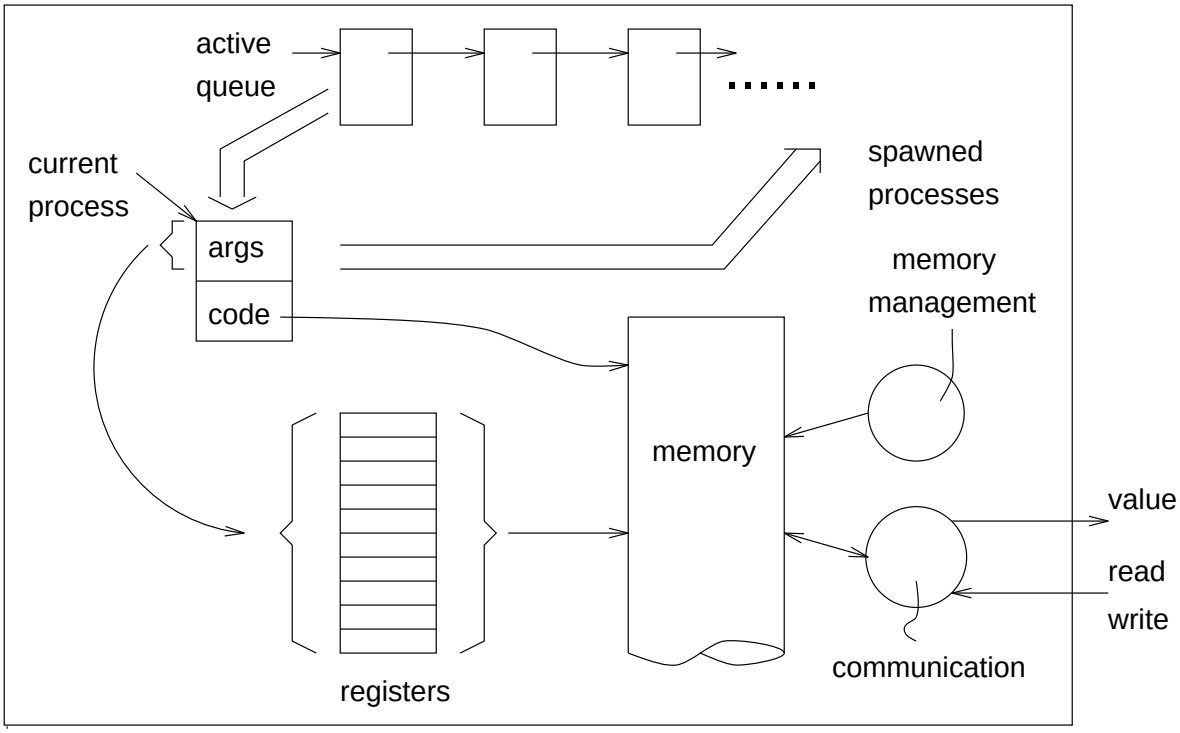

Figure 5: Single Computer Function

7 Run-Time Techniques

We conclude our discussion of the techniques used to map high-level concurrent programs onto parallel computers by describing the techniques used to execute the Core PCN code produced by the compiler.

Recall from Section 6.1 that Core PCN programs simply receive messages, modify state and spawn other processes. This basic model of computation is realized by a fine grain, concurrent, abstract machine. This machine comprises a number of computers connected via an interconnection network. Each computer is organized as shown in Figure 5 and is responsible for process scheduling, intercomputer communication, and memory management. The machine also incorporates facilities for performance evaluation [19, 32].

The abstract machine executes sequences of simple instructions that encode process control, guard evaluation, and data structure manipulation. In all, there are 33 instructions whose arguments are typically registers (R_i), program names (P), the number of arguments in a process (N), etc. Each instruction corresponds to a few physical machine instructions. Memory management and communication functions are used by the instructions but are not encoded directly.

7.1 Process Control

The abstract machine maintains an *active queue* containing runnable processes. Each process consists of a set of arguments and the location of the associated code. Conceptually, the basic execution algorithm is to repeatedly remove a *current process* from the active queue, load its arguments into machine registers, and execute the associated Core PCN procedure. For example, consider a process p(4,3,2,1) executing the following code:

```
p(a,b,c,d)
{ ? a > b  ->
    {| | q(a,b,d),
         r(b,c,d)
    }
}
```

When process p is scheduled, its arguments are loaded into machine registers R0 to R3. Since 4>3, the parallel composition is executed. One legitimate execution strategy is to spawn processes q and r, place them at the end of the active queue, terminate process p, and perform a context switch to execute another process from the queue. This strategy is simple but incurs considerable overhead. Hence, we use an alternative strategy: The current process proceeds directly to execute process q; only process r is spawned and placed into the active queue. This strategy is a form of *tail recursion optimization*, which can be applied as shown here even when recursion is not involved. It permits the efficiency of iteration to be achieved in many concurrent programs expressed in recursive form.

Notice that the arguments a and b for process q are already in the correct registers (R0,R1) for execution of process q. Hence, in order to execute process q, we use a single instruction to transfer the variable d to register R2. This optimization can be reapplied in the execution of process q. We limit the number of consecutive applications of the optimization, to guarantee that every process will eventually execute. After a fixed number of iterations, called a *timeslice*, a context switch is forced to occur. Table 1 summarizes the instructions for process scheduling and control.

Recall that PCN programs can call sequential procedures written in C, C++, or Fortran. The compilation transformations ensure that these calls occur as atomic actions as described in Section 6.2. The calls are encoded by using the call_foreign instruction. Arguments are always passed to such procedures using call by reference. This can be achieved efficiently because the PCN implementation records information about data types and data availability using tagged pointers. Hence, basic data types such as scalars and arrays can be represented in the same way as in sequential languages. Information can be passed in calls simply by stripping the tag from a pointer; this is achieved by the put_foreign instruction.

7.2 Guard Evaluation

Figure 6 outlines the structure of the compiled code for a Core PCN procedure (Section 6.1). All of the guards for a single procedure are encoded to form a discrimination

Table 1: Process Scheduling and Control

Instruction	Comment
fork P N	create an active process
recurse P N	execute a tail recursive call
halt	terminate the current process
default N	decide whether to suspend the current process
try L	if the following guard fails, go to L
copy R1 R2	copy from one register to an argument register
put_value R	place a value in a process argument
put_foreign R	prepare a foreign procedure argument
call_foreign N Address	invoke a foreign procedure
run R1 R2	invoke a procedure dynamically

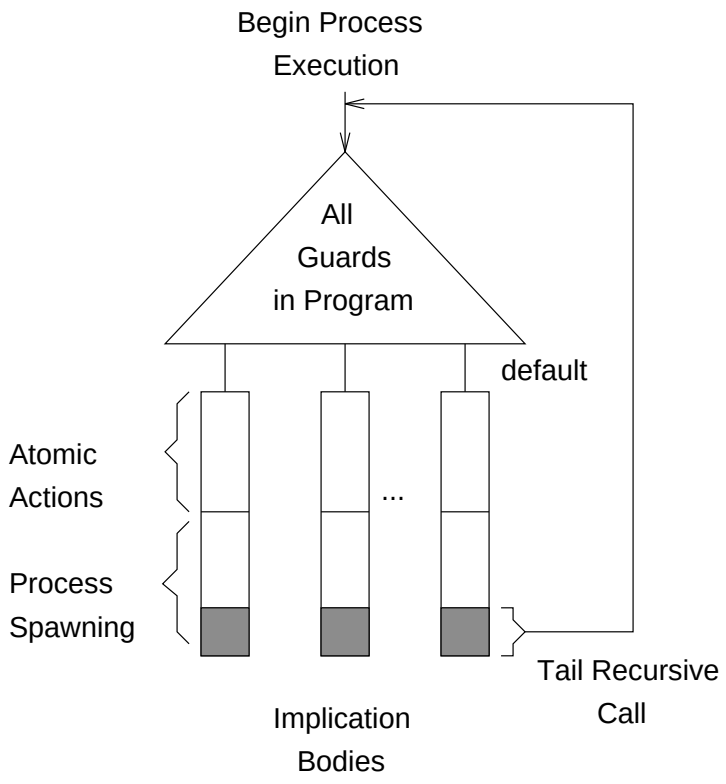

Figure 6: Compiled Program Form

Table 2: Guard Evaluation

Instruction	Comment
get_tuple R1 A R2	decompose an incoming message
equal R1 R2	compare for equality
not_equal R1 R2	compare for inequality
type R1 Tag	check the type of a value
le R1 R2	less than or equal
lt R1 R2	less than
data R	wait for data

network. This network simply decides which implication body to execute. There are three possible outcomes to guard evaluation. If any guard succeeds, then an associated implication body is executed. This involves immediate execution of the atomic actions, spawning of concurrent processes, and continued execution of the current process. If there are no procedure calls in the implication body, the current process terminates and a context switch occurs. If all guards fail, then the body associated with the default implication is executed. Finally, there may not be sufficient information available for any guard to succeed. In this case, the current process must be suspended. If suspension occurs, the procedure requires the value of one or more monotone variables. If only one variable is needed, then the process is attached to a queue of suspended processes associated with that variable. If multiple variables are required, then the process is placed in a global queue that is rescheduled periodically.

Table 2 summarizes the abstract machine instructions used to encode guard evaluation. These are the only abstract machine instructions that involve process synchronization.

7.3 Data Structure Manipulation.

The abstract machine provides a variety of instructions to manipulate arrays and monotone variables. Machine instructions are available to build these variables, transfer them between registers, perform arithmetic, deposit them in processes, etc. Table 3 summarizes these instructions.

7.4 Communication

Communication is necessary when processes located on different computers share a monotone variable. The algorithms used to implement communication follow from the representation chosen for monotone variables in a parallel computer network. Each variable is located at a *single* computer; all other instances of the variable are represented by intercomputer pointers termed *remote references* [39]. Intercomputer communication is necessary whenever a guard or assignment operation encounters a remote reference. This communication is achieved by using three message types: *read*, *write*, and *value*.

Table 3: Data Structure Manipulation and Arithmetic

Instruction	Comment
build_static R Type Size	build a statically sized array
build_dynamic Type R1 R2	build a dynamically sized array
build_monotone R	build a monotone variable
put_data R Type Size Value	place a literal in a register
define R1 R2	define monotone variable (send a message)
get_arg R1 R2 R3	extract an argument from a structure
get_element R1 R2 R3	get an element of an array
put_element R1 R2 R3	put an element into an array
copy_mut R1 R2	snapshot a variable for communication
coerce_mut R1 R2	change a data-type
length R1 R2	extract the length of a data structure
add R1 R2 R3	addition
sub R1 R2 R3	subtraction
mul R1 R2 R3	multiplication
div R1 R2 R3	division
mod R1 R2 R3	modulus

A *read* message is issued to request the value of a monotone variable located at a remote computer. It is generated when a guard test encounters a remote reference. Recall that the compilation transformations place all synchronization operations in guards. Hence, read messages may be issued only during guard evaluation. A computer receiving a *read* message responds with a *value* message when the value for the requested variable becomes available.

The *write* message is issued when an assignment operation is applied to a monotone variable represented by a remote reference. The message carries the value that is to be assigned. A computer receiving such a request completes the assignment at the specified location.

Messages are received and serviced by a computer whenever a context switch occurs. Hence, the use of a timeslice to force periodic context switches also has the effect of allowing overlapping of computation and communication.

7.5 Memory Management

Recall that PCN provides recursively defined data structures and dynamic memory allocation. Although it is possible to write programs that execute without consuming memory, a garbage collection algorithm is required in the general case. This algorithm reclaims memory occupied by data structures that are no longer accessible by any active process [13]. The current PCN implementation uses a simple asynchronous garbage collection technique for memory management. This technique allows computers to col-

lect independently by maintaining tables of remote references. These tables decouple the address spaces on different computers [22].

We are currently investigating programming and compiler techniques that will allow programs to be refined so as to avoid the need for garbage collection. This will allow the use of simpler memory management techniques.

7.6 Encoding the Octahedron Example

We conclude this description of the run-time techniques by encoding two fragments of the octahedral application. For clarity, these encodings do not take advantage of all opportunities for optimization. Program 7 encodes a fragment of the Core PCN compute procedure given in Program 6. This encoding demonstrates communication of an array on a stream, calling of sequential C code, and tail recursion optimization. In Program 8, we encode a fragment of the sphere procedure from Program 4. This encoding demonstrates the coupling of process spawning and tail recursion optimization.

8 Conclusion

We have described programming and compiler techniques that support the use of abstraction in concurrent program design. These techniques allow programmers to specify applications at a high level using reusable domain-specific abstractions. These abstractions can encapsulate design decisions concerned with decomposition, communication, mapping, load-balancing, scheduling, granularity control, and details of the physical machine.

These programming concepts are supported through compiler techniques that allow programs expressed in terms of abstractions to be compiled into efficient code for a variety of parallel architectures. Compilation proceeds in three primary stages. The first stage applies transformations to programs expressed in terms of a variety of abstractions. This stage yields programs in a simple compositional programming notation that implement abstractions through communication and synchronization. The second stage applies generic compilation transformations to generate programs in a machine-independent core notation. The third stage compiles this core notation to the instruction set of a concurrent, fine-grain, abstract machine. This machine can be implemented with run-time techniques based on the use of a portable emulator. Alternatively, the compilation pipeline can be extended to apply machine-specific transformations that generate native code for a particular architecture. These transformations can make use of specific machine features such as fine-grain process support or variable handling hardware.

The compiler is implemented as a small driver program that applies the abstraction, compilation, and machine-specific transformations. The transformations themselves are specified in a high-level program transformation notation. This notation is simply PCN augmented with operations for the manipulation of sets of programs. These operations provide building blocks that are used to construct libraries of reusable transformations.

All of the transformation, compilation and run-time system techniques described in

```
compute(step,mesh,ni,ei,si,wi,no,eo,so,wo,_DE)
double mesh[], edge[16];
{ ? step < 1000 ->
        { ; c_get_edge(0,edge,mesh), no=[edge | no1],
            ...,
            compute.1(step,mesh,ni,ei,si,wi,no1,eo1,so1,wo1,_DE)
        },
    default -> { ; c_dump_mesh(mesh), _DE = [] }
}

compute/11:  /* R0 = step, R1 = mesh, R2-9 = ni-wo, R10 = _DE */
    build_static double 16 11                /* R11 = edge */
    try L0
        put_data 12 1000                     /* R12 = integer(1000) */
        lt 0 12                              /* step < 1000 */
        put_data 12 0                        /* R12 = integer(0) */
        put_foreign 12                       /* 0 */
        put_foreign 11                       /* edge */
        put_foreign 1                        /* mesh */
        call_foreign c_get_edge 3            /* Call C procedure */
        build_static int 1 12                /* R12 = mutable integer */
        length 11 12                         /* R12 = length(edge) */
        build_dynamic double 12 13           /* R13 = mutable */
        copy_mut 11 13                       /* copy edge to message */
        build_monotone 14                    /* R14 = no1 */
        build_static tuple 2 15              /* R15 = [head | tail] */
            put_value 13                     /* head = message */
            put_value 14                     /* tail = no1 */
        define 6 15                          /* send message on "no" */
        ...
        copy 14 6                            /* no1 */
        copy 16 7                            /* eo1 */
        copy 17 8                            /* so1 */
        copy 18 9                            /* wo1 */
        recurse compute.1/11                 /* Branch to compute.1 */
L0: default 10                               /* Default implication */
    put_foreign 1                            /* mesh */
    call_foreign c_dump_mesh 1               /* Call C procedure */
    build_static tuple 0 11                  /* R11 = [] */
    define 10 11                             /* R10 = [] */
    halt                                     /* Terminate and context switch */
```

Program 7: Encoding the compute Procedure

```
sphere(c)
{|| rhombus(c,c,n0,e0,e3,n3),      /* Call 1 */
    ...
    rhombus(c,c,n2,e2,e1,n1)       /* Call 3 */
    rhombus(c,c,n3,e3,e2,n2)       /* Call 4 */
}

sphere/1:                          /* R0 = c */
    build_monotone 1               /* R1 = n0 */
    build_monotone 2               /* R2 = n3 */
    build_monotone 3               /* R3 = e3 */
    build_monotone 4               /* R4 = e2 */
    build_monotone 5               /* R5 = n2 */
    build_monotone 6               /* R6 = n1 */
    build_monotone 7               /* R7 = e1 */
    build_monotone 8               /* R8 = e0 */
    fork rhombus/6                 /* Call 1 */
        put_value 0                /* c */
        put_value 0                /* c */
        put_value 1                /* n0 */
        put_value 8                /* e0 */
        put_value 3                /* e3 */
        put_value 2                /* n3 */
    fork rhombus/6                 /* Call 2 */
    ...                            /* Arguments for Call 2 */
    fork rhombus/6                 /* Call 3 */
        put_value 0                /* c */
        put_value 0                /* c */
        put_value 5                /* n2 */
        put_value 4                /* e2 */
        put_value 7                /* e1 */
        put_value 6                /* n1 */
    copy 0 1                       /* c */
    recurse rhombus/6              /* Call 4 */
```

Program 8: Encoding the sphere Procedure

this paper have been implemented and are incorporated in a public-domain program development toolkit. The toolkit operates on a wide variety of networked workstations, multicomputers and shared-memory multiprocessors. It includes tools for defining program transformations, compiling concurrent programs, checking programs, debugging, performance analysis, and program animation. The toolkit has been used to design and implement substantial applications in several domains, including climate modeling and fluid dynamics [10, 27]. These programs use abstractions to coordinate the execution of thousands of lines of pre-existing C and Fortran code. Experimental studies show that the codes operate with predictable and impressive performance on a wide range of parallel computers.

The toolkit can be obtained by anonymous FTP. Both the toolkit and on-line documentation are located in directory pub/pcn at info.mcs.anl.gov and in directory pcn at sampson.caltech.edu.

9 Acknowledgments

We are grateful to Gil Weigand of DARPA, Gary Koob of ONR, and Nat Macon of NSF for their interest, encouragement and support. This work owes a great debt to other members of the research groups at both Argonne National Laboratory and the California Institute of Technology. In particular, Steve Tuecke is responsible for the development of the PCN emulator, Sharon Brunett and Dong Lin for the compilation transformations and compiler development, and Bob Olson for the network implementation and virtual machine implementation.

References

[1] Agha, G., *Actors*, MIT Press, 1986.

[2] Babb, R., Parallel processing with large grain data flow techniques, *IEEE Computer*, 17(7), 55–61, 1984.

[3] Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of processes, *CACM*, 21, 613–41, 1978.

[4] Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J., and Stevens, R., *Portable Programs for Parallel Processors*, Holt, Rinehart, and Winston, 1987.

[5] Boyle, J., and Muralidharan, M., Program reusability through program transformations, *IEEE Trans. Softw. Eng.*, SE-10(5), 574–588, 1984.

[6] Carriero, N., and Gelernter, D., *How to Write Parallel Programs*, MIT Press, 1990.

[7] Chandy, K. M., and Misra, J. *Parallel Program Design*, Addison-Wesley, 1988.

[8] Chandy, K. M., and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

[9] Chen, M., Choo, Y., and Li, J., Compiling parallel programs by optimizing performance, *J. Supercomputing*, 1(2), 171–207, 1988.

[10] Chern, I., and Foster, I., Design and parallel implementation of two methods for solving PDEs on the sphere, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991.

[11] Clark, K., and Gregory, S., A relational language for parallel programming, *Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architectures*, 1981, 171–178.

[12] Clocksin, W., and Mellish, C., *Programming in Prolog*, Springer-Verlag, 1981.

[13] Cohen, J., Garbage collection of linked data structures, *Computing Surveys*, 13(3), 341–367, 1981.

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

[15] Dally, W. J., et al., The J-Machine: A fine-grain concurrent computer, Information Processing 89, G. X. Ritter (ed.), Elsevier Science Publishers B.V., North Holland, IFIP, 1989.

[16] Dijkstra, E. W., Guarded commands, nondeterminacy and the formal derivation of programs, *CACM*, 18, 453–457, 1975.

[17] Dongarra, J. and Sorenson, D., Schedule: Tools for developing and analyzing parallel Fortran programs, *The Characteristics of Parallel Algorithms*, MIT Press, 1987.

[18] Foster, I., Automatic generation of self-scheduling programs, *IEEE Trans. Parallel and Distributed Systems*, 2(1), 68–78, 1991.

[19] Foster, I., Kesselman, C., and Taylor, S., Concurrency: Simple concepts and powerful tools, *Computer Journal*, 33(6), 501-507, 1990.

[20] Foster, I., and Stevens, R., Parallel programming with algorithmic motifs, *Proc. Intl Conf. on Parallel Processing*, Penn. State Univ. Press, 1989.

[21] Foster, I. and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1989.

[22] Foster, I. Tuecke, S., and Taylor, S., A portable run-time system for PCN, Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rept. ANL/MCS-TM-137, July 1991.

[23] Gajski, D., Padua, D., Kuck, D., and Kuhn, R. A second opinion on dataflow machines and languages, *IEEE Computer*, 15(2), 58–69, 1982.

[24] Gregory, S., *Parallel Logic Programming in PARLOG*, Addison-Wesley, 1987.

[25] Gregory, S., Foster, I., Burt, A., and Ringwood, G., An abstract machine for the implementation of PARLOG on uniprocessors, *New Generation Computing*, 6, 389–420, 1989.

[26] Halstead, R., Multilisp – A language for concurrent symbolic computation, *ACM Trans. Prog. Lang. Syst.*, 7(4), 501–538, 1985.

[27] Harrar, H., Keller, H., Lin, D., and Taylor, S., Parallel computation of Taylor-vortex flows, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991.

[28] Henderson, P., *Functional Programming*, Prentice-Hall, 1980.

[29] Hoare, C., Communicating sequential processes, *CACM*, 21(8), 666–677, 1978.

[30] Houri, A. and Shapiro, E., A sequential abstract machine for Flat Concurrent Prolog, Weizmann Institute Technical Report CS86-19, Rehovot, 1986.

[31] Kelly, P, *Functional Programming for Loosely-Coupled Multiprocessors*, MIT Press, 1989.

[32] Kesselman, C., *Integrating Performance Analysis with Performance Improvement in Parallel Programs*, Ph.D. thesis, UCLA, 1991.

[33] Martin, A., The torus: An exercise in constructing a processing surface, *Proc. Conf. on VLSI*, Caltech, 52–57, Jan. 1979.

[34] Parnas, D., On the criteria to be used in decomposing systems into modules, *CACM,* 15(12), 1053-1058, 1972.

[35] Safra, S., and Shapiro, S., Meta-interpreters for real, *Concurrent Prolog: Collected Papers*, MIT Press, 1987.

[36] Seitz, C. L., Multicomputers, *Developments in Concurrency and Communication*, C.A.R. Hoare (ed.), Addison-Wesley, 1991.

[37] Smith, R. D., Dukowicz, J. K., and Malone, R. C., Parallel ocean general circulation modeling, *Physica D* (to appear).

[38] Steele, G., Rabbit: A compiler for Scheme, MIT AI Lab TR/474, 1978.

[39] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[40] Warren, D.H.D., Applied logic — its use and implementation as a programming tool, SRI International Tech. Rep. 290, 1983.

[41] Wirth, N., Program development by stepwise refinement, *CACM,* 14, 221–227, 1971.

[42] Yang, J., and Choo, Y., Parallel program transformation using a metalanguage, *Proc. Conf. on Principles of Programming Languages*, 11–20, 1991.