Preprint MCS-P391-1093, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1994.

Language Constructs for Modular Parallel Programs

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

Abstract

We describe programming language constructs that facilitate the application of modular design techniques in parallel programming. These constructs allow us to isolate resource management and processor scheduling decisions from the specification of individual modules, which can themselves encapsulate design decisions concerned with concurrency, communication, process mapping, and data distribution. This approach permits development of libraries of reusable parallel program components and the reuse of these components in different contexts. In particular, alternative mapping strategies can be explored without modifying other aspects of program logic. We describe how these constructs are incorporated in two practical parallel programming languages, PCN and Fortran M. Compilers have been developed for both languages, allowing experimentation in substantial applications.

Keywords: modularity; parallel programming; programming languages; program composition; code reuse; virtual computer

1 Introduction

In sequential programming, modular and object-oriented design and programming techniques are well understood and widely used to reduce complexity, permit separate development of components, and encourage reuse [31, 27, 3]. In parallel programming, the situation is less advanced. Parallel programs are, for the most part, developed in an ad-hoc fashion, as monolithic entities that cannot easily be adapted to changing circumstances. Code reuse is rare, outside the specialized context of single-program multiple-data (SPMD) programming, where the same program is run on every processor and libraries can be called to perform common global operations [13, 25].

This paper presents programming language constructs that permit the benefits of modularity to be realized in parallel programs. The central ideas are as follows. First, process and data placement decisions within an individual module are specified with respect to a *virtual computer*; the embedding of this virtual computer within a physical or another virtual computer is specified when the module is invoked. This approach permits resource management and locality decisions to be separated from the specifications of individual modules and developed in a hierarchical fashion by using stepwise refinement techniques.

Second, virtual computer constructs are incorporated into *compositional* programming languages, in which concurrency is specified with explicit parallel constructs and interactions between concurrent processes are restricted so that neither physical location nor

execution schedule affects the result of a computation [7]. Languages with this property include Strand and PCN, in which processes interact by reading and writing shared single-assignment variables, and Fortran M, in which interactions occur via single-reader, single-writer virtual channels. Compositional languages permit design decisions concerned with mapping, data distribution, communication, and scheduling to be made separately and to be modified without changing other aspects of a design [15].

In this paper, we show how virtual computer constructs are integrated into two compositional parallel programming languages: PCN [9, 16] and Fortran M [14]. PCN is a C-like language that integrates ideas from concurrent logic programming and imperative programming; Fortran M is a small set of extensions to Fortran. In each case, virtual computers and related constructs have been introduced in a manner that is consistent with the base language, hence simplifying both comprehension and compilation. Both PCN and Fortran M have been used to develop a range of substantial parallel applications; these provide an empirical basis for evaluation of the concepts.

In summary, the contributions of this paper are as follows:

- The definition of programming language constructs that facilitate the modular construction of parallel programs.
- The instantiation of the constructs in practical parallel programming languages.
- Empirical evaluation in a range of applications.

The next three sections of this paper address each of these issues in turn, after which we review related work and present our conclusions.

2 Modularity and Parallel Programs

We use the term "modularity" to refer to two related program-structuring techniques:

1. Stepwise refinement [31] – the decomposition of a complex problem into simpler subproblems, each solved by a separate module with a well-defined interface.
2. Modular decomposition [27] – the isolation in separate modules of design decisions that are difficult, likely to change, or common to several program components.

The first of these techniques is more naturally applied *top down* in the design process and, if care is taken to ensure generality, produces modules that can be reused in different contexts. The second is more naturally applied *bottom up* and can reduce the cost of program modifications. Both techniques are central to object-oriented design [3].

It is instructive to examine how these techniques can be applied in sequential and parallel programming. For illustrative purposes, we consider a simple example: the convolution operation used, for example, in motion estimation algorithms in image processing [11]. Each pair of images is first processed by using a two-dimensional fast Fourier transform (FFT). The resulting matrices are then multiplied, and an inverse FFT is applied to the result. The various operations, and the dependencies between them, are illustrated in Figure 1.

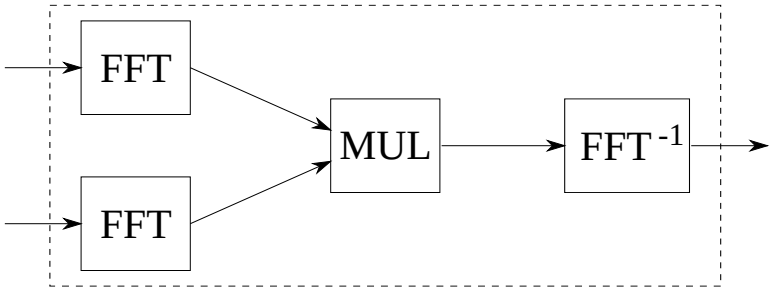

Figure 1: Convolution Algorithm Structure

In a sequential implementation, stepwise refinement may identify FFT, matrix multiplication, input, and output modules. The data structures to be input to and output from each module are specified in its interface; internal data structures and algorithmic details are encapsulated and not visible to other components. The interface specifies the size and perhaps the type of the data structures to be operated on, hence allowing the module to be used in different contexts. For example, the same FFT module can be called three times in the convolution example. The interface design may also address efficiency issues, for example by specifying that data is to be passed by reference rather than by value.

The same decomposition can be used in a parallel implementation. Now, however, we need to encapsulate not only data and computation but also concurrency, communication, process mapping, and data distribution. If we do not encapsulate this information, two modules mapped to the same processor might interfere, for example if one received messages intended for the other. When composing two modules, we prefer not to be aware of how each maps computation and data. (The mappings used in different modules can be made conformant to improve performance, but should not affect correctness.) Again, efficiency issues need to be addressed in the interface design; for example, by allowing for the exchange of distributed data structures between parallel modules.

In a sequential implementation, modular decomposition may identify the representation of matrices as common to several program components. Hence, we define a module that provides the required matrix operations: read an element, write an element, etc. The rest of the program uses these functions to access matrices; alternative matrix representations (e.g., linear storage or a linked-list representation of sparse matrices) can be substituted without changes to other components.

A design decision of equal significance in a parallel implementation is how computation and data are mapped to processors. For example, the various phases of the convolution problem illustrated in Figure 1 can be organized so as to execute (a) on disjoint sets of processors as a pipeline, (b) in sequence for each input data set, or (c) as some hybrid of these two approaches (Figure 2). We wish to localize the changes required to explore these alternatives, which do not affect the basic structure of the algorithm but may have different performance characteristics. These changes concern the allocation of computational resources (processors) to modules and the scheduling of components mapped to the same processor.

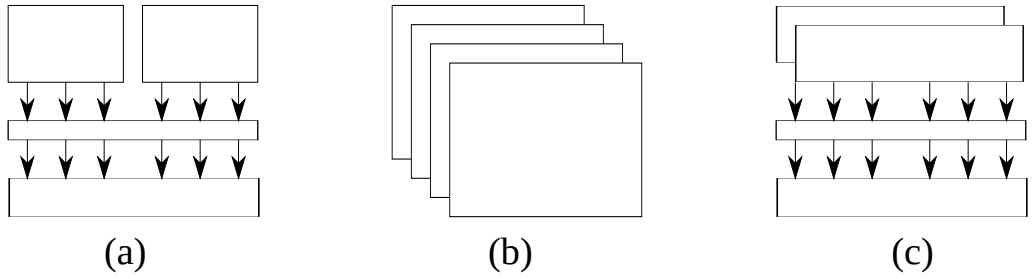

Figure 2: Alternative Mappings for the Parallel Convolution Algorithm

We have found that the design issues encountered in the convolution example are common to many parallel programming problems. In summary, the most important issues are as follows.

1. Parallel modules need to encapsulate not only computation and data, as in sequential programming, but also communication, concurrency, process mapping, and data distribution.

2. Resource allocation and scheduling decisions need to be isolated from module specifications.

3. Interfaces between parallel modules need to allow for the sharing of distributed data structures.

We now present the programming language concepts that we use to address these issues.

Virtual Computers. We remove resource allocation decisions from individual modules by specifying process and data mapping within a module relative to a *virtual computer*. The shape, size, and mapping of this virtual computer to physical processors are specified externally to the module. This mapping need not be one to one: several virtual computer nodes can be located on a single physical processor, or a virtual computer can correspond to a subset of a physical computer. For example, a parallel FFT module might be specified with respect to a one-dimensional array of processors with size specified in its interface. Only the virtual computer specification in the calling program need be changed to explore the different mappings illustrated in Figure 2.

Virtual computers control both resource allocation and locality. The size of a program component's virtual computer determines in part the computational resources available to that component. (The number of other components mapped to the same processors is also a factor.) The location of the virtual computer determines whether interactions with other components require inexpensive intraprocessor or expensive interprocessor communication. For example, compared with strategy (a), strategy (b) in Figure 2 permits different phases of the convolution operation to communicate inexpensively, because all operate on the same processors rather than on disjoint sets of processors. On the other

hand, it increases intramodule communication costs, because each module is distributed over many processors.

We define three operations on virtual computers: process mapping, data distribution, and embedding.

Process mapping is the mechanism by which a process is created on (or migrated to) a particular processor. Language constructs are provided to specify that a particular process is to initiate or continue execution on a particular node of a virtual computer. For example, an FFT module might create one process on each node of the virtual computer in which it executes.

Data distribution is the mechanism by which data structures are distributed over processors. Language constructs are provided to define the type and size of a distributed data structure and the mapping of its elements to the nodes of a virtual computer. For example, an FFT module might specify a blocked distribution of its input, output, and work arrays.

Embedding is the mechanism by which virtual computers are defined. An embedding function specifies the size and type of a computer and how its nodes are mapped to the nodes of an existing physical or virtual computer. Language constructs are provided for specifying embedding functions. In the convolution example, an embedding function would be used to define the subcomputers in which the various modules (FFT, etc.) execute.

Embedding functions allow the programmer to develop resource management and locality specifications in a hierarchical fashion. For example, assume that our convolution program is defined to execute in a two-dimensional virtual computer. This convolution code may itself be embedded in a larger program. When calling the convolution code, it suffices to create a two-dimensional virtual computer in order to specify how the convolution code (and hence its constituent modules) is mapped to physical processors.

Scheduling. It is often necessary (for correctness) or useful (for efficiency) for components mapped to the same processor to execute in an interleaved fashion. For example, in Figure 2, concurrent execution of different pipeline stages can improve efficiency by allowing overlapping of computation and communication: one can execute while others wait for data.

A programmer can control scheduling explicitly by merging code for different components into a single thread of control for each processor. However, this approach typically requires an intimate and complex intermingling of the different components, which not only compromises modularity but also hinders exploration of alternative scheduling strategies. Unfortunately, scheduling is an aspect of a parallel algorithm that is frequently changed. Execution on a different machine, minor changes to other components of an algorithm, or a new compiler can all change the temporal relationships between components and hence change the optimal schedule.

We address this problem by permitting the programmer to specify components as separate processes that execute in a data-driven manner. The execution schedule is then determined by the availability of data and by the scheduling algorithm used to select executable processes. This approach is, of course, well known in actors, dataflow, parallel functional programming, and concurrent logic programming [1, 6, 21, 15].

Interfaces. Concurrently-executing program components must be able to share data. As noted above, a mechanism is required that supports encapsulation, that is independent of resource allocation and scheduling decisions, and that is efficient on parallel computers. We achieve this as follows.

Logical Resources. In order that modules do not interfere in unexpected ways, interactions between program components are restricted to language objects passed via interfaces. Hence, concurrently-executing processes cannot operate on shared "global" variables or interact via physical resources such as processor or memory addresses.

Schedule Independence. In order that scheduling decisions can be changed without modifying other program components, we restrict operations on shared data to those for which execution order does not affect the result of a computation [7]. For example, two processes may share a channel on which one performs nonblocking sends while the other performs blocking receives. Or, several processes may share a single-assignment variable that one writes and several read. In both cases, the reader(s) block(s) until data is provided by the writer; hence, the schedule does not affect the result of a computation.

Global Address Space. In order that mapping decisions can be changed without modifying other program components, we make no distinction at the language level between local and remote references to objects. Hence, a process can operate on a resource regardless of its location [15]. This requires some form of global address space, as provided, for example, by virtual channel and single-assignment variables.

Concurrency. Concurrent module interfaces are achieved by permitting shared data (for example, arrays of virtual channels or single-assignment variables) to be distributed over processors and accessed concurrently by program components.

The first three of these requirements are satisfied by compositional languages such as PCN and Fortran M. The integration of data distribution constructs satisfies the fourth requirement.

3 Language Constructs for Modularity

The programming language concepts introduced in the preceding section have been incorporated in the PCN and Fortran M programming languages. In each case, it has proved possible to introduce the concepts in a manner that is consistent with other language concepts.

3.1 Program Composition Notation

Program Composition Notation (PCN) [9, 16] is a high-level concurrent programming language with a C-like syntax that combines features of concurrent logic languages and imperative languages. Programs are constructed by using three composition operators — parallel, sequential, and choice — to compose procedure calls, other compositions, or primitive operations. Concurrently executing processes interact by reading and writing single-assignment variables. Such variables are initially undefined and can be written at most once; an attempt to read an undefined variable suspends until the variable is written.

In both PCN and Fortran M, we specify that the virtual computer in which a process executes is by default the same as that process's parent, unless specified otherwise when the process is created. Information about this virtual computer can be obtained by inquiry

functions; in PCN, these are `topology()` and `nodes()`, which return a representation of its type (e.g., a term {`"mesh",16,32`}), and its size, respectively. Mapping, data distribution, and embedding operations within the process are performed relative to this implicit virtual computer. An alternative approach, which also has its merits, is to represent the virtual computer as a data structure that can be manipulated in the language. This approach is adopted in CC++ [8].

Mapping in PCN is specified by using the infix operator `@` to apply a mapping function to a process or block. A mapping function returns a node number within the current virtual computer. For example, in the following the `@` operator is used to locate a call to process `myproc()` on node `(i,j)` of a `M`×`N` mesh. The first code fragment is a parallel block that invokes procedure `myproc` on node `meshnode(i,j)` of the current virtual computer. The second code fragment is a definition for the `meshnode` function. This uses a PCN choice composition operator (a form of guarded command) to specify actions to be performed if the current topology is a mesh and the mesh location is valid (action: return the appropriate index within the mesh) or otherwise (action: return an error value). The infix operator `?=` is a term-matching primitive that tests and decomposes the term returned by `topology()`. All variables in PCN are local to the procedure in which they occur; single-assignment variables are not explicitly declared.

```
{|| ...
    myproc(...) @ meshnode(i,j)
    ...
}

function meshnode(i,j)
{ ? topology() ?= {"mesh",M,N}, i < M, j < N ->
        return(M*i+j),
    default -> return(-1)
}
```

Data distribution is supported in the form of blocked distributions of arrays of single-assignment variables. (In a blocked distribution, each processor is allocated a contiguous block of array elements [18].) Elements of a distributed array are accessed in the same manner as ordinary array elements. More complex distributions and data structures, such as those supported in data-parallel programming languages [30, 18], can be integrated in the same manner, but are not supported in the current PCN compiler.

A keyword `port` is used to declare these distributed arrays. For example, the following procedure declares a distributed array `P` of single-assignment variables with one element on each node of the current virtual computer. The syntax "`{|| i over 0..n ::`" is a parallel enumerator, used here to create `nodes()` instances of the `ringnode` process. The location function `node(i)` is assumed to return its argument; it is used to place each process on a separate node in the current virtual computer. Elements of `P` are used to connect these processes in a unidirectional ring; as `P` is distributed, the newly created processes can access elements efficiently, without the need to communicate with a central location.

```
ring()
```

```
port P[nodes()];
{|| i over 0..nodes()-1 ::
       ringnode(P[i],P[(i+1)%nodes()]) @ node(i)
}
```

Embedding is supported by the infix operator `in`, used to annotate a process call or block with an embedding function that returns a representation of the new virtual computer. For example, assume an embedding function `subarray(start,size)` which embeds a 1-D computer of specified `size` at location `start` in a parent computer of the same type but larger size. The following code fragment uses this function to invoke two of the FFT processes required for the convolution problem, each on a separate set of n processors.

```
{||
    ...
    call fft(...) in subarray(0,n),
    call fft(...) in subarray(n,n),
    ...
}
```

The PCN extensions that support virtual computers, data distribution, and embedding have been incorporated into a PCN compiler [17]. A programmable source-to-source transformation system is used to translate the extended language into PCN as well as calls to a runtime library that manages virtual computers and handles requests for distributed data. This compiler has supported extensive experimentation with the language extensions, as described in Section 4. The compiler is designed to optimize process mapping and access to distributed data, both of which take constant time irrespective of computer size. Execution of an embedding function takes time proportional to the square of the size of the new virtual computer; however, this cost is distributed over the physical processors on which the virtual computer is to be created and has not proved excessive on 500-processor computers.

3.2 Fortran M

Fortran M is a small set of extensions to Fortran designed to support the modular construction of scientific and engineering applications [14]. A primary goal in designing Fortran M was to provide a minimal set of extensions that were consistent with Fortran concepts. The temptation to "fix" Fortran was avoided. We describe Fortran M for two reasons. First, it shows how our concepts can be incorporated into a programming language with characteristics very different from those of PCN. Second, it allows us to provide a more comprehensive illustration of how distributed arrays are integrated with virtual computers.

Fortran M programs use parbegin/parend constructs to create processes that interact by sending and receiving messages on typed *ports* that reference single-writer, single-reader *channels*. Channel operations are modeled on Fortran file I/O constructs. Mapping is specified with respect to virtual computers. In keeping with Fortran concepts, virtual computers are N-dimensional arrays of virtual processors. As in High Performance Fortran (HPF) [20], a `PROCESSORS` declaration is used to define the size and shape

of the processor array that is to apply in a particular process, and the inquiry functions `Number_Of_Processors` and `Processor_Shape` permit a process to determine the size and shape of the computer in which it executes.

Mapping is specified by using the `LOCATION` annotation. An annotation of the form `LOCATION(`I_1`,...,`I_n`)` appended to a process call indicates that the call is to execute on node `(`I_1`,...,`I_n`)` of the current virtual computer. An executable `RELOCATE` statement can also be defined to request process migration; however, this is not currently supported in the Fortran M compiler due to the portability problems associated with process migration.

The following code fragment implements the ring structure for which PCN code was presented in the preceding section. The code declares arrays of inports and outports, creates channels, and then invokes the `ringnode` processes, passing the ports as arguments. Notice the `PROCESSORS` declaration and `LOCATION` annotation.

```
process ring
PROCESSORS(p)
inport  (integer) pi(p)
outport (integer) po(p)
do i = 1,p
   channel(in=pi(i), out=po(mod(i)+1))
enddo
processdo i = 1,p
   processcall ringnode(pi(i), po(i)) LOCATION(i)
endprocessdo
end
```

Data distribution is specified by using data distribution statements based on those in HPF, with the difference that these statements are interpreted with respect to the current virtual computer rather than an entire machine. For example, the following statements can be added to the previous program to specify that arrays `pi` and `po` are to be distributed.

```
DISTRIBUTE pi(BLOCK)
ALIGN po WITH pi
```

Embedding is specified with the `SUBMACHINE` annotation. When appended to a process call, this indicates that the call is to execute in a submachine of specified size and shape. Fortran 90 array constructors can be used to define the new virtual machine. For example, the following code fragment invokes a parallel FFT process in the ith row and column of an $N \times N$ processor array, respectively. Processes and distributed arrays created within each FFT process are distributed only over the processors on which that process executes.

```
processes
   processcall fft(...) SUBMACHINE(i, 1:N)
   processcall fft(...) SUBMACHINE(1:N, i)
endprocesses
```

A Fortran M compiler has been constructed that translates Fortran M programs into Fortran plus calls to a communication and process management library. This compiler is operational on both networks of workstations and parallel computers, and has been used in a variety of programming experiments, as described in Section 4.

4 Experiences

The language constructs described in this paper have been applied to a wide range of scientific and engineering problems, in areas as diverse as atmospheric modeling, computational biology, computational chemistry, and image processing. Here, we summarize some of these investigations, which provide insights into the strengths and weaknesses of the approach.

4.1 Parallel Solvers

We first describe a study of parallel algorithms for numerical methods used to solve partial differential equations in spherical geometry, in which PCN was used to prototype algorithm variants [10]. Three different numerical methods were considered: a spectral transform method on a regular latitude/longitude grid, a finite difference method on overlapping stereoscopic grids, and a control volume on an icosahedral-hexagonal grid. In each case, the resulting parallel algorithms are complex, and there was a need to explore algorithmic variants. We use the simplest of the three examples to illustrate the approach. The spectral transform algorithm comprises three components. It operates on a latitude-longitude grid, with the bulk of the computation being performed independently at each grid point; communication is encapsulated in an FFT performed in each latitude and a summation performed in each longitude. The PCN implementation is structured as follows. The routine `spectral` is executed in a two-dimensional mesh virtual computer created by a call to `spectral_mesh`. An FFT module is invoked in each row of this computer and a summation module in each column, by using embedding functions `row` and `col`, respectively. A modified version of the sequential code is executed on each processor. The missing arguments represent the port variables used for communication between the different modules.

```
main(lat,lon)
{|| spectral(lat,lon) in spectral_mesh(lat,lon) }

spectral(lat,lon)
...
{|| {|| i over 0..lat-1 :: fft(...) in row(i) },
    {|| i over 0..lon-1 :: sum(...) in col(i) },
    {|| i over 0..lat*lon-1 :: compute(...) @ node(i) }
}
```

Our approach proved to have two major benefits in this case. First, it proved possible to reuse FFT and summation modules. Both were available as PCN programs and could be called directly, without reengineering to execute in different contexts. In effect, no new *parallel* code had to be written beyond that shown above: only the sequential Fortran code required modification, to execute in the parallel harness. Second, experimentation with alternative mapping strategies was facilitated. For example, on a two-dimensional mesh computer such as the Intel Delta, it is possible to map each "row" of the spectral mesh to either a row or a submesh of the computer. Both have potential performance advantages: the former results in simpler communication patterns, while the latter makes more efficient use of available wires. We were able to experiment with both approaches.

These benefits were also observed when implementing the finite difference and control volume algorithms [10].

The study also revealed deficiencies in the PCN approach and tools. The use of two languages (PCN for the parallel harness, and Fortran for sequential computation) proved offputting to the applied mathematicians who assisted in the project. Also, although the PCN scheduler schedules computational tasks and remote reads correctly, a lack of preemption meant that remote read operations were sometimes delayed for extended periods awaiting completion of computationally intensive operations. As a short-term solution, we decomposed the `compute` process into many lightweight processes. This is not as expensive as it might sound, because of the low cost of switching between the lightweight threads used in PCN: less than 50 μsec on a Sun 4. (For example, the control volume code achieves 75%–90% parallel efficiency relative to a sequential Fortran code on 512 Intel Delta processors, depending on problem size, even when decomposed in this way.) In the long term, some form of preemption is required.

4.2 Earth System Models

In the second application that we consider, our constructs are applied in a multidisciplinary framework. An earth system model integrates models of the atmosphere, ocean, biosphere, etc. The developer of a parallel earth system model encounters challenging encapsulation issues, as component models may themselves be parallel programs. The ability to explore alternative mapping strategies is also important, since it can sometimes be desirable to execute the component models on different processors or even on different computers; in other situations, it may be preferable to multiprocess models on the same processors.

In order to explore the application of our approach to this class of problems, we have developed a Fortran M framework code that couples simple atmosphere and ocean models. Arrays of channels are used to transfer data between two parallel models and virtual computers are used to control process placement. This framework code has been used to explore performance issues. For example, the simple form of compositional data structure used in Fortran M — the channel — requires that data be copied when it is transferred from one module to another, even if these two modules execute on the same processor. An early concern was that high copying costs would necessitate that the portability obtained by a Fortran M implementation be sacrificed, in order to permit direct sharing of data. However, experiments showed that the cost of a modular decomposition based on Fortran M concepts was small: about 2.8% of total execution time on a Sun-4 workstation. Much of this overhead was due to the relatively high cost of switching between Fortran M processes, currently implemented as Unix processes, and the use of TCP/IP sockets for interprocess communication. Experiments with a prototype Fortran M compiler that uses Sun lightweight processes (threads) and shared-memory operations show that channel overhead can be reduced significantly. For example, Unix process switch cost (defined here as the time required to send a null message between two processes on the same processor) is around 1.8 msec; with threads, this is reduced to 0.37 msec, and greater reductions appear possible if specialized thread packages are used.

4.3 Education

PCN and Fortran M have been used to teach parallel programming to graduates and undergraduates in both computing and the sciences. Students were provided with libraries of modules — implementing global operations, mesh structures, load balancing libraries, and the like — which they then used in programming projects. This approach proved effective at two levels. Students were able to use some modules unchanged, without a deep understanding of their implementation. Other modules served as frameworks that they modified to suit a particular purpose. The sophistication of the programs developed by students was considerably higher when module libraries were used. Not surprisingly, the computing students showed a strong preference for PCN, while the science students preferred Fortran M.

5 Related Work

Several parallel languages and programming environments have been developed to support the modular construction of parallel programs. Borkhar et al. [4] propose that parallel programs be constructed by plugging together "cells," in a manner analogous to VLSI. They use this technique to generate efficient programs for the iWarp systolic processor. occam [24] has been used for similar purposes. The target hardware limits the programs that can be specified in these systems: in the iWARP work the contiguous submesh is the only virtual computer supported and the number of channels is limited. Griswold et al. propose *process ensembles* as a means of organizing data, computation, and communication [19]. However, they do not consider hierarchies of virtual computers. Browne et al. propose a compositional calculus for specifying interconnections between *software chips* [5]. The integration of this calculus into a programming notation is not discussed, and the notion of virtual computer is absent.

Some recent work on parallel message-passing libraries has explored the use of "process groups" and "communication contexts" to support the encapsulation of communication operations in parallel libraries [28, 2]. 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 [12]. As in this paper, concurrent structures can be defined and composed with other structures to build concurrent programs. However, resource allocation and locality are not addressed.

Virtual computers have been used to achieve portability by hiding information concerning the size and topology of a physical computer. Martin [26], Hudak [23], and Taylor [29] have investigated notations for specifying process mapping on a (potentially infinite) processing surface. In data-parallel languages [30, 18], data distribution is specified with respect to a virtual computer, as proposed here; however, hierarchies cannot be defined. While these systems succeed in decoupling mapping or data distribution from other aspects of a parallel algorithm, they do not permit resource allocation decisions to be isolated from module definitions. For example, it is not straightforward to invoke the same module on processor subsets that may be overlapping or of different sizes.

Some of the ideas developed in this paper have also been applied to CC++, extensions to C++ designed to support compositional parallel programming [8]. In CC++, virtual computers are represented explicitly, as arrays of pointers to *processor objects*. Mapping

is achieved by invoking a function in a processor object. Data distribution and embedding are not supported directly but can be specified as CC++ functions. CC++ mapping statements have semantic content (as functions can access data contained in a processor object); hence, mapping is not independent of other aspects of a design.

6 Conclusions

Research in sequential programming has demonstrated the value of modular design. In this paper, we have described programming language concepts that facilitate the application of modular design principles in parallel programming. We have also show how these concepts can be realized in practical parallel programming languages and reported experiences using these languages to solve large-scale programming problems. These experiences show that the familiar benefits of modular programming can be achieved in a parallel context. Modules implementing useful parallel algorithms can be reused. Existing applications can be integrated into larger systems without changes to their internal structure or implementation. Major structural changes concerning the mapping of computation and data to processors do not require changes to component modules.

A significant aspect of this work is that it is now possible to build truly modular and hence reusable parallel libraries. In a manner analogous to (but more flexible than) VLSI cell libraries, we can define libraries of parallel program components with specified internal logic and external interface but encapsulating no resource allocation or scheduling decisions. These components can be used unchanged in a wide variety of contexts or modified by the user to solve related problems. We are currently constructing libraries of this sort in several areas.

We have assumed in this paper that modules are implemented with the same technology: for example, PCN or Fortran M. However, modules can also be be constructed using different techniques (e.g., message-passing libraries or data-parallel languages) as long as implementations do not encapsulate mapping or scheduling decisions.

Acknowledgments

This work was supported by the Office of Scientific Computing, U.S. Department of Energy under Contract W-31-109-Eng-38. I am grateful to Robert Olson and Steven Tuecke for their work on the PCN and Fortran M compilers, and to Mani Chandy and Carl Kesselman for numerous discussions.

References

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

[2] V. Bala and S. Kipnis, "Process Groups: A mechanism for the coordination of and communication among processes in the Venus collective communication library," Preprint, IBM T. J. Watson Research Center, 1992.

[3] G. Booch, *Object-oriented Design*. Benjamin/Cummings, 1991.

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

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

[6] Cann, D. C., J. T. Feo, and T. M. DeBoni, "Sisal 1,2: High performance applicative computing," in *Proc. Symp. Parallel and Distributed Processing*, IEEE Press, 1990, pp. 612–616.

[7] K. M. Chandy and I. Foster, "A deterministic notation for cooperating processes," Preprint MCS-P346-0193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. 60439, 1993.

[8] K. M. Chandy and C. Kesselman, Compositional parallel programming in CC++, Technical Report, Caltech, 1992.

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

[10] I. Chern and I. Foster, "Design and parallel implementation of two methods for solving PDEs on the sphere," in *Parallel Computational Fluid Dynamics '91*, Elsevier Science Publishers B.V., 1991.

[11] A. Choudhary and R. Ponnusamy, "Parallel implementation and evaluation of a motion estimation system algorithm using several data decomposition strategies," *J. Parallel and Distributed Computing*, no. 14, pp. 50–65, 1992.

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

[13] J. Dongarra, R. van de Geijn, and D. Walker, "A look at scalable dense linear algebra libraries," in *Proc. Scalable High Performance Computing Conf.*, Williamsburg, Virginia, 1992.

[14] I. Foster and K. M. Chandy, "Fortran M: A language for modular parallel programming," Preprint MCS-P237-0992, Argonne National Laboratory, Argonne Ill. 60439, 1992.

[15] I. Foster, C. Kesselman, and S. Taylor, "Concurrency: Simple concepts and powerful tools," *The Computer Journal* vol. 33, no. 6, pp. 501–507, 1990.

[16] I. Foster, R. Olson, and S. Tuecke, "Productive parallel programming: The PCN approach," *Scientific Programming*, vol. 1, no. 1, 1992.

[17] I. Foster and S. Taylor, A compiler approach to scalable concurrent program design, *ACM Trans. Prog. Lang. Syst.* (to appear).

[18] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Technical Report TR90-141, Computer Science, Rice Univ., Houston, Texas, 1990.

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

[20] High Performance Fortran Forum, *High Performance Fortran Language Specification Version 0.4*, Technical Report, CITI/CRPC, Rice University, November 6, 1992.

[21] P. Henderson, *Functional Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1980.

[22] C. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[23] P. Hudak, "Para-functional programming," *IEEE Computer*, pp. 60–70, 1986.

[24] Inmos Limited, *occam 2 Reference Manual*. Prentice Hall, 1988.

[25] M. Lemke and D. Quinlan, "A parallel C++ array class library for architecture-independent development of structured grid applications," *ACM SIGPLAN Notices* vol. 28, no. 1, pp. 21–23, 1992.

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

[27] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM* vol. 15, no. 2, pp. 1053–1058, 1972.

[28] A. Skjellum and A. Leung, "Zipcode: A portable multicomputer communication library atop the Reactive Kernel," in *Proc. 5th Distributed Memory Concurrent Computing Conf.*, IEEE Press, 1990, pp. 767–776.

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

[30] Thinking Machines Corporation, *CM Fortran Reference Manual*, Thinking Machines, Cambridge, Mass., 1989.

[31] N. Wirth, "Program development by stepwise refinement," *Commun. ACM* vol. 14, pp. 221–227, 1971.