Basic Fortran Support in Legion

Adam J Ferrari and Andrew S. Grimshaw
Department of Computer Science, University of Virginia

Technical Report CS-98-11
March 4, 1998

Abstract

Fortran is the most widely used programming language for high-performance scientific computing applications, yet in the past the Legion system has not supported objects implemented in Fortran. This paper describes the design and interface of the Legion Basic Fortran Support (BFS) system. This system consists of compiler and runtime library that allow the description of Legion object interfaces in a Fortran-like Interface Description Language (IDL), and the implementation of Legion objects using Fortran. The system also supports remote method invocations on Legion objects through the use of pseudo-comments: Legion BFS directives embedded in normal Fortran comment lines. These method invocations are processed using a macro-dataflow model similar to that provided by the Mentat Programming Language, thus allowing both inter- and intra-method parallelism.

1. Introduction

Fortran is the most widely used programming language for high-performance scientific computing applications. In the high-performance computing community it has received more attention than any other single language. Indeed, many parallel Fortran dialects exist, ranging from Fortran M to Fortran 90 and HPF. In the past the Legion project has not attempted to support Fortran in its native form, opting instead to concentrate on object-oriented languages more closely matched to the Legion programming model. The effort to provide similar compiler support for Fortran that we provide for C++ was beyond the scope of the project.

An alternative to full-fledged compiler support has been under discussion in the Legion project for some time and is now available. This alternative, which we call *Basic Fortran Support* or *BFS*, is a compromise between full-fledged compiler support and essentially no support. The basic idea is simple and harks back to the pseudo-comments used in early parallelizing Fortran compilers such as Paraphrase [4]. Rather than creating an extended Fortran dialect that includes coarse-grain objects and attempting to parse and perform data-dependence analysis on Fortran programs, we provide a set of Legion directives that can be embedded in Fortran code in the form of pseudo-comment lines. The programmer uses these embedded pseudo-comments to call member functions on Legion objects and to obtain the results of these member functions. The pseudo-comment syntax is intentionally similar to Fortran, and is easy for Fortran programmers to learn.

The pseudo-comments are translated by a simple preprocessor, `legion_bfs`, into calls to a BFS support library, which in turn directly invokes the Legion run-time system. The pseudo-comments direct the preprocessor to invoke member functions on Legion objects. Keywords in the pseudo-comments indicate whether the invocation is a subroutine-style invocation or a function, whether the call should be made synchronously or asynchronously, and whether the call is to a stateless object or to stateful object. The execution model is that of Legion—each “remote” member function invocation executes in a separate address space. Thus, all communication between the caller and callee is via parameters and

returns values—there is no global memory. In the case of stateless objects, the back-end run-time system will create new object instances to service function invocations as needed.

In addition to invocations on Legion objects, the Fortran program can define Fortran-based Legion objects using a Fortran-like *interface description language* (*IDL*) [5]. The IDL is translated by the `legion_bfs` filter, which generates Fortran skeletons that can be linked to Fortran code in order to implement the object interface. The Fortran implementation code may itself be the output of the `legion_bfs` preprocessor, allowing Fortran code to act as both client and server in a Legion system.

This document is designed to give the reader a basic overview of Legion BFS and to permit the reader to begin using the features of the "language" rapidly. In Section 2 we describe the BFS programming interface. In Section 3 and Section 4 we provide two simple example programs that illustrate the BFS programming model. In Section 5 we discuss the current status of the BFS implementation and enumerate some of the limitations of the current implementation as a guide to possible future development of the system. In Section 6 we include a partial grammar for the BFS IDL and directives as an aid to understanding the language.

2. Programming Interface

The Legion BFS programming interface consists of two basic parts: client-side method invocation through pseudo-comment directives, and server-side object development through the BFS IDL. In "client" code (i.e. Fortran code that will invoke methods on Legion objects) pseudo-comments are used to describe method invocations. These pseudo-comments are translated into calls to a BFS support library, which in turn uses the services provided by the Legion system to support method invocation (see Figure 1). Fortran code that will be used to implement Legion objects (i.e. Fortran code that will be called through a Legion member function interface) must be described by programmer-supplied BFS IDL interfaces. These IDL interfaces are translated by the `legion_bfs` filter, which generates Mentat Programming Language (MPL) [1] skeleton code. This skeleton code is compiled using the `legion_mplc` compiler and linked to Fortran implementation code (see Figure 2).

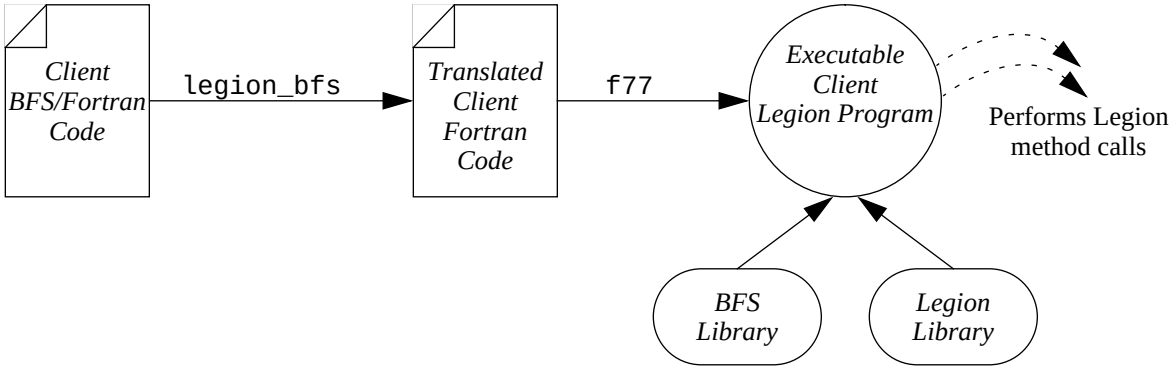

FIGURE 1. Creation of a Legion client program using BFS. The client Fortran code, with embedded pseudo-comments describing Legion method invocations, is translated to standard

It is certainly possible that "server" code—code that will be called through Legion member functions—may need to call the methods of other Legion objects. In such cases, the server implementation code includes the appropriate BFS pseudo-comment directives, is translated by `legion_bfs`, and is then linked to the appropriate IDL (also translated by BFS), as depicted in Figure 3. Thus, a single Fortran subroutine or function might both be called through Legion, and call other objects through Legion. In the following two sections, we examine the server and client programming interfaces in more detail. The combination of these two (as depicted in Figure 3) is straightforward, and is thus left to the reader.

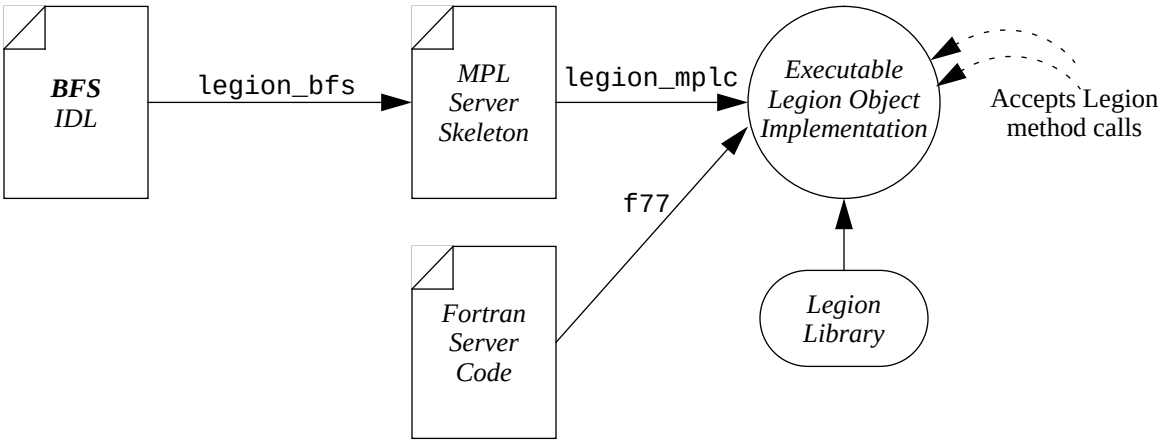

FIGURE 2. Creation of a Legion object using BFS. The object's interface, described in an IDL file, is translated by the `legion_bfs` filter to create MPL skeleton code. This skeleton code, together with

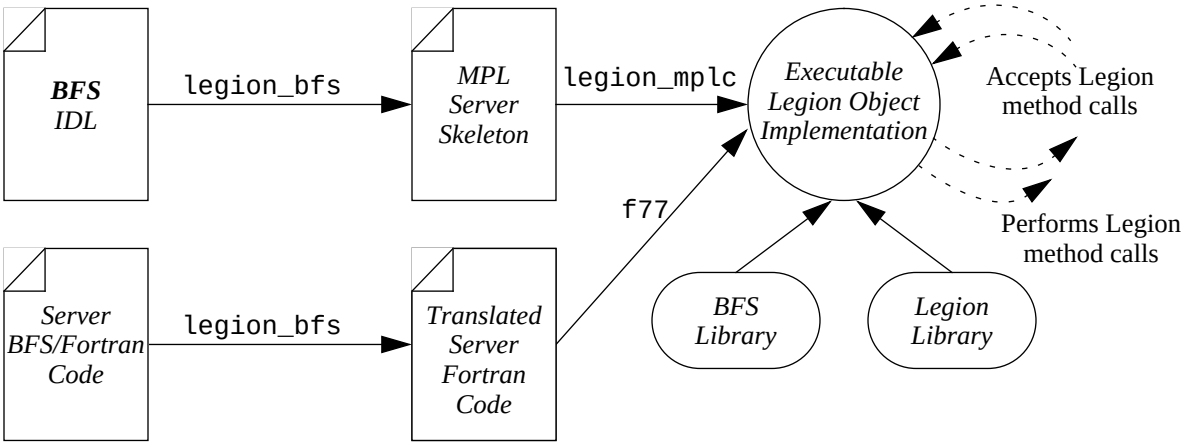

FIGURE 3. Creation of a Legion object that performs Legion method invocations. Creation of such objects involves a combination of the "client" and "server" mechanisms.

2.1 Server-side Programming

We first consider the problem of implementing Legion objects in Fortran—that is, the process of taking subroutines and functions defined in Fortran and making them available as Legion methods. Fortran code wrapped in this manner can be called in parallel from anywhere in a Legion network. In Section 2.2 we examine the programming interface used to invoke functions on Legion objects—i.e. the client side interface.

2.1.1 Class Styles

BFS supports the definition of two basic kinds of Legion objects: stateless and stateful. Stateless objects are useful for making purely functional services available. Stateless objects support methods whose outputs are pure functions of their inputs—the operation of a stateless object is not affected by previous method invocations or evolving object state. Because of this, methods invoked on stateless objects are automatically load-balanced among all available instances of the given stateless class, thus allowing good performance in variably loaded or heterogeneous networks. Despite this advantage, it is often more natural to deal with objects that maintain state between method invocations. To enable this programming style, BFS supports stateful classes. Instances of these classes have normal method

invocation semantics: methods are invoked on a specific object, and the object "remembers" its state from one invocation to the next.

2.1.2 Mechanics

To make Fortran subroutines and functions available in the form of a Legion object, we must "wrap" the Fortran code that implements the functions in a C++/MPL skeleton, invoke the MPL compiler, and link the generated MPL objects with the Fortran implementation code provided by the user. This process is automated through the use of a simple IDL that follows the same form as the caller-side specification of the function (discussed in Section 2.2). The user specifies the name and parameters of all functions in an interface specification (i.e. IDL) file. The `legion_bfs` filter parses this interface specification file and generates an MPL program that matches the interface. The user then compiles and links the program using `legion_mplc`. No changes to the Fortran code, nor any C++ or MPL programming on the part of the user is required.

Suppose we wish to make a Fortran module containing a function called `function1` and a subroutine called `subroutn1` available in the form of a stateless Legion object. The following IDL interface specification file would be used:

```
C Comments allowed
LEGION STATELESS CLASS my_class
   real function1(integer)
   subroutn1(INOUT integer I, INOUT real dimension (*,*))
CLASS END
```

The syntax for stateful objects is similar. For example:

```
LEGION CLASS my_class
   mysub(inout integer I)
   initialize(integer I, character path)
CLASS END
```

There should be exactly one class specification in each input file. BFS IDL files require a "`.bfs`" suffix.

2.1.3 Compilation

To compile an object, first translate the IDL file using `legion_bfs`. Given a file with a "`.bfs`" suffix, legion_bfs will generate a "`.c`" and a "`.h`" MPL file. The resulting MPL file should be compiled using `legion_mplc`. The object file containing the Fortran implementation code should be specified on the `legion_mplc` link line. For example:

```
$ legion_bfs my_class.bfs
  Parsing class my_class
$ f77 -c my_class_impl.f
$ legion_mplc my_class.c my_class_impl.o -o my_class
```

2.2 Client-side Programming

In this section, we examine the client-side (i.e. method invocation) programming interface. The client-side programming interface is based on the use of pseudo-comments. The programmer embeds BFS directives into normal Fortran code using comments that begin with the prefix `LEGION`. This "annotated" Fortran code is then translated by the `legion_bfs` filter, which generates standard Fortran code. This resulting code is then compiled and linked against a BFS support library and the Legion library.

2.2.1 Include Files

Before methods can be invoked on objects of a given class, the BFS translator must have a

description of the class's interface. This interface description will be used by `legion_bfs` to generate the appropriate calls to the Legion run-time system. The mechanism for communicating object interfaces to the translator is the `INCLUDE` directive, which is used to "include" BFS IDL files in BFS Fortran code. For example, the following statement

```
C LEGION INCLUDE my_class.bfs
```

should precede any methods called on instances of the class `my_class`.

2.2.2 Method Invocation

In this section we cover the BFS syntax used to invoke methods. We start with a very simple example: a blocking remote method invocation that takes a single parameter and returns a single result. To call remote method `function1` defined by stateless class `my_class`, which takes an integer parameter and returns a real result, the following syntax could be used:

```
C LEGION SYNCH FUNCTION real X = (my_class)function1(integer I)
```

This would perform a blocking remote procedure call (RPC) on an instance of `my_class`: calling `function1`, passing the integer parameter `I` as an input, and placing the real return value in the variable `X`. Note that the complete method invocation is contained within a single Fortran comment line. Note also that this method is invoked on a stateless Legion class, whose instances are therefore pure functional units. Thus, this method will be serviced not by a specific named object but by any available (or newly created) instance of `my_class`.

The above example performs a standard, *blocking* RPC on a Legion object. To achieve parallelism between remote method invocations, a non-blocking RPC mechanism is necessary. In BFS, non-blocking RPCs are achieved through the use of the `ASYNCH` specifier.

```
C LEGION ASYNCH FUNCTION real X  = (my_class)function1(integer I)
```

This statement causes the same function to be executed but does not wait for the return value to be placed in `X`. When this statement is executed, the method `function1` begins to be processed at the remote object, and the caller immediately proceeds. The caller can later block for the result using the statement:

```
C LEGION BLOCK real X
```

Since the variable `X` was previously named as the result of an asynchronous method invocation, when this statement is encountered the caller blocks for the result and assigns it to `X`. The use of asynchronous method calls allows methods to execute in parallel. For example, if `function1` is time consuming, a caller of this function might perform other work in parallel before blocking for the result. This parallel work could include the invocation of other Legion methods, such as additional calls to `function1`.

Unlike Fortran, which uses call-by-reference parameter passing, the default BFS parameter passing convention is call-by-value. So, if the variable `I` in the above example is modified by `function1`, that change would not be propagated back to the caller. Other parameter-passing semantics are supported through the use of the key words `IN`, `OUT`, and `INOUT`. For example:

```
C LEGION ASYNCH FUNCTION real X = (my_class)function2(INOUT integer I)
```

This specifies that the variable `I` is call-by-value-result. The value of `I` is passed to the function, and when the function terminates the new value of `I` will be copied from the callee to the caller.

The above examples demonstrate functions—methods that produce a return value. Standard Fortran subroutines are also supported. For example:

```
c LEGION SUBROUTINE (my_class)subroutn1(INOUT integer I)
```

The analogous asynchronous call is:

```
C LEGION ASYNCH SUBROUTINE (my_class)subroutn1(INOUT integer I)
C LEGION BLOCK integer I
```

So far, we have only considered scalar parameters However, routines of interest often deal with arrays. To pass the array A into a method subroutn2, the following syntax is used:

```
C LEGION SUBROUTINE (my_class)subroutn2(INOUT integer dimension(10,10) A)
```

The effect is to pass the array A into subroutn2, and, when the method completes, to copy the array back to the caller. The maximum number of dimensions is not fixed, though the user should be aware that there must be adequate memory to copy parameters, and that large parameters require more communication time.

To this point, we have only demonstrated calls on stateless objects. As described in Section 2.1, BFS also supports stateful objects—objects that maintain state between method invocations. Unlike stateless invocations, which are performed on any object of a given class, stateful invocations must be performed on a specific, named object instance. In BFS, Legion objects are identified by integer *OID*s (Object IDentifiers) that are only valid in the local address space. OIDs are obtained in two ways: as the result of object creation requests, and as the result of looking up object names in Legion context space [3].

Object creation is supported through the CREATE directive. For example:

```
C LEGION CREATE OBJ = NEW my_stateful_class
```

This statement will cause the creation of a new object my_class. An OID that refers to the new object is stored into the integer variable OBJ. In addition to creating new objects, BFS programs can bind to existing objects. An OID for an existing object, which is named in Legion context space, can be obtained using the LOOKUP directive:

```
C LEGION LOOKUP OLDOBJ = my_object
```

This statement will look up the existing object named "my_object" in Legion context space, and store an OID that refers to "my_object" in OLDOBJ.

The syntax for invoking methods on stateful objects is similar to that used with stateless objects, but introduces the need to specify a target object using an OID. For example:

```
C Legion SUBROUTINE OBJ->subroutn1(parameter list)
```

If your program is using multiple classes with methods of the same name, you must specify the class explicitly to disambiguate method invocations. For example:

```
C Legion SUBROUTINE (my_stateful_class)OBJ->subroutn1(parameter list)
```

This more verbose syntax is always acceptable, and will generally lead to easier to read, safer code.

Unlike memory that is declared within a program, stateful objects can persist indefinitely beyond the lifetime of the program that creates them. To avoid creating "garbage" objects—objects that are no longer needed by any programs but are still consuming system resources—BFS programs must clean up its stateful objects before terminating. Stateful objects may be deleted using the DESTROY directive.

```
C LEGION DESTROY OBJ
```

This statement will destroy the object referred to by the OID OBJ.

2.2.3 Compilation

Fortran code containing the pseudo-comment directives discussed in this section must first be preprocessed by the `legion_bfs` translator. Given a file with a "`.f`" suffix, the translator will produce a new Fortran source file with the suffix "`.trans.f`". This resulting file should be compiled by a standard Fortran compiler, and linked against the Legion libraries using the following flags (assuming your primary C++ compiler is g++):

```
-L$LEGION/lib/$LEGION_ARCH/g++ -lLegion -lLegionBFS
```

3. Example Program: Stateful Objects

In this section, we present a simple but complete program using stateful objects. Consider the Fortran code depicted in Figure 4. To make this code available in the form of a Legion object, we define an object interface for it using the IDL depicted in Figure 5. Note that since the Fortran code in Figure 4 depends on state that is maintained between method invocations (i.e. the common block variable `A`), we use a stateful Legion object to wrap the code.

```
C File: my_class_impl.f
    subroutine sub(I)
       integer I,A
       common A
       A = I
       I = I + 1
    end

    integer function func(I,J)
       integer I,J,A
       common A
       func = I + J + A
       return
    end
```

FIGURE 4. Example Fortran code that we wish to make available in the form of a

```
C File: my_class.bfs
LEGION CLASS my_class
   sub(inout integer i)
   integer func(integer i, integer j)
CLASS END
```

FIGURE 5. BFS IDL suitable for wrapping the Fortran code depicted in Figure 4 in

Once the IDL depicted in Figure 5 has been translated by `legion_bfs`, compiled by `legion_mplc`, and linked to the object code resulting from a Fortran compilation of the code in Figure 4, programs can create objects of the class "`my_class`" and invoke methods on them. In Figure 6 we depict a simple BFS Fortran main program that demonstrates object creation, a remote subroutine call, a remote function call, and object deletion. Note that the integer variable `OID` is used as a local reference to the Legion object created in the program. Also note that in this simple example we use synchronous method invocation, since no parallelism was possible between the two method invocations. In Section 4 we consider the use of asynchronous methods and stateless objects, both of which have the potential to offer improved performance.

The output of the program depicted in Figure 6 will be:

```
1 + 2 + 2 = 5
```

```
C File: example1.f
    program example1
    implicit none
    integer OID,I,J,K
    I = 1
    J = 2

    call legion_fortran_setup()
C LEGION INCLUDE my_class.bfs
C LEGION CREATE  OID = NEW my_class
C LEGION SYNCH SUBROUTINE (my_class)OID->sub(INOUT integer I)
C LEGION SYNCH FUNCTION integer K=(my_class)OID->func(integer I, integer J)

    write(*,*) (I-1), ' +', I,' +', J,' =', K

C LEGION DESTROY OID
    call legion_fortran_cleanup()
    stop
    end
```

FIGURE 6. A BFS Fortran program that uses the object interface defined by the IDL in

4. Example Program: Stateless Objects

The previous example used stateful objects, since the wrapped Fortran code maintained state from one method invocation to the next. However, consider the code depicted in Figure 7.

```
C File: dprod.f
    real function dprod(X, Y, N)
        real X(*), Y(*)
        integer N, I
        dprod = 0.0
        do I=1,N
            dprod = dprod + X(I)*Y(I)
        end do
        return
    end

    real function add(A, B)
        real A, B
        add = A + B
        return
    end
```

FIGURE 7. Example Fortran code that we wish to make available in the form of a Legion object. Note, both functions neither rely on state set up by previous calls nor

Both of the functions defined in this file are purely functional—they are free of side effects, and do not rely on state set by previous methods in producing their results. Because of these features, this Fortran module can be wrapped in a stateless Legion object, and thus benefit from improved performance through automatic load balancing. The BFS IDL required to wrap this code is depicted in Figure 8.

Figure 9 depicts a simple Fortran main program that uses the stateless class "`dprod_object`". This example performs two dot product operations and computes the sum of their results. Note that the two dot product operations are completely independent of one another—they operate on entirely disjoint data. Thus, we use `ASYNCH` methods to allow the functions to proceed in parallel. A further benefit of using `ASYNCH` methods comes from data dependence analysis performed by BFS: since the results of the

```
C File: dprod_object.bfs
LEGION STATELESS CLASS dprod_object
   real dprod(real dimension(*) X, real dimension(*) Y, integer N)
   real add(real A, real B)
CLASS END
```

FIGURE 8. BFS IDL suitable for wrapping the Fortran code depicted in Figure 7 in

dot product operations are needed to perform the sum operation, the results of these methods are forwarded directly to the object that will perform the sum operation. Direct forwarding of results, as afforded by ASYNCH methods, improves performance by reducing communication. Instead of the results D1 and D2 being sent first to the main program and then to the sum operation, the parameters can skip the middle hop and go directly to the sum operation. This optimization is especially important when array parameters are used, as they consume the most communication resources.

```
C File: example2.f
   program example2
   implicit none
   integer I
   real X1(3), Y1(3), X2(3), Y2(3), D1, D2, SUM
C LEGION INCLUDE dprod_object.bfs
   call legion_fortran_setup()
   do I = 1,3
      X1(I) = 1.0
      Y1(I) = 2.0
      X2(I) = 0.5
      Y2(I) = 4.0
   end do

C LEGION ASYNCH FUNCTION REAL D1 = dprod(REAL DIMENSION(3) X1,
         REAL DIMENSION(3)  Y1, INTEGER 3)
C LEGION ASYNCH FUNCTION REAL D2 = dprod(REAL DIMENSION(3) X2,
         REAL DIMENSION(3)  Y2, INTEGER 3)
C LEGION SYNCH FUNCTION REAL SUM = add(REAL D1, REAL D2)
C LEGION FREE D1
C LEGION FREE D2

   write(*,*) 'SUM =',SUM

   call legion_fortran_cleanup()
   stop
   end
```

Note, each call to dprod is on a single line

FIGURE 9. A BFS Fortran program that uses the object interface defined by the IDL

The output of the program depicted in Figure 9 is:

```
SUM = 12.
```

5. Current Status and Limitations

The BFS compiler and interface described in Section 2 is currently available as a standard part of the Legion distribution. BFS is available on all supported platforms, but use of the system requires the availability of a Fortran compiler.

The implementation currently has some practical limitations:

- The BFS type system is currently very limited: only REAL, INTEGER, LOGICAL, COMPLEX, and

CHARACTER are supported. Types such as DOUBLE PRECISION and sized types such as INTEGER*8 are not yet available.

- The BFS IDL translator (`legion_bfs`) currently translates "`.bfs`" files to MPL files. Since `legion_mplc` does not currently support "OUT" or "INOUT" parameters, at most one "OUT" or "INOUT" parameter can be specified for any single BFS subroutine, and no "OUT" or "INOUT" parameters may be specified for BFS functions.

6. BFS syntax

This grammar is provided as an aid to understanding the BFS syntax, and as a guide to implementing the syntax. It is neither complete nor suitable for automatic processing.

```
<stmt>         := LEGION <stmt body>
<stmt body>    := <method> | <block> | <free> | <create> | <destroy> | <lookup>

<method>       := <methodstart> <method name> <param list> |
                  <methodstart> <OID>-><method name> <param list>
<methodstart> :=<synch> <subroutine> | <synch> <function>
<sync>         := ASYNCH | SYNCH
<method name>  := <id>
<subroutine>   := SUBROUTINE (<class>) | SUBROUTINE
<class>        := <id>
<function>     := FUNCTION <var> = (<class>) | FUNCTION <var> =
<OID>          := <int expr>
<param list>   := (<params>)
<params>       := <param>, <params> | <param>
<param>        := <mode> <type> <var> | <type> <var> |
                  INTEGER <int> | REAL <real>
<mode>         := IN | OUT | INOUT

<block>        := BLOCK <type> <var>
<free>         := FREE <var>
<create>       := CREATE <var> = NEW <class>
<destroy>      := DESTROY <var>
<lookup>       := LOOKUP <var> = <id>

<var>          := <id> | <id> <indeces>
<indeces>      := (<index list>)
<index list>   := <int expr>, <index list> | <int expr>
<type>         := <scalar type> | <vector type>
<scalar type>  := REAL | INTEGER | LOGICAL | COMPLEX | CHARACTER
<vector type>  := <scalar type> <dim>
<dim>          := DIMENSION (<dims>)
<dims>         := <int> | * | <int>, <dims> | *, <dims>
<int>          := any integer literal
<int expr>     := any valid fortran expression that evaluates to type INTEGER
<real>         := any real literal
<id>           := any valid fortran identifier
```

References

[1] Grimshaw, A.S., “Easy-to-use object-oriented parallel processing with Mentat,” *IEEE Computer*, pp. 39-51, May 1993.

[2] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, “Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,” ACM Transactions on Computer Systems, Vol. 14, No. 2, May 1996.

[3] Legion Research Group, Legion 1.0 Basic User Manual, 1998. Available from: *http://legion.virginia.edu*.

[4] Kuck, D. Lawrie, R. Cytron, A. Sameh and D. Gajski, “The Architecture and Programming of the Cedar System,” Cedar Document no. 21, University of Illinois at Urbana-Champaign, Department of Computer Science, August, 1983.

[5] Object Management Group, “The Common Object Request Broker: Architecture and Specification,” Revision 2.0, July 1995 (updated July 1996).