HPF_CRAFT Language

Tom MacDonald and *Andrew Meltzer*, Cray Research, Inc,
655-F Lone Oak Drive, Eagan, Minnesota 55121

ABSTRACT: *HPF_CRAFT is an implicit multi-threaded parallel model that simplifies data distribution and work distribution. This paper describes the HPF_CRAFT language and its relationship to both CRAFT and HPF.*

1 Introduction

This document is an overview of a language called HPF_CRAFT that is a merging of CRI's CRAFT language and HPF. HPF_CRAFT is being developed in partnership with The Portland Group, Inc. (PGI). The language is fundamentally CRAFT, but implemented within the superstructure of PGI's implementation of HPF, and influenced by the CRAFT-90 specification presented at the Denver CUG in 1995.

CRAFT features and a multi-threaded execution model allow the user to take advantage of the flexibility of an SPMD programming model and the low-level process control available with CRAFT. The HPF syntax allows the user to more easily port codes and provides a more well known user interface to reduce the amount of learning necessary to use the language.

To facilitate ease of understanding, use, and interoperability with HPF, as well as to attempt to conform to a more standard syntax, HPF syntax is used wherever possible. CRAFT syntax is used for features that are extensions to the basic overlapping functionality of HPF and CRAFT.

The underlying model of HPF_CRAFT is multi-threaded. This model is the foremost reason that the language is CRAFT. The execution performance of HPF_CRAFT is as good and possibly better than CRAFT. HPF_CRAFT will work seamlessly with SHMEM, Message Passing, and any other model that is inter-operable with CRAFT.

1.1 Goals of HPF_CRAFT

In defining the HPF_CRAFT language we have followed these guidelines, in order of importance:

1. The language must offer as good or better performance than the equivalent CRAFT.
2. Where it does not conflict with the above, the language should be as compatible as possible with HPF to provide a familiar and non-proprietary solution.
3. The language should be intercallable with Fortran 90, HPF, C++, and C.
4. The language provides the same execution model as the message passing and SHMEM libraries.
5. The language and implementation must be stable and reliable.

1.2 HPF, Kernel HPF, and HPF_CRAFT

HPF2 is the current High Performance Fortran Forum process, due to be complete during the summer of 1996. This process will develop a document describing the next HPF specification, HPF 2.0, along with one or more subsets and extensions (e.g., extrinsic environments). The HPF 1.1 document is currently available and the language is complete. HPF 2.0 and any subsets are in the process of creation and are still in flux. Kernel HPF has been proposed by CRI and is one subset being considered. It is a performance oriented subset of HPF 2.0. Kernel HPF bears a strong resemblance to CRAFT-90 except that it is largely a single-threaded language.

HPF_LOCAL is an extrinsic environment defined by the HPF committee to describe a portable SPMD programming language that is callable from HPF and understands HPF data layout information (but cannot itself reference data that is not local to a processor.) There is a well defined interface between HPF_LOCAL routines and HPF routines.

HPF_CRAFT is syntactically a near perfect superset of Kernel HPF, except that it employs a multi-threaded model. It is built upon the machine model of HPF_LOCAL, and uses that same well-defined interface.

2 Performance

The multi-threaded semantics supported by HPF_CRAFT allow greater programming flexibility along with performance as good or better than CRAFT. In almost all cases, HPF_CRAFT constructs (directives and intrinsics) can be replaced by equivalent CRAFT constructs.

Areas of HPF that introduce unpredictable performance profiles have been removed from Kernel HPF and are not included in HPF_CRAFT (with the exception that array extents can have a non-power-of-two size). Consequently, there is no technical reason that the performance of HPF_CRAFT cannot be equivalent to the performance of CRAFT.

2.1 General Execution Model

In HPF_CRAFT all PEs begin executing in parallel, with data defaulting to a replicated distribution (each PE gets a copy of the data storage unless specified otherwise by the user), as is currently the case in CRAFT. Consequently I/O works identically to CRAFT I/O and the SHMEM libraries and message passing libraries are as easily integrated as with CRAFT.

In short, the execution model is that of CRAFT.

3 Language Features

In this section the features of HPF_CRAFT are described. Features are distinguished as to whether they are derived from CRAFT or HPF.

3.1 Data Alignment and Distribution Directives

The HPF_CRAFT directives are chosen to have HPF syntax whenever possible. In a few instances additional functionality is added beyond the defined functionality in CRAFT-90 to make HPF_CRAFT more similar to Kernel HPF in appearance and usage. This added functionality is only there when no performance impact occurs if the feature is not used, and no surprises occur when it is. The **SHARED** directive was abandoned in favor of the HPF-style data mapping syntax.

3.1.1 The DISTRIBUTE Directive

The distribution formats available in HPF 2.0 are identical to those available in CRAFT with the exception that Kernel HPF has a **CYCLIC** distribution (**:BLOCK(1)** directive in CRAFT parlance). This distribution is added to HPF_CRAFT.

Degenerate can be specified in HPF_CRAFT, but instead of a '**:**', the '*****' that HPF uses in its distribution directive to specify an on-processor dimension is used.

The available distribution formats in HPF_CRAFT are:

- **BLOCK**
- **CYCLIC**
- ***** (degenerate)

CHARACTER arrays and variables cannot be explicitly mapped in HPF_CRAFT.

3.1.2 The ALIGN Directive

The **ALIGN** directive is syntactically and semantically identical to the HPF 2.0 align directive. The following are rules placed on the **ALIGN** directive in HPF_CRAFT

- Alignments may not contain offsets or strides.
- The *align-subscript-use* in the HPF 1.1 syntax rules should be replaced with *align-dummy.*
- Dimensions may not be permuted.
- A '*****' is allowed in either the *align-spec* or the *alignee.*

For example the following is correct code:

```
!HPF$   ALIGN A(I,J) WITH B(I,J)
```

But the following is not:

```
!HPF$   ALIGN A(I,J) WITH B(I+1,J)
```

The above rules allow for replication or collapse of dimensions but restrict arbitrary alignments.

3.1.3 The TEMPLATE Directive

Templates cannot be made semantically equivalent to the CRAFT **GEOMETRY** directive. The **GEOMETRY** directive is more like a macro which expands the memory layout part of the directive definition than it is like the **TEMPLATE** directive. The **TEMPLATE** directive requires the declaration of extents, and the elements of an array to be aligned to the template elements.

The **TEMPLATE** directive in HPF_CRAFT is identical to Kernel HPF.

3.1.4 The GEOMETRY Directive

The **GEOMETRY** directive is retained in the model, though a given **GEOMETRY** directive could be replaced in HPF_CRAFT by (possibly multiple) templates. A geometry need only be the same rank as the arrays which are mapped onto it.

3.1.5 The PROCESSORS Directive

The **PROCESSORS** directive's closest analog in CRAFT is the weights in the original implementation of CRAFT. The **PROCESSORS** directive in HPF 2.0 is a restricted form of the general directive allowed in HPF 1.1. The following restrictions apply:

- the product of the extents must exactly match the number of processors, or the processor arrangement must be scalar, and
- if an **ONTO** clause is not specified, a default arrangement is provided which is identical for all distributees that have identical shapes and identical explicit mappings.

3.1.6 Cray Pointers

The CRI Fortran compiler, CF90, includes Cray pointers; these are treated just as Cray pointers are treated in CRAFT and with the CRAFT syntax and semantics. Pointees can be distributed but a pointer to a mapped (distributed) object must point to the whole object.

3.1.7 Private Objects

As in CRAFT, private data objects are the default. Objects may also be declared explicitly to be private by using the **PE_PRIVATE** directive of CRAFT. The behavior of private objects is identical to the behavior in CRAFT.

3.1.8 Data Distribution Feature Comparison

The CRAFT **SHARED** directive is replaced in HPF_CRAFT by the **DISTRIBUTE** directive. Shared objects (now called *explicitly mapped* objects) retain the same meaning in HPF_CRAFT as they have in CRAFT.

The HPF **PROCESSORS** directive has been added.

The HPF **TEMPLATE** directive has been added.

The HPF **ALIGN** directive has been added.

The CRAFT **GEOMETRY** directive is retained.

The CRAFT **PE_PRIVATE** directive is retained.

CRAFT data distribution being private by default is retained.

CRAFT Cray pointers may point to distributed (*explicitly mapped*) objects.

3.2 Subprogram Interfaces

Although CRAFT permitted the remapping of arguments across subroutine boundaries, CRAFT-90 did not. Kernel HPF requires interface blocks when arguments are re-mapped across subroutine boundaries. HPF_CRAFT requires interface blocks wherever Kernel HPF requires them. This eases a restriction added to the CRAFT-90 specification, provides greater programming flexibility, but is designed to ensure that in all cases the caller is able to do the remapping (thus eliminating unwanted runtime checks). It also increases the compatibility between HPF and HPF_CRAFT without compromising the performance of the model.

An explicit interface is required in the following cases:

- The dummy argument has the **INHERIT** attribute.
- The mapping of a dummy argument is not the same as the mapping of the corresponding actual argument, and at least one of the following two conditions is true:
  1. the dummy argument is explicitly mapped, or
  2. the actual argument is an explicitly mapped whole array or a section of an explicitly mapped array.

3.2.1 Shared-to-Private Coercion

Shared-to-private coercion in HPF_CRAFT is implemented as in CRAFT, with the same restrictions and rules. This is an extension in HPF_CRAFT.

In addition to CRAFT style shared-to-private coercion, data may be coerced to local using the **HPF_LOCAL** extrinsic interface, in which case all data is considered private to the PE onto which it was distributed in the called routine, and the routine acts as an individual node program on each PE. This interface requires that the caller see an explicit interface with the **HPF_LOCAL** extrinsic name in it.

3.2.2 The INHERIT Directive

The **INHERIT** directive is not considered to be a high-performance feature, but its impact can be isolated to the places where it is used. The **INHERIT** directive can be instrumental when coding library and general purpose routines.

The **INHERIT** directive is essentially identical to the **UNKNOWN_SHARED** directive of CRAFT. This directive was excluded from the CRAFT-90 specification because library developers found that it did not provide high enough performance. In HPF it is useful in interface blocks, because it allows users to have a single interface for many distributions. Within the subroutine the mappings of the arrays can be tested and separate routines can be called for each mapping. In CRAFT explicit interfaces are not required, so this problem does not arise.

This feature does not impact the performance of codes that do not use it because an explicit interface is required when a dummy argument has the **INHERIT** attribute.

3.2.3 Subroutine Interface Comparison

Remapping across subprogram interfaces has been re-introduced. It is available in the original implementation of CRAFT, but was removed from the CRAFT-90 specification. In HPF_CRAFT it is available but it is also guaranteed that the caller may re-map data.

The **INHERIT** directive was added to HPF_CRAFT.

Shared-to-private coercion is not available in HPF.

3.3 PURE Functions and Subroutines

Pure functions and subroutines are not in CRAFT but are included in HPF_CRAFT for HPF compatibility reasons, and because they are sometimes needed when using the **FORALL** statement. These functions are not currently necessary to enable any behavior in HPF_CRAFT alone, but this feature may be useful if an HPF programmer desires to call an HPF_CRAFT subroutine.

3.4 Loops and Array Operations

In CRAFT there are two methods of specifying implicit work sharing: using array syntax and using the **DOSHARED** directive. In HPF there are also two ways to specify implicit work sharing: array syntax and the **INDEPENDENT** directive. The keyword **DOSHARED** has been replaced with the HPF **INDEPENDENT** syntax. Array syntax has not changed.

While the meaning of **INDEPENDENT** can easily be transformed to match that of **DOSHARED**, the syntax changes are not quite so simple. Two different forms of the **INDEPENDENT** directive are specified. The first is similar to the use of **INDEPENDENT** in HPF. The second is more syntactically similar to the CRAFT style used in the **DOSHARED** directive.

3.4.1 INDEPENDENT without the ON Clause

In HPF_CRAFT the meaning of the **INDEPENDENT** directive for a **DO** loop is functionally equivalent to what the meaning of **DOSHARED** would be without the **ON** clause. It asserts that there are no loop carried dependencies. The compiler is forced to pick a processor on which to execute each iteration. If there are distributed arrays within the loop, one of these can be chosen. If not, any distribution may be chosen. **INDEPENDENT** directives on loops that are tightly nested are merged and executed as if they were single **DOSHARED** directives. Inner **INDEPENDENT** loops that are not tightly nested are ignored.

3.4.2 INDEPENDENT with the ON Clause

In CRAFT the **DOSHARED** directive is applied to the first of a group of tightly nested loops and may apply to more than one of them. This more easily facilitates the use of the **ON** clause. The HPF **INDEPENDENT** directive applies only to a single loop nest. HPF_CRAFT allows either syntax for compatibility.

The **INDEPENDENT** directive is extended so that multiple loop nests can be named using a syntax very similar to the syntax of CRAFT, only the keyword **INDEPENDENT** replaces the keyword **DOSHARED**.

The syntax and semantics of **INDEPENDENT** with the **ON** clause are different from its syntax and semantics without the **ON** clause. With the **ON** clause the directive states that there are no cross-processor dependencies, but there may be dependencies between iterations on a processor. It also indicates which loop iterations it refers to. With the **ON** clause, **INDEPENDENT** has exactly the same semantics as the CRAFT **DOSHARED** directive. Syntactically the keyword **DOSHARED** can be replaced by the keyword **INDEPENDENT.** For example, where in CRAFT the directive might have been:

```
!DIR$ DOSHARED (I,J) ON A(J,I)
```

In HPF_CRAFT it is:

```
!HPF$ INDEPENDENT (I,J) ON A(J,I)
```

If the **ON** clause is used, **INDEPENDENT** must be used in this form.

The **INDEPENDENT** directive may optionally not include the **ON** clause at all. If programmers want to take advantage of the functionality of the **ON** clause in the **INDEPENDENT** directive, they can simply use the **INDEPENDENT** directive in the same way they previously used the **DOSHARED** directive..

3.4.3 The NEW Clause

An HPF independent loop optionally may have a **NEW** clause. The **NEW** clause is not required by CRAFT because in CRAFT data defaults to private and values may differ from processor to processor.

In CRAFT, however, private data has slightly differing semantics from the **NEW** clause. Iterations of a **DOSHARED** loop have a defined ordering for each PE so a private data item can be used beyond a single iteration of the loop. The values of data items named in a **NEW** clause may not be used beyond a single iteration. The **NEW** clause asserts that the **INDEPENDENT** directive is valid if new objects are created for the variables named in the clause for each iteration of the loop.

The **NEW** clause requires the compiler to generate a temporary, which must be used in place of the user variable. This is also the behavior in HPF_CRAFT as well. It is retained for compatibility reasons. The variables named in a **NEW** clause apply only to the immediately subsequent loop nest.

3.4.4 Array Syntax

Array syntax is treated the same in CRAFT as it is in HPF so no changes are required for HPF_CRAFT.

3.4.5 The FORALL Statement and Construct

The **FORALL** statement and the **FORALL** construct are part of HPF_CRAFT. **FORALL** is not a part of the original implementation of CRAFT but was planned for CRAFT-90. It is also part of Kernel HPF.

There are no changes or incompatibilities with either CRAFT or Kernel HPF.

FORALL INDEPENDENT is analyzed syntactically as a **FORALL**, but treated as an **INDEPENDENT** loop. These semantics are consistent with HPF.

3.4.6 Work Sharing Comparison

The **NEW** clause is added to HPF_CRAFT for compatibility with Kernel HPF. It adds no new semantics or functionality to HPF_CRAFT.

DOSHARED is replaced with **INDEPENDENT** and may be used with or without the **ON** clause.

The **INDEPENDENT** directive may refer to more than one (tightly nested) loop nest at a time and may be combined with the **ON** clause.

When used with the **ON** clause, the **INDEPENDENT** directive has the same meaning as the **DOSHARED** directive.

The semantics of parallel loop execution follow those of CRAFT.

3.5 Intrinsic and Library Procedures

The HPF library is supplied for HPF_CRAFT. In addition the CRAFT library routines are also provided.

3.6 Storage and Sequence Association

The storage and sequence association rules are identical to those in CRAFT and Kernel HPF; there is no sequence or storage association for data that is not private. Private data retains the sequence and storage rules of standard Fortran.

3.7 Parallel Execution

Due to the differences in models (multi-threaded vs. single-threaded) this is the area of greatest change with respect to Kernel HPF. The model used is that of CRAFT. This model fits easily on top of the HPF_LOCAL extrinsic environment (which is a defined portion of HPF). The HPF_LOCAL environment has a well-defined interface with HPF that will be used for the HPF_CRAFT extrinsic environment.

In essence, HPF_CRAFT is CRAFT extensions embedded within the HPF_LOCAL extrinsic.

3.7.1 Parallelism Inquiry Intrinsics

These directives are provided with their CRAFT semantics. They are an extension to HPF. The **IN_DOSHARED** is changed to **IN_INDEPENDENT**. The supported parallelism inquiry intrinsics are:

- **IN_PARALLEL**
- **IN_INDEPENDENT**

3.7.2 STOP and ABORT

The **STOP** and **ABORT** statements behave just as they do in CRAFT.

3.7.3 Sequential Regions

In HPF_CRAFT the **MASTER**/ **END MASTER** construct remains, retaining the syntax and semantics that it has in CRAFT, the **COPY** clause is also retained unchanged.

3.7.4 Task Identity

N$PES is augmented with the equivalent HPF intrinsic **NUMBER_OF_PROCESSORS()**. The **MY_PE()** intrinsic is augmented with the equivalent HPF name **MY_PROCESSOR()**. Both versions are available because of the ubiquity of their use and the convenience of the CRAFT names.

3.7.5 Deprecated/Altered CRAFT Features for Parallel Execution

- There are no changes from the CRAFT syntax or semantics.
- **NUMBER_OF_PROCESSORS** and **MY_PROCESSOR()** have been added with the same meaning as **N$PES** and **MY_PE()** respectively for compatibility with HPF.

3.7.6 Parallel Execution Comparison

The execution model is the CRAFT multi-threaded model.

The **IN_INDEPENDENT** intrinsic replaces the CRAFT **IN_DOSHARED** intrinsic, and the CRAFT **IN_PARALLEL** intrinsic is retained.

The CRAFT **N$PES** and **MY_PE()** are retained, and the HPF equivalents are added.

The CRAFT **MASTER** and **END MASTER** directives with the **COPY** clause is retained.

The **STOP** statement and **ABORT** function have the CRAFT semantics.

3.8 Synchronization

In CRAFT there are a number of synchronization primitives and directives. There are none available native in HPF. The entire set of CRAFT primitives is included. Shared data coherence points are identical to those of CRAFT.

3.8.1 Program Barrier Directives

Explicit barriers are not necessary in the single-threaded HPF model, and barrier removal only occurs when automatically detectable. In CRAFT, barriers serve many useful purposes and may be removed by the user. The CRAFT barrier syntax and semantics are retained. These directives are:

- **!HPF$ BARRIER**
- **!HPF$ NO BARRIER**

3.8.2 The REDUCE Directive

The **REDUCE** directive is new in Kernel HPF. The **REDUCE** directive is identical to CRAFT's **ATOMIC UPDATE** directive. In Kernel HPF the **REDUCE** directive may only apply to intrinsic types and intrinsic operators. This behavior is adopted by HPF_CRAFT.

3.8.3 Synchronization Primitives

CRAFT defines a large set of synchronization primitives unavailable in HPF. All of these primitives are available in HPF_CRAFT. They have the same syntax and semantics as currently implemented for CRAFT-90.

These include:

- **SET_BARRIER()**
- **WAIT_BARRIER()**
- **TEST_BARRIER()**
- **BARRIER()**
- **SET_LOCK()**
- **CLEAR_LOCK()**
- **TEST_LOCK()**
- **CRITICAL / END CRITICAL**
- **SET_EVENT()**
- **CLEAR_EVENT()**
- **WAIT_EVENT()**
- **TEST_EVENT()**

3.9 Input and Output

All I/O in HPF_CRAFT will retain its CRAFT syntax and semantics. Private I/O retains its CRAFT syntax and semantics.

3.10 The PE_RESIDENT Directive

PE_RESIDENT is retained with its CRAFT semantics. If used, any loop which contains it and also uses the **INDEPENDENT** directive is required to have the **ON** clause. A **PE_RESIDENT** array may not be used in array syntax operations.

All restrictions on dummy arguments for CRAFT-90 also apply to HPF_CRAFT programs.

3.11 Intrinsics

All intrinsics available in CRAFT are available in HPF_CRAFT. The intrinsics available in both CRAFT and HPF are renamed to match the HPF naming conventions.

3.11.1 Data Mapping functions

The data mapping functions are retained in HPF_CRAFT with their current meaning. The functions are:

- **HIIDX**
- **LOWIDX**
- **BLKCT**
- **PES**
- **HOME**

3.11.2 Parallel Prefix and Parallel Scan Functions

CRAFT contains a set of parallel prefix and parallel scan functions:

- **PREMAX**
- **PREMIN**
- **PREPROD**
- **PRESUM**
- **SCANMAX**
- **SCANMIN**
- **SCANPROD**
- **SCANSUM**

These functions are replaced by the HPF library functions with the same (or added) functionality.

4 Other CRAFT Features

4.1 Memory Allocation Directive

This directive makes the shared memory library codes easier to use with HPF_CRAFT and is very easy to implement. It is basically a way for the programmer to direct the compiler to put data on the shared heap or the shared stack. It lets users ensure that data is stored at the same offset on all PEs. The directive is available in HPF_CRAFT.

- **!HPF$ SYMMETRIC**

5 Extensions

5.1 HPF_CRAFT Extrinsic

HPF_CRAFT defines a new extrinsic environment called HPF_CRAFT. The extrinsic tells the compiler to compile the subroutine as CRAFT code. The environment uses the standard HPF **EXTRINSIC** syntax

```
EXTRINSIC(HPF_CRAFT)
```

The HPF_CRAFT environment may also be used indicated by a compile-time switch.

Appendix A: HPF_CRAFT Features

This appendix contains a complete list of the directives available in HPF_CRAFT.

- **DISTRIBUTE** with the following data layouts
  - **BLOCK**
  - **CYCLIC**
  - Degenerate (*)
- **ALIGN**
- **TEMPLATE**
- **PROCESSORS**
- **PE_PRIVATE**
- **INDEPENDENT** with the following extensions
  - **ON**
  - **NEW**
- **FORALL**
- **IN_INDEPENDENT**
- **IN_PARALLEL**
- **MASTER**
- **END MASTER**
- **PARALLEL_ONLY**
- **SERIAL_ONLY**
- **PARALLEL_AND_SERIAL**
- **NUMBER_OF_PROCESSORS**
- **MY_PROCESSOR()**
- **N$PES**
- **MY_PE()**
- **BARRIER**
- **NO BARRIER**
- **REDUCE**
- **INHERIT**
- **SET_BARRIER()**
- **WAIT_BARRIER()**
- **TEST_BARRIER()**
- **BARRIER()**
- **SET_LOCK()**
- **CLEAR_LOCK()**
- **TEST_LOCK()**
- **CRITICAL / END CRITICAL**
- **SET_EVENT()**
- **CLEAR_EVENT()**
- **WAIT_EVENT()**
- **TEST_EVENT()**
- **PE_RESIDENT**
- **HIIDX**
- **LOWIDX**
- **BLKCT**
- **PES**
- **HOME**
- **HPF_DISTRIBUTION**
- **PREMAX**
- **PREMIN**
- **PREPROD**
- **PRESUM**
- **SCANMAX**
- **SCANMIN**
- **SCANPROD**
- **SCANSUM**
- **SYMMETRIC**
- **EXTRINSIC(HPF_CRAFT)**

Appendix B: CRAFT-90 Differences

This appendix lists the differences between HPF_CRAFT and CRAFT-90.

Data Distribution

- The **SHARED** directive has been replaced in HPF_CRAFT by the **DISTRIBUTE** directive. Shared objects (now called *explicitly mapped* objects) retain the same meaning in HPF_CRAFT as they have in CRAFT.

- The **CYCLIC** distribution has been added.
- The **PROCESSORS** directive has been added.
- The **TEMPLATE** directive has been added.
- The **ALIGN** directive has been added.
- Arrays of derived types may not be distributed.

Subprograms and Subprogram Interfaces

- Remapping across subprogram interfaces has been re-introduced. It is available in the original implementation of CRAFT, and was removed for CRAFT-90. In this model it is available but it is guaranteed that the caller may re-map data.
- **PURE** subroutines and functions have been added.
- The **INHERIT** directive has been added.

Work Sharing

- The **NEW** clause is added for compatibility with Kernel HPF
- **DOSHARED** is replaced with **INDEPENDENT**.
- **INDEPENDENT** may be used with or without the **ON** clause.

Synchronization

- The **ATOMIC UPDATE** directive has been replaced by **REDUCE.**

Libraries and Intrinsics

- The HPF Library is included in the language.

Parallel Execution

- **NUMBER_OF_PROCESSORS** and **MY_PROCESSOR()** have been added with the same meaning as **N$PES** and **MY_PE()** respectively (for compatibility with HPF).
- The **EXTRINSIC(HPF_CRAFT)** environment has been added.

Appendix C: Kernel HPF Differences

This appendix lists the differences between HPF_CRAFT and Kernel HPF.

Data Distribution

- **PE_PRIVATE** directive has been added.
- Data distribution default is private.
- Cray pointers may point to distributed objects.
- The **GEOMETRY** directive has been added.

Subprograms and Subprogram Interfaces

- Shared-to-private coercion has been added.
- **PE_RESIDENT** directive has been added.

Loops and Array Operations (Work Sharing)

- The **INDEPENDENT** directive may refer to more than one (tightly nested) loop nest at a time and may be combined with the **ON** clause.
- When used with the **ON** clause, the **INDEPENDENT** directive has the same meaning as the **DOSHARED** directive.
- The semantics of the parallel execution of the loop follow those of CRAFT.

Parallel Execution

- The execution model is multi-threaded.
- The **IN_INDEPENDENT** and **IN_PARALLEL** intrinsics are added.
- **N$PES** and **MY_PE()** have been added.
- **MASTER / END MASTER** with the **COPY** clause is added.
- **STOP** and **ABORT** have CRAFT semantics.
- **PARALLEL_ONLY**, **SERIAL_ONLY**, and **PARALLEL_AND_SERIAL** directives have been added.

Libraries and Intrinsics

- **HIIDX, LOWIDX, BLKCT, PES, HOME** intrinsics are added.
- **IN_PARALLEL** and **IN_INDEPENDENT** are added.

Synchronization

- **BARRIER / NO BARRIER** have been added.
- CRAFT synchronization primitives has been added.

Input and Output

- Altered to CRAFT semantics.

Memory Allocation Directive

- The **SYMMETRIC** directive has been added.