Split-Java

David E. Gay
CS263 project report

June 17, 1997

1 Introduction

Split-C [2] is an explicitly parallel programming language designed as a set of simple extensions to C [3]. These extensions are: explicit parallelism with all processes created "ab initio", a global address space with explicit global pointers, "spread" arrays - arrays whose elements are distributed amongst all the processors, special asynchronous assignment operators, and simple synchronisation primitives.

Java [1] is an object-oriented language designed for Internet applications, in some respects close to C++. It already includes some support for parallelism in the form of multi-threading.

This project aims to design a language called "Split-Java", close in spirit to Split-C, that will be an explicitly parallel version of Java incorporating extensions similar to those of Split-C.

2 Split-Java

This section presents an informal definition of Split-Java by describing the parts of the language that are changed, the motivations for these changes and the various options considered. A more formal definition is in Section A.

2.1 Requirements

Split-Java is an upwardly-compatible extension of Java augmented with the Split-C features (see [2, pp.6-9]):

- *Multiple persistent threads*: a Split-Java program has one thread per processor from the start of execution. These threads all survive until the program terminates. Synchronisation primitives are provided.
- *Global address space*: references can be made to any object on any processor, but global and local references are distinguished. Global references include the processor number and a local reference.
- *Spread arrays*: a spread array is an array whose elements are split amongst the processors in a cyclical fashion: the first element on the first processor, the second on the second, the $n+1$th, where n is the number of processors, on the first, and so on.
- *Special assignment operators*: the usual assignment operator `=` is augmented with a *split-phase* assignment operator `:=`. This initiates a remote read or write, but allows execution to continue before the remote answer is received. An even simpler *signaling* assignment operator `:-` initiates a remote write but does not provide any acknowledgment to the writer. For both these operators, simple synchronisation primitives are available.

Special versions of these operators are provided for array slices to allow efficient transmission of large quantities of data between processors.

Split-Java is defined from Java, with changes in the following areas:

- Objects and object references.
- Threads and thread synchronisation.
- Assignment operators.
- Arrays and bulk data transfer.

The other features of the Java language are preserved. In particular, Split-Java is a safe language, i.e. all runtime errors cause exceptions. It is also garbage collected, which raises the issue of garbage collection with remote object references. The rest of this section explores these issues in more detail. It does not purport to be a complete definition, for that see Appendix A.

2.2 Objects

Reference to data in another process is by a *remote object reference*, which includes the processor number and the local reference. As in Split-C, remote references must be explicitly typed with the `global` keyword:

```
global Object foo;
```

There are no `global` primitive types:

```
global int bad; // Error!
```

Fields can be read and written via a global reference, subject to the Java `public`, `protected` and `private` visibility restrictions. There are also special assignment operators, as described in Section 2.4.

Method invocation presents several options:

- Forbid method calls on global references. This is undesirable because it forces a non-object-oriented programming style.
- The distributed programming model: Method calls are executed on the processor that holds the object, remote calls send the arguments and results across the network. This is potentially very slow.
- Method calls are executed locally, as usual. `this` is a global, not a local, reference. This design raises the issue of how to handle the fact that now `this` can be local or global. Again, several solutions are possible:
  - The same code can be used for the local and global case. To avoid constant tests at each implicit reference to `this`, the implementation of every method should be duplicated[1] to provide separate code for each case.

    In this case, the type of `this` is unclear: it could be either a local reference, in which case the special assignment operators of Section 2.4 cannot be used, or it can be a global reference in which case the special operators must be emulated in the locally-compiled version.
  - Invocation of methods via global references can be restricted to methods that are specially declared with a `remote` keyword. In these methods, `this` is a global object reference. As in the previous solution, a special local version can be compiled for performance. If a `remote` method is invoked on a local reference, this reference is first implicitly cast to a global reference.

[1] This need only be done for methods that are potentially called via global references, so the code expansion should be far less than a factor of two.

- Separate implementations of methods can be provided by the programmer for the local and global cases. Only methods with global implementations can be called via global references. This is similar to the previous solution, with overloading of the method name based on the type of the object on which the method is called.

None of these solutions is clearly better than the others, but the `remote` keyword approach is cleaner from a software engineering perspective: it is much clearer which methods potentially get executed, it is not necessary to check the type of the reference. The selection of method is already complicated enough by inheritance and overloading without adding a separate, independent overloading mechanism.

Allowing global references has two impacts on garbage collection:

- It is not so easy to move objects if a copying garbage collector is used, as the remote references are not so easily available as the local ones.
- The only references to an object may be remote, so it is not as obvious when an object is "dead".

These problems, and other issues are discussed in the literature on parallel and distributed garbage collection, e.g. [4] or [5]. The appropriate garbage collection will depend on the architecture of the target, e.g. SMPs versus distributed-memory machines.

2.3 Arrays

Arrays in Java are simple one-dimensional vectors of references or primitive types. To allow multi-dimensional arrays, the reference can be to another array. But each element is independent, so a triangular array can be created with:

```
byte[][] newArray = new byte[n][]; // Allocate row pointer array
for (int i = 0; i < n; i++)
  newArray[i] = new byte[i];
```

Split-Java provides a number of extensions to arrays, some of which are implied by the existence of global references:

- There may be global references to arrays.
- Arrays of references can be either arrays of local references or of global references.
- Taking this together with Java-style arrays means that there can be a local reference to an array of global references to an array of local references to objects of type A. This can be a little confusing, especially when spread arrays are also considered.
- Arrays may be spread across processors, as described in the requirements section. Apart from the question of location, spread array references behave like global array references (see in particular the discussion of global references to arrays of local references below).
- Split-Java provides real multi-dimensional arrays.
- Slices can be copied between arrays, possibly remotely. See Section 2.4.

2.3.1 Syntax

A global reference to an array is very similar to any other global reference:

```
global double[] foo; // foo is a global reference to an array of double
```

If the base type of the array is a reference type, then this syntax produces a global reference to an array of local references:

```
global A[] bar;
```

There is therefore a need for a different syntax to express types such as 'array of global references to A's'. And, as mentioned above one can have 'arrays of global references to arrays of local references to A's', or 'arrays of global references to arrays of global references to A's'. So it must be possible to attach a qualifier with each dimension that indicates whether the references stored there are local or global. This is done with the `[] global` notation

So to get an array of global references, one writes:

```
A[] global baz; // baz is an array of global references to A's
```

With multiple dimensions this can be a little complex:

```
global double[][] twod;
  // twod is a global reference to an array of local references to arrays of doubles

double[] global [] twod-split;
  // twod-split is a reference to an array of global references to arrays of doubles
  // (each row of twod-split can be on a different processor)

global A[][] global strange;
  // strange is a global reference to an array of local references to an
  // array of global references to A's
```

Spread arrays are written with {} instead of `[]`, and can be combined with normal arrays ({} `global` plays the same role as `[] global`):

```
double{} pickle; // pickle is a spread array of doubles
double[]{} chutney; // chutney is an array of references to spread arrays of doubles

global double[]{} gchutney; // a global reference to chutney

A{} fun; // A spread array of local references to A's
A({} global) morefun; // A spread array of global references to A's
```

It is an error to try and declare a global reference to a spread array:

```
global A{} bad; // error!
double[] global {} worse; // error!
```

Both spread and normal arrays can be multi-dimensional. The number of dimensions, but not their sizes, must be specified in declarations. The separator is ",":

```
double multi1[,]; // A 2-dimensional array of doubles
double spreadmulti{,}; // A spread 2-dimensional array

multi1 = new double[20,20]; // Allocate a 20x20 array of doubles
```

```
multi1[3,2] = 42.0 + spreadmulti[0,0]; // Read & write elements

global A[][,] global stranger;
  // strange is a global reference to an array of local references to a
  // 2-dimensional array of global references to A's
```

A reference to a multi-dimensional array must provide all the dimensions.

The `length` element, defined on all arrays returns the number of elements in a multi-dimensional array. A new `int[] size` method returns an array of integers giving the size of each dimension:

```
double[,,] multi3 = new double[10,10,20];
int[] multi3size = multi3.size(); // multi3size = { 10, 10, 20 }
```

To allocate a spread array, all processors must call the same `new` statement, passing it the same arguments. The threads will stall until everyone has reached the allocation point, i.e. it is an implicit barrier (see Section 2.5). A `new` statement is a spread array allocation if it specifies the size of a spread dimension:

```
double foo{,} = new double{200,10}; // spread array allocation
double bar[]{} = new double[10]{}; // NOT a spread array allocation
```

2.3.2 Reading and Writing

Reading and writing array elements is now far more complicated than before. Ignoring spread arrays for the time being, in a statement $x[i, j, \ldots]$ x can be one of the following:

- A local reference to an array of a primitive type, of a local reference type, or another local array. The behaviour here is as in Java.
- A local reference to an array of global object references, or of global array references. The behaviour is also unchanged, except that global references are read or written.
- A global reference to an array of primitive type. Reads or writes are remote accesses and can use the special assignment operators described below.
- A global reference to an array of global object references, or of global array references. Reads or writes are remote accesses and can use the special assignment operators. The values read or written are global references.
- A global reference to an array of local object references, or of local array references. Reads are allowed, but they return a *global* reference to the processor containing the array. Global references may be written to such arrays, the compiler inserts a run-time check to ensure that the global reference belongs to the processor owning the array element and throws an exception if not.

  Comment: this is a somewhat ugly special case, as assignments in Java do not otherwise require run-time checks (by design). But the two other options, forbidding such assignments or requiring special syntax are even more unattractive.

For spread arrays, a statement $x[i, j, \ldots]$ is defined as follows:

1. The processor p on which the element resides is computed using the cyclical defined above, assuming x is stored in column-major order.
2. The rules for indexing are as above, where spread arrays should be treated like global arrays. The only special case is a spread array of local references: In that case, the global reference created by reading the local value l from processor p is $< p, l >$.

2.4 Data Movement

Data accessible via global references can be read or written with the usual `=` operator. These are either field or array accesses:

```
o1.field1 = o2.field2 + 23;
a1[3] = a2[0,0];
```

If `o1` is global, the expression on the right hand side is evaluated, and a global write is performed. Fields can be read from global references in expressions, these imply a global read to fetch the value of the expression from its processor. Both the global read and write operations are synchronous. The same rules apply to array references.

These rules imply that if both `o1` and `o2` are global, then `o2.field2` will be read remotely, then its value written remotely to `o1.field1`.

The special assignment operators `+=`, `*=`, ... can also be used with global pointers: `E1 op= E2` is the same as `E1 = E1 op E2` except that E1 is evaluated only once. However there will be both a global read and a global write of the field or array element named by E1. The increment and decrement operators `++` and `--` can also be used, and will also cause a global read and write.

2.4.1 Split-phase Assignment

Split-phase assignment is used to initiate an asynchronous global read or write operation. This is useful to hide the latency of these operations when their effects are not immediately needed. These asynchronous reads and writes are called *get* and *put* respectively.

Syntactically, they are written `E1 := E2` where at least one of `E1` or `E2` must be a global field or array access. If `E1` is global, then it is a split-phase write, otherwise it is a split-phase read:

```
global A foo;
A bar;
global double{} big;
double x;

foo.field1 := bar.field1; // A split-phase write
x := big[23]; // A split-phase read
```

After a get or put has been initiated, the value of the element being written is undefined until its value has been synchronised. There are various ways Split-Java could provide this:

- Global synchronisation: A static `java.split.global_sync()` method can be called that will wait for all outstanding gets and puts of this processor to complete.
- Per-object synchronisation: Each object has a `sync()` method that waits for all outstanding gets and puts of this object (which might be an array) to complete.
- A structured synchronisation construct from which control can exit only when all gets and puts initiated inside it complete:

  ```
  sync {
    foo.field1 := bar.field1; // A split-phase put
    x := big[23]; // A split-phase get
  }
  // Control reaches here when the put & get are complete
  ```

Exceptions and `return` statements are also delayed until the gets and puts are complete.

To implement the per-object synchronisation requires a per-object put and get counter. This overhead seems excessive, so Split-Java only provides global and structured synchronisation.

To ensure safety, Split-Java guarantees the atomicity of these updates. A program will see either the new or the old value.

Split-Java also provides an unacknowledged *store* operation which simply writes a value to a remote processor: `E1 :- E2` where `E1` must be a global field or array element, `E2` can be any value but will be read with a normal synchronous read if it is remote. As with gets and puts, the update is atomic.

The writer cannot check when the store is complete, but there are several possible synchronisation methods:

- A global store synchronisation: `java.split.all_store_sync` must be called on all processors (and includes an implicit barrier synchronisation), it waits for all stores to all processors to complete.
- A local store synchronisation: `java.split.store_sync(`n`)` will stall the current processor until n store's have been received.
- A per-object store synchronisation: o`.store_sync(`n`)` will stall the current processor until n store's have been received for object o - o must be a local object.

Again, per-object synchronisation requires one counter per object. So only the global and local store synchronisations are provided by Split-Java.

Stores, split-phase reads and writes are statements and not expressions: a split-phase read cannot be an expression as its value is not available, so syntactically a split-phase write cannot be either. For consistency, the other special assignment operator is also considered a statement.

2.4.2 Arrays

The data movement primitives described above only allow small amounts of data - primitive data types or object references - to be moved at a time. This is a performance bottleneck. To allow efficient movement of large quantities of data, Split-Java provides array-slice assignment, which can be used either locally or remotely and with the normal (=) or special (`:=`, `:-`) assignment operators.

Syntactically, these bulk data movements look like assignments:

```
global double{}[] a1; // A spread array of local arrays
double[] a2; // A local array

// Allocate arrays
a1 = new double{20}[20];
a2 = new double[10];

a2[0:9:1] = a1[3][1:19:2]; // copy odd elements
a1[5][5:14:1] = a2[0:9:1]; // and write them somewhere else...
```

The types of the two arrays must be the same, except that one (and only one) may be a global or spread array reference. Each slice is specified as $[start_1 : end_1 : step_1, start_2 : end_2 : step_2, \ldots]$. The number of dimensions must match the dimension of the array, and the number of elements specified in each dimension (i.e. $(end + 1 - start)/size$) must be the same on the left and right hand sides.

If the two array expressions specified by a slice copy evaluate to references to the same array then the copy behaves as if all elements are first read, then all the elements are written. This applies even if one of the array references is remote.

Spread arrays may be copied with this notation, but performance will be slower because data has to be read or written from several processors (data is less likely be transferred in bulk, though this depends on the amount of data being copied and the stride):

```
double{} a1;
double[] a2;

a2[0:10:1] = a1[1:11:1]; // Reads data from up to 10 processors...
```

The special assignments operators work on array slices as they do on primitive and reference types. The same synchronisation operations are available. For `store_sync`, each array elements counts as 1 store, so a destination processor should do `store_sync(10)` if it expects to receive an array slice with 10 elements.

The array slice operations are atomic at the array element level, to ensure safety. Threads will observe either the new or the old value for any given array element.

The copies are not done in any particular order. This implies that if two different writes refer to overlapping portions of the same array then the results could be a jumble of values from both writes.

2.5 Threading and Parallelism

Java and Split-C have different models of parallelism:

- Java has explicit thread creation and deletion, shared memory semantics and simple monitor style synchronisation on a per-object basis.
- In Split-C there is one thread per processor, these threads live for the whole duration of the program. Remote accesses are explicit and the main synchronisation primitive is a *barrier*, that all threads must cross simultaneously.

Split-Java is based on Split-C's style of parallelism, but must somehow integrate the Java features. The rest of this section first discusses the issue of threads, and then the options for synchronisation in Split-Java.

2.5.1 Java Threads

There are a number of options for dealing with Java threads:

- They can be simply removed. This is the simplest solution and it creates the least complications for synchronisation. It would however make Split-Java not upwardly compatible with Java.
- Each processor could have as many threads as it wishes, created as usual, and able to use all the Split-Java facilities. However this complicates the semantics of spread array allocation and other global operations.
- Each processor could distinguish the main thread (the one that exists at program startup) on each processor, and forbid synchronisation operations (including spread array allocation) from happening in other threads. This is not very clean and difficult to enforce at compile-time.

2.5.2 Synchronisation

The basic Split-C synchronisation primitive is `barrier()`. When this function is called a thread stalls until the threads on all the other processors have called `barrier()`, at which point all threads resume execution. In Split-Java, in the presence of multiple threads per processor, there are two options:

1. All threads on all processors must call the `barrier()` method.
2. One thread per processor must call `barrier()`. It is a runtime error if more than one thread per processor does this.

The first solution is definitely the most logical.

Java methods can have a `synchronized` keyword that means that invocations of that method will acquire a per-object (or per-class if the method is `static`) lock on entry and release it on exit. In Split-Java methods can also be characterised as `remote` (see Section 2.2) which means that the code may be executed on another processor than the one owning the object. In this case, the lock must be acquired and released from the remote processor. The `synchronized` statement is extended in the same fashion for global object references.

As defined above, and in Split-C, when allocating a spread array with a statement like:

```
double{} bark = new double{100};
```

every thread must execute the allocation statement - there is an implicit barrier at the start of the allocation. When execution resumes, the spread array has been allocated with some elements on each processor, and each thread receives the spread array reference as a result of the call to `new`. When there is more than one thread per processor the same options as for `barrier()` are available:

1. All threads on all processors must execute the call to `new`. This is bizarre because only one thread per processor really needs to help in the allocation.
2. One thread per processor must call `new`. This conflicts with the first choice for `barrier()`, because of the implicit barrier at the start of the allocation.

Neither of these solutions is very good. But, because of the implicit barrier, the choice here should follow that made for `barrier()`.

Another solution would be to change the semantics of spread array allocation: any processor could allocate a spread array, without requiring synchronisation at the language level[2]. If the value needed to be available in all threads, the value broadcast facilities described in Section 2.6.4 would be used.

2.5.3 Remote Invocation

Split-C allows the programmer to request the atomic execution on another processor of simple functions by calling the `atomic()` function. There are many restrictions on this facility, for instance the atomic function may not read or write any global values.

Split-Java already provides atomicity through the use of the `synchronized` keyword on methods. As discussed in Section 2.2, execution of a method should proceed on the local processor, so Split-Java does not provide this facility.

2.6 Miscellaneous

2.6.1 The java.split class

The `java.split` class contains all the static class methods specific to Split-Java described above, all the global operations of Section 2.6.4 and the following miscellaneous operations:

- `int nprocessors()`: returns the number of processors available.
- `int myproc()`: returns the number of the current processor, between 0 and `nprocessors()` -1.

[2] This might constrain the possible implementations of spread arrays

2.6.2 Global-Local Conversions

Local references can be converted to global references. The resulting reference has processor id equal to `java.split.myproc()`. This conversion happens implicitly when a call is made to a `remote` method via a local reference. As this conversion requires no runtime check, it can also be done explicitly or as an *Assignment conversion* (see [2, Section 3.3]), i.e. the following code is legal:

```
A foo = new A;
global A bar = foo; // make a global pointer to foo

bar = (global A)foo; // explicit casts are ok too
```

Remote references to arrays of local references also implicitly create global references from local ones.

A local reference can be created from a global one only on the machine on which the referred object resides by an explicit cast. Such casts cause a runtime check, if it fails an `NonLocalReference` exception is thrown. For instance:

```
global{} A spreadfoo;
global A anyfoo;
A myfoo;

anyfoo = spreadfoo[10]; // Valid everywhere
if (java.split.myproc() == 9)
  myfoo = (A)(spreadfoo[9]); // Element 9 is local...
```

2.6.3 Iterating over Spread Arrays

When a computation needs to be done for all the elements of a spread array, it is most efficient if each processor operates on the elements that are stored locally. By providing language support for this common operation, the compiler can avoid the overhead of checking for a remote access, and can avoid creating a global reference for spread arrays of local references (as the reference will be known to be local).

Split-Java provides a high-level control structure methods to support this: `scan` iterates over the elements of a multi-dimensional spread array, or a slice thereof.

`scan` is used as follows:

```
double{,} multi2 = new double{20,10};
double x;
int i, j;

scan (multi2[0:19:1,0:9:2], x, i, j)
  multi2[i, j] += 10.0; // or multi2[i, j] = x + 10.0;
```

The `scan` control structure takes a spread array, and optional slice, as first argument. The second element must be a variable of the base type of the array. These are followed by as many integer variables as the array has dimensions. The `scan`'s body is a statement that is executed once for each local element of the spread array that is in the slice (or all local elements without a slice). At each iteration:

- The second argument (`x` above) is set to the value of the corresponding element. A global reference is *not* created in this case.
- The third through last arguments are set to the index of this element.

- Within the body of the `scan`, references to the current element (i.e. `multi2[i, j]` in the example) are recognised as being local and implemented without the need for a remote read.

The iterations proceed in the order of the elements in memory, assuming a column-major array layout.

It would also be possible to provide low-level read and write operations on spread arrays that would modify the ith local element of a spread array, but it is unclear how useful this would be. For instance:

```
double{} multi = new double{100};
double x;
int i;

multi.lwrite(3, x); // Write x in the third local element
// This is the same as:
multi[3 * java.split.nprocs() + java.split.myproc()] = x;
x = multi.lread(i); // Read the ith local element
// This is the same as:
x = multi[i * java.split.nprocs() + java.split.myproc()];
```

For one-dimensional arrays, the compiler can recognise the index expression, but this is not so easy when there are more than two dimensions.

2.6.4 Global Operations

Split-Java provides the same global communication operations as Split-C (see [2, Section 10.8]), as static methods in the `java.split` class. All threads on all processors must execute the operation. This section only gives the differences with respect to Split-C.

Reductions or scans to processor 0 (methods such as `all_reduce_to_one_dadd`) return the value to all threads on processor 0. The value returned on other processors is the *default* value for the type being reduced (see [1, §2.3]). The operand type in the method names is nothing for `int`, `'l'` for `long` and `'d'` for `double`. There is also an `all_bcast_o` method to broadcast a global object reference.

3 Conclusion

3.1 Implementability

The new syntax, detailed in the appendix, can be added to Java without undue problems - an existing `lex+yacc`-based Java grammar was modified to include them. The addition of slice operations requires a post-processing of each statement to verify that slices are only used in legal contexts (scan, array slice assignment).

Actually implementing Split-Java would be a large project, though it should not present any major difficulties, except for the issue of parallel garbage collection.

3.2 Remarks

The extensions for arrays are the most complex part of the Split-Java language: arrays can now be multi-dimensional or spread, and the interactions with local and global references can get quite complicated.

This project did not come up with a good way of integrating threads with Split-C style SPMD parallelism. So Split-Java does not include Java threads. This is still an open issue.

In summary, this report has proposed a design for Split-Java, a fairly small extension to Java incorporating the explicit parallelism features of Split-C and better support for arrays, which is necessary for numerical computation. These extensions do not introduce special cases into Java, except for some cases of array assignment that require a run-time check (when no other form of assignment does). Split-Java has a clean and simple model for objects in the parallel framework: `this` is a global reference in methods that can be invoked on processors that do not contain the object. The language is not a superset of Java because of the removal of threads, this might be a concern for an implementation that wished to reuse the Java libraries.

References

[1] The Java^{tm} Language Specification, version 1.0Beta. Technical report, Sun Microsystems, Oct 95.

[2] David E. Culler et al. Introduction to Split-C. Technical report, University of California Berkeley, Computer Science Division, Apr 95.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[4] Bernard Lang, Christian Quiennec, and José Piquer. Garbage Collecting the World. In *Nineteenth ACM Symposium on Principles of Programming Languages*, January 1992.

[5] Ravi Sharma and Mary Lou Soffa. Parallel Generational Garbage Collection. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, October 1991.

A Split-Java Specification

Split-Java is specified as a set of changes to [1], the Java language reference manual.

1 Lexical Structure

1.5 Keywords

The following keywords are new: `global` `remote` `scan` `sync`.

1.9 Operators

The following operators are new: `:=` `:-`.

2 Types and Values

References come in two kinds: *local* and *global*. Global references are references to values stored on other processors. Variables and array elements must be explicitly declared `global` to accept global references.

Type:
    BasicType
    `global` *BasicType*

BasicType:
    PrimitiveType
    ClassType
    InterfaceType
    ArrayType

`global` is a qualifier on the entire type, not on the base type of the declaration. `global A[]` is a global reference to an array of `A`'s, not an array of global references to `A`'s.

`global` can not be applied to a primitive type or to a spread array type.

2.2 Reference Types and Values

ArrayType:
    BasicType `[` *Dimensions* `]`
    BasicType `[` *Dimensions* `]` *global*
    BasicType { *Dimensions* }
    BasicType { *Dimensions* } *global*

Dimensions:
    / empty */*
    , *Dimensions*

2.2.2 Arrays

Arrays are local to one processor (`[]` declaration) or spread across processors ({} declaration). The arrays have as many dimensions as the number of commas included between the brackets, plus one (i.e. `[,]` is a two-dimensional array). The type of the array elements is *BasicType* or a global reference to *BasicType* (which cannot then be primitive) if the `global` keyword is included.

2.3 Standard Default Values

Global references also have default value `null`.

3 Conversions

3.2 Conversions on Reference Values

A local reference to a value of type `T` can be converted to a global reference to a value of type `T`, and is of course a global reference to a value on the current processor.

A global reference `X` to a value of type `T` can be converted to a local reference to a value of type `T` if and only if `X` is a global reference to the current processor.

There is a special case when assigning to arrays of local references, see below.

3.3 Assignment Conversion

Local references are assignable to global references, but not vice-versa except as discussed next.

Assignment of a global reference to an array of local references (typically a spread or global array of local references) is allowed without a cast, but causes a run-time check. The global reference must be to the processor that owns the array element being assigned to. For a local array, that is the current processor. For a global array, it is the processor that owns the array. For a spread array, it is the processor that owns the particular element being assigned to.

5 Program Structure

5.8 Split-Java Startup - NEW

A Split-Java program is started by loading a particular package on all processors. This package must contain a public `main` class, with a public `run` method. An object of this class will be instantiated on all processors, and it's `public void run()` method invoked.

6 Class and Interface Type Declarations

6.3 Variable Declarations

6.3.1 Variable Modifiers

One copy of each field variables declared `static` is stored on each processor. No consistency is maintained between these fields.

6.3.3 Variable Initializers

Remove the definition of *ArrayInitializers* here, see §7.3 for details.

6.4 Method Initializers

6.4.1 Method Modifiers

The *MethodModifiers* also include: `remote`.

If x is a global reference to a type T, then only `remote` methods of T may be called on x. The execution remains local. Within methods qualified with `remote`, `this` and `super` are global references instead of a local ones. Accesses to `static` fields within the method will be remote accesses to the processor owning `this`.

`static` methods may be declared `remote`. If they are invoked by qualifying a global object reference, then references to `static` fields within the will be remote accesses to the processor owning the invoking object. Otherwise the accesses will be to the local data for that class.

A `remote synchronized` method will acquire and release the lock for the object remotely.

6.4.9 Method Overriding

It is not possible to add or remove the `remote` keyword when overriding a method.

6.5 Constructor Method Declaration

6.5.5 Object Creation

Objects never migrate.

6.6 Automatic Storage Management and Finalization

An object will not be deleted as long as it is referenced either locally or remotely.

6.7 Class Loading and Initialization

6.7.1 Class Loading

Class loading is handled independently on each processor. Therefore the set of classes loaded on each processor my be different.

7 Arrays

Global references to arrays may not be assigned to variables of type `Object`, but only to variables of type `global Object`. References to spread arrays may not be assigned to either.

Split-Java arrays are multi-dimensional. The number of dimensions, but not their size, is part of the array type. It is equal to the number of commas between the array specifier (`[]` or {}), plus one. Indices in each dimension range from 0 to $size - 1$.

Split-Java arrays can be *spread*. To indicate a spread array type, use {} instead of `[]`. A spread array has it's elements spread across all processors, in a cyclic distribution: element 0 is on processor 0, element 1 on processor 1, ..., element n (where n = number of processors) on processor 0, and so on. For multi-dimensional arrays, the elements are numbered in column-major order.

As in Java, it is also possible to have arrays of references to arrays, but multi-dimensional arrays should be used where possible.

7.1 Array Types

The elements of arrays are either primitive values or references. By default, these references are assumed to be local. To make an array contain global references, the `[]` or {} specifier must be followed by a `global` keyword. So:

```
A[] x;  // Local reference to an array of local references to A
A[] global x; // Local reference to an array of global references to A
global A[] x; // Global reference to an array of local references to A
```

It is an error to declare a global reference to a spread array.

7.3 Array Initialization

The array initialiser syntax is unchanged, but it's semantics depend on the type of the array.

For multi-dimensional arrays, the appropriate depth of array initialisers is required, and all the lengths at the same depth must be the same (otherwise the multi-dimensional array is ill-formed).

If initialisation includes any spread dimensions, then the same synchronisation requirements are present as for allocation of spread arrays with `new`. See §9.8.2.

7.4 Array Length and Size

On multi-dimensional arrays, the `.length` field contains the total number of elements in the array.

Arrays also have a `.size` field, of type `int[]`, that contains the array's size in each dimension. Element 0 of size is the size of the left-most dimension, element 1 of the next-to-leftmost, and so on.

7.6 Array Allocation and Reclamation

See §9.8.2.

8 Blocks and Statements

8.3 Statements

Statements also include:

Statement:

  ...
  SplitAssignmentStatement
  ArraySliceStatement

8.8 Iteration Statements

Iteration statements also include:

IterationStatement:

  ...
  `scan` (*ComplexPrimaryOrName* $Slice_{opt}$, *Identifier* , *Indices*) *Statement*

ComplexPrimaryOrName:
  ComplexPrimary
  Name

Indices:

Identifier

Indices , *Identifier*

`scan` uses *ComplexPrimaryOrName* instead of *Expression* for syntactic reasons (the grammar is hard to write otherwise).

8.8.4 The `scan` *Statement - NEW*

The `scan` statement iterates over a spread array, returning only the elements local to the current processor. The first element must evaluate to a value of spread array type, the number of indices must be equal to the number of dimensions of this array. The indices must be variables of type `int`, the identifier must be a variable of the type of the elements of the spread array.

The statement is executed once for each element of the array that is stored on the local processor, with the identifier bound to the value of that element and the indices bound to its index. The elements are scanned in column-major order.

If an optional slice is included, the scan is restricted to the elements on the current processor that are in that slice. Slices are defined in §8.13.

8.10 Guarding Statements

Guarding statements also include:

GuardingStatement:

...

`sync` *Statement*

8.10.1 The `synchronized` *Statement*

If the expression evaluates to a global reference, then the lock is acquired and released remotely.

8.10.3 The `sync` *Statement - NEW*

`sync` first executes its statement. It then waits till all split-phase assignments started directly in the statement are complete before execution resumes. It does not wait for split-phase assignments that were started in methods called from within statement to complete, i.e. it only considers lexically enclosed split-phase assignments.

Any jump or `throw` statement that would cause the `sync` statement to terminate is caught and delayed until the split-phase assignments are complete.

8.12 Split-phase Assignment Statements - NEW

Normal reads or writes of values via global references are synchronous. To get asynchronous reads or writes, the special split-phase assignment statements must be used.

SplitAssignmentStatement:

Expression `:=` *Expression* ;

Expression `:-` *Expression* ;

The left hand side must be an assignable expression, i.e. an array element or a field.

For `:=`, if the left hand side is accessed via a global reference or spread array then the statement is a *split-phase write*. The right hand side is evaluated as usual, and a write request is sent to the remote processor specified by the global reference.

If the left hand side is accessed via a local reference, or is a local variable, then the right hand side must be accessed via a global reference or spread array. The statement is a *split-phase read*. A read request is sent to the processor specified by the global reference, the result will be stored in the local variable, field or array element when the read request is answered.

Completion of split-phase assignments can be detected with the `sync` statement or the `java.split.global_sync()` method. See Appendix B.

`:-` is an *unacknowledged store*. The left hand side must be accessed via a global reference or spread array, the right hand side can be any expression. A write request is sent to the processor specified by the global reference, but the writer can not check for completion of this write.

To synchronise stores, `java.split.all_store_sync()` or `java.split.store_sync(`n`)` can be used. See Appendix B.

Both stores and split-phase reads and writes are atomic.

8.13 Array-slice Statements - NEW

ArraySliceStatement:
 ComplexPrimaryOrName Slice SliceAssignmentOp ComplexPrimaryOrName Slice

SliceAssignmentOp:
 `=`
 `:=`
 `:-`

Slice:
 `[` *DimensionSlices* `]`

DimensionSlices:
 DimensionSlice
 DimensionSlices `,` *DimensionSlice*

DimensionSlice:
 Expression `:` *Expression* `:` *Expression*

ComplexPrimaryOrName is used instead of *Expression* for syntactic reasons.

The types of the two arrays must be the same, except that one (and only one[3]) may be a global or spread array reference. Each slice is specified as $[start_1 : end_1 : step_1, start_2 : end_2 : step_2, \ldots]$. The number of dimensions must match the dimension of the array, this is checked at compile-time. The number of elements specified in each dimension (i.e. $(end + 1 - start)/size$) must be the same on the left and right hand sides. If this is not the case, an `ArraySliceIncompatible` exception is thrown at run-time.

If the two array expressions specified by a slice copy evaluate to references to the same array then the copy behaves as if all elements are first read, then all the elements are written. This applies even if one of the array references is remote.

The special assignments operators work on array slices as they do on primitive and reference types. For `:-` the left-hand side must be global or spread and the right-hand side local. The same synchronisation operations are available as with the normal split-phase assignment statements.

The array slice operations are atomic at the array element level only. The copies are not done in any particular order.

[3] If both could be remote then this would either require staging of the copy via the invoking processor, or would be equivalent to remote execution of the statement on one of the processors owning the references. For spread-spread copies, this would be even more complicated.

9 *Expressions*

9.2 Type of an Expression

The discussion of run-time versus compile-time types extends naturally to global object references. Values of type `Object` or `global Object` will never contain references to spread arrays.

The section on run-time errors here seems wrong: the error when assigning to array elements contradicts §3.3 and §9.20.1 on casts in assignments, and should cause an exception anyway. Also failed casts cause exceptions (§3.4). Thus the only cause for run-time errors should be uncaught exceptions.

9.4 Primary Expression

9.4.1 `this` *and* `super`

In `remote` methods, `this` and `super` are global references.

9.5 Array Access

ArrayAccess:
    Name `[` *IndexExpressions* `]`
    ComplexPrimary `[` *IndexExpressions* `]`

IndexExpressions:
    Expression
    IndexExpressions `,` *Expression*

The array expression can evaluate to a local reference to an array of elements of type T, to a global reference to an array of elements of type T or to a reference to a spread array of elements of type T.

The compile-time type of the array must have as many dimensions as there are index expressions. All index expressions must be promotable to type `int`.

An `ArrayIndexOutOfBoundsException` is thrown if any of the indices is out of the bounds of the corresponding dimension.

If the array expression is a global or spread reference to an array, and the type T is a local reference, then special processing occurs on reads and writes of elements: for reads, the local reference is implicitly converted to a global reference to the processor owning the element. Writes must be of global references, there is a run-time check to ensure that the global reference points to the processor owning the element. See §3.3.

9.6 Field Access

9.6.1 Field Access through an Object or Array Reference

The compile-time type of the primary expression may also be a global reference of type T. Arrays have two named fields, `length` and `size`.

If the field is `static`, the access is to the copy of that field on the processor owning the object.

9.6.2 Field Acess through a Simple Name

See §6.4.1 for the rules for accessing `static` fields in `remote` methods.

9.6.3 Field Access through a Qualified Name

See §6.4.1 for the rules for accessing `static` fields in `remote` methods.

9.7 Method Calls

There is one further restriction on method applicability:

- If the invoking object is accessed through a global reference, then the method must be declared `remote`.

9.8 Allocation Expressions

DimExpr is changed to:

DimExpr:
  `[` *DimensionSizes* `]`
  `[` *DimensionSizes* `]` *global*
  { *DimensionSizes* }
  { *DimensionSizes* } *global*

DimensionSizes:
  Expression
  DimensionSizes , *Expression*

Dims:
  `[` *Dimensions* `]`
  `[` *Dimensions* `]` *global*
  { *Dimensions* }
  { *Dimensions* } *global*

Dimensions:
  / empty */*
  , *Dimensions*

Dims:
  `[` *SubDimensions*

9.8.2 Allocating New Arrays

The discussion is mostly unchanged, except that "*DimExpr* expression" should read "*Dimension* expressions".

Explicit multi-dimensional arrays exist alongside arrays of arrays. The arrays whose sizes are given are allocated in the specified size, the last array whose size is given is initialised to its standard default value (§2.3).

If the `new` statement is allocating a spread array (i.e. if any of the dimensions with sizes is specified with {}) then all processes of the Split-Java program must execute this allocation statement simultaneously. Processes will block till everyone reaches this particular statement. All processes must pass the same size arguments for all the arrays and dimensions.

9.9 Postfix Expressions

Variables accessed via global references can be incremented or decremented. The operation is decomposed into a remote read and a remote write, with no atomicity guarantees.

9.10 Unary Operators

The comments of §9.9 above apply also to pre-decrementing and pre-incrementing.

9.15 Equality Operators

9.15.3 Object Equality Operators `==` *and* `!=`

Two global references may be compared for equality, the result is `true` if they refer to the same object, otherwise the result is `false`.

9.20 Assignment Operators

9.20.1 Simple Assignment Operator `=`

One exception to the assignment conversion rule is when assigning to global or spread arrays of local references. These assignments involve a run-time check, to ensure that the local reference being assigned belongs to the processor owning the element. See §3.3.

If the variable being assigned is accessed global, then a synchronous remote write is performed.

9.20.2 Compound Assignment Operators

Variables accessed via global references can be assigned with the compound assignment operators. The operation is decomposed into a remote read and a remote write, with no atomicity guarantees.

10 Collected Java Grammar

The changes described above are incorporated.

B The `java.split` Class

```
public class java.split {
  // Processor information
  public static int nprocessors();
  public static int myproc();

  // Synchronisation primitives
  public static void barrier();
  public static void global_sync();
  public static void all_store_sync();
  public static void store_sync(int n);

  // Broadcasts
  public static int all_bcast_i(int i);
  public static int all_bcast_l(long l);
  public static int all_bcast_d(double d);
  public static global Object all_bcast_o(global Object o);

  // Reductions and scans
```

```
// Reductions
public static int all_reduce_to_one_add(int n);
public static int all_reduce_to_one_mult(int n);
public static int all_reduce_to_one_max(int n);
public static int all_reduce_to_one_min(int n);
public static int all_reduce_to_one_or(int n);
public static int all_reduce_to_one_xor(int n);
public static int all_reduce_to_one_and(int n);
public static int all_reduce_to_all_add(int n);
public static int all_reduce_to_all_mult(int n);
public static int all_reduce_to_all_max(int n);
public static int all_reduce_to_all_min(int n);
public static int all_reduce_to_all_or(int n);
public static int all_reduce_to_all_xor(int n);
public static int all_reduce_to_all_and(int n);

public static long all_reduce_to_one_ladd(long n);
public static long all_reduce_to_one_lmult(long n);
public static long all_reduce_to_one_lmax(long n);
public static long all_reduce_to_one_lmin(long n);
public static long all_reduce_to_one_lor(long n);
public static long all_reduce_to_one_lxor(long n);
public static long all_reduce_to_one_land(long n);
public static long all_reduce_to_all_ladd(long n);
public static long all_reduce_to_all_lmult(long n);
public static long all_reduce_to_all_lmax(long n);
public static long all_reduce_to_all_lmin(long n);
public static long all_reduce_to_all_lor(long n);
public static long all_reduce_to_all_lxor(long n);
public static long all_reduce_to_all_land(long n);

public static double all_reduce_to_one_dadd(double n);
public static double all_reduce_to_one_dmult(double n);
public static double all_reduce_to_one_dmax(double n);
public static double all_reduce_to_one_dmin(double n);
public static double all_reduce_to_all_dadd(double n);
public static double all_reduce_to_all_dmult(double n);
public static double all_reduce_to_all_dmax(double n);
public static double all_reduce_to_all_dmin(double n);

// Scans
public static int all_scan_add(int n);
public static int all_scan_mult(int n);
public static int all_scan_max(int n);
public static int all_scan_min(int n);
public static int all_scan_or(int n);
public static int all_scan_xor(int n);
public static int all_scan_and(int n);

public static long all_scan_ladd(long n);
public static long all_scan_lmult(long n);
public static long all_scan_lmax(long n);
public static long all_scan_lmin(long n);
public static long all_scan_lor(long n);
```

```
  public static long all_scan_lxor(long n);
  public static long all_scan_land(long n);

  public static double all_scan_dadd(double n);
  public static double all_scan_dmult(double n);
  public static double all_scan_dmax(double n);
  public static double all_scan_dmin(double n);
  public static double all_scan_dor(double n);
  public static double all_scan_dxor(double n);
  public static double all_scan_dand(double n);

}
```

B.1 Processor Information

`nprocessors()` returns the number of processors used by this Split-Java program.

`myproc()` returns the number of the current processor, with the first processor numbered 0 and the last one `nprocessors() - 1`.

B.2 Synchronisation Primitives

This section, and the subsequent ones, will use the expression "f must be called simultaneously by all processors" to indicate that all processors must call f. Processes that call f before the others will wait till everyone has called f. Once this has occurred, execution of f will proceed on all processors. It is an error for one processor to call a different such function g if some processors are already waiting in f. If this occurs, the program will be terminated (there are no clear exception semantics for this case).

`barrier()` must be called simultaneously by all processors. It does not do any work.

`global_sync()` waits for all outstanding split-phase reads and writes of this processor to complete.

Each processor has a store counter to count incoming stores. It starts at 0 and is incremented by one for each store of a primitive value or a reference. Stores of array slices increment it by the number of elements in the slice.

`all_store_sync` waits for all stores across all processors to complete. It resets the store counter on all processors to 0. It must be called simultaneously by all processors.

`store_sync`(n) waits for this processor's store counter to reach n, then decrements it by n.

B.3 Broadcasts

The `all_bcast_`x functions must be called simultaneously on all processors. The value of the argument passed on processor 0 is returned as the result of `all_bcast_`x on all processors.

B.4 Reductions and Scans

The `all_reduce_to_`... and `all_scan`... functions must be called simultaneously on all processors. They compute the result of applying the specified operation to the arguments passed by each processor. They can be applied to values of type `int`, `long` or `double`.

They differ in what results the various processors get:

- The `all_reduce_to_one_`...functions return the result to processor 0, and return the standard default value for that type to all other processors.
- The `all_reduce_to_all_`...functions return the result to all processors.
- The `all_scan_`...functions return to each processor the result of applying that operation on all processors with number less than or equal to its own (the scans are inclusive).