Information Hiding in Parallel Programs

Ian Foster

CRPC-TR92276
May 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

PREPRINT MCS-P290-0292

Information Hiding in Parallel Programs

by

Ian Foster

May 1992

Mathematics and Computer Science Division

Argonne National Laboratory

Information Hiding in Parallel Programs

Ian Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

Abstract

A fundamental principle in program design is to isolate difficult or changeable design decisions. Application of this principle to parallel programs requires identification of decisions that are difficult or subject to change, and the development of techniques for hiding these decisions. We experiment with three complex applications, and identify mapping, communication, and scheduling as areas in which decisions are particularly problematic. We develop computational abstractions that hide such decisions, and show that these abstractions can be used to develop elegant solutions to programming problems. In particular, they allow us to encode common structures, such as transforms, reductions, and meshes, as *software cells* and *templates* that can reused in different applications. An important characteristic of these structures is that they do not incorporate mapping, communication, or scheduling decisions: these aspects of the design are specified separately, when composing existing structures to form applications. This separation of concerns allows the same cells and templates to be reused in different contexts.

Keywords: information hiding; parallel programming; program composition; reuse; software cell; template; virtual topology

1 Introduction

A fundamental principle in program design is to isolate difficult or changeable design decisions, so that interfaces between program components are simple and unlikely to change [24]. Application of this principle has been found to reduce design complexity, facilitate reuse of components, and decrease the cost of modifications. In sequential programming, good programmers routinely hide decisions concerned with data structures, storage management, and hardware-dependent features. Multicomputer programs, which coordinate the activities of hundreds or thousands of processors, necessarily involve additional design decisions. Hence we ask: Which of these decisions are particularly problematic? Are there program-structuring techniques that can isolate these decisions?

We have conducted a series of programming experiments over the past several years in an effort to answer these questions. These experiments have proceeded in conjunction with an interdisciplinary investigation of the methods and algorithms required to execute climate models on multicomputers. Numerical methods used in climate models are often complex; parallel algorithms for these methods have correspondingly rich structures, with unusual domain decompositions, irregular communication structures, and architecture-dependent mapping strategies. Hence, such algorithms provide a demanding testbed for design techniques.

In the first phase of these experiments, we designed and implemented numerous variants of three atmospheric modeling algorithms. This exercise revealed sources of complexity, impediments to reuse, and costly design changes. Analysis showed that most difficulties could be traced to design decisions concerned with the *mapping* to processors of the subtasks and subdomains produced by an initial functional and/or domain decomposition, the *communication* between these components, or the *scheduling* of tasks mapped to the same processor.

In the second phase, we explored techniques for isolating such decisions. This led us to develop an integrated set of four computational abstractions. Each of the first three is responsible for hiding a different class of design decision: *virtual topologies* for mapping, *virtual channels* for communication, and *lightweight processes* for scheduling. The fourth, the *port array*, is used when specifying communication decisions involving process ensembles. Together, these abstractions allow us to develop program components without committing to mapping, communication, or scheduling decisions, and to introduce these decisions in a stepwise fashion when composing components to form more complex programs. The components composed in this fashion can define the distributed computation structures that we call *software cells*, or may be *templates* that define a class of possible cells, with the code to be executed in the cell provided as a parameter. In this respect, the work complements and extends previous work [13], in which virtual channels and lightweight processes were used to decouple mapping, communication, and scheduling decisions, but cells and templates were not supported.

Run-time support required by the abstractions can be developed on an ad-hoc basis for each application, integrated into message-passing tools, or encapsulated in programming language constructs. We prefer the third approach, and have developed programming language support for the abstractions for both Fortran and the concurrent language PCN [6, 14]. Here, we work with PCN, which is supported by a public-domain compiler developed at Argonne National Laboratory and Caltech [15].[1]

The rest of the paper is as follows. In Section 2, we present the three atmospheric modeling algorithms used in programming experiments. In Sections 3 and 4, we analyze design problems in these algorithms and introduce the computational abstractions that we use to overcome these problems. In Section 5, we show how these abstractions are encapsulated in PCN. In Section 6–9, we present a set of cells and templates and use these to develop implementations for the algorithms. In Section 10, we compare our approach with other work in parallel program design. We conclude in Section 11.

We have chosen to focus on a small set of atmospheric modeling algorithms in this paper so as to provide the reader with a detailed understanding of our approach. However, the techniques have also been applied in areas as diverse as computational chemistry, computational biology, and optimization, each time with excellent results. Hence, we feel justified in arguing that the approach is of general utility, and in recommending its use to other developers of complex parallel programs.

[1] The software is accessible by anonymous FTP from `info.mcs.anl.gov`, directory pub/pcn.

2 Parallel Algorithms for Climate Modeling

Computer climate models have at their core a numerical method used to simulate atmospheric motion. This method must address the "pole problem": the singularities that arise at the poles in conventional coordinate systems. Accuracy and computational requirements are also important. Recently, with the advent of parallel supercomputers, suitability for parallel execution has become an important concern.

We present parallel algorithms for a spectral transform method on a latitude-longitude mesh, a control volume method on an icosahedral mesh, and a finite difference method on a composite mesh. Space does not permit more than a cursory description of the methods themselves; the interested reader is referred to the excellent comparative article by Browning, Hack, and Swarztrauber [5]. All three algorithms have been implemented and extensively evaluated on parallel computers, including the 528-processor, 20-Gflops Intel DELTA supercomputer [9, 12, 27].

2.1 Spectral Transform Algorithm

The spectral transform method is popular because of its spectral accuracy and its avoidance of the pole problem. An arbitrary scalar field $a(\lambda, \mu)$ on a latitude-longitude mesh (*physical space*) is approximated by a truncated series of its spectral coefficients a_n^m (*spectral space*) as follows:

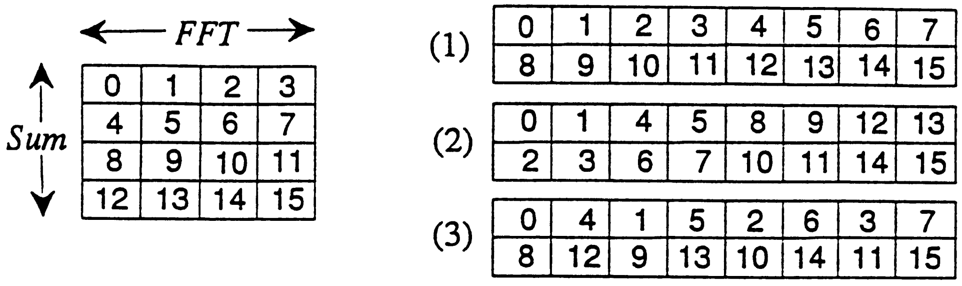
$$a(\lambda,\mu) = \sum_{m=-M}^{m=M} \sum_{n=|m|}^{M} a_n^m P_n^m(\mu) e^{im\lambda}, \qquad (1)$$

where P_n^m are the associated Legendre functions.

Computation is performed in both physical and spectral space. Data is transferred between the two spaces by forward and inverse spectral transforms. As Equation 1 suggests, the forward transform can be implemented by a fast Fourier transform (FFT) followed by a Legendre transform. The FFT operates on each latitude independently to produce a set of intermediate quantities. The Legendre transform then operates on each column of the intermediate array independently to produce the spectral coefficients. The inverse spectral transform operates in the reverse sequence.

We obtain a parallel algorithm by decomposing the latitude/longitude mesh in two dimensions to obtain equal-sized submeshes. Other data structures are also decomposed appropriately. Parallel FFTs must be performed between submeshes containing the same latitude, and parallel summations between submeshes containing the same longitude [12, 27]. In addition, parallel reductions over the entire mesh are required to produce diagnostic quantities.

The subdomains produced by decomposition can be mapped to the nodes of a parallel computer in several ways. Figure 1 shows three possible mappings of a latitude/longitude mesh decomposed into 4×4 submeshes to a 2×8 mesh computer. The first mapping results in FFTs being performed along each row of the parallel computer, and global summations along disjoint columns. The second mapping clusters nodes involved in the same FFT, while the third mapping clusters nodes involved in the same summation. The optimal mapping depends on machine architecture, problem size, and other factors [12].



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(1)

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

(2)

0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15

(3)

0	4	1	5	2	6	3	7
8	12	9	13	10	14	11	15

Figure 1: Alternative Mappings for the Spectral Transform

2.2 Control Volume/Icosahedral Mesh Algorithm

The icosahedral method is an explicit grid-point method and hence requires less communication than the spectral transform. In addition, the icosahedral grid is quasi-uniform on the sphere. This grid is constructed from a spherical icosahedron with 12 nodes and 20 equilateral triangular surfaces. Each spherical triangle is further partitioned into N^2 smaller triangles based on geodesic arguments. All points in this grid have six surrounding triangles, except for the 12 principal nodes of the icosahedron, which have only five. At each point, we connect by geodesic curves the centroids of the neighboring triangles to produce the hexagons or pentagons that serve as the control area elements for numerical integration [9].

For convenience in implementation, each triangle is joined with one of its neighbors to form a rhombus; each of these 10 rhombi then contains an $N \times N$ mesh. The two polar points are located in two separate "polar rhombi."

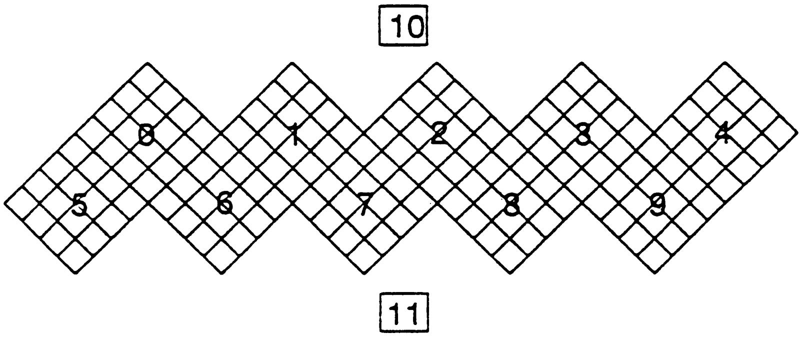


Figure 2: Icosahedral Mesh Domain Decomposition

A second-order, conservative control volume method is implemented on this grid. The use of hexagonal and pentagonal control elements means that computation at each point requires data from either five or six neighboring points. For simplicity in exposition, we shall assume a five-point stencil in subsequent discussion. However, the extension to the mixed six/seven-point stencil is straightforward [9].

We obtain a parallel algorithm by decomposing each nonpolar rhombus into a number (say C^2) of subrhombi. This gives a total of $10C^2 + 2$ subdomains: $10C^2$ meshes containing $(N/C)^2$ points and two individual points (the polar rhombi). This organization is illustrated in Figure 2, for C=4. Communication must be performed to obtain values from neighboring subdomains during integration. In addition, a global reduction is required to compute diagnostics. The design of an efficient mapping is complicated by the irregular domain. On some parallel computers, it may be desirable to place two or more subdomains on the same processor.

2.3 Finite Difference/Composite Mesh Algorithm

Our second explicit grid-point method avoids the pole problem by the use of two equal-sized, overlapping square meshes, centered at the north and south poles. The surface of the sphere is mapped onto these meshes as follows. The area on the sphere falling above a specified L degrees of southern latitude (i.e., the northern hemisphere and an overlap region) is mapped onto one mesh by using a stereographic projection; the southern hemisphere is projected onto the other mesh in a similar fashion.

During computation, points within the projected region on each mesh are updated by using a finite difference method. The finite difference computation within each mesh requires the values of points within that mesh but outside the projected region. These values are obtained by interpolation from the other mesh.

The parallel algorithm partitions each mesh into some number (e.g., $C \times C$) of equal-sized submeshes or *charts*, as illustrated in Figure 3. (The solid circles delimit the projected regions in each mesh, and the dashed circles the interpolation regions.) Note that the number of points falling within the projected region, and hence the amount of computation, varies from chart to chart. Communication requirements also vary significantly. Charts containing points within the projected or interpolation regions must exchange boundary values with those neighboring charts that also satisfy this criterion. Some of these charts must also exchange interpolation data with charts in the other mesh. Each chart hence communicates with between 0 and 4 neighbors within its mesh and with 0 or more charts in the other mesh. In addition, global reductions are required to compute diagnostics.

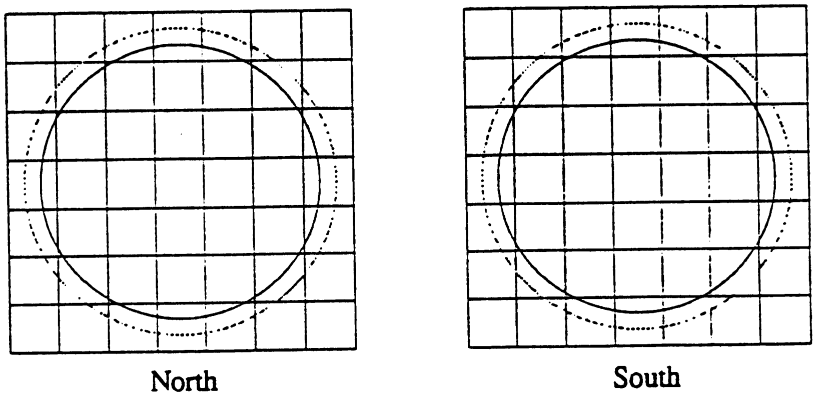


Figure 3: Composite Mesh Domain Decomposition

The mapping of subdomains to the nodes of a parallel computer is complicated by a need for load balancing. As the amount of computation and communication performed can vary significantly from chart to chart, we would like to map charts to processors in a way that minimizes load imbalance while maintaining locality of communication between neighbors. An optimal mapping is likely to require placement of several charts on a single processor.

3 What Should be Hidden?

The three parallel algorithms share common characteristics. All are based on straightforward (if irregular) domain decompositions. All have complex communication requirements. All require potentially complex and machine-dependent mappings which may place several subdomains on some processors. All can be decomposed into simpler building blocks such as mesh computations, FFTs, and reductions; however, each algorithm uses these building blocks in different contexts and for different purposes.

As implementors of these algorithms, we wish to experiment with algorithmic and architectural alternatives; hence, it should be easy to modify target architecture, mappings, communication structures, grid point stencils, etc. It should be possible to reuse common structures, so as to avoid duplication of effort involved in developing and optimizing code. Finally, we wish to reduce the effect of communication latency by overlapping computation and communication.

Analysis of these characteristics and requirements leads us to identify three classes of design decision that are either difficult or likely to change: mapping, communication, and scheduling. Mapping decisions are architecture-dependent and hence likely to change. Communication decisions are difficult to change if, as is the case in many first-generation message-passing systems, the location and identity of message receivers must be explicitly specified in sender code. Scheduling is problematic if an algorithm overlaps computation and communication: in most systems, this can be achieved only by intimate and complex intermingling of the code for the various tasks mapped to a single processor.

We use a single example to illustrate the value of isolating mapping, communication, and scheduling decisions. Recall that the spectral algorithm involves several parallel reductions: one per longitude, and one over the whole mesh. It should be possible to reuse the same code for each of these operations. However, as each reduction involves a different set of processors, a different set of communication partners, and different scheduling requirements, reuse is possible only if mapping, communication, and scheduling decisions can be isolated and specified separately from the reduction algorithm. In the next section, we shall see how this separation is achieved.

4 Information-Hiding Abstractions

We describe four abstractions that can be used to isolate mapping, communication, and scheduling decisions: virtual topologies, virtual channels, lightweight processes, and port arrays.

A *virtual topology* consists of a number of virtual processors, or *nodes*. Mapping inside a program component is specified with respect to such a topology; the mapping of the nodes of this topology to physical processors (or to the nodes of another virtual topology) is specified by a separate mapping function, introduced when the component is incorporated in a larger program. Hence, mapping decisions (number and placement of virtual processors) are isolated from algorithmic specifications. Hierarchies of such topologies can be defined, allowing the programmer to compose layouts when composing program components.

A *virtual channel* is a named, single-writer communication stream that supports read and write operations; tasks can communicate if and only if they share a virtual channel. A program component interacts with other components by reading and writing virtual channels passed as arguments; the identity of these components is specified when the component is incorporated in a larger program. Hence, communication decisions (identity and location of communication partners) are isolated from algorithmic specifications. The single-writer property of virtual channels means that a composition of two or more program components that communicate via virtual channels is guaranteed to be deterministic if the components are themselves deterministic. This property greatly simplifies program development.

A program component can be decomposed into one or more *lightweight processes.* A process is delayed if it is waiting for data on a virtual channel; otherwise, it is executable. Executable processes are selected for execution according to some fair scheduling algorithm. Hence, scheduling decisions are isolated in a scheduling algorithm: algorithmic specifications need indicate only how tasks are to be distributed to virtual processors and how tasks are to communicate.

The *port array* is a distributed array of virtual channels, with a specified number of elements per node of the virtual topology in which it is created. An important characteristic of a port array is that when passed as an argument to a program component executing in a different virtual topology, its indices are remapped so that the new program component is able to view it as a local port array declared in the new topology. This makes it possible to isolate communication decisions from algorithmic specifications, even when dealing with ensembles of processes. A cell executing in a virtual topology can select input and output channels from port arrays passed as arguments; connections between components are established by the code that sets up these port arrays. As the port array is distributed, these connections can be established in constant time.

4.1 Software Cells and Templates

An important aspect of these abstractions is that they permit us to define what we call cells and templates. A *software cell* is a reusable parallel program component that executes within a virtual topology. A cell definition specifies the processes that are to execute within the topology, the code to be executed by these processes, and the port arrays to be used for intercell communication. A cell definition does *not* specify mapping decisions (these are encapsulated in the mapping function used to define the virtual topology), intercell communication decisions (these are encapsulated in the code that sets up the port arrays passed as arguments), or scheduling decisions (these are encapsulated in the scheduling algorithm that supports lightweight processes). Hence, a cell definition can be reused without change in different contexts.

The code to be executed by a cell may be specified by parameters, in which case we refer to the cell definition as a *template*. Templates increase the potential for reuse by allowing the same parallel structure to be used for different purposes.

As our terminology suggests, there are similarities between these mechanisms and constructs used in VLSI design. A software cell is much like a VLSI cell, and the port arrays used to connect cells resemble the arrays of pins used for the same purpose in VLSI.

4.2 Virtual Topologies

We provide additional information on virtual topologies. A topology comprises one or more nodes and an associated type indicating how these nodes are organized. Simple topologies include the *point* (a single node), the one-dimensional *array*, the two-dimensional *mesh*, and the balanced binary *tree*. We shall also define more complex application-specific topologies.

A *mapping* function is used to embed a virtual topology in a physical or virtual topology. It specifies the type and size of the new topology and the embedding of its nodes in the existing topology. A mapping may perform one or more of the following types of transformation.

Reshaping: The new topology has a different type from its parent, or its nodes are ordered differently — for example, a mapping that embeds an array of size $m * n$ in a mesh of size $m \times n$.

Restriction: The new topology does not embed nodes in every node of its parent — for example, a mapping that embeds an array in a row of a mesh.

Expansion: The new topology embeds more than one node in a node of the parent topology — for example, a mapping that embeds two nodes in every parent node.

For example, the following pseudo-code specifies a restriction mapping `row(r)` that embeds an array in the `r`th row of a mesh. The function `topology()` returns the parent topology type. As the mesh has size `m`×`n`, the new topology has type *array* and size `m`; the function `oldnode` specifies that node `i` ($0 \leq$`i<m`) is located in node `i+r*m` of the mesh.

```
function row(r):
   if topology() != {"mesh",m,n} or r < 0 or r >= n then error
   else return(
               {"array",m},         /* Type of new topology */
               m,                   /* Size of new topology */
               oldnode(i) = i + r*m /* Embedding function   */
              )
```

A *location* function is used to specify relative or absolute positions within a topology. It maps a node number and a topology type to a node number. As examples, we specify location functions `node(i)`, which computes an absolute location `i` in any topology, and `north`, which computes a relative location in a mesh. The function `location()` returns the node number of the procedure that executes it.

```
function node(i):            /* Absolute location */
   if i < 0 or i >= nodes()-1 then error
   else
       return i              /* New location */

function north:              /* Relative location in mesh */
   if topology() != {"mesh",m,n} or location() < m then error
   else
       return location()-m   /* New location */
```

5 Using the Abstractions

Our presentation has so far focused on concepts. We now examine how virtual topologies, virtual channels, lightweight processes, and port arrays can be used to develop parallel programs.

Although some multicomputers and operating systems incorporate certain of these abstractions as primitive mechanisms [25, 11, 28], it will in general be necessary to provide compile-time or run-time support. There is much to be gained from standardizing this support so that it can be reused in many applications. It is also desirable to define interfaces that encourage or enforce correct usage.

One viable approach is to incorporate the necessary concepts in an extended message-passing library. This can be layered on top of an existing message-passing library, and may implement virtual topologies by accessing an indirection table when sending messages, lightweight processes by using operating system facilities, and virtual channels by message types.

We prefer the alternative approach of encapsulating the abstractions in language constructs. Responsibility for implementation then rests in a compiler. This reduces potential for programmer error, allows more succinct specifications, and facilitates compile-time verification and optimization.

The language constructs required to support the abstractions are not complex and can in principle be defined for any language. For example, we have defined appropriate extensions for both Fortran and the high-level concurrent language Program Composition Notation (PCN) [6, 14]. We choose to work with PCN here, as this already provides elegant representations of virtual channels and lightweight processes and can be extended straightforwardly to support virtual topologies and port arrays. We describe those aspects of the extended PCN language that are relevant to the present discussion.

5.1 Extended PCN

The syntax of PCN is similar to that of the C programming language.

Lightweight Processes. A PCN solution to a programming problem is a set of procedures, each with the following general form ($k, l \geq 0$).

```
name(arg$_1$,...,arg$_k$)
declaration$_1$, ..., declaration$_l$;
block
```

A block is a call to a PCN procedure (or to a procedure in a sequential language such as Fortran or C), a composition, or a primitive operation such as assignment. A composition is written { op block$_1$, ..., block$_m$}, $n > 0$, where op is one of "||", ";", or "?", indicating that the blocks block$_1$, ..., block$_m$ are to be executed concurrently, in sequence, or as a set of guarded commands, respectively. Blocks in a parallel composition execute as lightweight processes.

Virtual Channels. Interprocess communication is expressed in terms of read and write operations on specialized *definitional variables.* These variables are distinguished by a lack of declaration, are initially undefined, can be written (defined) once using the primitive operator "=", and once written cannot be modified. A process that requires the value of an undefined variable blocks until the required data is available.

A shared definitional variable can be used to exchange a sequence (stream) of values between a producer and a consumer. The producer sends a message by defining the shared variable to be a structured term (a *tuple*) containing the message plus a new variable. This process can then be repeated with the new variable. For example, the producer could use the following sequence of operations to communicate the messages "hello" and "goodbye" to any process with a reference to x. (The notation {e$_1$,...,e$_n$} denotes a tuple with elements e$_1$,...,e$_n$.)

```
x = {"hello",t},    t = {"goodbye",t2},    t2 = []
```

A shared definitional variable provides an elegant implementation of a virtual channel, with read and write operations on the variable corresponding to send and receive operations. A powerful feature of this data type is that it can be included in messages. This allows channels to be established dynamically, as in the stream communication protocol outlined in the preceding paragraph.

Location and Mapping. Programs invoke mapping functions to execute subcomputations within different virtual topologies. An annotation "in M" on a block denotes invocation of mapping function M; it causes the block to execute within the virtual topology returned by M.

Programs invoke location functions to place subcomputations on specific virtual processors. An annotation "@ L" on a block location denotes invocation of location function L; it causes the block to execute on the virtual processor with index returned by L.

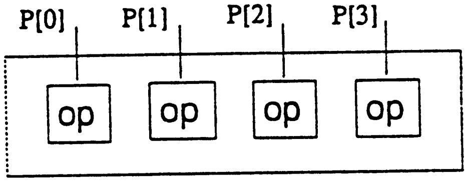
Port Arrays. A port declaration creates a one-dimensional distributed array of definitional variables. A declaration "port P[N];" creates a port array P with N elements, distributed blockwise across the nodes of the virtual topology in which the port array is declared. For example, a declaration "port p[2*nodes()];" creates a port array p with

2*nodes() elements; p[2*i] and p[2*i+1] are located on the ith node of the current topology (0≤i<nodes(), where the function nodes() returns the number of nodes in that topology).

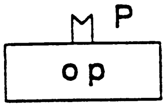
Cells and Templates. Cell and template definitions use location functions to place procedure calls within the current topology and may pass port array elements to individual procedure calls. For example, the following template executes the procedure named by parameter op on every node of the current topology, passing each call the local element of port array P. The quantification i over 0..nodes()-1 causes i to range over the nodes, and the location function node(i) locates the ith call to op on the i node. The variable op is quoted in the parallel composition to indicate that it is being used as a variable, not a string. For clarity, we capitalize variable names denoting port arrays in this and subsequent programs.

```
replicate(op,P)
port P[];
{|| i over 0..nodes()-1 : 'op'(P[i]) @ node(i) }
```

Execution of this procedure in a four-node array topology creates the following set of processes, with the lines representing the port elements passed as arguments: the cell's interface to the outside world. These port elements can be used to establish connections to other cells.



For brevity, we shall sometimes use the following more compact representation of a cell. This represents the same cell as the preceding figure, and indicates that the port P is to be used for input.

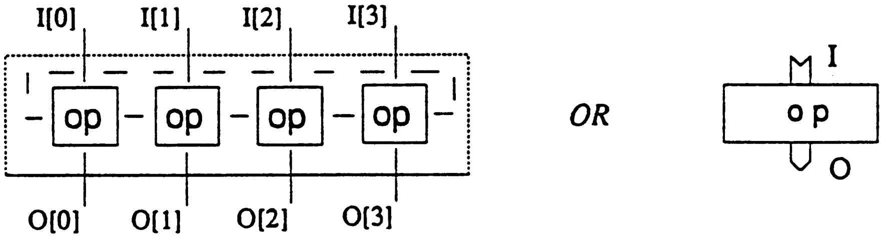


5.2 Ring Pipeline Example

We use an example to illustrate how PCN programs are developed by composing simpler cells and templates. The following template creates a single process in each node of the current topology and uses the local port array S to establish internal communication streams between neighboring processes, so that each process has two streams, one shared with each neighbor. The code to be executed at each node is provided as a parameter. In addition, the ith node of this ring pipeline structure is given elements I[i] and O[i] of the two port arrays I and O passed as parameters, so as to allow communication with the outside world. The "%" represents the modulus operator. As in C, the dimension of an array passed as an argument is not specified.

```
ring_pipe(op,I,O)
port S[nodes()], I[], O[];
{|| i over 0..nodes()-1 :
        'op'(I[i],O[i],S[(i+1)%nodes()],S[i]) @ node(i)
}
```

The process structure created by this procedure can be drawn as follows, with the solid lines indicating the port connections to the outside world and the dotted lines representing internal streams.



The following procedures implement simple input and output cells. The procedure load reads values from a file and sends them to successive elements of the port array P; the procedure store writes to a file values received on successive elements of port array Q. Both use the sequential composition operator to sequence I/O operations.

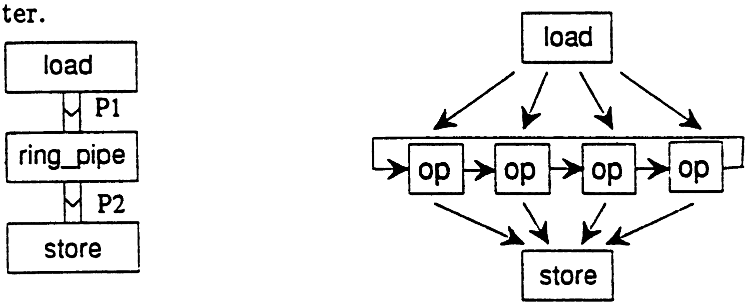
```
load(file,P)
port P[];
{ ; i over 0..nodes()-1 : read(file,stuff), P[i] = stuff }

store(file,Q)
port Q[];
{ ; i over 0..nodes()-1 : write(file,Q[i]) }
```

We compose the three cells to obtain a program main that reads data from infile, executes a user-supplied function in the ring pipeline (e.g., a naive N-body algorithm), and finally writes results to outfile.

```
main(infile,outfile)
port P1[nodes()], P2[nodes()];
{|| load(infile,P1),
    ring_pipe(nbody(),P1,P2),
    store(outfile,P2)
}
```

Data flows from load to ring_pipe via port array P1 and from ring_pipe to store via port array P2. This is illustrated in the following figure, which shows the cell structure represented by program main, and the process structure that would be created on a four-processor computer.



This program can be developed and tested on a uniprocessor. We can then experiment with alternative mappings. Let us assume that the program main is invoked in a ring topology (i.e., main(infile,outfile) in ring). Mapping decisions are then encapsulated entirely within the mapping function ring. Possible mappings include the following. (1) One ring node is placed on each processor. This is simple, but may prevent placement of ring neighbors on adjacent processors. (2) Two ring nodes are placed on each processor. This ensures that ring neighbors can always be located on adjacent processors. (3) Many ring nodes are placed on each processor. This can be useful if there is significant variation in the amount of work performed at different pipeline nodes, or if it desirable to overlap computation and communication.

This example, although simple, illustrates most aspects of our approach. (One important exception is restriction mappings, which will be demonstrated in subsequent examples.) Note how easy it is to change mapping decisions. No change to the application code is required: only a mapping function need be modified, even if a mapping places more than one subdomain (process) on a physical processor. Note also how logically distinct program components are separated and packaged as reusable cells and templates. This separation is possible because the cell definitions specify only local decisions; mapping decisions are encapsulated in the mapping function that implements the ring topology, communication decisions in the code that sets up the port arrays P1 and P2, and scheduling decisions in a scheduling algorithm.

6 Building Blocks

The climate modeling algorithms of Section 2, like the ring pipeline, can be constructed from simpler building blocks. Here, we present six such cells and templates: fft, reduce, mesh, mesh_io, mesh_io2, and router. Some of these are illustrated in Figure 4. All six interact with other components by means of port arrays. The first five operate loosely synchronously: that is, each computation phase consumes one data item from each element of an input port and produces one data item on each element of an output port. The sixth operates asynchronously: data items can be received and processed independently. In the Appendix, we show that the fft and reduce cells can be expressed as instances of a common template, butterfly. We also present an implementation of mesh_io2.

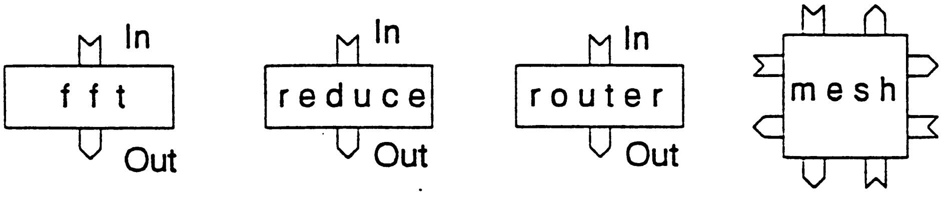


Figure 4: Building Blocks

Cell fft(in_port,out_port): The fft procedure defines a cell that computes the fast Fourier transform of distributed data. The data to be transformed is input on the elements of in_port. The fft nodes compute the transform and output the transformed data on the elements of out_port.

Template reduce(op,in_port,out_port): The reduce template is used to define cells that reduce distributed data by using a specified binary operator (e.g., maximum, addition) and distribute the reduced value to the nodes participating in the reduction. The data to be reduced is input on in_port. The reduce nodes reduce the data by using op, and output a copy of the reduced value on each element of out_port.

Template mesh(op,nsi,nso,wei,weo): A variety of mesh templates can be defined to implement different communication patterns, boundary conditions, etc. The mesh template invokes op in each node of a 2-dimensional mesh and establishes communication streams between each node and its north, east, south, and west neighbors. Communication streams to nodes on the edges of the mesh are taken from ports nsi, nso, wei, and weo. The ports nsi and nso provide input and output on the north and south edges of the mesh, and ports wei and weo provide input and output on the west and east edges.

Template mesh_io(op,i,o,nsi,nso,wei,weo): The related template mesh_io in addition associates an element of an input and output port with each node in the mesh.

Template mesh_io2(op,i1,o1,o1,o2): The template mesh_io2 provides two input and output ports and no edge connections.

Cell router(in_port,out_port): A router cell provides general routing capabilities. A message with the general form {i,msg} appended to any element of in_port causes the specified msg to be appended to the ith element of out_port.

The router cell abstracts traditional message-passing facilities, with three extensions. First, routing is performed within a user-defined virtual topology rather than a physical computer. Second, messages transmitted via the router cannot interfere with other communications. Third, the router provides a termination mechanism: it shuts down, defining all elements of out_port to be the special element nil (□), if all elements of in_port are defined to be nil.

7 Spectral Transform Implementation

We now develop implementations for the algorithms presented in Section 2. Similar strategies are employed in each case. We first define a virtual topology with the same structure as the domain decomposition. Then, we develop the implementation by composing various cells (e.g., mesh, FFT, reduction) in the framework of this topology. Finally, we specify the mapping of the complete program, using the mapping function that defines the topology.

We first examine the spectral transform algorithm (Section 2.1). Recall that this decomposes the latitude/longitude mesh into lat×long subdomains.

7.1 Virtual Topologies

We define a *spectral mesh* topology with the same structure as the domain decomposition: i.e., lat×long nodes, organized as a mesh. The mapping function spectral_mesh for this topology will be discussed in Section 7.3. We shall also require mapping functions row(r) and col(c), which define subtopologies of type *array* comprising the rth row and cth column of the spectral mesh, respectively. The mapping function row(r) has been defined in Section 4.2; col(c) is similar.

7.2 Code Sketch

Recall that our parallel algorithm involves lat parallel FFTs (each operating on data from long subdomains), long parallel summations (each operating on data from lat subdomains), and one parallel summation that operates on all subdomains (Section 2.1). As we illustrate in Figure 5, this structure can be constructed by composing two existing components (reduce and fft cells: Section 6) with the code to be executed within a single node (submesh: shown as n).

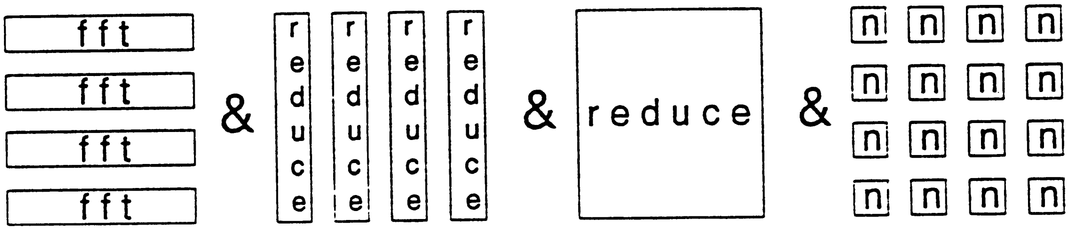


Figure 5: Code Structure for Spectral Transform

This composition is specified in Figure 6. The top level composes a summation cell with the procedure spectral, which executes within the *spectral mesh* topology. The procedure spectral composes lat FFT cells, one per row of the spectral mesh; long summation cells, one per column; and lat×long submesh processes, one per node. The summation cells are instances of the reduce template.

Six port arrays are used to connect co-located processes from the various cells. The ports i and o connect the global reduce cell to the rest of the program. The ports fti, fto, lri, and lro connect the fft and local reduce cells to the submesh processes. In total, each submesh process is passed six communication streams as arguments. These are three input/output pairs, to the global summation cell, a FFT cell, and a local summation cell, respectively.

```
sphere(lat,long)
port I[nodes()], O[nodes()];
{|| spectral(I,O,lat,long) in spectral_mesh(lat,long),
    reduce(sum(),I,O)
}

spectral(I,O,lat,long)
port I□, O□;
port Fti[nodes()], Fto[nodes()], Lri[nodes()], Lro[nodes()];
{|| {|| i over 0..lat-1     : fft(Fti,Fto) in row(i) },
    {|| j over 0..long-1    : reduce(sum(),Lri,Lro) in col(j) },
    {|| k over 0..nodes()-1 :
            submesh(k,I[k],O[k],Fti[k],Fto[k],Lri[k],Lro[k]) @ node(k)
    }
}
```

Figure 6: Code Sketch for Spectral Transform

Several aspects of Figure 6 bear careful study. First, consider the hierarchy of virtual topologies used in this program. At the top level, we have some unspecified topology. The mapping function spectral_mesh *reshapes* this topology to create a structure with the same shape as the spectral transform's domain decomposition: this allows the parallel algorithm to be developed independently of mapping issues. The mapping functions row and col *restrict* the spectral mesh topology to create array subtopologies: this serves both to locate the fft and reduce cells correctly within the spectral mesh and to allow reuse of these cell definitions, which create an appropriate process structure within the virtual topology in which they are invoked. Finally, the location function node replicates the submesh procedure throughout the spectral mesh topology.

Second, consider the techniques used to compose the cells spectral and reduce. The ports I and O connect co-located process pairs from the two structures. The spectral cell will generate periodically a data item on each element of I. The reduce cell receives these values, performs internal communication to compute their sum, and outputs the sum on each element of O. The result computed is independent (modulo rounding differences) of the mapping employed in the spectral structure, as specified by the mapping function spectral_mesh, as long as this mapping is one-to-one from nodes of the spectral mesh to nodes of its parent topology, that is, as long as spectral_mesh only reshapes and does not expand or restrict. The techniques used to compose cells when a mapping is not one-to-one are discussed in Section 9.3.

It is apparent that a substantial part of the program is formed from pre-existing building blocks; the application-specific code is primarily concerned with putting these blocks together. Our abstractions, by separating mapping, communication, and scheduling decisions from algorithm specifications, allow cells (in this case, fft and reduce) to be used in different contexts without modification. Mapping decisions are encapsulated in the row and column operators that define the subtopologies in which the fft and reduce cells are executed. Communication decisions are encapsulated in the declarations of the port arrays. Multiple processes (FFT, reduction, submesh) mapped to the same processor are scheduled according to the availability of data.

We complete this code sketch by outlining in Figure 7 the code executed within a single submesh. This represents the application-specific computational code that must be provided to complete the program. In essence, each submesh alternates between performing computation with local data and communicating with cells that perform FFT, local summation, and global summation operations. The submesh procedure initiates an FFT or summation by sending a message on the appropriate communication stream. The processes constituting this structure perform the transform or summation and eventually return a result. This communication is encapsulated in the procedure exchange, which sends a message in (to = [in|to1]) and concurrently awaits a reply out (fr ?= [o|f] -> ...).

7.3 Mapping

We explore alternative mappings by changing the mapping function spectral_mesh. (The global summation structure is mapped separately.) For example, the following procedure implements mapping (1) in Figure 1. It uses the identity function (oldnode(i) = i) to embed each node of the new topology in the corresponding node of the old topology.

```
function spectral_mesh(lat,long):
   if topology() != {"mesh",m,n} or m*n != lat*long then error
   else return(
               {"mesh",lat,long},
               lat*long
               oldnode(i) = i
              )
```

The following procedure implements mapping (2) in Figure 1. The mapping function is more complex, but remains manageable.

```
function spectral_mesh(lat,long):
   if topology() != {"mesh",m,n} or m*n != lat*long then error
   else return(
               {"mesh",lat,long},
               lat*long,
submesh(id,to_gs,fr_gs,to_fft,fr_fft,to_sum,fr_sum)
{ ; init(id,state),
    compute(state,to_gs,fr_gs,to_fft,fr_fft,to_sum,fr_sum)
}

compute(state,to_g,fr_g,to_f,fr_f,to_s,fr_s)
{ ; compute1(state,fftdata),
    exchange(fftdata,result1,to_f,fr_f,to_f1,fr_f1),
    compute2(result1,state,sumdata),
    exchange(sumdata,result2,to_s,fr_s,to_s1,fr_s1),
    compute2(result2,state,diagnostics),
    exchange(diagnostics,globsum,to_g,fr_g,to_g1,fr_g1),
    compute(state,globsum,to_g1,fr_g1,to_f1,fr_f1,to_s1,fr_s1)
}

exchange(in,out,to,fr,to1,fr1)
{|| to = [in|to1],
    fr ?= [o|f] -> {|| out=o, fr1=f}
}
```

Figure 7: Spectral Transform — Submesh Code

```
oldnode(i) = mm + nn*m
    where ls = sqrt(long)
          lt = i/long, ln = i%long
          mm = (lt%(m/ls))*ls + ln%ls
          nn = (lt/(m/ls))*ls + ln/ls
)
```

8 Control Volume/Icosahedral Mesh Implementation

Recall that the problem domain in the second algorithm consists of two polar points and ten equal-sized rhombi (meshes) and that each mesh is partitioned into C^2 submeshes (Section 2.2).

8.1 Virtual Topologies and Mapping

As in the spectral transform, we define a specialized virtual topology with the same structure as the domain decomposition (Figure 2). The *icosahedral mesh* topology comprises ten $C \times C$ mesh structures and two polar nodes. We assume a mapping function `icosahedral_mesh` that defines a mapping for this topology. We shall also require mapping functions `rhombus(i)` and `pole(j)`, defined below, which create subtopologies (of type *mesh* and *point*, respectively) comprising the nodes in the ith mesh or jth pole.

```
function rhombus(i):
    if topology() != {"icosahedral_mesh",C} or
          i < 0 or i > 9 then error
    else return(
                {"mesh",C,C},
                C*C,
                oldnode(i) = r*C*C + i
               )

function pole(i):
    if topology() != {"icosahedral_mesh",C} or
          i < 0 or i > 1 then error
    else return(
                {"point",1},
                1,
                oldnode(i) = 10*C*C + i
               )
```

The mapping function icosahedral_mesh is not presented here. We have observed that on modern multicomputers such as the 528-node Intel Touchstone DELTA, performance is not particularly sensitive to the mapping employed [9], as high-performance communication reduces the importance of locality. On machines where communication locality is important, we can fold the virtual topology (locating two or more nodes per processor) so as to reduce message latency.

8.2 Code Sketch

The parallel algorithm involves nearest-neighbor communication within each rhombus and more complex communication between rhombi and the poles. In addition, global reductions are used to compute diagnostic information (Section 2.2).

The code sketch in Figure 8 shows that this structure can be constructed by composing two pre-existing components (mesh_io and reduce: Section 6) and an application-specific procedure (interconnect). Ten calls to the mesh_io template establish intrarhombus communication channels and create the processes that handle computation in submeshes. One call to reduce creates the cell used to perform global reductions. The interconnect procedure establishes interrhombus communication channels.

Note that the parallel program is specified as a template, with the code to be executed at polar and nonpolar nodes provided as arguments. The mapping functions "in rhombus(_)" and "in pole(_)" locate the rhombus cells and pole processes within the icosahedral mesh topology.

The interconnect procedure establishes interrhombus communication channels by connecting elements of ports nsi, nso, wei, and weo. For example, the following fragment establishes communications between the southern edges of rhombi 0–4 and the northern edges of rhombi 5–9. The rest of the procedure is similar.

```
{|| i over 0..4, j over 0..c-1 :
```

```
sphere(meshop,poleop)
port I[nodes()], O[nodes()];
{|| ico(meshop,poleop,I,O) in icosahedral_mesh,
    reduce(maximum(),I,O)
}

ico(meshop,poleop,I,O)
port I[],O[];
port Nsi[nodes()],Nso[nodes()],Wei[nodes()],Weo[nodes()];
{|| {|| i over 0..9 :
            mesh_io(meshop,I,O,Nsi,Nso,Wei,Weo) in rhombus(i)
    },
    'poleop'(0,I,O,Nsi,Nso,Wei,Weo) in pole(0),
    'poleop'(1,I,O,Nsi,Nso,Wei,Weo) in pole(1),
    interconnect(Nsi,Nso,Wei,Weo)
}
```

Figure 8: Code Sketch for Icosahedral Mesh

```
        {|| Nsi[i*c*c+j] = Nso[(i+5)*c*c+(c-1)*c+j])
            Nso[i*c*c+j] = Nsi[(i+5)*c*c+(c-1)*c+j])
        } @ node(i*c*c+j)
}
```

As in the spectral code, a substantial part of the program is formed from existing building blocks. It is also instructive to examine what is involved in making various modifications. The reduction algorithm can be changed simply by substituting an alternative template. A change to the numerical method's stencil involves substitution of an alternative mesh template and some changes to `interconnect`. As always, alternative mappings are specified simply by changing a mapping function. In each case, modifications are restricted to a small piece of code.

9 Finite Difference/Composite Mesh Implementation

Recall that in the third algorithm the problem domain consists of two equal-sized meshes and that each mesh is partitioned into C^2 equal-sized charts (Section 2.3).

9.1 Virtual Topologies and Mapping

As in the first two algorithms, we define a specialized virtual topology with the same structure as the domain decomposition (Figure 3). The *composite mesh* topology comprises two meshes, each partitioned into $C \times C$ charts. We assume a mapping function `composite_mesh` that defines the mapping of this topology to a physical computer. We shall also require mapping functions northern and southern, which create subtopologies of type *mesh*, comprising just the northern and southern component of the composite mesh, respectively.

Mapping is complicated by the fact that different charts perform varying amounts of computation and communication. An optimal mapping should probably locate a varying number of charts on different processors so as to minimize load imbalances. The use of the composite mesh topology makes the implementation of this mapping strategy straightforward. The composite mesh program locates one chart in each node of the composite mesh topology. The composite_mesh mapping function maps a variable number of nodes to each physical processor.

9.2 Code Sketch

The parallel algorithm involves nearest-neighbor communication between charts within each mesh and more complex intermesh communication for the purposes of interpolation. In addition, global reductions are used to compute diagnostic information (Section 2.3).

The code sketch in Figure 9 shows that this structure can be constructed by composing three pre-existing components: mesh_io2, reduce, and router. The procedure solve composes the procedure composite (in the composite mesh topology) and a reduce template, used to implement the global summation. The procedure composite composes two invocations of the mesh_io2 template and a router cell. Recall that the mesh_io2 template takes two pairs of port arrays as arguments; in this case, these are used for communication with the reduce and router cells.

The calls to mesh_io2 create sets of C^2 chart processes in the northern and southern meshes and establishes the communication streams needed for communication between neighboring charts in each mesh. Each chart process (except those on a mesh edge) has connections to north, south, east, and west neighbors. Each process must determine whether to engage in computation and which connections to use for communication.

The interpolation communication structure is complex and not easily defined in terms of indices into port arrays. Fortunately, it is straightforward to construct the communication structure dynamically by using the router. First, each chart process determines the data that it requires from other charts. Then, each chart uses the router to send a message to each chart from which it requires data. This message contains a description of the required data and a new definitional variable to be used for subsequent exchanges. For example, if a chart A requires data D from a chart B, it uses the router to send a message $\{D, S\}$ to B, where S is a definitional variable. Upon receipt of this message, B records the data to be sent to A (D) and the stream on which it must be sent (S). As A also records S, this exchange dynamically establishes a channel between A and B. Note that this exchange takes advantage of the virtual channel's status as a first-class data structure, which allows it to be included in messages.

Finally, each chart process possesses streams to and from neighbors in the grid, streams on which it is to send interpolation data, and streams on which it is to receive interpolation data. All necessary communication channels have been established, and each chart process can proceed to execute a compute–communicate cycle similar to that defined for the spectral transform submesh process (Figure 7).

```
main()
{|| solve() in composite_mesh }

solve()
port I[nodes()], O[nodes()];
{|| composite(I,O),
    reduce(maximum(),I,O)
}

composite(I,O)
port I[], O[], Ri[nodes()], Ro[nodes()];
{|| mesh_io2(chart(0),I,O,Ri,Ro) in northern,
    mesh_io2(chart(1),I,O,Ri,Ro) in southern,
    router(Ri,Ro)
}
```

Figure 9: Code Sketch for Composite Mesh

9.3 Many-to-One Communication

The program in Figure 9 creates two cells (reduce and composite) within a composite mesh topology. A deficiency of this formulation is that the composite_mesh mapping is unlikely to be optimal for the reduce cell, as we have assumed in Section 9.1 that it would be optimized for the composite cell. We would prefer to create the reduce network directly on the underlying computer; in this way, any clever embeddings developed for the reduce cell can be exploited. Hence, we rewrite the top level of the composite mesh code as follows.

```
solve()
port I[nodes()], O[nodes()];
{|| composite(I,O) in composite_mesh,
    reduce(maximum(),I,O)
}
```

Recall that this technique is applied in the spectral transform and icosahedral codes (Figures 6 and 8). We did not apply it in Figure 9 because the mapping function composite_mesh may *expand* as well as reshape the topology in which it is applied. That is, it may place several nodes of the new topology on each node of the original topology. This creates difficulties when we compose a cell defined on the new topology (e.g., composite) with a cell defined on the parent topology (e.g., reduce): there is no longer a one-to-one relationship between nodes in the two cells. (A similar situation arises if the mapping function used to produce the offspring topology performs a restriction operation.)

This turns out to be a common-situation; in fact, although we have ignored the possibility, it can also arise in the spectral transform and icosahedral algorithms if the mapping functions spectral_mesh or icosahedral_mesh expand and/or restrict as well as reshape. Hence, we provide the following mechanism for dealing with many-to-one communication. Consider two ports, the first of dimension 1 in the parent topology and the second of arbitrary dimension in the expanded and/or restricted offspring topology. The primitive operation connect, when invoked with these two ports as arguments, defines each element of the first port to be a tuple containing those elements of the second port that are mapped to the corresponding parent topology node.

More precisely: Let P be a port of dimension 1 defined in a topology T, and Q be a port of dimension q defined in S, the subtopology of T obtained by the mapping function F. The primitive operation connect(P,Q) defines each element $P[i]$ of P, $0 \leq i <$sizeof(T), to be a tuple containing those elements $Q[j * q + l]$, $0 \leq j <$sizeof(S), $0 \leq k < q$, such that the mapping function F locates the jth node of S in the ith node of T.

We use this mechanism in the procedure composite. Two additional ports are defined, Li and Lo, and two connect calls are used to define each element of I and O to be a tuple of port elements. The reduce cell most be modified to deal with tuples of streams as input and output.

```
composite(I,O)
port I[], O[];
port Ri[nodes()], Ro[nodes()], Li[nodes()], Lo[nodes()];
{|| mesh_io2(chart(NORTH),Li,Lo,Ri,Ro) in northern,
    mesh_io2(chart(SOUTH),Li,Lo,Ri,Ro) in southern,
    router(Ri,Ro),
    connect(I,Li),
    connect(O,Lo)
}
```

10 Related Work

Program structuring and reuse are important themes in parallel computing research. These themes are particularly visible in CSP [19], functional programming [21, 18], concurrent logic programming [10, 16], object-oriented programming [1], and Unity [7]. Most of these systems are based on lightweight process and message-passing ideas similar to those explored in this paper. However, for a variety of reasons, they do not support the same forms of composition and reuse. CSP's processes and channels cannot be created dynamically or communicated in messages. Some of these restrictions can be removed [3], but the overall model is static rather than dynamic. We have not emphasized the dynamic aspect of our approach in this paper, but in practice we frequently generate or change the configuration of software cells during program execution. In concurrent functional and logic programming, the concept of a distributed array of channels is absent, making it difficult to specify the composition of process ensembles. In concurrent object-oriented programming, the concept of a channel is absent, making it difficult to specify deterministic computations.

There are clearly similarities between our ideas and fundamental concepts of VLSI design [23]. In particular, the notions of cell and port are used in an analogous manner, to control complexity in large circuits. However, it is important not to be misled by the obvious analogies. Software is more flexible than hardware and admits more general and elegant solutions to many problems. For example, a software cell is not restricted to two dimensions and need not be mapped to contiguous processors; software cells and wires can overlap.

Concepts similar to our software cells and ports have been proposed by several researchers, notably the iWARP group [2], Griswold et al. [17], and Browne et al. [4]. The focus of iWARP is the generation of efficient programs for a systolic processor. Hence, the forms of programs that can be specified is limited: the contiguous submesh is the only topology supported and the number of channels is limited. Griswold et al. propose process ensembles as a means of organizing data, computation, and communication. However, their concepts do not support hierarchies of topologies. Browne et al. propose a compositional calculus for specifying interconnections between software chips. The integration of this calculus into a programming notation is not discussed, and the notion of virtual topology is absent.

The use of virtual topologies to abstract mapping decisions was first proposed by Martin [22]. Hudak [20] and Taylor [26], among others, have used similar ideas to specify mapping decisions in declarative programming systems. In Hudak's scheme, arbitrary integer functions can be used to specify both relative and absolute locations. Taylor uses them to specify relative locations in an infinite computing surface. However, the important concepts of mapping function and software cells are absent from these proposals.

Chien and Dally's Concurrent Aggregates (CA) language allows the definition of homogeneous collections of objects called *aggregates*; messages addressed to an aggregate are routed to one of its members [8]. As in our proposal, concurrent structures can be defined and composed with other structures to build concurrent programs. However, issues associated with spatial organization of such structures are not addressed.

We conclude with a note concerning what is currently the most common approach to multicomputer programming, which we will refer to as "heavyweight message passing". In this approach, applications are structured as heavyweight processes, one per physical processor; processes communicate by sending and receiving messages. Unfortunately, this structure fails to isolate design decisions concerned with mapping, communication, and scheduling. As only one process can be located on each processor, and messages must be directed to processors, logically disjoint design decisions become inextricably intertwined and are clearly visible in module interfaces. As a consequence, programs become complex and inflexible, and reusable libraries are hard to define. It is important to realize that in many cases these difficulties are not inherent in programming problems but are instead artifacts of this particular approach.

11 Conclusions

Research in sequential programming has demonstrated the value of isolating design decisions that are difficult or likely to change. We have argued that in multicomputer programs, design decisions concerned with mapping, communication, and scheduling are problematic and hence deserving of encapsulation. We have described four computational abstractions that together allow these decisions to be isolated and hidden. *Virtual topologies*, *virtual channels*, and *lightweight processes* provide mechanisms for isolating mapping, communication, and scheduling decisions, while *port arrays* allow these techniques to be applied to program components that must execute on many processors.

In addition to describing the abstractions, we have shown how they can be encapsulated in programming language constructs, and have outlined solutions to three substantial parallel programming problems. These code sketches indicate how the abstractions allow complex programs to be developed by composing existing templates, and how mapping, communication, and scheduling decisions can be separated from algorithmic specifications. As a consequence, changes that in most programming systems would be extremely difficult (e.g., exploring a different mapping) can be accomplished by changing a few lines of code. Similarly, overlapping of computation and communication, normally difficult to accomplish, is achieved automatically.

We believe that the most significant aspect of this work is that it is now possible to define truly modular (and hence reusable) parallel libraries. In particular, we are able to define *cells*: parallel program components with specified internal logic and external communication ports, but encapsulating no mapping, communication, or scheduling decisions. Instead, these decisions are isolated in the code used to compose cells to form applications. Thus, the same cell can be used unchanged in different contexts. Even greater flexibility is provided by *templates*: cell definitions parameterized with the code to be executed at each node.

We have assumed in this paper that all cells are implemented with the same technology (PCN). However, there is no reason why cells should not be implemented using different techniques (e.g., message-passing libraries or data-parallel languages), as long as implementations do not encapsulate mapping, communication, or scheduling decisions.

Acknowledgments

This research was supported by the Applied Mathematical Sciences subprogram and the Atmospheric and Climate Research Division of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38. Development of PCN was sponsored by the National Science Foundation's Center for Research in Parallel Computation under Contract NSF CCR-8809615. We thank fellow participants in the CHAMMP climate modeling program at Argonne National Laboratory, the National Center for Atmospheric Research, and Oak Ridge National Laboratory for assistance with development of sequential and parallel codes.

References

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

[2] Borkar, S., et al., iWarp: An integrated solution to high-speed parallel computing, *Proc. Supercomputing Conf.*, 330–339, 1988.

[3] Brinch Hansen, P., Joyce — a programming language for distributed systems, *Softw. P. and E.*, 17, 29–50, 1987.

[4] Browne, J., Werth, J., and Lee, T., Intersection of parallel structuring and reuse of software components, *Proc. Intl Conf. on Parallel Processing*, Penn. State Press, 1989.

[5] Browning, G., Hack, J., and Swarztrauber, P., A comparison of three numerical methods for solving differential equations on the sphere, *Mon. Wea. Rev.*, 117 (5), 1989.

[6] Chandy, C., and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

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

[8] Chien, A., and Dally, W., Concurrent Aggregates, *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, ACM, 1990, 187–196.

[9] 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.

[10] 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.

[11] 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.

[12] Foster, I., Gropp, W., and Stevens, R., The parallel scalability of the spectral transform method, *Mon. Wea. Rev.*, March 1992.

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

[14] Foster, I., Olson, R., and Tuecke, S., Productive parallel programming: The PCN approach, *Scientific Programming*, 1(1), 1992 (in press).

[15] Foster, I., and Taylor, S., A compiler approach to scalable concurrent program design, Technical Report, Argonne National Laboratory, 1992.

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

[17] Griswold, W., Harrison, G., Notkin, D., and Snyder, L., Port ensembles: A communication abstraction for nonshared memory parallel programming, *Proc. Intl Conf. on Parallel Processing*, Penn. State Press, 1990.

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

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

[20] Hudak, P., Para-functional programming, *IEEE Computer*, 60–70, Aug 1986.

[21] Kahn, G., The semantics of a simple language for parallel programming, *Proc. IFIP Congress '74*, North Holland, 1974.

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

[23] Mead, C., and Conway, L., *Introduction to VLSI Systems*, Addison Wesley, 1980.

[24] Parnas, D., On the criteria to be used in decomposing systems into modules, *CACM* 15(2), 1053–1058, 1972.

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

[26] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, 1989.

[27] Walker, D., Drake, J., and Worley, P., Parallelizing the spectral transform method — Part II, Tech. Rep. ORNL/TM-11855, Oak Ridge National Laboratory, Oak Ridge, Tenn., 1991. (Available from DOE Office of Scientific and Technical Information.)

[28] Young, M., et al., The duality of memory and communication in Mach, *Proc. 11th Symp. on Operating System Principles*, ACM, 63–76, 1987.

Appendix: Implementation of Building Blocks

We present PCN implementations for three building blocks used in this paper. The brief descriptions that accompany the programs highlight selected features but do not explain all details.

Butterfly Template. Both the `reduce` and `fft` cells (Section 6) can be defined in terms of a `butterfly` template, which creates cells with a butterfly network as their internal communication structure. The template operates loosely synchronously and is invoked as `butterfly(op,In,Out)`. Each time data arrives on the port array `In`, the supplied `op` is invoked with the message and a list of butterfly communication streams as arguments. The value returned by `op` is output on `Out`.

A possible implementation is given in Figure 10. The code verifies that the number of processors (p) is a power of two, declares a port `Ps` of dimension $\log_2 p$ (for the butterfly communication streams), and creates a `streams` and `bflynode` process on each node of the current topology. Each `streams` process collects $\log_2 p$ input streams and $\log_2 p$ output streams for its node. The `bflynode` procesres handle the actual computation: when a message arrives on `in`, a process invokes the supplied `op` with the message and a vector of communication streams as arguments. The value returned by the operator is output on `out`.

The supplied program maps the ith node of the butterfly ($0 \leq$ i $< p$) to the ith node of the current topology. This mapping is efficient in an array topology but not necessarily in other topologies. A more sophisticated implementation would specify alternative mappings for different architectures.

Reduce Template. The `reduce` template reduces a set of values received on one port array, using a supplied binary operator, and broadcasts the result of the reduction operation on another port array. This operation can be programmed with a butterfly network, as shown in Figure 11. Note the use of the `butterfly` template, parameterized with the `reduce_bfly` procedure. In each of $\log_2 p$ phases (one per element on the list of streams passed to `reduce_bfly`), each node sends a partially reduced value to another node in the network, receives a partially reduced value, and performs a reduction operation.

Mesh Template. An implementation of the `mesh_io2` template is provided in Figure 12. This program actually implements a torus with wrap-around connections between west/east and north/south edges, as the edge connections are ignored by the composite mesh code that uses this template, and it is simpler to wrap the edges connections than to leave them unconnected. Note how the call to op is passed the appropriate elements of `I1`, `O1`, `I2`, and `O2`, as well as `N`, `E`, `S`, and `W`. (The macro `I(i,j)` is used to compute port array indices, as two-dimensional port arrays are not supported directly.) The location function `mesh_node` is used to locate the calls to op within the mesh topology.

```
butterfly(op,In,Out)
port In[], Out[];
{|| power_of_two(nodes(),r),
    { ? r == "false" -> error(),
        default -> {|| log2(nodes(),l2),
                       bfly(op,In,Out,l2)
                   }
    }
}

bfly(op,In,Out,l2)
port In[], Out[], Ps[l2*nodes()];
{|| i over 0..nodes()-1 :
        {|| streams(i,l2,l2-1,i,1,Ps,is,os),
            bflynode(op,In[i],Out[i],is,os) } @ node(i)
}

streams(i,l2,l,ii,k,Ps,is,os)
port Ps[];
{ ? l >= 0 -> {|| { ? ii%2 == 0 -> si = {"+",Ps[l*l2 + i+k]},
                      ii%2 != 0 -> si = {"-",Ps[l*l2 + i-k]}
                  },
                  so = Ps[l*l2 + i],
                  is = [si|is1], os = [so|os1],
                  streams(i,l2,l-1,ii/2,k*2,Ps,is1,os1)
              },
    default -> {|| is=[], os=[]}
}

bflynode(op,in,out,is,os)
{ ? in ?= [id|in1] ->
        {|| ‘op‘(id,od,is,os,is1,os1),
            out = [od|out1],
            bflynode(in1,out1,is1,os1)
        }
}
```

Figure 10: Butterfly Template

```
reduce(op,In,Out)
port In[], Out[];
{|| butterfly(reduce_bfly(op),In,Out) }

reduce_bfly(op,in,out,is,os,is1,os1)
{ ? os ?= [o|os2] ->
    {|| o = [in|o1], os1 = [o1|os3], /* Send IN on O */
        is ?= [{_,[v|i]}|is2] ->     /* Recv V on I */
        {|| `op`(in,v,newv),         /* Reduce IN and V */
            is1 = [{_,i}|is3],       /* Prepare to recurse */
            reduce(op,newv,out,is2,os2,is3,os3)
        }
     },
     os ?= [] -> {|| out=in, is1=[], os1=[]}  /* Done */
}
```

Figure 11: Reduction Template

```
#define I(i,j) ((((i)+m)%m)*n + ((j)+n)%n)

mesh_io2(op,I1,O1,I2,O2)
port I1[],O1[],I2[],O2[];
port N[nodes()],E[nodes()],S[nodes()],W[nodes()];
{ ? topology() ?= {"mesh",m,n} ->
    {|| i over 0..m-1 :
        {|| j over 0..n-1 :
            `op`(I1[I(i,j)], O1[I(i,j)], I2[I(i,j)], O2[I(i,j)],
                 {N[I(i,j)],  E[I(i,j)],  S[I(i,j)],  W[I(i,j)],
                  S[I(i-1,j)], W[I(i,j+1)],
                  N[I(i+1,j)], E[I(i,j-1)]}
            ) @ mesh_node(i,j)
    }
}
```

Figure 12: Mesh Template

Mathematics and Computer Science Division
Building 221
Argonne National Laboratory
Argonne, Illinois 60439-4844

Recent Preprints:

J. N. Lyness, "On Handling Singularities in Finite Elements," MCS-P271-1091.

G. Kirlinger and G. F. Corliss, "On Implicit Taylor Series Methods for Stiff ODEs," MCS-P272-1191.

Robert M. Corless and George F. Corliss, "Rationale for Guaranteed ODE Defect Control," MCS-P273-1191.

Christian H. Bischof, Biswa N. Datta, and Avijit Purkayastha, "A Parallel Algorithm for the Multi-input Sylvester-Observer Equation," MCS-P274-1191.

E. Lusk and L. Wos, "Benchmark Problems in Which Equality Plays the Major Role," MCS-P275-1191.

J. N. Lyness and T. Sorevik, "Lattice Rules by Component Scaling," MCS-P276-1191.

Mark Jones and Paul Plassmann, "Scalable Iterative Solution of Sparse Linear Systems," MCS-P277-1191.

C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, "ADIFOR: Fortran Source Translation for Efficient Derivatives (ADIFOR Working Note #4)," MCS-P278-1291.

M. K. Kwong, "A Dirichlet Problem with Infinite Multiplicity," MCS-P279-1291.

J. Zhang and L. Wos, "Automated Reasoning and Enumerative Search, with Applications to Mathematics," MCS-P280-1291.

J. Garner, M. Spanbauer, R. Benedek, K. J. Strandburg, S. Wright, and P. Plassmann, "Critical Fields of Josephson-Coupled Superconducting Multilayers," MCS-P281-1291.

Ping Tak Peter Tang, "Dynamic Condition Estimation and Rayleigh-Ritz Approximation," MCS-P282-1291.

Richard G. Carter, "A Worst-Case Example Using Linesearch Methods for Numerical Optimization with Inexact Gradient Evaluations," MCS-P283-1291.

Gui-Qiang Chen, Qiang Du, and Eitan Tadmor, "Spectral Viscosity Approximations to Multidimensional Scalar Conservation Laws," MCS-P284-1291.

Larry Wos, "The Problem of Reasoning from Inequalities," MCS-P285-0192.

Christian H. Bischof and Xiaobai Sun, "A Divide-and-Conquer Method for Tridiagonalizing Symmetric Matrices with Repeated Eigenvalues," MCS-P286-0192.

Ewing Lusk, "Performance Visualization for Parallel Programs," MCS-287-0192.

C. Bischof, A. Carle, G. Corliss, and A. Griewank, "ADIFOR Automatic Differentiation in a Source Translator Environment (ADIFOR Working Note #5)," MCS-P288-0192.

L. Wos, "The Problem of Choosing between Using and Avoiding Equality Predicates," MCS-P289-0192.

W. McCune and L. Wos, "Application of Automated Deduction to the Search for Single Axioms for Exponent Groups," MCS-P291-0292.

P. Plassmann and M. Jones, "The Effect of Many Color Orderings on the Convergence of Iterative Methods," MCS-P292-0292.

P. Takac, "Dynamics on the Attractor for the Complex Ginzburg-Landau Equation," MCS-P293-0392.