Language Constructs and Runtime Systems for Compositional Parallel Programming

Ian Foster[1] and Carl Kesselman[2]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.
[2] Beckman Institute, California Institute of Technology, Pasadena, CA 91125, U.S.A.

Abstract. In task-parallel programs, diverse activities can take place concurrently, and communication and synchronization patterns are complex and not easily predictable. Previous work has identified *compositionality* as an important design principle for task-parallel programs. In this paper, we discuss alternative approaches to the realization of this principle. We first provide a review and critical analysis of Strand, an early compositional programming language. We examine the strengths of the Strand approach and also its weaknesses, which we attribute primarily to the use of a specialized language. Then, we present an alternative programming language framework that overcomes these weaknesses. This framework uses simple extensions to existing sequential languages (C++ and Fortran) and a common runtime system to provide a basis for the construction of large, task-parallel programs. We also discuss the runtime system techniques required to support these languages on parallel and distributed computer systems.

1 Introduction

Parallel programming is widely regarded as difficult: more difficult than sequential programming, and perhaps (at least this is our view) more difficult than it needs to be. In addition to the normal programming concerns, the parallel programmer has to deal with the added complexity brought about by multiple threads of control: managing their creation and destruction, and orchestrating their interactions via synchronization and communication. Parallel programs must also manage a richer set of resources than sequential programs, controlling for example the mapping and scheduling of computation onto multiple processors.

As in sequential programming, complexity in program development can be managed by providing appropriate programming language constructs. Language constructs can help both by supporting encapsulation so as to prevent unwanted interactions between program components, and by providing higher-level abstractions which leverage programmer effort by allowing compilers to handle mundane, error-prone aspects of parallel program implementation. For example, the various languages that have been developed to support data-parallel programming achieve both these goals, albeit for a restricted class of programs [7,

9, 17]. Data-parallel programs exploit the parallelism inherent in applying the same operation to all or most elements of large data structures. Data-parallel languages avoid unwanted interactions by enforcing sequential semantics. They use data distribution statements to provide a high-level, abstract syntax for specifying data placement, freeing the programmer from the labor of partitioning computation and translating between global and local addresses.

Our research goal is to develop language constructs and associated tools to support the more general class of *task-parallel* applications, in which multiple unrelated activities can take place concurrently. Task parallelism arises in time-dependent problems such as discrete-event simulation, in irregular problems such as those involving sparse matrices, and in multidisciplinary simulations coupling multiple, possibly data-parallel, computations. The challenge when developing language constructs for task-parallel programming is to provide the modularity and abstraction needed for ease of programming while maintaining the generality needed to support arbitrary parallel computations.

Compositionality has been proposed as a design principle for task-parallel programs. A compositional programming system is one in which properties of program components are preserved when those components are composed in parallel with other program components. That is, the behavior of the whole is a logical combination of the behavior of the parts. Compositionality can simplify program development by allowing program components to be developed and tested in isolation and then reused in any environment.

In this paper, we describe various language constructs and runtime system techniques that have been proposed to support compositionality. We first use the example of Strand to show how the basic ideas of compositional programming can be supported using a small number of simple concepts, namely monotone operations on shared objects, a uniform addressing mechanism, and parallel composition. Then, we show how these same concepts have been incorporated into the Compositional C++ and Fortran M extensions to the sequential languages C++ and Fortran, hence providing a more flexible and accessible implementation of the ideas. Finally, we examine the runtime system techniques used to support these various compositional programming languages on parallel and distributed computer systems.

2 New Languages: Strand and PCN

One particularly elegant and satisfying approach to compositional task-parallel programming is to define a simple language that provides just the essential elements required to support this programming style. This language can be used both as a language in its own right and as a *coordination language,* providing a parallel superstructure for existing sequential code. These dual roles require a simple, uniform, highly-parallel programming system in which:

- the structure of the computation, the number of concurrently-executing threads of control, and the placement of these threads can vary dynamically during program execution,

- communication and synchronization operations are introduced into a program via high-level abstractions which can be efficiently implemented by the language compiler,
- patterns of communication can change dynamically,
- the functional behavior of parallel program modules is independent of the scheduling or processor allocation strategy used,
- arbitrary parallel modules can be combined and will function correctly, and
- modules written in other languages can be incorporated.

These goals motivate the design both of Strand and of the CC++ and Fortran M languages described below.

2.1 Strand Design

The Strand language integrated ideas from earlier work in parallel logic programming [8], dataflow computing [1], and imperative programming [15] to provide a simple task-parallel programming language based on four related ideas:

- single assignment variables,
- a global, shared namespace,
- parallel composition as the only method of program composition, and
- a foreign language interface.

Single-assignment variables provide a unified mechanism for both synchronization and communication. All variables in Strand follow the single-assignment rule [1]: a variable is set at most once and subsequently cannot change. Any attempt by a program component to read a variable before it has been assigned a value will cause the program component to block. All synchronization operations are implemented via reading and writing these variables. New variables can be introduced by writing recursive procedure definitions.

Strand variables also define a global namespace. A variable can refer to any object in the computation, even another variable. The location of the variable or object being referenced does not matter. Thus, Strand does not require explicit communication operations: processes can communicate simply by reading and writing shared variables.

Unlike most programming languages which support only the sequential composition of program components, Strand supports only parallel composition. A parallel composition of program components executes as a *concurrent interleaving* of the components, with execution order constrained only by availability of data, as determined by the single-assignment rule.

The combination of single-assignment variables, a global namespace, and parallel composition means that the behavior of a Strand program is invariant to the placement and scheduling of computations. One consequence of this invariance is that Strand programs are compositional: a program component will function correctly in any environment. Another consequence is that the specification of the location of a computation is orthogonal to the specification of

the computation. To exploit these features, Strand provides a mapping operator which allows the programmer to control the placement of a computation on a parallel computer.

By allowing modules written in sequential languages to be integrated into Strand computations, the foreign language interface supports the use of Strand as a coordination language. Sequential modules that are to be integrated in this way must implement pure functions. The interface supports communication between foreign modules and Strand by providing routines that allow foreign language modules to access Strand variables passed as arguments.

2.2 Strand Critique

Unlike many parallel programming systems developed in a research environment, Strand has been used extensively for application development in areas as diverse as computational biology, discrete event simulation, telephone exchange control, automated theorem proving, and weather modeling. This work provides a broad base of practical experience on which we can draw when evaluating the strengths and weaknesses of the Strand approach. Analysis of this experience indicates three particular strengths of the Strand constructs:

- The use of parallel composition and a high-level, uniform communication abstraction simplifies development of task-parallel applications featuring dynamic creation and deletion of threads, complex scheduling algorithms, and dynamic communication patterns. Complex distributed algorithms can often be expressed in a few lines of code using Strand constructs.
- Parallel composition and single assignment variables also enforce and expose the benefits of a compositional programming model. This eases program development, testing, and debugging, and the reuse of program components.
- The recursively-defined data structures and rule-based syntax that Strand borrows from logic programming are useful when implementing symbolic applications, for example in computational biology.

This same analysis also reveals four significant weaknesses which limit the utility of the Strand system, particularly for larger scientific and engineering applications.

- While the use of a separate coordination language for parallel computation is conceptually economical, it is not universally popular. Writing even a simple program requires that a programmer learn a completely new language, and the logic-based syntax is unfamiliar to many.
- The foreign language interface is often too restrictive for programmers intent on reusing existing sequential code in a parallel framework. In particular, it is difficult to convert sequential code into single program/multiple data (SPMD) libraries, as this typically requires the ability to embed parallel constructs in existing sequential code, something that Strand does not support. As a consequence, combining existing program modules with Strand can require significant restructuring of those modules.

- The Strand abstractions provide little assistance to the programmer intent on applying domain decomposition techniques to regular data structures. In these applications, the principal difficulties facing the programmer are not thread management or scheduling, but translating between local and global addresses, problems which have been addressed in data-parallel languages.
- The use of a new language means that program development tools such as debuggers and execution profilers have to be developed from scratch; it also hinders the application of existing sequential development tools to sequential code modules.

2.3 Program Composition Notation

In a related research project stimulated in part by Strand and the Unity system [5], Chandy and Taylor investigated the feasibility of integrating single-assignment variables and concurrent composition with conventional imperative programming. This led to the development of Program Composition Notation (PCN) [6]. Like Strand, PCN provides a parallel programming model based on single-assignment variables, a global address space, and concurrent composition. Its major contribution is to show how this model can be integrated with the conventional world of "multiple-assignment" variables and sequential composition. This produces a programming language that is both more complex and more expressive than Strand. In retrospect, however, it appears that while PCN addressed some Strand deficiencies, these were probably not the important ones. PCN still suffers from the four essential weaknesses identified in the preceding subsection.

3 Language Extensions: CC++ and FM

The weaknesses of the Strand approach appear to derive in large part from the use of a new language to express parallel computation. This observation suggests an alternative approach to compositional programming in which traditional languages, such as C++ and Fortran, are extended to provide the central strengths of Strand: compositionality and high-level specification of communication and synchronization. (Support for symbolic applications appears less fundamental.) In principle, these language extensions can address Strand's weaknesses by providing a common framework for parallel and sequential programming and simplifying the integration of existing code. It would also be desirable for these extensions to support the specification of data-parallel computations.

The design of a language extension that supports compositional parallel programming requires some analysis of what makes a programming language "compositional." Compositionality in Strand is achieved using three mechanisms. Single-assignment variables provide both an interaction mechanism based on monotonic operations on shared state, and a uniform address space; parallel composition provides a concurrent interleaving. (State changes on single-assignment variables are monotonic in that the value of a variable cannot be changed once

written [4].) Together, these mechanisms ensure that neither the order in which program components execute, nor the location of this execution, affect the result computed. Other mechanisms can provide the same capabilities. For example, nonblocking send and blocking receive operations on a virtual channel data type are also monotonic, and could form the basis for a compositional programming language.

These various consideration lead to the following additional design goals for compositional programming languages.

- A language should define just a small set of new language constructs; these new constructs should be compatible with the basic concepts of the sequential base language.
- The new constructs should provide monotonic operations on shared program state, so as to support compositionality.
- The new constructs should be easily embedded in existing sequential code, so as to facilitate the development of parallel SPMD libraries.
- The language should retain support for flexible communication and synchronization structures, and a data-driven execution model.
- The language should support interoperability, both with other compositional languages and with data-parallel languages.

These design goals have motivated the development of the parallel programming framework illustrated in Figure 1. Compositional programming is supported by small sets of extensions to C++ and Fortran 77 called Compositional C++ (CC++) and Fortran M (FM), respectively. A common runtime system, Nexus, is used by the compilers developed for both languages, facilitating interoperability. We describe the language extensions in the following.

3.1 Compositional C++

Compositional C++ [3], or CC++, is a general-purpose parallel programming language based on C++. CC++ defines six new keywords, designed to provide an essential set of capabilities from which many different types of parallel programs could be constructed. For example, we can write CC++ libraries that implement parallel programming paradigms such as synchronous virtual channels, actors, data flow, and concurrent aggregates [16].

CC++ is not a purely compositional programming language. In order to guarantee compositionality, unacceptable restrictions would have to be made on the C++ constructs that are available in CC++. Thus, in designing CC++, our approach was to provide constructs that would enable rather than guarantee the construction of compositional modules. In most instances, compositional modules can be obtained by following simple programming conventions [4].

CC++ provides three different mechanisms for creating threads of control: the parallel block, the parallel loop, and spawned functions. The first two have a parbegin/parend semantics, while the spawned function creates an independent thread.

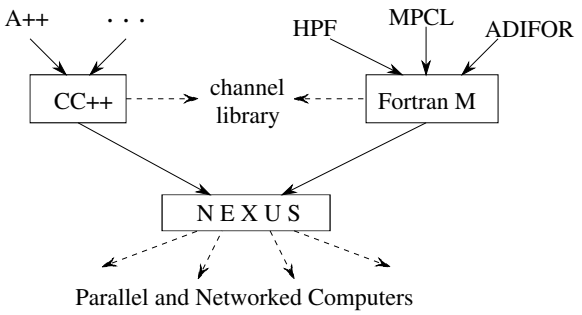

Fig. 1. A task-parallel programming framework based on language extensions (CC++ and FM), a common runtime system, and libraries implementing common abstractions such as virtual channels. The language extensions can be used to construct libraries supporting a range of programming models, including message passing (MPCL), data parallelism (A++, HPF), and parallelism extracted automatically from derivative computations (ADIFOR).

CC++ borrows the idea of a single-assignment variable from Strand. In CC++, a single-assignment variable is called a synchronization, or `sync` variable, and is distinguished by the type modifier `sync`. A CC++ program can contain both `sync` and regular C++ variables. Programs that contain only `sync` variables will be compositional. To support the development of compositional programs containing regular C++ variables, CC++ introduces atomic functions. Within an instance of a given C++ class, only one atomic function is allowed to execute at a time. The operations specified in the body of an atomic function execute without interference. Thus, an atomic function is like a monitor [14]. If all accesses to a shared C++ variable takes place within the body of an atomic function, than the resulting program is compositional.

The remaining aspects of C++ deal with the allocation of computation to processors and the methods used to access data on different processors. The central issue is what happens to global and static data in a CC++ program. Our approach is to introduce a structure called a *processor object*. A processor object is a virtual processor, containing a private copy of all global and static data. Like other C++ objects, a processor object has a type declared by a class definition, encapsulates functions and data, and can be dynamically created and destroyed. Each instance of a processor object contains an address space from which regular objects can be allocated. As in Strand, the functional behavior of the program is independent of where the processor objects are placed.

CC++ distinguishes between inter-processor object and intra-processor object references: a pointer that can refer to an object in another processor object must be declared to be `global`. Global pointers provide CC++ with both a global name space and a two-level locality model that can be manipulated di-

rectly by a program. A global pointer can be dereferenced like any other C++ pointer. However, dereferencing a global pointer causes an operation to take place in the processor object referenced by that global pointer. Thus in CC++, communication abstractions are provided by operations on `global` pointers, while synchronization abstractions are provided by `sync` pointers.

In summary, CC++ integrates parallel composition with sequential execution. It uses global pointers to provide a uniform global address space and `sync` variables and atomic functions to implement compositional interactions between program components.

3.2 Fortran M

Fortran M (FM) [11] is a small set of extensions to Fortran 77 for task-parallel programming. FM is designed to support both the modular construction of large parallel programs and the development of libraries implementing other programming paradigms. For example, FM libraries have been used to integrate SPMD message-passing computations and data-parallel HPF programs into a task-parallel framework [10], and to implement distributed data structures. Although simple, the FM extensions provide the essential mechanisms required for compositional programming. Program components can encapsulate arbitrary concurrent computations and can be reused in any environment.

Concepts such as pointers and dynamic memory allocation are foreign to Fortran 77. Hence, the FM design bases its communication and synchronization constructs on an existing concept: file I/O. FM programs can dynamically create and destroy *processes*, single-reader/single-writer virtual files (*channels*), and multiple-writer, single-reader virtual files (*mergers*). Processes can encapsulate state and communicate by sending and receiving messages on channels and mergers; references to channels, called *ports*, can be passed as arguments or transferred between processes in messages, providing a restricted global address space.

FM processes are created by process block and process do-loop constructs with parbegin/parend semantics. Arguments passed to a process are copied in on call and back on return; common blocks are local to each process. A channel is a typed, first-in/first-out message queue with a single sender and a single receiver; the merger is similar but allows for multiple senders. FM constructs allow the programmer to control process placement by specifying the mapping of processes to *virtual computers*: arrays of virtual processors. Mapping decisions do not effect program semantics. A novel aspect of the FM extensions is that even complex programs can be guaranteed to be deterministic [2].

In summary, FM integrates parallel composition with sequential execution. It uses channels both to provide a uniform global address space and to implement compositional interactions between program components.

4 Runtime Systems

Compilers for parallel languages rely on the existence of a runtime system. The runtime system defines the compiler's view of a parallel computer: how computational resources are allocated and controlled and how parallel components of a program interact, communicate and synchronize with one another.

Runtime systems for data-parallel languages are concerned primarily with the efficient realization of collective operations in which all processors communicate at the same time, in a structured fashion. Runtime systems for compositional task-parallel languages such as Strand, PCN, CC++, and FM are more complex, as they must support:

- multiple, concurrent threads of control;
- a data-driven execution model;
- dynamic allocation and deletion of threads, shared variables, and other resources;
- a global address space, whether based on single-assignment variables, global pointers, or channels;
- asynchronous access to remote resources; and
- efficient sequential execution.

In addition, task-parallel programs are often required to execute in heterogeneous environments such as networked collections of multiprocessors.

4.1 Strand and PCN: Interpreter-based Runtime Systems

The implementation technology used to support the requirements just listed depends in part on what aspect of program performance is to be optimized. The goal of Strand and PCN implementation efforts was to provide highly efficient support for concurrent composition and lightweight processes. These goals were met using a interpreter- and heap-based runtime system. (Similar techniques have been used in abstract machines for Id and other functional languages [18].) Programs are compiled to the instruction set of an abstract machine. A portable interpreter for this abstract machine handles the data-driven scheduling of lightweight processes. References to shared variables are tagged, and a runtime test is used to determine when a read or write operation is applied to an off-processor reference. The operation is then implemented as a call to a machine-dependent communication library. This design allows the efficient execution of programs that create thousands of processors and switch frequently between threads of control. A disadvantage is that the use of a heap-based storage system and an interpreter hinders efficient execution of sequential code.

4.2 CC++ and FM: The Nexus Runtime System

An alternative approach to runtime system design is to focus on enabling efficient execution of sequential code. This implies an execution model based on a

"heap of stacks" rather than a simple heap, so that code generated by optimizing sequential language compilers can be used unchanged. Executable code generated by these compilers is linked with a runtime library implementing the basic abstractions needed for task-parallel execution, using existing message-passing and thread systems when possible. This approach is taken in the runtime system called Nexus that is used by both CC++ and FM compilers.

Nexus Interface. Nexus provides five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests [12]. Associated services provide direct support for light-weight threading, address space management, communication, and synchronization. A computation consists of a set of *threads*, each executing in an address space called a *context*. An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context. It can also generate asynchronous *remote service requests*, which invoke procedures in other contexts. Nodes, contexts, threads, and global pointers can be created and destroyed during program execution. The abstractions have the following properties.

- The node abstraction supports dynamic acquisition and release of potentially heterogeneous processor resources.
- The context abstraction supports the creation of multiple address spaces in a single node. (This corresponds to the CC++ processor object and the FM process.)
- The thread abstraction supports the creation of multiple threads of control.
- The global pointer supports the implementation of a uniform global address space. (This corresponds to the CC++ global pointer and is used to implement the FM channel.)
- The remote service request provides access to remote resources.

Nexus as a Compiler Target. The translation from CC++ and FM constructs to the Nexus abstractions is fairly straightforward. For example, an FM process is implemented as a thread executing in a dedicated Nexus context, with the context's data segments used to hold process state. This context must be allocated by the FM compiler prior to creating the thread, and deallocated upon process termination. As an optimization, processes without state can be implemented as threads in a preexisting context containing the appropriate code. This optimization can reduce process creation costs and, in some systems, scheduling costs, and is important for fine-grained applications. A channel is implemented as a message queue data structure maintained in the context of the receiving process; an outport is implemented as a data structure containing a Nexus global pointer to the channel data structure. A send operation is compiled to code which packs message data into a buffer and invokes a remote service request to a compiler-generated handler which enqueues the message onto the channel. A receive operation is compiled to code which unpacks a pending message into variables or suspends on a condition variable in the channel data structure if no messages are pending.

Heterogeneity. A novel aspect of the Nexus design is that it supports heterogeneity at multiple levels, allowing a single computation to utilize different programming languages, executables, processors, and network protocols. In order to support heterogeneity, the Nexus implementation encapsulates thread and communication functions in thread and protocol modules, respectively, that implement a standard interface to low-level mechanisms. Current thread modules include POSIX threads, DCE threads, C threads, and Solaris threads. Current protocol modules include local (intracontext) communication, TCP sockets, PVM, IBM's EUI message-passing library, and Intel NX message-passing. Protocol modules for MPI, SVR4 shared memory, Fiber Channel, AAL-5 (ATM Adaptation Layer 5) for Asynchronous Transfer Mode (ATM), and remote memory operations such as the get and put operations on the Cray T3D are planned or under development. When communicating between contexts on a global pointer, Nexus uses the most efficient protocol available to the two contexts.

Interoperability. Nexus provides a basis for interoperability between diverse parallel languages. Interoperability involves a range of both mundane and complex issues relating to data structures, subroutine calling conventions, and the like. Our focus is on those issues that are particular to parallel computing. Because CC++ and FM are both implemented using Nexus facilities, parallel structures in the two languages can both coexist and interact. For example, an FM program can invoke a CC++ program, specifying the contexts in which it is to execute and passing as arguments an array of Nexus global pointers representing the inports or outports of channels. The CC++ program can then apply send or receive functions to these global pointers to transfer data between contexts executing FM code and contexts executing CC++ code.

5 Conclusions

The goal of compositional programming is to simplify parallel program development by allowing complex programs to be developed from simpler components. In this paper, we have discussed a variety of approaches to the realization of this goal. A review of Strand, an early compositional programming language, indicates both the advantages of a compositional approach and the disadvantages of using a specialized language. A description of Compositional C++ and Fortran M shows how the advantages of compositionality can be exploited in more familiar settings by extending existing languages with appropriate constructs. Finally, a description of the runtime support required for compositional programming languages indicates that a relatively small set of simple mechanisms suffices to support complex task-parallel computations on parallel and distributed computer systems.

6 Acknowledgments

The Strand system was developed with Steve Taylor. We gratefully acknowledge the many contributions of Mani Chandy to the work on CC++ and FM, and the outstanding implementation efforts of John Garnett, Tal Lancaster, Robert Olson, James Patton, Mei Su, Steven Tuecke, and Ming Xu. This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615.

References

1. Ackerman, W.: Data flow Languages. Computer **15**(2), (1982), 15–25
2. Chandy, K.M., Foster, I.: A deterministic notation for cooperating processes. Preprint, Argonne National Laboratory (1993)
3. Chandy, K.M., Kesselman, C.: CC++: A declarative concurrent object-oriented programming notation. Research Directions in Object Oriented Programming, MIT Press (1993)
4. Chandy, K.M., Kesselman, C.: The derivation of compositional programs. Proc. 1992 Joint Intl Conf. and Symp. on Logic Programming, MIT Press (1992)
5. Chandy, K.M., Misra, J.: Parallel Program Design. Addison-Wesley (1988)
6. Chandy, K.M., Taylor, S.: An Introduction to Parallel Programming. Jones and Bartlett (1992)
7. Chapman, B., Mehrotra, P., Zima, H.: Programming in Vienna Fortran. Scientific Programming **1**(1) (1992) 31–50
8. Clark, K., Gregory, S.: A relational language for parallel programming. Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architectures (1981) 171–178
9. Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., Wu, M.: Fortran D language specification. Rice University TR90-141 (1990)
10. Foster, I., Avalani, B., Choudhary, A., Xu, M., A compilation system that integrates High Performance Fortran and Fortran M. Proc. 1994 Scalable High Performance Computing Conf., IEEE Computer Science Press (1994) 293–300
11. Foster, I., Chandy, K.M.: Fortran M: A language for modular parallel programming. J. Parallel and Distributed Computing (to appear)
12. Foster, I., Kesselman, C., Tuecke, S.: Nexus: Runtime support for task-parallel programming languages. Preprint, Argonne National Laboratory (1994)
13. Foster, I., Taylor, S.: Strand: New Concepts in Parallel Programming. Prentice Hall (1989)
14. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM **17**(10) (1974) 549–557
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1984)
16. Kesselman, C.: Implementing parallel programming paradigms in CC++. Proc. Workshop on Parallel Environments and Tools, SIAM (to appear)
17. Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M.: The High Performance Fortran Handbook. MIT Press (1994)
18. von Eicken, T., Culler, D., Goldstein, S., Schauser, K.: TAM — A compiler controlled threaded abstract machine. J. Parallel and Distributed Computing (1992)