Parallel Programming with Algorithmic Motifs

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

Abstract

We describe an approach to parallel program development based on the use of algorithmic motifs: useful parallel program structures, that users can combine with application-specific routines to construct parallel programs. Key features of our proposal are the use of a high level concurrent programming language and automated source-to-source transformations to implement motifs. These features support the construction of new motifs from old by both modification and composition. This in turn encourages an exploratory programming style, in which programmers experiment to develop good parallel implementations of applications. The use of a high level language encourages the user to view libraries implementing motifs as archives of expertise that can be consulted, modified, and extended when developing parallel programs. We illustrate the application of the approach using examples.

Keywords: algorithmic motif; source-to-source transformation; high level language; composition; parallel algorithm

1 Introduction

Large MIMD computers pose a serious software engineering problem: the flexibility offered by multiple instruction streams is hard to exploit because of the complexity of specifying and coordinating their execution. Hence, MIMD computers are frequently used in a semi-SIMD fashion [9].

A promising solution to this problem is to provide libraries of algorithmic motifs that encapsulate useful parallel programming abstractions. These libraries provide implementations of parallel algorithms, in which certain components are unspecified; a user slots in application-specific procedures to obtain a specific parallel program. For example, a tree reduction motif may coordinate the application of an unspecified operator at each node in a binary tree; it can be used to evaluate any associative operation, simply by specifying the missing operator. Ideally, the use of motifs permits the labor of designing and implementing parallel algorithms to be amortized over many programs of a similar class.

This general idea underlies several existing or proposed parallel programming tools. For example, the DIME package developed at Caltech is intended to support the development of finite element codes. A user provides code defining the actions to be performed at a single node in a triangular mesh; the DIME system maintains the mesh data structure on a parallel computer and handles communication and load balancing [11]. The Argonne monitor macros and Schedule packages support load-balancing on shared-memory computers. A user provides a set of procedures and defines data dependencies between them; the system schedules their execution appropriately [2, 5]. Or-parallel Prologs provide an implementation of parallel search. The user provides logic clauses that specify a search problem and the system explores the corresponding search tree [10]. Cole proposes "algorithmic skeletons" to support several classes of parallel program, and provides complexity analyses for various problems on a two-dimensional grid architecture [4].

These systems permit programmers to reuse the parallel code implementing the motifs "as is" when developing new applications. Unfortunately, these systems typically support reuse only "as is". Yet two other forms of reuse are helpful when, as is often the case, no pre-existing motif quite fits the bill. Reuse through modification permits a user to define variants of existing motifs that provide modified functionality. For example, a scheduler motif might be adapted to the demands of a highly parallel computer by introducing additional levels in its manager/worker hierarchy. Reuse through composition permits a user to construct a new motif by combining new code with implementations of more primitive abstractions. Examples of composition will be presented in later sections.

We describe in this paper an approach to the design and implementation of parallel algorithms that enables reuse by both modification and composition. Key features of the approach are the use of a high level language to express concurrent algorithms and the use of automated source-to-source transformations to implement motifs. The use of a high level language facilitates program development, maintenance, and reuse. Transformations are used to restructure applications to fit the requirements of particular motifs; this approach facilitates experimentation with alternative motifs in a single application. Transformations are also the key to motif composition. In addition the use of automatically applied transformations can speed the parallel program development process.
2 The Approach

We now introduce the two ideas that distinguish our approach, namely, high level language and source-to-source transformations.

2.1 High Level Language

The parallel programming systems referred to in the introduction implement motifs in low level languages such as Fortran or C. These languages are to a large extent write-only notations: They can be used to provide efficient implementations of algorithms, but can be read only with difficulty. Hence, they do not provide a good medium for archiving expertise.

The use of a high level language is central to our approach. It permits concise and readable specifications of complex algorithms; thus, libraries implementing motifs become archives of expertise that can be consulted and adapted for new purposes.

High level languages are often rejected because of perceived inefficiency. However, efficient execution of high level components is not critical to our approach, for two reasons. First, we assume a multilingual approach to parallel programming, in which low level, computationally-intensive components of applications are implemented in low level languages [7]. The high level language is used primarily to construct parallel programs from these sequential components. Second, although a motif implementation (which provides scheduling routines, etc.) may encapsulate significant complexity, it is rare that significant time is spent executing its routines. If these routines do form a bottleneck, then the motif in question is probably not scalable, and an alternative approach needs to be explored.

A high level language for concurrent programming should permit concise and elegant specifications of process management, communication, and synchronization. It should encourage a modular approach to program design. Finally, it should be possible to achieve reasonably efficient implementations on a variety of parallel architectures.

We have used the concurrent programming language Strand [8] as the primary programming language in our work to date; other notations such as PCN [3] could also be employed. Although it is not reasonable to expect one language to be ideal for all purposes, we have been pleasantly surprised by Strand's versatility; furthermore, it is available on shared-memory computers, hypercubes, mesh machines, transputer surfaces, and other parallel computers.

A Strand program is a collection of guarded rules with the form

\[ H : \{-\} G_1, \ldots, G_m \mid B_1, B_2, \ldots, B_n \quad m, n > 0 \]

where \( H \) is the rule head, \( \{-\} \) denotes implication, the \( G_i \)s are the rule guard, \( \mid \) is the commit operator, and the \( B_j \)s are the rule body. Rules in which the heads have the same name (say \( p \)) and number of arguments (say \( k \)) are grouped into a process definition; this will sometimes be referred to as \( p/k \). The head and guard of a rule define conditions that must be satisfied before a process can execute (reduce); the body specifies new processes to replace the process if these conditions are satisfied. The program in Figure 1 illustrates the notation.

\[
\begin{align*}
\text{go}(N) & : - \text{producer}(N,Xs,\text{sync}), \text{consumer}(Xs). & \% R1 \\
\text{producer}(N,Xs,\text{sync}) & : - & \% R2 \\
& \quad N > 0 \mid \\
& \quad Xs ::= [X | Xs1], N1 \equiv N - 1, \text{producer}(N1,Xs1,X). & \% R3 \\
\text{producer}(0,Xs,:) & : - Xs ::= []. & \% R3 \\
\text{consumer}([X | Xs]) & : - X ::= \text{sync}, \text{consumer}(Xs). & \% R4 \\
\text{consumer}([]) & . & \% R5
\end{align*}
\]

Figure 1: Example Strand Program

The notation \([\text{Head} \mid \text{Tail}]\) denotes a list structure with a Head and a Tail. An alternative tuple notation can also be used to denote structured data: A term \([T1, \ldots, Tn]\) denotes a tuple with the terms \(T1, \ldots, Tn\) as arguments. Strings beginning with uppercase letters denote variables, while those with lowercase letters denote constants. The assign primitive, \("=\)\), is used to assign values to variables, which have the single assignment property: The value of a variable is initially undefined and, once provided, cannot be modified. Attempts to assign to a variable that has a value are signaled as run-time errors.

The state of a computation is represented by a pool of lightweight processes. Execution proceeds by repeatedly selecting and attempting to reduce processes in this pool. For example, if an initial process pool contains the single process \text{go}(4), this can be immediately reduced with rule R1 to create two new processes, producer and consumer.

Processes communicate by reading and writing shared variables. The example program illustrates a common communication structure, in which a producer incrementally instantiates a shared variable to a list structure (R2), hence communicating a stream of values (the list elements) to a consumer (R4). In the example, the values communicated are themselves variables (\([X, X', \ldots\] ). The consumer process acknowledges these "communications" by assigning each variable that it receives the value \text{sync} (R4).

The availability of data serves as the synchronization mechanism. Conditions expressed by non-variable terms in a rule head define dataflow constraints: A rule cannot be used to reduce a process until a process's arguments match its own. In consequence, producer and consumer communicate synchronously. The consumer process waits
for a communication from producer; the recursive call to producer waits for the variable it has communicated to be assigned the value sync. After sending 4 messages, the two processes terminate.

2.2 Transformations

The implementation of a motif comprises both a source-to-source transformation and a library program. Hence, we often denote a motif by a pair \( \{T, L\} \), where \( T \) denotes a transformation and \( L \) a library. A motif is applied to an application program to yield a new program. This process is achieved in two stages. First, the transformation is applied, yielding a modified application program. Second, the library code is linked with the modified application. Thus, if \( A \) is an application program and \( M = \{T, L\} \) is a motif, then the application of \( M \) to \( A \) yields a new program \( A' = M(A) = T(A) \cup L \).

Source-to-source transformations are used to massage application code into forms required by libraries or other motifs. Thus, a programmer is able to express applications in a form that is convenient to him, without foregoeing the opportunity to experiment with different motifs when developing a parallel program. In contrast, purely template-based systems require that the programmer restructure an application to suit the requirements of a particular motif, typically by embedding calls to library routines. This requirement hinders experimentations, as the application must be manually restructured to utilize alternative decompositions.

To better understand the use of transformations, consider a scheduler motif concerned with dynamically allocating tasks to idle processors. It is easy to define a library program which creates a set of worker processes and distributes data structures representing tasks to idle workers. However, it would be inconvenient if programmers had to embed explicit calls to this scheduler in their programs and manually construct data structures representing tasks. Fortunately, these functions can be incorporated automatically by an application-independent transformation. The programmer only needs to supply pragma specifying tasks and data dependencies [6].

A second advantage of transformations is that they permit the development of new motifs by composition. As the output of a motif is itself a program, it is often possible to implement new motifs by providing transformations and libraries to be composed with the transformations and libraries defining existing motifs. For example, an existing motif \( M_1 = \{T_1, L_1\} \) may provide part, but not all, of the functionality required in some application. Rather than implementing a new motif \( M \) from scratch, we may be able to implement a simpler motif \( M_2 = \{T_2, L_2\} \) that, when composed with \( M_1 \), provides the required functionality. That is, \( M = M_2 \circ M_1 \), and \( M(A) = M_2(M_1(A)) = T_2(T_1(A) \cup L_1) \cup L_2 \).

As described in Chapter 9 of [8], source-to-source transformations can be implemented easily in Strand because of the language's simple, recursively-defined structure. Programs are represented as structured terms and transformations as programs that manipulate these terms.

3 Case Study

We illustrate the application of our techniques by describing the development of two different implementations for a tree reduction motif. We motivate this material using a concrete application taken from computational biology, namely, the generation of alignments of multiple sequences of RNA from different but related organisms. This application first generates a binary "philogenetic tree", in which subtrees represent clusters of more closely related organisms. Reduction of this tree using an "align-node" function produces the desired alignment. We assume that the philogenetic tree and align-node functions are given, and we focus on the tree reduction component of the problem.

The node evaluation function required to complete implementation of the multiple sequence alignment algorithm is still being implemented. When it is complete, the motifs developed in this section will permit us to develop parallel implementations based on two different random mapping strategies.

3.1 Towards a Tree Reduction Motif

Although in our application the operation to be performed at each node is complex, we can illustrate the issues involved in tree reduction using a simpler example. Consider the following tree in which each non-leaf node represents a multiplication or addition operation.

![Tree Diagram]

Reduction of this tree corresponds to evaluation of the expression \((3^2)+((2+1))\) and yields the value 24 at the root.

Many different techniques can be used to reduce a tree on a parallel computer. A static partition of the tree is probably ideal in the simple arithmetic example. In contrast, our biology application requires a more dynamic algorithm, as the time required at each node is non-uniform and cannot easily be predicted. We develop here a dynamic algorithm that satisfies this requirement. This algorithm employs a divide-and-conquer strategy to reduce a tree. Starting at the root, it recursively creates processes to reduce left and right subtrees and to compute a value using the values computed for the subtrees. At each node, one of the subtree evaluations is spawned to another, randomly-selected processor. This random
mapping should produce a reasonably balanced load if \(|\|\text{Nodes}\| > |\|\text{Processors}\|)\).

A complete Strand implementation of the dynamic algorithm is presented in Figures 2 and 3. The program is divided into four distinct parts. Part A defines the evaluation function `eval` that computes the value at a node. Part B reduces a tree using the divide-and-conquer strategy; at each node, it proceeds to further evaluate one subtree and sends a reduce message containing the other subtree to another, randomly-selected processor. Two low-level primitives, `rand.num` and `distribute`, are used to achieve this communication. Part C is concerned with the implementation of random process mapping; it defines a server process that dispatches reduce processes in response to reduce messages. Finally, Part D (Figure 3) creates a network of servers and establishes connectivity between servers. We will say a little bit more about Figure 3 later. However, we emphasize that the reader is not expected to understand the workings of the code in detail. The precise details of how the server network is created are not relevant to this discussion. What is important to note is that a relatively complex process network is specified in only a page of code. Clearly, code of this brevity cannot be too difficult to understand, maintain, and modify.

% Part A.
\[
\begin{align*}
\text{eval}(\text{"+/",L,R,Value}) & \rightarrow \text{Value} = L + R. \\
\text{eval}(\text{"*",L,R,Value}) & \rightarrow \text{Value} = L \times R. \\
\text{eval}(\text{"---",Value}) & \rightarrow \text{Value} = \text{integer}(1) \mid \text{Value} = 1.
\end{align*}
\]

% Part B.
\[
\begin{align*}
\text{reduce}(&\text{tree}(V,L,R,\text{Value},DT) \rightarrow \\
& \quad \text{length}(DT,N), \text{rand.num}(N,O), \\
& \quad \text{distribute}(O,DT,\text{reduce}(R,\text{RV}), \\
& \quad \text{reduce}(L,\text{LV},DT), \text{eval}(V,\text{LV},\text{RV},\text{Value}), \\
& \quad \text{reduce}(\text{leaf}(L),\text{Value},..) \rightarrow \text{Value} = L.)
\end{align*}
\]

% Part C.
\[
\begin{align*}
\text{server}(&\text{reduce}(T,V) \mid \text{ln},DT) \rightarrow \\
& \quad \text{reduce}(T,V,DT), \text{server}(\text{ln},DT). \\
\text{server}(\{..\})
\end{align*}
\]

% Part D. (See Figure 3)

Figure 2: Tree Reduction Algorithm

Several comments can be made about this program. The first is that although the program as a whole can only be used to reduce arithmetic expressions, parts B-D define a completely application-independent tree reduction motif that can be used to reduce trees using any associative operator. The programmer can use this program to implement a range of tree reduction algorithms, simply by providing alternative definitions for the `eval` function. For example, defining `eval` to invoke the "align-node" function provides a solution to the sequence alignment problem.

\[
\begin{align*}
\text{create}(N,\text{Msg}) & \rightarrow \text{create1}(\text{Msg},N,[],[]). \\
\text{create1}(\text{Msg},N,\text{ls},\text{os}) & \rightarrow \\
& \quad J > 0 \mid \text{server.init}(N,\text{O},\text{O}) \& J, \quad J \rightarrow J -1, \\
& \quad \text{create1}(\text{Msg},N,\text{L},[\text{ls}],\text{[O | Os]}). \\
\text{create1}(\text{Msg},N,\text{ls},\text{os}) & \rightarrow \\
& \quad \text{form.js}(N,\text{ls}, \text{first}(\text{Msg},\text{ls},\text{ls},1), \text{form.os}(\text{os},\text{ls},1)). \\
\text{form.js}(N,\text{L},[\text{ls}]) & \rightarrow \text{form.js}(N,\text{L}, \text{ls}), \\
& \quad \text{form.js}([-]).
\end{align*}
\]

\[
\begin{align*}
\text{form.js}(N,\text{O}) & \rightarrow N > 0 \mid 1 := [-1, N], \quad \text{form.js}(N,\text{L},1). \\
& \text{form.js}(0,\text{L}) \rightarrow 1 := [1].
\end{align*}
\]

\[
\begin{align*}
\text{first}(\text{M},[[\text{S} | \text{S}], \text{ls}, \text{ls},1]) & \rightarrow S := [M | \text{S}1], \quad \text{ls} := [[\text{S}1 | S], \text{ls}]. \\
\text{form.os}(\text{O} | \text{Os}, \text{ls}) & \rightarrow \text{form.os}(\text{ls}, \text{ls}, \text{ls}, 1), \text{form.js}(\text{os}, \text{ls}). \\
\text{form.js}([..]).
\end{align*}
\]

\[
\begin{align*}
& \quad \text{form.js}([\text{S} | \text{S}], \text{ls}, [1, \text{ls}, 1, \text{O}]) \rightarrow \\
& \quad \quad \quad \text{O} := [\text{S}1 | \text{O}], \quad \text{ls} := [[\text{S}1 | \text{S}], \text{ls}2, \text{ls}]. \\
& \quad \text{form.js}(\text{ls}, [1, \text{ls}, 0]) \rightarrow \text{O} := [1], \quad \text{ls} := [1].
\end{align*}
\]

\[
\begin{align*}
\text{server.init}(N,\text{Ins},\text{Outs}) & \rightarrow \\
& \quad \text{combine}(\text{Ins}, \text{ls}), \text{merge}(\text{ls}, \text{ln}), \\
& \quad \text{make.tuple}(N, \text{Outs}), \text{fill}(1, \text{Outs}, \text{Out}). \\
& \quad \text{server}(\text{ln}, \text{Out}).
\end{align*}
\]

\[
\begin{align*}
& \quad \text{combine}([\text{S} | \text{Ins}], \text{ls}) \rightarrow \text{ls} := [\text{merge}() | \text{ls}1], \text{combine}(\text{ins}, \text{ls}1). \\
& \quad \text{combine}([\text{ls}], \text{ls}) \rightarrow \text{ls} := [1].
\end{align*}
\]

\[
\begin{align*}
& \quad \text{fill}([1 | \text{S} | \text{O}], \text{Out}) \rightarrow \text{put.arg}([1, \text{Out}, \text{S}]), \quad \text{ls} := \text{ls} + 1, \text{fill}([1, \text{Out}, \text{O}], \text{Out}). \\
& \quad \text{fill}([-1, \text{..}]).
\end{align*}
\]

Figure 3: Server Library Program

A second observation is that parts B, C, and D of the program are concerned with distinct functions: tree reduction, random mapping, and creation of a server network, respectively. This clean separation of concerns suggests that it might be useful to distinguish one low level motif to support the server abstraction and other, higher-level motifs to support random mapping and tree reduction.

A third observation is that the code in Part B is not as clear as we would like. The following, more abstract specification is easier to understand. This provides a more succinct specification of a divide-and-conquer tree reduction algorithm. The pragma `random` is used to identify the process that is to be dispatched for execution on a randomly-selected processor.

\[
\begin{align*}
\text{reduce}(&\text{tree}(V,L,R,\text{Value}) \rightarrow \\
& \quad \text{reduce}(R,\text{RV}) \quad \text{random}, \text{reduce}(L,\text{LV}). \\
& \quad \text{eval}(V,\text{LV},\text{RV},\text{Value}). \\
& \quad \text{reduce}([\text{leaf}(L),\text{Value}]) \rightarrow \text{Value} = L.
\end{align*}
\]

This alternative expression of the tree reduction algorithm
motivates two further observations. First, there is no reason why we should not express the algorithm in the alternative, more convenient form: It can easily be converted to the form in Figure 2 (Part B) with a simple source-to-source transformation. Second, this transformation is, if implemented properly, completely application-independent: It will support the use of the "@random" pragma in any program.

In summary, we have shown that our tree reduction motif can be decomposed into three distinct layers: divide-and-conquer tree reduction (Part B); random mapping (a transformation, and Part C); and servers (Part D). Each of these layers can be viewed as a motif in its own right.

The purpose of this exercise in decomposition was to show how motifs can be isolated in parallel programs. In practice, a user of a motif-based programming system would take basic motifs such as servers and random mapping from libraries, and construct a new tree reduction motif by composition. Hence, the user would not need to be aware of the implementation details embodied in Figures 2 and 3; he would only need to provide the four-line program given above.

In the rest of this section, we develop implementations for the server, random mapping, and tree reduction motifs and show how more complex motifs can be built up from simpler motifs by composition.

3.2 Server Motif

The Server motif provides the programmer with a fully connected set of named servers, each capable of initiating computations upon receipt of messages from other servers. These computations can in turn generate further messages. This is a relatively low level abstraction, its main purpose is as a building block for other, higher-level constructs.

Recall that the server code in Figure 2 takes two arguments: an input stream of messages and a tuple of output streams to other servers. The code invoked by the server to handle messages (reduce) calls low level primitives such as distribute, which appends a message to an output stream. This explicit manipulation of the output stream tuple is clumsy. Hence, the Server motif provides a transformation which allows programmers to express server definitions in a more convenient format. This permits a programmer to provide just a definition for a one-argument server process (server/1) and an initial incoming message. The server definition may include calls to send, nodes, and halt operations. A call send(Name,Message) requests that a specified message be passed to a named server. A call nodes(N) is used to determine the number N of servers in operation. A call halt requests that the message halt be sent to each server. The server definition must include both a rule for each message type sent in the server code, and a rule to handle the halt message.

Recall that a motif is implemented by a library and a transformation. In this case, the library code handles the creation of the server network, message routing, termination, etc.; the transformation introduces the output stream tuple and translates calls to operations such as send into calls to low-level primitives. The transformed user program is linked with the library to produce an executable program.

![Figure 4: The Server Motif](image)

The relationship between the library code and the user program is illustrated in Figure 4. In this figure, the circles represent processes and the arrows communication channels. The first part of the figure represents the application-independent server network created by the library code. The second part represents the application-specific server code (the solid circle) and the initial message (m). The third part shows how the application-specific code is embedded in each network node.

Server Library

The complete server library code has already been presented, in Figure 3. It is invoked with a call create(N,Msg), where N is the number of servers to create and Msg is an initial incoming message to be passed to some server.

Execution of this program generates a network of N server processes. Each server executes a process server/2, where the second argument is a tuple containing output streams to each of the other servers and the first argument is a stream corresponding to an intermingling of all the output streams directed to that server. The N input streams are combined into a single stream by a merge primitive [8]. The servers are spawned on separate processors using a low-level Strand environment feature (not a motif) that permits the placement of processes on named processors; this feature is invoked by the @J notation in the second rule in Figure 3.

Server Transformation

The application-specific code to which the Server motif is applied must provide a definition for a process server/1 which expects a stream of incoming messages as its only argument. The server library program (Figure 3) invokes a process server/2, with the additional second argument being a tuple of streams to other servers. The Server
transformation introduces the additional second argument and also translates send, nodes, and halt operations into calls to low-level primitives. These changes are achieved with the following steps.

1. Add as an additional argument a new, unique variable (e.g., DT) to both process definitions that includes a call to the send, nodes, or halt primitives, and the process definitions of these processes' ancestors in the call graph.

2. Replace each call `send(Node,Msg)` with a call `distribute(Node,Msg,DT)`, where DT is the additional argument introduced in step (1). The distribute primitive sends a message on a numbered stream in the tuple of server streams represented by DT.

3. Replace each call `nodes(N)` with a call `length(DT,N)`, where DT is the additional argument introduced in step (1). The length primitive call determines the size of the DT tuple, which corresponds to the number of servers, and thus assigns N this number.

4. Replace each call `halt` with a call to a process that appends the message `halt` to each server stream contained in DT, hence broadcasting this message to all servers.

The extended application program generated by this transformation is linked with the server library (Figure 3) to produce the output of the `Server` motif. This is a program that can be executed on a parallel computer.

The application of the `Server` motif is illustrated in Figure 5. This figure will be used in subsequent discussion to illustrate motif composition. It is divided into four sections, each representing the program produced when a specified motif is applied to the code in the previous section. We are interested at this point in the fourth section, which contains the program produced when the `Server` motif is applied to the code in the third section. Note the changes to the server definition and the translation of `send`.

3.3 Random Mapping Motif

We now consider the first of several motifs to be constructed using the `Server` motif as a building block. The `Random` motif supports the `Random` pragma, used to specify processes to be migrated to other, randomly selected processors. This motif can be implemented by a composition of the `Server` motif with a simple motif, `Rand`. That is, `Random = Server o Rand`. The motif `Rand` is defined by an empty library and the following transformation.

1. Replace each call "P@random" (where P represents an arbitrary process) with the sequence of calls `nodes(N), rand_num(N,R), send(R,P)`. A call `rand_num(N,R)` computes a random integer value R in the range (1,N);

hence, this instruction sequence sends a message representing the process P to a randomly selected server.

2. Augment the application program with a process definition `server/1` containing rules for the process used to initiate execution of the application, for each process type annotated with `@random`, and for the halt message. For example, the rule for a process with name p and n arguments has the following form; the `Vi` are unique variables.

\[
\text{server}([\{p(V1,\ldots,Vn)\} \mid n]) := p(V1,\ldots,Vn), \text{server}(\text{In}).
\]

This rule initiates execution of a process `p(V1,\ldots,Vn)` upon receipt of the corresponding message.

Application of the `Rand` motif to an application program produces a new program that includes both a `server/1` definition and `send` calls. This is illustrated in Figure 5; the third section in this figure is the output generated when `Rand` is applied to the code in the second section. Note the changes to reduce and the new server definition. Note also that the code produced is in the form required by the `Server` motif. Application of the `Server` motif to `Rand`'s output introduces additional arguments and code to produce an executable parallel program.

The `Random` motif described here does not provide for termination detection in an application. If this is required, the associated transformation can be extended to thread a short circuit [8] through the application program and to add code to invoke the `Server` motif's halt operation when the application terminates.

3.4 Tree Reduction Motif I

We can now complete the definition of the tree reduction motif introduced in Section 3.1. We achieve this by defining a simple motif `Tree1` comprising the identity transformation and the following library program.

\[
\begin{align*}
\text{reduce}(\text{tree}(V,L,R),\text{Value}) := & \\
\text{reduce}(R,RV@random), & \\
\text{reduce}(L,LV), \text{eval}(V,LV,RV,\text{Value}). & \\
\text{reduce}(\text{leaf}(L),\text{Value}) := & \text{Value} := L.
\end{align*}
\]

As shown in Figure 5, linking this program with a user-supplied node evaluation program produces a program in the form required by the `Rand` motif. Hence, the `Tree1` motif can be composed with the `Random` motif to obtain a tree reduction motif `Tree-Reduce-1`. That is,

\[
\text{Tree-Reduce-1} = \text{Random o Tree1} = \text{Server o Rand o Tree1}
\]

The various stages of the composition process are illustrated in Figure 6. At each stage, a (possibly identity) transformation is applied to the output of the previous stage (or to the node evaluation function, in the first stage)
and a (possibly empty) library program is incorporated. Figure 5 shows the programs produced at each stage of the composition.

% Output of `Tree-Reduce-1` motif.
eval(...) := ...

reduce(tree(V,L,R),Value) :=
  reduce(R,RV) @ random,
  reduce(L,LV), eva(V,LV,RV,Value).
reduce(leaf(L),Value) := Value := L.

% Output of `Rand` motif.
eval(...) := ...

reduce(tree(V,L,R),Value) :=
  nodes(N), rand.num(N,O), send(O,reduce(R,RV)),
  reduce(L,LV), eva(V,LV,RV,Value).
reduce(leaf(L),Value) := Value := L.

server([reduce(T,V) | In]) :=
  reduce(T,V), server(In)
server([halt | ...]).

% Output of `Server` motif.
eval(...) := ...

reduce(tree(V,L,R),Value,DT) :=
  length(DT,N), rand.num(N,O),
  distribute(O,DT,reduce(R,RV)),
  reduce(L,LV,DT), eva(V,LV,RV,Value).
reduce(leaf(L),Value, ...) := Value := L.

server([reduce(T,V) | In,DT]) :=
  reduce(T,V,DT), server(In,DT).
server([halt | ...]).

... Server Library Code ... (Figure 3)

Figure 5: The Three Stages of `Tree-Reduce-1`

3.5 Tree Reduction Motif II

The tree reduction motif defined in previous sections can initiate multiple computations on the same processor simultaneously, as each reduce message received by a server causes the initiation of an independent computation. This is potentially problematic in the genetic sequence alignment application, as each invocation of the node evaluation function can create large intermediate data structures. We present an alternative tree reduction motif, `Tree-Reduce-2`, that addresses this problem. Each tree node is allocated to a randomly selected processor. The value of a node is computed when its offspring's values are available and is then sent to the processor on which its parent is located. At each processor, computation is sequenced so that only a single node evaluation is active at any given time. This reduces memory consumption.

We assume that each node in the tree is assigned an integer label and a unique integer identifier. Leaf nodes are labeled with a processor number, selected randomly with the restriction that sibling nodes have the same label; non-leaf nodes are labeled with the label of their left-hand offspring. Each node is also given both the identifier and label of its parent. The parent label at a node is used to determine the processor to which the node's value should be sent when computed; the labeling used here ensures that an interprocessor communication is required for at most one of each node's offspring values. Labels are generated in a preprocessing step introduced by the transformation.

Initially, leaf values are sent to the processors indicated by their label. Values received at each processor are maintained in a queue. When values for two nodes with the same parent are available, a process is created to compute the value of that parent. Values are computed in sequence at each processor.

The implementation of this alternative tree reduction motif builds on the `Server` motif (Section 3.2). It exploits a server's ability to maintain a history of messages received and to process subsequent messages in ways that depend on this history. A motif `Tree-Reduce` augments the user-supplied node evaluation function with termination-detection code and integrates this with a tree reduction library program. Applying the `Tree-Reduce` motif to an application program generates a definition for `server1`; application of the `Server` motif to this program produces an executable program. That is, `Tree-Reduce = Server o Tree-Reduce`.

Tree Reduction Library

The `Tree-Reduce` library code is presented in Figure 7. As before, our intention here is not to explain all details of the program's execution but to demonstrate that our techniques permit concise specifications of complex concurrent algorithms. The tree is assumed to be represented by a tuple in which the i-th element contains the data value, parent identifier, and parent label for the tree node with identifier i. The root node is distinguished by a parent identifier of -1. As before, the process eval represents the user-supplied node evaluation function.

The code in Figure 7 defines a process `server1` that handles value, tree, init, and halt messages (R2-5). The init message represents the initial message used to start a computation; it provides the tree to be reduced. A node receiving an init message (R4) first sends a copy of the tree to each server (using a tree message: R11) and then dispatches a value message for each leaf node (R13). A value message is evaluated with respect to the state maintained at the server to which it is directed (R2-6-8). This state contains the tree in question, a list of pending values, and a solution variable. Evaluation of the initial value mes-
3.6 Discussion

We have shown in this section how basic motifs such as Server and Random can be used as building blocks with which other motifs can be composed to construct higher level motifs. The use of composition greatly simplified development of the two tree reduction motifs: The first is implemented with five lines of code, and the second with a page of library code and a simple transformation (to introduce termination detection code) that would normally be available in a library. In contrast, the node evaluation code for the sequence alignment application currently exceeds 2000 lines of Strand and C. Hence, the use of motifs permits a parallel version of our code to be developed with only a small incremental effort, even if no tree reduction motif is available.

Two different tree reduction motifs were developed in this section. These provide the same interface to the user, who need provide only a node evaluation function to obtain a parallel program. However, the two motifs implement different parallel algorithms. The second is better suited to our application. We suspect that many applications will benefit from specialized motifs tailored to their particular requirements.

4 Conclusion

Algorithmic motifs can make parallel programming easier by replacing the difficult problem of coordinating the actions of many processors with the simpler problem of choosing, and perhaps tailoring, an existing motif. However, effective use of motifs requires techniques to facilitate reuse, not only "as is", but also through modification and composition.

We have described a novel framework for the implementation of motifs. Two key features of this framework encourage reuse. A high level language is used to express concurrent components of motifs; this facilitates both the creation of new code and the maintenance, understanding, and modification of existing code. Hence, collections of motifs become archives of expertise that can be consulted and adapted for new purposes. The second key feature is the implementation of motifs as source-to-source transformations. Transformations can be used to automatically restructure application-specific code; this permits users to express applications in a convenient, motif-independent format. Transformations also support the development of new motifs by composition.

Our (admittedly limited) experience with these techniques suggests that they can significantly simplify the task of generating good parallel programs. If procedures that implement the necessary application-specific functions are available, then in many cases the use of motifs can virtually eliminate the incremental cost of generating a parallel program. Another benefit of the approach is that it encourages programmers to experiment with the use of alternative motifs in a single application. We believe that this exploratory approach often produces better parallel programs than could be obtained with a less powerful system.

It is our intention to build a comprehensive parallel programming system based on these ideas. To date, we have implemented a number of motifs and developed basic building blocks for motif construction. The design and implementation of one motif, a scheduler, is described in [6]; building blocks include key transformations. We hope that the availability of basic motifs and building blocks will encourage others to develop the body of more specialized motifs required to produce a generally useful system.

In the future, we plan to develop new motifs and explore their application in program development. Areas in which motifs seem appropriate include search, sorting, grid problems, divide and conquer, and various graph theory problems.
Figure 7: Tree Reduction Library

Acknowledgments

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The multiple sequence alignment application discussed in Section 3 was provided to us by Ross Overbeck.

References


