
Challenges in Designing an HPF Debugger

High Performance Fortran (HPF) provides directive-based data-parallel extensions to Fortran 90. To achieve parallelism, DIGITAL's HPF compiler transforms a user's program to run as several intercommunicating processes. The ultimate goal of an HPF debugger is to present the user with a single source-level view of the program at the control flow and data levels. Since pieces of the program are running in several different processes, the task is to reconstruct the single control and data views. This paper presents several of the challenges involved and how an experimental debugging technology, code-named Aardvark, successfully addresses many of them.

As we learn better ways to express our thoughts in the form of computer programs and to take better advantage of hardware resources, we incorporate these ideas and paradigms into the programming languages we use. Fortran 90^{1,2} provides mechanisms to operate directly on arrays, e.g., $A=2 * A$ to double each element of A independent of rank, rather than requiring the programmer to operate on individual elements within nested `DO` loops. Many of these mechanisms are naturally data parallel. High Performance Fortran (HPF)^{3,4} extends Fortran 90 with data distribution directives to facilitate computations done in parallel. Debuggers, in turn, need to be enhanced to keep pace with new features of the languages. The fundamental user requirement, however, remains the same: Present the control flow of the program and its data in terms of the original source, independent of what the compiler has done or what is happening in the run-time support. Since HPF compilers automatically distribute data and computation, thereby widening the gap between actual execution and original source, meeting this requirement is both more important and more difficult.

This paper describes several of the challenges HPF creates for a debugger and how an experimental debugging technology, internally code-named Aardvark, successfully addresses many of them using techniques that have applicability beyond HPF. For example, programming paradigms common to explicit message-passing systems such as the Message Passing Interface (MPI)⁵⁻⁷ can benefit from Aardvark's methods.

The HPF compiler and run time used is DIGITAL's HPF compiler,⁸ which produces an executable that uses the run-time support of DIGITAL's Parallel Software Environment.⁹ DIGITAL's HPF compiler transforms a program to run as several intercommunicating processes. The fundamental requirement, then, is to give the appearance of a single control flow and a single data space, even though there are several individual control flows and the data has been distributed. In the paper, I introduce the concept of logical entities and show how they address many of the control flow challenges. A discussion of a rich and flexible data model that easily handles distributed data follows. I then point out difficulties imposed on user interfaces, especially when the program is not in a completely

consistent state, and indicate how they can be overcome. Sections on related work and the applicability of logical entities to other areas conclude the paper.

Logical Entities

From the programmer's perspective, an HPF program is a single process/thread with a single control flow represented by a single call stack consisting of single stack frames. A debugger should strive to present the program in terms of these single entities. A key enabling concept in the Aardvark debugger is the definition of logical entities in addition to traditional physical entities. Generally, a *logical entity* collects several physical entities into a single entity. Many parts of Aardvark are unaware of whether or not an entity is logical or physical, and a debugger's user interface uses logical entities to present program state.

A *physical entity* is something that exists somewhere outside the debugger. A physical process exists within the operating system and has memory that can be read and written. A physical thread has registers and (through registers and process memory) a call stack. A physical stack frame has a program counter, a caller stack frame, and a callee stack frame. Each of these has a representation within the debugger, but the actual entity exists outside the debugger.

A logical entity is an abstraction that exists within the debugger. Logical entities generally group together several related physical entities and synthesize a single behavior from them. In C++ terms, a process is an abstract base class; physical and logical processes are derived classes. A logical process contains as data members a set of other (probably physical) processes. The methods of a logical process, e.g., to set a breakpoint, bring about the desired operations using logical algorithms rather than physical algorithms. The logical algorithms often work by invoking the same operation on the physical entities and constructing a logical entity from the physical pieces. This implies that some operations on physical entities can be done in isolation from their logical containers. Aardvark makes a stronger statement: Physical entities are the building blocks for logical entities and are first-class objects in their own right. This allows physical entities to be used for traditional debugging without any additional structure.¹⁰

A positive consequence of this object-oriented design is that a user interface can often be unaware of the physical or logical nature of the entities it is managing. For example, it can set a breakpoint in a process or navigate a thread's stack by calling virtual methods declared on the base classes.

Some interesting design questions arise: What is a process? What is a thread? What is a stack frame? What operations are expected to work on all kinds of processes but actually only work on physical processes? Experience to date is inconclusive. Aardvark currently defines the

base classes and methods for logical entities to include many things that are probably specific to physical entities. This design was done largely for convenience.

Sometimes a logical entity is little more than a container of physical entities. A logical stack frame for threads that are in unrelated functions simply collects the unrelated physical stack frames. Nevertheless, logical stack frames provide a consistent mechanism for collecting physical stack frames, and variants of logical stack frames can discriminate how coordinated the physical threads are. The concept of logical entities does not apply to all cases, though. Variables have values, and there does not seem to be anything logical or physical about values. Yet, if a replicated variable's values on different processors are different, there is no single value and some mechanism is needed. Rather than define logical values, Aardvark provides a differing values mechanism, which is discussed in a later section of the same name.

Controlling an HPF Process

Users want to be able to start and stop HPF programs, set breakpoints, and single step. From a user interface and the higher levels of Aardvark, these tasks are simple to accomplish—ask the process or thread, which happens to be logical, to perform the operation. Within the logical process or thread, however, the complexity varies, depending on the operation.

Starting and Stopping

Starting and stopping a logical thread is straightforward: Start or stop each component physical thread. Some race conditions require care in coding, though. For example, starting a logical thread theoretically starts all the corresponding physical threads simultaneously. In practice, Aardvark serializes the physical threads. In Figure 1, when all the physical threads stop, the logical thread is declared to be stopped. Aardvark then starts the logical thread at time “+” and proceeds to start each physical thread. Suppose the first physical thread (thread 0) stops immediately, at time “*.” It might appear that the logical thread is now stopped because each physical thread is stopped. This scenario does not take into account that the other physical threads have not yet been started. Timestamping execution state transitions, i.e., ordering the events as observed by Aardvark, works well; a logical thread becomes stopped only when all its physical threads have stopped after the time that the logical thread was started. An added complexity is that some reasons for stopping a physical thread should stop the other physical threads and the logical thread. In this case, pending starts should be cancelled.

Breakpoints

Setting a breakpoint in a logical process sets a breakpoint in each physical process and collects the physical

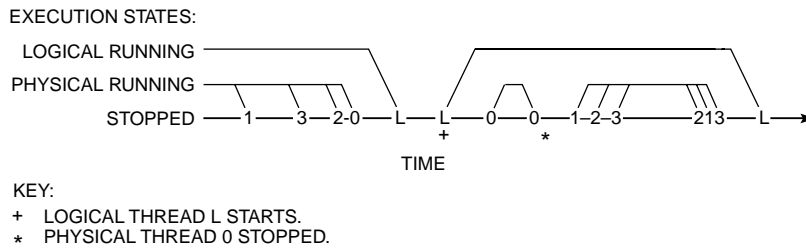


Figure 1
Determining When a Logical Thread Stops

breakpoint representations into a logical breakpoint. For HPF, any action or conditional expression is associated with the logical breakpoint, not with the physical breakpoints. Consider the expression `ARRAY(3,4).LT.5`. Even if the element is stored in only one process, the entire HPF process needs to stop before the expression is evaluated; otherwise, there is the potential for incorrect data to be read or for processes to continue running when they should not. This requires each physical process to reach its physical breakpoint before the expression can be evaluated. Once evaluated, the process remains stopped or continues, depending on the result. For HPF, a breakpoint in a logical process implies a global barrier of the physical processes.

Recognizing and processing a thread reaching a logical breakpoint is somewhat involved. Aardvark's general mechanism for breakpoint determination is to ask the thread's operating system model if the initial stop reason could be a breakpoint. If this is the case, the operating system model provides a comparison key for further processing.

For physical DIGITAL UNIX threads, a `SIGTRAP` signal could be a breakpoint, with the comparison key being the program counter address of the potential breakpoint instruction. This comparison key is then used to search the breakpoints installed in the physical process to determine which (if any) breakpoint was reached. If a breakpoint was reached, the stop reason is updated to be "stopped at breakpoint." All this physical processing happens before the logical algorithms have a chance to notice that the physical thread has stopped. Therefore, by the time Aardvark determines that a logical thread has stopped, any physical threads that are stopped at a breakpoint have had their stop reasons updated.

For a logical thread, the initial (logical) stop reason could be a breakpoint if each of the physical threads is stopped at a breakpoint, as shown in Figure 2. The comparison key in this case is the logical stop reason itself. The breakpoints of the component stop reasons are then compared to the component breakpoints of the installed logical breakpoints to determine if a logical breakpoint was reached. If there is a match, the logical thread's stop reason is updated.

Aardvark achieves the flexibility of vastly different types of comparison keys (machine addresses and logical stop reasons) by having the comparison key type be the most basic Aardvark base class, which is the equivalent of Java's `Object` class, and by using run-time typing as necessary.

Single Stepping

Single stepping a logical thread is accomplished by single stepping the physical threads. It is not sufficient to single step the first thread, wait for it to stop, and then proceed with the other threads. If the program statement requires communication, then the entire HPF program needs to be running to bring about the communication. This implies that single stepping is a two-part process—initiate and wait—and that the initiation mechanism must be part of the exposed interface of threads.

As background, running a thread in Aardvark involves continuing the thread with a *run reason*. The

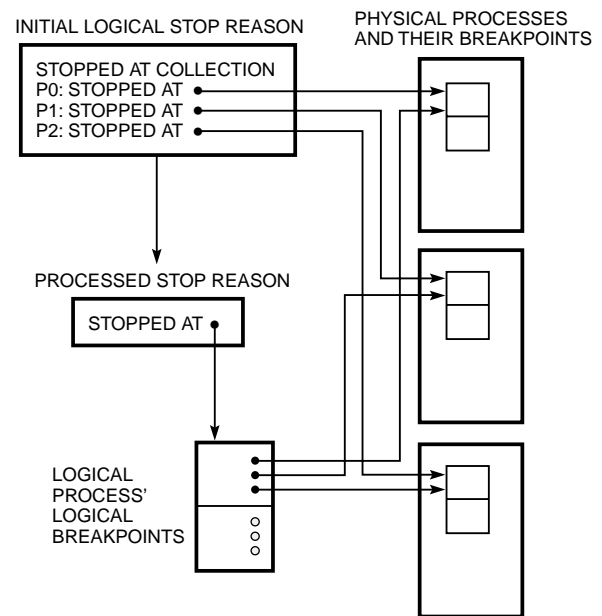


Figure 2
Logical Breakpoint Determination

run reason is empowered to take the actions (e.g., setting or enabling temporary breakpoints) necessary to carry out its task. In this paper, the word *empowered* means that the reason has a method that will be called to do reason-specific actions to accomplish the reason's semantics. This relieves user interfaces and other clients from figuring out how to accomplish tasks. As a result, Aardvark defines a "get single-stepping run reason" method for threads. Clients use the resulting run reason to continue the thread, thereby initiating the single-step operation.

Therefore, single stepping a logical thread in Aardvark involves calling the (logical) thread's "get single-stepping run reason" method, continuing the thread with the result, and waiting for the thread to stop. The "get single-stepping run reason" method for a logical thread in turn calls the "get single-stepping run reason" method of the component (physical) threads and collects the (physical) results into a logical single-stepping run reason. When invoked, the logical reason continues each physical thread with its corresponding physical reason.

Single stepping dramatically demonstrates the autonomy of the physical entities. When continuing a (logical) thread with a (logical) single-stepping run reason, the physical threads can start, stop, and be continued asynchronously to each other and without any intervention from a user interface, the logical entities, or other clients. This is especially true if the thread was stopped at a breakpoint. In this case, continuing a physical thread involves replacing the original instruction, machine single stepping, putting back the breakpoint instruction, and then continuing with the original run reason. Empowering run reasons (and stop reasons) to effect the necessary state transitions enables physical entities to be autonomous, thus relieving the logical algorithms from enormous potential complexity.

Coordinating Physical Entities

The previous discussion describes some logical algorithms. The section "Starting and Stopping" describes using timestamps to determine when a logical thread becomes stopped (see Figure 1), and the section "Breakpoints" describes a logical thread possibly reaching a breakpoint (see Figure 2). The physical entities need to be coordinated so that the logical algorithms can be run. In Aardvark, this is done with a *process change handler*. A process change handler is a set of callbacks that a client registers with a process and its threads, allowing the client to be notified of state changes. For example, if a user interface is notified that a thread has stopped and that the reason is a UNIX signal, the user interface can look up the signal in a table to determine if it should continue the thread (possibly discarding the actual signal) or if it should keep the thread stopped.

In the context of HPF, a user interface registers its process change handler with the logical HPF process. During construction of the logical process, Aardvark registers a *physical-to-logical process change handler* with the physical processes. It is this physical-to-logical handler that coordinates the physical entities. When the first physical thread stops, as at time "*" in Figure 1, the handler is notified but notices that the timestamps do not indicate that the logical thread should be considered to have stopped. When the last physical thread stops, the handler then synthesizes a "stopped at collection" logical stop reason, as in Figure 2, and informs the (logical) thread that it has stopped.

Aardvark defines some callbacks in process change handlers that are for HPF and other logical paradigms. These callbacks allow a user interface to implement policies when a thread or process goes into an intermediate state. For example, at time "*" in Figure 1 a physical thread has stopped but the logical thread is not yet stopped. Whenever a physical thread stops, the handler's "component thread stopped" callback is invoked. A possible user interface policy is¹¹

- If the component thread stopped for a *nasty* reason, such as an arithmetic error, try to stop all the other component threads immediately in order to minimize divergence among the physical entities.
- If this is the first component thread that stopped for a *nice* reason, such as reaching a breakpoint, start a timer to wait for the other component threads to stop. If the timer goes off before all the other component threads have stopped, try to stop them because it looks like they are not going to stop on their own.
- If this is the last component thread, cancel any timers.

The user interface can provide the means for the user to define the timer interval, as well as other attributes of policies. These policies and their control mechanisms are not the responsibility of the debug engine.

Examining an HPF Call Stack

When an HPF program stops, the user wants to see a call stack that appears to be a single thread of control. Sometimes this is not possible, but even in those cases, a debugger can offer a fair amount of assistance. The HPF language provides some mechanisms that also need to be considered. The `EXTRINSIC(HPF_LOCAL)` procedure type allows procedures written in Fortran 90 to operate on the local portion of distributed data. This type is useful for computational kernels that cannot be expressed in a data-parallel fashion and do not require communication. The `EXTRINSIC(HPF_SERIAL)` procedure type allows data to be mapped to a single process that runs the procedure. This type is useful for calling inherently serial code, including user interfaces,

which may not be written in Fortran. DIGITAL's HPF compiler also supports *twinning*, which allows serial code to call parallel HPF code. All these mechanisms affect the call stack or how a user navigates the call stack. They require underlying support from the debugger as well as user interface support.

Logical Stack Frames

Aardvark's logical entity model applies to stack frames: logical stack frames collect several physical stack frames and present a synthesized view of the (logical) call stack. Currently, Aardvark defines four types of logical stack frames to represent different scenarios that can be encountered:

1. Scalar, in which only one physical thread is semantically active
2. Synchronized, in which all the threads are at the same place in the same function
3. Unsynchronized, in which all the threads are in the same function but at different places
4. Multi, in which no discernible relationship exists between the corresponding physical threads

Aardvark's task is to discover the proper alignment of the physical frames of the physical threads, determine which variant of logical frame to use in each case, and link them together into a call stack. Ideally, all logical frames are synchronized, which means that the program is in a well-defined state. This is true most of the time with HPF; the Single Program Multiple Data (SPMD) nature of HPF causes all threads to make the same procedure calls from the same place, and breakpoints are barriers causing the threads to stop at the same place.

Aardvark's alignment process starts at the outermost stack frames of the physical threads (the ones near the Fortran PROGRAM unit) and then progressively examines the callees (toward where the program stopped). Starting from the innermost frames is an error-prone approach. If the innermost frames are in different functions, Aardvark might construct a multiframe when the frames are actually misaligned because the physical stacks have different depths. As discussed in the section on twinning, depth is not a reliable alignment mechanism either. Starting at the outermost frames follows the temporal order of calls and also correctly handles recursive procedures. The disadvantage of starting at the outermost frames is that each physical thread's entire stack must be determined before the logical stack can be constructed. Usually the programmer only wants the innermost few frames, so time delays in the construction process can reduce the ease of use of the debugger.¹²

Much of the time, the physical stack frames are at the same place because the SPMD nature of HPF causes the physical threads to have the same control

flow. When a procedure is called, each thread executes the call and executes it from the same place. A logical breakpoint is reached when the physical threads are stopped at the same place at the corresponding physical breakpoints. These cases lead to synchronized frames. The most common cause of an unsynchronized frame is interrupting the program during a computation. Even in this case, the divergence is usually not very large. One reason for a multiframe is the interruption of the program while it is communicating data between processes. In this case, the code paths can easily diverge, depending on which threads are sending, which are receiving, and how much data needs to be moved. Scalar frames are created because of the semantic flow of the program: the main program unit is written in either a serial language or an HPF procedure called an `EXTRINSIC(HPF_SERIAL)` procedure type.

The result of the alignment algorithm is a set of frames collected into a call stack. The normal navigation operations (e.g., up and down) apply. Variable lookup and expression evaluation work as expected, also. Variable lookup works best for synchronized frames and, for HPF, works for unsynchronized frames as well. For multiframes, variable lookup generally fails because a variable name `VAR` may resolve to different program variables in the corresponding physical frames or may not resolve to anything at all in some frames. This failure is not because of a lack of information from the compiler but rather because multiframes are generally not a context in which a string `VAR` has a well-defined semantic.

Experience to date suggests that multiframes are of interest largely to the people developing the run-time support for data motion. Nevertheless, the point of transition from synchronized to unsynchronized to multi tells the user where control flows diverged, and this information can be very valuable.

Narrowing Focus

Using the previously mentioned techniques sometimes results in a cluttered view of the state of the entire program and difficulty in finding relevant information. Aardvark provides two ways to help. The first aid is a Boolean *focus* mask that selects a subset of the processes and then re-applies the logical algorithms. For properly chosen subsets, this can turn a stack trace with many multiframes into a stack trace with synchronized frames. A narrowed focus can also look behind the scenes of the twinning mechanism described in the next paragraph. The second aid is to view a single physical process in isolation, effectively turning off the parallel debugging algorithms. This technique is useful for debugging `EXTRINSIC(HPF_LOCAL)` and `EXTRINSIC(HPF_SERIAL)` procedures. The ability to retrieve the physical processes from a logical process is the major item that enables viewing a process in isolation; as mentioned before, physical entities are first-class objects.

Twinning

DIGITAL's HPF provides a feature called twinning in which a scalar procedure can call a parallel HPF procedure. This allows, for example, the main program consisting of a user interface and associated graphics to be written in C and have Fortran/HPF do the numerical computations. The feature is called twinning because each Fortran procedure is compiled twice. The *scalar twin* is called from scalar code on a designated process. Its duties include instructing the other processes to call the scalar twin, distributing its scalar arguments according to the HPF directives, calling the *HPF twin* from all processes, distributing the parallel data back onto the designated process after the HPF twin returns, and finally returning to its caller. The HPF twin is called on all processes with distributed data and executes the user-supplied body of the procedure.

At the run-time level, the program's entry point is normally called on a designated process (process 0), and the other processes enter a dispatch loop waiting for instructions. Conceptually, such a program starts in scalar mode and at some point transitions into parallel mode. An HPF debugger should represent this transition. Aardvark accomplishes this by having knowledge of the HPF twinning mechanism. When it notices physical threads entering the dispatch loop, Aardvark creates a scalar logical frame corresponding to the physical frame on process 0. It then processes procedure calls on process 0 only, creating more scalar frames, until it notices that the program transitions from scalar to parallel. This transition happens when all processes call the same (scalar twin) procedure: process 0 does so as a result of normal procedure calls; processes other than 0 do so from their dispatch loops. At this point, a logical frame is constructed that will likely be synchronized, and the frame processing described previously applies. The result is the one desired: a scalar program transitions to a parallel one.

DIGITAL's HPF goes a step further: it allows `EXTRINSIC(HPF_SERIAL)` procedures to call HPF code by means of the twinning mechanism. When an `EXTRINSIC(HPF_SERIAL)` procedure is called, processes other than 0 call the dispatch loop. When the scalar code on process 0 calls the scalar twin, the other processes are in the necessary dispatch loop. Aardvark tracks these calls in the same way as in the previous paragraph, noticing that processes other than 0 have called the dispatch loop and eventually call a scalar twin.

User Interface Implications

User interfaces and other clients must be keenly aware of the concept of logical frames and the different types of logical frames. Depending on the type of frame, some operations, such as obtaining the function name

or the line number, may not be valid. Nevertheless, a user interface can provide useful information about the state of the program. The program used for the following discussion has a serial user interface written in C and uses twinning to call a parallel HPF procedure named `HPF_FILL_IN_DATA` (see Figure 3). The HPF procedure uses a function named `MANDEL_VAL` as a non-data-parallel computational kernel. The program was run on five processes. (Twinning is a DIGITAL extension. Most HPF programs are written entirely in HPF. This example, which uses twinning, was chosen to demonstrate the broader problem.)

Figure 4 shows the program interrupted during computation. Line 2 of the figure contains a single function name, `MANDEL_VAL`. Line 3 contains the function's source file name but lists five line numbers, implying that this is an unsynchronized frame. In fact, the user interface discovered that Aardvark created an unsynchronized logical frame. Instead of trying to get a single line number, the user interface retrieved the set of line numbers and presented them. In lines 4 through 10, the user interface also presented the range of source lines encompassing the lines of all the component processes. This user interface's `up` command (line 21) navigates to the calling frame. In this example, the frame is synchronized, causing the user interface to present the function's source file and single line number (line 26), followed by the single source file line (line 27).

Figure 5 shows a summary of the program's call stack when it was interrupted during computation. The summary is a mix of unsynchronized, synchronized, and scalar frames. Frame #0 (line 2) is unsynchronized, and the various line numbers are presented. Its caller, frame #1 (line 3), is synchronized with a single line number. All this is consistent with the previous discussion. Frame #1 is the HPF twin of the scalar twin in frame #2. The scalar twin of frame #2 is expected to be called by scalar code, confirmed by frames #3 and #4. Frame #5 is part of the twinning mechanism; process 0 is at line 499, while the other processors are all at line 506.

Narrowing the focus to exclude process 0 shows a different call stack summary (lines 9 through 16 of Figure 5). The new frame #0 (line 11) continues to be unsynchronized, but all the other frames are synchronized. The twinning dispatch loop (line 14) replaces the scalar frames of the global focus (lines 5 and 6). This replacement causes the new call stack, corresponding more closely to the physical threads, to have fewer frames than the global call stack.

Interrupting the program while idle within the user interface shows more about twinning and also shows a multiframe (see Figure 6). Most of the frames are scalar except for the twinning mechanism (frame #7, line 9) and the initial run-time frame (frame #8, line 10). Narrowing the focus to exclude process 0 shows the twinning mechanism while waiting. The twinning


```

subroutine hpf_fill_in_data(target, w, h, ccr, cci, cstep, nmin, nmax)
  integer, intent(in)      :: w, h
  byte, intent(out)       :: target(w,h)
  real*8, intent(in)      :: ccr, cci, cstep
  integer, intent(in)     :: nmin, nmax
!hpf$ distribute target(*, cyclic)

  integer :: cx, cy
  cx = w/2
  cy = h/2

  forall(ix = 1:w, iy = 1:h)
    target(ix,iy) = mandel_val(CMPLX(ccr + ((ix-cx)*cstep),
                                     cci + ((iy-cx)*cstep),
                                     KIND=KIND(0.0D0)),
                               nmin, nmax)

contains

  pure byte function mandel_val(x, nmin, nmax)
  complex(KIND=KIND(0.0D0)), intent(in) :: x
  integer, intent(in)                  :: nmin, nmax

  integer                               :: n

  real(kind=KIND(0.0D0)) :: xorgr, xorgi, xr, xi, xr2, xi2, rad2
  logical                 :: keepgoing

  n = -1
  xorgr = REAL(x)
  xorgi = AIMAG(x)
  xr = xorgr
  xi = xorgi

  do
    n = n + 1
    xr2 = xr*xr
    xi2 = xi*xi
    xi = 2*(xr*xi) + xorgi
    keepgoing = n < nmax
    rad2 = xr2 + xi2
    xr = xr2 - xi2 + xorgr
    if (keepgoing .AND. (rad2 <= 4.0)) cycle
    exit
  end do

  if (n >= nmax) then
    mandel_val = nmax-nmin
  else
    mandel_val = MOD(n, nmax-nmin)
  end if

end function mandel_val

end subroutine hpf_fill_in_data

```

Figure 3

HPF_FILL_IN_DATA Procedure (Source Code for Figures 4 and 5)

mechanism at frames #5 and #6 (lines 23 and 24) is similar to the mechanism at frames #3 and #4 (lines 14 and 15) of Figure 5. In Figure 6, they do not call a scalar twin but rather call the messaging library to receive instructions from process 0. The messaging library, however, is often not synchronized among the peers, and frame #2 (line 15) shows a multiframe. This user interface shows a multiframe as a collection of one-line summaries of the physical frames (lines 16 through 20).

Examining HPF Data

Examining data generally involves determining where the data is stored, fetching the data, and then presenting it. HPF presents difficulties in all three areas. Determining where data is stored requires rich and flexible data-location representations and associated operations. Fetching small amounts of data can be done naively, one element at a time, but for large amounts of data, e.g., data used for visualization, faster

```

1      Thread is interrupted.
2      #0: MANDEL_VAL(X = <<differring COMPLEX(KIND=8) values>>, NMIN = 255, NMAX = 510)
3      at mb.hpf.f90:45,44,45,40,39
4          39          xr2 = xr*xr
5          40          xi2 = xi*xi
6          41          xi = 2*(xr*xi) + xorgi
7          42          keepgoing = n < nmax
8          43          rad2 = xr2 + xi2
9          44          xr = xr2 - xi2 + xorgr
10         45          if (keepgoing .AND. (rad2 <= 4.0)) cycle
11
12     debugger> print x
13         $1 = #<DIFFERING-VALUES
14             #0: (-0.66200000000000003,-0.114)
15             #1: (-0.59599999999999997,-0.113)
16             #2: (-0.65300000000000002,-0.112)
17             #3: (-0.93799999999999994,-0.10600000000000001)
18             #4: (-0.56600000000000006,-0.11)
19         >
20
21     debugger> up
22         #1: hpf$hpf_fill_in_data_(TARGET = <<non-atomic= INTEGER(KIND=1), DIMENSION(1:400, 1:400)>>,
23             W = 400, H = 400,
24             CCR = -0.76000000000000001, CCI = -0.02, CSTEP = 0.001,
25             NMIN = 255, NMAX = 510)
26         at mb.hpf.f90:14
27             14          forall(ix = 1:w, iy = 1:h)          &
28
29     debugger> info address target
30         #<locative_to_hpf_section 5 peers of type INTEGER(KIND=1), DIMENSION(1:400,1:400) >
31         type          INTEGER(KIND=1), DIMENSION(1:400,1:400)
32         phys_count    5
33         addresses
34             0:          0x11fff71f0
35             1:          0x11fff7000
36             2:          0x11fff7000
37             3:          0x11fff7000
38             4:          0x11fff7000
39         arank          2
40         trank          2
41         diminfos      dlower  dupper  plower  pupper  ... dist_k
42             0          1        400      1        400 ... collap
43             1          1        400      1         80 ... cyclic
44
45     debugger> info address target(100,100)
46         #<locative_in_peer in peer 4 ... >
47         type          INTEGER(KIND=1)
48         peernum       4
49         locative      #<locative_to_memory at dmem address 0x11fff8e13 of type INTEGER(KIND=1) >

```

Figure 4
Program Interrupted during Computation

methods are needed. Displaying data can usually use the techniques inherited from the underlying Fortran 90 support, but some mechanism and corresponding user interface handling is needed when replicated data has different values.

Data-Location Representations

Representing where data is stored is relatively easy to do in languages such as C and Fortran 77: the data is in a register or in a contiguous block of memory. Fortran 90 introduced assumed-shape and deferred-shape arrays,¹³ where successive array elements are not necessarily adjacent in memory. HPF allows the array

to be distributed so that successive array elements are not necessarily stored in a single process or address space. These lead to data that can be stored discontinuously in memory as well as in different memories.

Fortran 90 also introduced array sections, vector-valued subscripts, and field-of-array operations,¹⁴ which further complicate the notion of where data is stored. Although evaluating an expression involving an array can be accomplished by reading the entire array and performing the operations in the debugger, this approach is inefficient, especially for a result that is sparse compared to the entire array. A standard technique is to perform address arithmetic and fetch only


```

1  debugger> where
2      > #0(unsync) MANDEL_VAL at mb.hpf.f90:45,44,45,40,39
3      #1(synchr) hpf$hpf_fill_in_data_ at mb.hpf.f90:14
4      #2(synchr) hpf_fill_in_data_ at mb.hpf.f90:1
5      #3(scalar) mb_fill_in_data at mb.hpf.c:45
6      #4(scalar) main at mb.c:421
7      #5(unsync) _hpf_twinning_main_usurper at [...]/Libhpf/hpf_twin.c:499,506,506,506,506
8      #6(synchr) __start at [...]/alpha/crt0.s:361
9  debugger> focus 1-4
10 debugger> where
11     > #0(unsync) MANDEL_VAL at mb.hpf.f90:<none>,44,45,40,39
12     #1(synchr) hpf$hpf_fill_in_data_ at mb.hpf.f90:14
13     #2(synchr) hpf_fill_in_data_ at mb.hpf.f90:1
14     #3(synchr) _hpf_non_peer_0_to_dispatch_loop at [...]/Libhpf/hpf_twin.c:575
15     #4(synchr) _hpf_twinning_main_usurper at [...]/Libhpf/hpf_twin.c:506
16     #5(synchr) __start at [...]/alpha/crt0.s:361

```

Figure 5
Control Flow of a Twinned Program Interrupted during Computation

```

1  debugger> where
2      > #0(scalar) __poll at <<unknown name>>:41
3      #1(scalar) <<disembodied>> at <<unknown>>:459
4      #2(scalar) _XRead at <<unknown name>>:1110
5      #3(scalar) _XReadEvents at <<unknown name>>:950
6      #4(scalar) XNextEvent at <<unknown name>>:37
7      #5(scalar) HandleXInput at mb.c:58
8      #6(scalar) main at mb.c:452
9      #7(unsync) _hpf_twinning_main_usurper at [...]/Libhpf/hpf_twin.c:499,506,506,506,506
10     #8(synchr) __start at [...]/alpha/crt0.s:361
11 debugger> focus 1-4
12 debugger> where
13     > #0(unsync) __select at <<unknown name>>:<none>,41,<none>,41,41
14     #1(unsync) TCP_MsgRead at [...]/Libhpf/msgtcp.c:<none>,1057,<none>,1057,1057
15     #2(multi)
16         <none>
17         _TCP_RecvAvail at [...]/Libhpf/msgtcp.c:1400
18         swtch_pri at <<unknown name>>:118
19         _TCP_RecvAvail at [...]/Libhpf/msgtcp.c:1400
20         _TCP_RecvAvail at [...]/Libhpf/msgtcp.c:1400
21     #3(unsync) _hpf_Recv at [...]/Libhpf/msgmsg.c:<none>,434,488,434,434
22     #4(synchr) _hpf_RecvDir at [...]/Libhpf/msgmsg.c:509
23     #5(synchr) _hpf_non_peer_0_to_dispatch_loop at [...]/Libhpf/hpf_twin.c:563
24     #6(synchr) _hpf_twinning_main_usurper at [...]/Libhpf/hpf_twin.c:506
25     #7(synchr) __start at [...]/alpha/crt0.s:361

```

Figure 6
Control Flow of a Twinned Program Interrupted While Idle in Scalar Mode

the actual data result at the end of the operation. The usual notion of an address, however, is that it describes the start of a contiguous block of memory.

Richer data-location representations are necessary. These representations can include registers and contiguous memory, but they also need to include discontinuous memory and data distributed among multiple processes. The representations should also include the results of expressions involving array sections, vector-valued subscripts, and field-of-array operations, thereby extending address arithmetic to data-location arithmetic. Aardvark defines a *locative* base class that has a virtual method to fetch the data. A variety of

derived classes implement the data-location representations needed.

DIGITAL's Fortran 90 implements assumed-shape and deferred-shape arrays using descriptors that contain run-time information about the memory address of the first element, the bounds, and per-dimension inter-element spacing.¹⁵ Aardvark models these types of arrays almost directly with a derivation of the locative class that holds the same information as the descriptor. Performing expression operations is relatively easy. An array section expression adjusts the bounds and the inter-element spacing. A field-of-array operation offsets the address to point to the compo-

ment field and changes the element type to that of the field. A vector-valued subscript expression requires additional support; the representation for each dimension can be a vector of memory offsets instead of bounds and inter-element spacing.

All arrays in HPF are qualified, explicitly or implicitly, with `ALIGN`, `TEMPLATE`, and `DISTRIBUTE` directives.¹⁶ DIGITAL's HPF uses a superset of the Fortran 90 descriptors to encode this information. Aardvark models HPF arrays with another derivation of the locative class that holds information similar to the HPF descriptors. The most pronounced difference is that Aardvark uses a single locative to encode the descriptors from the set of processes. Aardvark knows that the local memory addresses are potentially different on each process and maintains them as a vector, but currently assumes that processor-independent information is the same on all processes and only encodes that information once.

Referring again to Figure 4, line 22 shows that the argument `TARGET` is an array, and line 29 is a request for information about the location of its data. (See also Figure 3 for the full source, including the declaration and distribution of `TARGET`.) Figure 4, line 32 shows that there are five processes, and lines 34 through 38 show the base address within each process. The addresses for processes 1 through 4 happen to be the same, but the address for process 0 is different. Lines 39 and 40 show that the rank of the array (`rank`) and the rank of the template (`trank`) are both 2. Lines 42 and 43 show the dimension information for the array. The declared bounds are `1:400, 1:400`, but the local physical bounds are `1:400, 1:80` and the distribution is `(* , CYCLIC)`. This is all accurate; distributing the second dimension on five processes causes the local physical size for that dimension (80) to be one-fifth the declared bound (400).

Performing expression operations on HPF-based locatives is more involved than for Fortran 90. Processing a scalar subscript not only offsets the base memory address but also restricts the set of processors determined by the dimension's distribution information. Processing a subscript triplet, e.g., `from:to:stride`, involves adjusting the declared bounds and the alignment; it does not adjust the template or the physical layout. As in Fortran 90, processing a vector-valued subscript in HPF requires the locative to represent the effect of the vector. For HPF, the representation is pairs of memory offsets and processor set restrictions. Processing a field-of-array operation adjusts the element type and offsets each memory address.

When selecting a single array element by providing scalar subscripts, another type of locative is useful. This locative describes on which process the data is stored and a locative relative to that selected process. For example, line 45 of Figure 4 requests the location information of a single array element. The result

shows that it is on process 4 at the memory address indicated by the contained locative.

Fetching HPF Data

As just mentioned, locatives provide a method to fetch the data described by the locative. For a locative that describes a single distributed array element (e.g., Figure 4, lines 45 through 49), the method extracts the appropriate physical thread from the logical thread and uses the contained locative to fetch the data relative to the extracted physical thread. For a locative that describes an HPF array, Aardvark currently iterates over the valid subscript space, determines the physical process number and memory offset for each element, and fetches the element from the selected physical process. For small numbers of elements, on the order of a few dozen, this technique has acceptable performance. For large numbers of elements, e.g., for visualization or reduction operations, the cumulative processing and communication delay to retrieve each individual element is unacceptable. This performance issue also exists for locatives that describe discontinuous Fortran 90 arrays. The threshold is higher because there is no computation to determine the process for an element, and the process is usually local rather than remote, eliminating communication delays.

The primary bottleneck is issuing many small data retrieval requests to each (remote) process. This involves many communication delays and many delays related to retrieving each element. What is needed is to issue a smaller number of larger requests. The smaller number reduces the number of communication transactions and associated delays. Larger requests allow analysis of a request to make more efficient use of the operating system's mechanisms to access process memory. For example, a sufficiently dense request can read the encompassing memory in a single call to the operating system and then extract the desired elements once the data is within the debugger.

Although not implemented, the best solution, in my opinion, is to provide a "read (multidimensional) memory section" method on a process in addition to the common "read (contiguous) memory" method. If the process is remote, as it usually is with HPF, the method would be forwarded to a remote debug server controlling the remote process. The implementation of the method that interacts with the operating system would know the trade-offs to determine how to analyze the request for maximum efficiency.

Converting a locative describing a Fortran 90 array section to a "read memory section" method should be easy: they represent nearly the same thing. For a locative that describes a distributed HPF array, Aardvark would need to build (physical) memory section descriptions for each physical process. This can be done by iterating over the physical processes and building the memory section for each process. It is

also possible to build the memory sections for all the processes during a single pass through the locative, but the performance gains may not be large enough to warrant the added complexity.

Differing Values

Using HPF to distribute an array often partitions its elements among the processes. Scalars, however, are generally replicated and may be expected to have the same value in each process. There are cases, though, where seemingly replicated scalars may not have the same value. `DO` loops that do not require data to be communicated between processes do not have synchronization points and can become out of phase, resulting in their indexes and other privatized variables having different values. Functions called within a `FORALL` construct often run independently of each other, causing the arguments and local variables in one process to be different from those in another. A debugger should be aware that values might differ and adjust the presentation of such values accordingly.

Aardvark's approach is to define a new kind of value object called *differing values* to represent a value from a semantically single source that does not have the same value from all its actual sources. A user interface can detect this kind of value and display it in different ways, for example, based on context and/or the size of the data.

Referring again to Figure 4, the program was interrupted while each process was executing the function `MANDEL_VAL` called within a `FORALL`. Line 2 shows that the argument `x` was determined to have differing values. This user interface does not show all the values at this point; a large number of values could distract the user from the current objective of discovering where the process stopped. Instead, it shows an indication that the values are different along with the type of the variable. Notice that the other two arguments, `NMIN` and `NMAX`, are presented as integers; they have the same value in all processes. Line 12 requests to see the value of `x`. Line 13 again shows that the values are different, and lines 14 through 18 show the process number and the value from the process.

To build a differing values object, Aardvark reads the values for a replicated scalar from each process. If all the values are bit-wise equal, they are considered to be the same and a standard (single) value object is returned. Otherwise, a differing values object is constructed from the several values. For numeric data, this approach seems reasonable. If the value of a scalar integer variable `INTVAR` is 4 on all the processes, then 4 is a reasonable (single) value for `INTVAR`. If the value of `INTVAR` is 4 on some processors and 5 on others, no single value is reasonable. For nonnumeric data and pointers, there is the possibility of false positives and false negatives. The ideal for user-defined types is to compare the fields recursively. Pointers that are seman-

tically the same can point to targets located at different memory addresses for unrelated reasons, leading to different memory address values and therefore a false positive. To correctly dereference the pointers, though, Aardvark needs the different memory address values. In short, it is reasonable to test numeric data and create a single value object or a differing values object, and it appears reasonable to do the same for nonnumeric data, despite the possibility of a technically false kind of value object.

Currently, differing values do not participate in arithmetic. That is, the expression `INTVAR.LT.5` is valid if `INTVAR` is a single value but causes an error to be signaled if `INTVAR` is a differing value. Many cases could be made to work, but some cases defy resolution. In the `INTVAR.LT.5` case, if all values of `INTVAR` are less than 5 or all are greater than or equal to 5, then it is reasonable to collapse the result into a single value, `.TRUE.` or `.FALSE.`, respectively. If some values are less than 5 and some are not, it also seems reasonable to create a differing values object that holds the differing results. What if `INTVAR.LT.5` is used as the condition of a breakpoint and some values of `INTVAR` are less than 5 and some are not? The breakpoint should probably cause the process (and all the physical processes) to remain stopped. It is unclear whether arithmetic on differing values would be useful to users or if it would lead to more confusion than it would clear up.

Unmet Challenges

HPF presents a variety of challenges that Aardvark does not yet address. Some of these challenges are not in common practice, giving them low priority. Some are recent with HPF Version 2.0 and are being used with increasing frequency. Some of the challenges, for example, a debugger-initiated call of an HPF procedure, are tedious to address correctly.

Mapped Scalars

It is possible to distribute a scalar so that the scalar is not fully replicated.¹⁷ The compiler would need to emit sufficient debugging information, which would probably be a virtual array descriptor with an array rank of 0 and a nonzero template rank. Aardvark would probably model it using its existing locative for HPF arrays, also with an array rank of 0 and appropriate template information.

Replicated Arrays

Unless otherwise specified, DIGITAL's HPF compiler replicates arrays. It is possible to replicate arrays explicitly and to align arrays (and scalars) so that they are partially replicated. Currently, Aardvark does not detect a replicated array, despite the symbol table or run-time descriptor indicating that it is replicated. As a result, Aardvark determines a single process from which to fetch each array element. For fully replicated

arrays, Aardvark should read the array from each process and process them with the differing values algorithms. Correctly processing arrays that are partially replicated is not as easy as processing unreplicated or fully replicated arrays. If the odd columns are on processes 0 and 1, while the even columns are on processes 2 and 3, no single process contains the entire array. The differing values object would need to be extended to index the values by a processor set rather than a single process.

Update of Distributed and Replicated Objects

Aardvark currently supports limited modification of data. It supports updating a scalar object (scalar variable or single array element) with a scalar value, even if the object is distributed or replicated. Even this can be incorrect at times. Assigning a scalar value to a replicated object sets each copy, which is undesirable if the object has differing values. Assigning a value that is a differing values object is not supported. More importantly (and more subtly), Aardvark is not aware of shadow or halo copies of data that are stored in multiple processes, so updating a distributed object updates only the primary location.

Distributed Array Pointers

HPF Version 2.0 allows array pointers in user-defined types to be distributed and allows fully replicated arrays of such types. For example, in

```

type utype
  integer, pointer :: compptr(:)
  !hpf$ distribute compptr(block)
end type

type (utype) :: scalar, array(20)

```

the component field `compptr` is a distributed array pointer. Aardvark does not currently process the array descriptor(s) for `scalar%compptr` at the right place and as a result does not recognize the expression as an array. As mentioned earlier, Aardvark reads a replicated array element from a single process. To process `array(1)%compptr`, all the descriptors are needed, e.g., for the base memory addresses in the physical processes. The use of this relatively new construct is growing rapidly, elevating the importance of being supported by debuggers.

Ensuring a Consistent View

A program can have its physical threads stop at the same place but be in different iterations of a loop. Aardvark mistakenly presents this state as synchronized and presents data as if it were consistent. This is what is happening in Figures 4 and 5; `hpf$hpf_fill_in_data` (frame #1) is in different iterations of the `FORALL`. With compiler assistance, it is possible to annotate each thread's location with iteration counts in addition to traditional line numbers.¹⁸

The resulting set of locations can be compared to a location in the conceptually serial program to determine which threads have already reached (and perhaps passed) the serial location and which have not yet reached it. A debugger could automatically, or under user control, advance each thread to a consistent serial location. For now, Aardvark's differing values mechanism is the clue to the user that program state might not be consistent.

Calling an HPF Procedure

Having a debugger initiate a call to a Fortran 90 procedure is difficult in the general case. One difficulty is that copy-in/copy-out (making a temporary copy of array arguments and copying the temporary back to its origin after the call returns) may be necessary. HPF adds two more difficulties. First, the data may need to be redistributed, which amounts to a distributed copy-in/copy-out and entails a lot of tedious (but hopefully straightforward) bookkeeping. Second, an HPF thread's state is much more complex than a collection of physical thread states. When a debugger initiates a uniprocessor procedure call, it generally saves the registers, sets up the registers and stack according to the calling convention, lets the process run until the call returns, extracts the result, and finally restores the registers. The registers are generally the state that is preserved across a debugger-initiated procedure call. For HPF, and in general for other paradigms that use message passing, it may be necessary to preserve the run-time state of the messaging subsystem in each process. This preservation probably amounts to making uniprocessor calls to messaging-supplied save/restore entry points, allowing the messaging subsystem to define what its state is and how it should be saved and restored. Although logical entities would be used to coordinate the physical details, this is a lot of work and has not been prototyped.

Related Work

DIGITAL's representative to the first meeting of the HPF User Group reported a general lament among users about the lack of debugger support.^{19,20} Browsing the World Wide Web reveals little on the topic of HPF debugging, although some efforts have provided various degrees of sophistication.

Multiple Serial Debuggers

A simplistic approach to debugging support is to start a traditional serial debugger on each component process, perhaps providing a separate window for each and providing some command broadcast capability. Although this approach provides basic debugging, it does not address any of the interesting challenges of HPF debugging.

Prism

The Prism debugger (versions dating from 1992), formerly from Thinking Machines Corporation, provides debugging support for CM Fortran.^{21,22} The run-time model of CM Fortran is essentially single instruction, multiple data (SIMD), which considerably simplifies managing the program. The program gets compiled into an executable that broadcasts macroinstructions to the parallel machine, even on the CM-5 synchronized multiple instruction, multiple data (MIMD) machine. Prism primarily debugs the single program doing the broadcasting. Therefore, operations such as starting, stopping, and setting breakpoints can use the traditional uniprocessor debugging techniques. Prism is aware of distributed data. When visualizing a distributed array, however, it presents each process's local portion and conceptually augments the rank of the array to include a process axis. For example, a two-dimensional 400×400 array distributed (`* , CYCLIC`) on five processes is presented as a $400 \times 80 \times 5$ array. For explicit message sending programs, Prism controls the target processes and provides a "where graph," which has some of the visual cues that Aardvark's logical frames provide.

TotalView

Recent (1997) versions of the TotalView debugger, from Dolphin Interconnect Solutions, Inc., provide some support for the HPF compiler from The Portland Group, Inc.^{23,24} TotalView provides "process groups," which are treated more like sets for set-wide operations than like a synthesis into a single logical entity. As a result, no unified view of the call stacks exists. TotalView can "dive" into a distributed HPF array and present it as a single array in terms of the original source. Distributed data is not currently integrated into the expression system, however, so a conditional breakpoint such as `A(3,4).LT.5` does not work. TotalView is being actively developed; future versions will likely provide more complete support for HPF.

Applicability to Other Areas

Many of the techniques that Aardvark incorporates can apply to other areas, including the single program, multiple data (SPMD) paradigm, debugging optimized code, and interpreted languages.

Single Program, Multiple Data

Logical entities can be used to manage and examine programs that use the SPMD paradigm. This is true for process-level SPMD, which is commonly used with explicit message sending such as MPI,^{5,6} and for thread-level SPMD such as directed decomposition.²⁵⁻²⁷ Aardvark's twinning algorithms can be used in both cases. Process-level SPMD is similar to

DIGITAL's HPF; the equivalent of twinning requires a stylistic way of coding and declaring a dispatch loop. Thread-level SPMD usually has a pool of threads waiting in a dispatch loop, requiring Aardvark to know some mechanics of the run-time support.

The differing values mechanism can apply to data in SPMD paradigms. DIGITAL's recent introduction of Thread Local Storage (TLS),²⁸ modeled on the Thread Local Storage facility of Microsoft Visual C++²⁹ with similarities to `TASKCOMMON` of Cray Fortran,³⁰ provides another source of the same variable having potentially differing values in different thread contexts.

Debugging Optimized Code

Aardvark's flexible locative subsystem and its awareness of nonsingular values (i.e., differing values) can be the basis for "split-lifetime variables." In optimized code, a variable can have several simultaneous lifetimes (e.g., the result of loop unrolling) or no active lifetime (e.g., between a usage and the next assignment). New derivations of the locative class can describe the multiple homes or the nonexistent home of a variable. Fetching by means of such a locative creates new kinds of values that hold all the values or an indication that there is no value. User interfaces become aware of these new kinds of values in ways similar to their awareness of differing values.

Aardvark's method of asking a thread for a single-stepping run reason and empowering the reason to accomplish its mission can be the basis for single stepping optimized code. Optimized code generally interleaves instructions from different source lines, rendering the standard "execute instructions until the source line number changes" method of single stepping useless. If instead the compiler emits information about the semantic events of a source line, Aardvark can construct a single-stepping run reason based on semantic events rather than line numbers. Single stepping an optimized HPF program immediately reaps the benefits since logical stepping is built on physical stepping.

Interpreted Languages

Logical entities can be used to support debugging interpreted languages such as Java³¹ and Tcl.³² In this case, the physical process is the operating system's process (the Java Virtual Machine or the Tcl interpreter), and the logical process is the user-level view of the program. A logical stack frame encodes a procedure call of the source language. This is accomplished by examining virtual stack information in physical memory and/or by examining physical stack frames, depending on how the interpreter is implemented. Variable lookup within the context of a logical frame would use the interpreter-managed symbol tables rather than the symbol tables of the physical process.

Summary

HPF presents a variety of challenges to a debugger, including controlling the program, examining its call stack, and examining its data, and user interface implications in each area. The concept of logical entities can be used to manage much of the control complexity, and a rich data-location model can manage HPF arrays and expressions involving arrays. Many of these ideas can apply to other debugging situations. On the surface, debugging HPF can appear to be a daunting task. Aardvark breaks down the task into pieces and attacks them using powerful extensions to familiar ideas.

Acknowledgments

I am grateful to Ed Benson and Jonathan Harris for their unwavering support of my work on Aardvark. I also thank Jonathan's HPF compiler team and Ed's Parallel Software Environment run-time team for providing the compiler and run-time products that allowed me to test my ideas.

References and Notes

1. *Programming Language Fortran 90*, ANSI X3.198-1992 (New York, N.Y.: American National Standards Institute, 1992).
2. J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener, *Fortran 90 Handbook* (New York, N.Y.: McGraw-Hill, 1992).
3. High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*. This specification is available by anonymous ftp from softlib.rice.edu in the directory pub/HPF. Version 2.0 is the file hpf-v20.ps.gz.
4. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel, *The High Performance Fortran Handbook* (Cambridge, Mass.: MIT Press, 1994).
5. MPI Forum, "MPI-2: Extensions to the Message-Passing Interface," available at <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html> or via the Forum's documentation page <http://www.mpi-forum.org/docs/docs.html>.
6. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference* (Cambridge, Mass.: MIT Press, 1995).
7. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI* (Cambridge, Mass.: MIT Press, 1994).
8. J. Harris et al., "Compiling High Performance Fortran for Distributed-memory Systems," *Digital Technical Journal*, vol. 7, no. 3 (1995): 5-23.
9. E. Benson et al., "Design of Digital's Parallel Software Environment," *Digital Technical Journal*, vol. 7, no. 3 (1995): 24-38.

10. It is possible to always use logical entities, but sometimes it is easier to work with the building blocks when additional structure might be cumbersome. In a similar vein, Fortran could eliminate scalars in favor of arrays of rank 0, but Fortran chooses to retain scalars because of their ease of use.
11. In this policy, *nasty* and *nice* are generic names for categories. Which particular stop reasons fall into which category is a separate design question. Once the category is determined, the policy presented can be performed.
12. Debuggers often build a (physical) call stack from innermost frame to outermost frame, interleaving construction with presentation. The interleaving gives the appearance of progress even if there are occasional delays between frames. The total time required to reach the outermost frames of each physical thread, which must occur before construction of a logical call stack can begin and before any presentation is possible, could be noticeable to the user.
13. An assumed shape array is a procedure array argument in which each dimension optionally specifies the lower bound and does not specify the upper bound. For example,

```
REAL :: ARRAY_ARG_2D(:,4:)
```

A deferred shape array has either the `ALLOCATABLE` or `POINTER` attribute, specifies neither the lower nor upper bound, and often contains local procedure variables or module data. For example,

```
REAL, ALLOCATABLE :: ALLOC_1D(:)
REAL, POINTER :: PTR_3D(:, :, :)
```

14. An array section occurs when some subscript specifies more than one element. This can be done with a subscript-triplet, which optionally specifies the lower and upper extents and a stride, and/or with an integer vector, for example,

```
ARRAY_3D(ROW1:ROWN , COL1::COL_STRIDE , PLANE_VEC)
```

A field-of-array operation specifies the array formed by a field of each structure element of an array, for example,

```
TYPE (TREE) :: TREES(NTRESS)
REAL :: TREE_HEIGHTS(NTRESS)
TREE_HEIGHTS = TREES%HEIGHT
```

In general, each of these specifies discontinuous memory.

15. "DEC Fortran 90 Descriptor Format," *DEC Fortran 90 User Manual* (Maynard, Mass.: Digital Equipment Corporation, June 1994).
16. The HPF array descriptor for the variable `ARRAY` in the HPF fragment

```
!HPF$ TEMPLATE T(NROWS,NCOLS)
!HPF$ DISTRIBUTE T(CYCLIC,BLOCK)
REAL :: ARRAY(NCOLS/2,NROWS)
!HPF$ ALIGN ARRAY(I,J) WITH T(J,I*2-1)
```

contains components corresponding to each of the ALIGN, TEMPLATE, and DISTRIBUTE directives. Often an array is distributed directly, causing the ALIGN and TEMPLATE directives to be implicit, for example,

```
REAL :: MATRIX(NROWS,NCOLS)
!HPF$ DISTRIBUTE MATRIX(BLOCK,BLOCK)
```

17. The variable SCALAR in

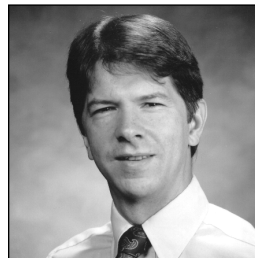
```
!HPF$ TEMPLATE T(4,4)
!HPF$ DISTRIBUTE T(CYCLIC,CYCLIC)
!HPF$ ALIGN SCALAR WITH T(*,2)
```

is partially replicated and will be stored on the same processors that the logical second column of the template T is stored.

18. R. Cohn, *Source-Level Debugging of Automatically Parallelized Programs*, Ph.D. Thesis, Carnegie Mellon University (October 1992).
19. HPF User Group, February 23–26, 1997, Santa Fe, New Mexico. Information about the meeting is available at <http://www.lanl.gov/HPF/>.
20. In a trip report, DIGITAL’s representative Carl Offner reported the following: “Many people complained about the lack of good debugging support for HPF. Those who had seen our [Aardvark-based] debugger liked it a lot.... [An industrial HPF user] complained emphatically about the lack of good debugging support.... [Another industrial HPF user’s] biggest concern is the lack of good debugging facilities.”
21. *Prism User’s Guide* (Cambridge, Mass.: Thinking Machines Corporation, 1992).
22. *CM Fortran Programming Guide* (Cambridge, Mass.: Thinking Machines Corporation, 1992).
23. *TotalView: User’s Guide* (Dolphin Interconnect Solutions, Inc., 1997). This guide is available via anonymous ftp from <ftp.dolphinics.com> in the totalview/DOCUMENTATION directory. At the time of writing the file is TV-3.7.5-USERS-MANUAL.ps.Z.
24. *PGHPF User’s Guide* (Wilsonville, Ore.: The Portland Group, Inc., 1997).
25. *KAP Fortran 90 for Digital UNIX* (Maynard, Mass.: Digital Equipment Corporation, October 1995).
26. “Fine-Tuning Power Fortran,” *MIPSpro POWER Fortran 77 Programmer’s Guide* (Mountain View, Calif.: Silicon Graphics Inc, 1994–1996).
27. “Compilation Control Statements and Compiler Directives,” *DEC Fortran 90 User Manual* (Maynard, Mass.: Digital Equipment Corporation, forthcoming in 1998).
28. *Release Notes for [Digital UNIX] Version V4.0D* (Maynard, Mass.: Digital Equipment Corporation, 1997).
29. “The Thread Attribute,” in *Microsoft Visual C++: C++ Language Reference (Version 2.0)* (Redmond, Wash.: Microsoft Press, 1994): 389–391.

30. *CF90 Commands and Directives Reference Manual* (Eagan, Minn.: Cray Research, Inc., 1993, 1997).
31. J. Gosling and H. McGilton, *The Java Language Environment: A White Paper* (May 1996). This paper is available at <http://www.javasoft.com/docs/white/langenv/> or via anonymous ftp from <ftp.javasoft.com> in the directory docs/papers, for example, the file whitepaper.ps.tar.Z.
32. J. Ousterhout, *Tcl and the Tk Toolkit* (Reading, Mass: Addison-Wesley, 1994).

Biography



David C. P. LaFrance-Linden

David LaFrance-Linden is a principal software engineer in DIGITAL’s High Performance Technical Computing group. Since joining DIGITAL in 1991, he has worked on tools for parallel processing, including the HPF-capable debugger described in this paper. He has also contributed to the implementation of the Parallel Software Environment and to the compile-time performance of the HPF compiler. Prior to joining DIGITAL, David worked at Symbolics, Inc. on front-end support, networks, operating system software, performance, and CPU architecture. He received a B.S. in mathematics from M.I.T. in 1982.