Introduction to Split-C[1]

Version 1.0

David E. Culler
Andrea Dusseau
Seth Copen Goldstein
Arvind Krishnamurthy
Steven Lumetta
Steve Luna
Thorsten von Eicken
Katherine Yelick

Computer Science Division — EECS
University of California, Berkeley
Berkeley, CA 94720
Split-C@boing.CS.Berkeley.EDU

April 25, 1995

Split-C is a parallel extension of the C programming language primarily intended for distributed memory multiprocessors. It is designed around two objectives. The first is to capture certain useful elements of shared memory, message passing, and data parallel programming in a familiar context, while eliminating the primary deficiencies of each paradigm. The second is to provide efficient access to the underlying machine, with no surprises. (This is similar to the original motivation for C—to provide a direct and obvious mapping from high-level programming constructs to low-level machine instructions.) Split-C does not try to obscure the inherent performance characteristics of the machine through sophisticated transformations. This combination of generality and transparency of the language gives the algorithm or library designer a concrete optimization target.

This document describes the central concepts in Split-C and provides a general introduction to programming in the language. Both the language and the document are undergoing active development, so please view the document as working notes, rather than the final language definition.

[1]This work was supported in part by the National Science Foundation as a Presidential Faculty Fellowship (number CCR-9253705), Research Initiation Award (number CCR-9210260), and Infrastructure Grant (number CDA-8722788), by Lawrence Livermore National Laboratory (task number 33), by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by the Semiconductor Research Consortium under contracts 92-DC-008 and 93-DC-008, and by AT&T. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Contents

1 Introduction

Split-C is a parallel extension to the C programming language designed for large, distributed memory multiprocessors. Following the C tradition, Split-C is a general-purpose language, but not a "very high level" language, nor a "big" one. It strives to provide the programmer enough machinery to construct powerful parallel data structures and operate on these in a machine independent fashion with reasonable clarity. At the same time, it does not attempt to hide the fundamental performance characteristics of the machine through elaborate language constructs or visionary compilation. Whereas C "deals with the sort of objects that most sequential computers do,"[1] the extensions in Split-C deal with the additional operations that most collections of computers support. In either case, we expect the compiler to be reasonably good at address calculations, instruction scheduling, and local storage management, with the usual optimizations that pertain to these issues.

Large-scale multiprocessors introduce two fundamental concerns: there is an active thread of control on each processor and there is a new level of the storage hierarchy which involves access to remote memory modules via an interconnection network. The Split-C extensions address these two concerns under the assumption that the programmer must think about these issues in designing effective data structures and algorithms and desires a reasonable means of expressing the results of the design effort. The presence of parallelism and remote access should not unduly obscure the resulting program. The underlying machine model is a collection of processors operating in a common global address space, which is expected to be implemented as a physically distributed collection of memories. The global address space is two dimensional from the viewpoint of address arithmetic on global data structures and from a performance viewpoint, in that each processor has efficient access to a portion of the address space. We may call this the *local portion* of the global space. Split-C provides access to global objects in a manner that reflects the access characteristics of the interprocessor level of the storage hierarchy.

Split-C attempts to combine the most valuable aspects of shared memory programming with the most valuable aspects message passing and data parallel programming within a coherent framework. The ability to dereference global pointers provides access to data without prearranged co-ordination between processors on which the data happens to reside. This allows sophisticated, linked data structures to be constructed and used. Split-phase access, *e.g.*, prefetch, allows global pointers to be dereferenced without causing the processor to stall during access. The global address space and the syntactic support for distributed data structures provides a means of documenting the global data structures in the program. This global structure is usually lost with traditional message passing, because it is implicit in the communication patterns. Algorithms that are natural to state in terms of message passing are more efficient within a global address framework with bulk transfer; they are as easy to express, and the fundamental storage requirements of the algorithm are made explicit. Traditional shared-memory loses the inherent *event* associated with transfer of information, so even simple global operations such as tree summation are hard to express efficiently. Split-C allows notification to be associated with access to the global addresses using an approach similar to split-phase access. Data parallel programming involves phases of local computation and phases of global communication. The global communication phases are often very general, say scattering data from each processor to every other, so the global address is very useful, but there is no need to maintain consistency on a per-operation basis. Split-C is built upon an active message substrate[AM], so the functionality of the language can easily be extended by libraries that use the lowest level communication primitive directly, while providing meaningful abstractions within a global address framework.

This paper is intended to introduce the pilot version of Split-C. Section 2 provides an overview of the basic concepts in the language. Sections 3 through 8 explain these concepts in more detail, describe the syntax and provide simple examples. Section 9 discusses optimization strategies, and Section 10 lists the library functions available to the Split-C programmer as well as the primitives used by the Split-C compiler.

2 Split-C Primitives Overview

The extensions introduced in Split-C attempt to expose the salient features of modern multiprocessor machines in a generic fashion. The most obvious facet is simply the presence of multiple processors, each following an independent thread of control. More interesting is the presence of a very large address space that is accessed by these threads. In all recent large-scale multiprocessors this is realized by storage resources that are local to the individual processors. This trend is expected to continue. Split-C provides a range of access methods to the global address space, but encourages a "mostly local" programming style. It is anticipated that different architectures will provide varying degrees of support for direct access to remote memory. Finally, it is expected that global objects will often be shared and this requires an added degree of control in how they are accessed.

Split-C provides the following extensions to C:

- *Multiple persistent threads:* A Split-C program is parallel *ab initio*. From program begin to program end there are `PROCS` threads of control within the same program image.[2] Each thread has a unique number given by a special variable `MYPROC` that ranges from 0 to `PROCS`-1. Generally, we will use the term processor to mean the thread of control or process on that processor. A variety of convenient parallel control structures can be built on this substrate and several are provided as C preprocessor (`cpp`) macros, but the basic language definition does not prescribe dynamic thread manipulation or task scheduling. A small family of global synchronization operations are provided to co-ordinate the entire collection of threads, *e.g.*, `barrier`. No specific programming paradigm, such as data parallel, data driven, or message passing, is imposed by the language. However, these programming paradigms can be supported as a matter of convention.

- *2D Global Address Space:* Any processor can access any object in a large global address space. However, the inherent two dimensional structure of the underlying machine is not lost. Each processor "owns" a specific region of the address space and is permitted to access that region via standard, local pointers. Rather than introducing a complicated set of mapping functions, as in Fortran-D, or mysterious mappings in the run-time system, as in CM-Fortran or C*, simple mapping rules are associated with multidimensional structures and global pointer types. Sophisticated mappings are supported by exploiting the relationship between arrays and pointers, as is common in C.

- *Global pointers:* A global pointer refers to an arbitrary object of the associated type anywhere in the system. We will use the term *global object* to mean an object referenced by a global pointer. A global object is "owned" entirely by a processor, which may have efficient access to

[2] This is termed the *split-join* model in [Brooks].

the object though standard pointers. A new keyword `global` is introduced to qualify a pointer as meaningful to all processors. Global pointers can be dereferenced in the same manner as standard pointers, although the time to dereference a global pointer is considerably greater than that for a local pointer, perhaps up to ten times a local memory operation (i.e., a cache miss). The language provides support for allocating global objects, constructing global pointers from local counterparts, and destructuring global pointers. (In general, global objects may contain local pointers, but such pointers must be interpreted *relative* to the processor owning the global object.)

A pointer in C references a particular object, but also defines a sequence of objects that can be referenced by arithmetic operations on the pointer. In Split-C the sequence of objects referenced by a standard pointer are entirely local to the processor. Address arithmetic on a `global` pointer has the same meaning as arithmetic on a standard pointer by the processor that owns the object. Hence, all the objects referenced relative to a `global` pointer are associated with one processor.

- *Spread pointers:* A second form of global pointer is provided which defines a sequence of objects that are distributed or *spread* across the processors. The keyword `spread` is used as the qualifier to declare this form of global pointer. Consecutive objects referenced by a spread pointer are "wrapped" in a helical fashion through the global address space with the processor dimension varying fastest. Each object is entirely owned by a single processor, but the consecutive element, (*i.e.*, that referenced by `++`) is on the next processor.

- *Spread arrays:* The duality in C between pointers and arrays is naturally extended to spread pointers and arrays that are spread across processors, called spread arrays. Spread arrays are declared by inserting a "spreader", `::`, which identifies the dimensions that are to be spread across processors. All dimensions to the left of the spreader are wrapped over the processors. Dimensions to the right of the spreader define the object that is allocated within a processor. The spreader position is part of the static type, so efficient code can be generated for multidimensional access. Indexing to the left of the spreader corresponds to arithmetic on `spread` pointers while indexing to the right of the spreader corresponds to arithmetic on `global` pointers. The `&` operator applied to an array expression yields a pointer of the appropriate type. Generic routines that operate independent of the input layout utilize the duality between arrays and pointers to eliminate the higher dimensions.

- *Split-phase assignment:* A new assignment operator, `:=`, is introduced to split the initiation of a global access from the completion of the access. This allows the time of a global access to be masked by other useful work and the communication resources of the system to be effectively utilized. In contrast, standard assignments stall the issuing processor until the assignment is complete, to guarantee that reads and writes occur in program order. However, there are restrictions on the use of split assignments. Whereas the standard assignment operator describes arbitrary reads and one write, the split assignment operator specifies either to *get* the contents of a global reference into a local one or to *put* the contents of a local reference into a global one. Thus, arbitrary expressions are not allowed on the right hand side of a split assignment. The `:=` initiates the transfer, but does not wait for its completion. A `sync` operation joins the preceeding split assignments with the thread of control. A local variable assigned by a get (similarly, a global variable assigned by a put) is guaranteed to have its new value only after the following `sync` statement. The value of the variable prior to the `sync` is not defined. Variables appearing in split assignments should not be modified (either directly or through aliases) between the assignment and the following `sync`, and variables on the left hand side should not be read during that time. The order in which puts take effect is only constrained by `sync` boundaries; between those boundaries the puts may be reordered. No limit is placed on the number of outstanding assignments.

- *Signaling assignment:* A weaker form of assignment, called *store* and denoted `:-`, is provided to allow efficient data driven execution and global operations. Store updates a global location, but does not provide any acknowledgement of its completion to the issuing processor. Completion of a collection of such stores is detected globally using `all_store_sync`, executed by all processors. For global data rearrangement, in which all processors are cooperating to move data, a set of stores by the processors are followed by an `all_store_sync`. In addition, the recipient of store can determine if certain number of stores to it have completed using `store_sync`, which takes the expected number of stores and waits until they have completed. This is useful for data driven execution with predictable communication patterns.

- *Bulk assignment:* Transfers of complete objects are supported through the assignment operators and library routines. The library operations allow for bulk transfers, which reflect the view that, in managing a storage hierarchy, the unit of transfer should increase with the access time. Moreover, bulk transfers enhance the utility of split-phase operations. A single word get is essentially a binding prefetch. The ability to prefetch an entire object or block often allows the prefetch operation to be moved out of the inner loop and increases the distance between the time where the get is issued and the time where the result is needed. The assignment and split-assignment operators transfer arbitrary data types or structs, as with the standard C assignment. However, C does not provide operators for copying entire arrays. Bulk operations are provided to operate on arrays.[3]

- *Synchronizing assignment:* Concurrent access to shared objects, as occurs in manipulating

[3] It is anticipated that Split-C will support range or triplet syntax, *ala* Fortran90, to copy portions of arrays.

linked data structures, requires that the accesses be protected under a meaningful locking strategy. Split-C libraries provide a variety of atomic access primitives, such as fetch-and-add, and a general facility for constructing locking versions of structs and manipulating them under mutual exclusion, single writer multiple reader, or other strategies.

3 Control Paradigm

The control paradigm for Split-C programs is a single thread of control on each of `PROCS` processors from the beginning of `splitc_main` until its completion. The processors may each follow distinct flow of control, but join together at rendezvous points, such as `barrier()`. It is a SPMD model in that every processor executes the same logical program image. Each processor has its own stack for automatic variables and its own static or external variables. Static spread arrays and heap objects referenced global pointers provide the means for shared data. Processors are numbered rom 0 to `PROCS` $-$ 1, with the pseudo-constant `MYPROC` referring to the number of the executing processor.

Figure 1 shows a simple Split-C program to compute an approximation of π through a Monte Carlo integration technique. The idea is to throw darts into the unit square, $[0,1) \times [0,1)$ and compute the fraction of darts that hit within the unit circle. This should approximate the ratio of the areas, which is $\pi/4$. Although the example is contrived, it illustrates several important aspects of the language.

All processors enter `splitc_main` together. They can each obtain command line arguments in the usual fashion. In this case the total number of trials is provided; this represents the work that is to be divided among the processors. Each processor computes the number of trials that it is to perform, initializes its random number generator with a seed based on the value of `MYPROC`, and conducts its trials. The processors join at the `barrier` and then all co-operate to sum the `hits` into `total_hits` on processor 0. Finally, processor 0 prints the result.

In general, the code executed by different processors is varied using a standard library of control macros. These typically involve a test of `MYPROC`, as in the case of `on_one` which tests for `MYPROC` $= 0$. More interesting macros, such as `for_my_2D`, will appear in later examples; this is used for iterating over sets of indexes that correspond to locally owned data. The library contains a set of these control macros for hiding the index arithmetic in some common control patterns, and users can easily define their own.

Split-C programs may mix this kind of control parallelism, in which different processors are executing different code, with data parallelism, in which global operations such as scans or reductions require the involvement of all processors. The global operations are provided by library operations, which by convention are named with the prefix `all_`. The assumption is that all processors execute these within a reasonably short time frame. If some processors are significantly behind the others, then performance will degrade, and it some processors fail to execute the operation at all, the program may hang. We will discuss these global operations further in Section 5, since they are frequently used with spread arrays.

All Split-C files should include `<split-c/split-c.h>`, which defines language primitives, such as `barrier` and pseudo-constants, such as `MYPROC` and `PROCS`. (The current installation uses `.sc` as the file type for Split-C files, which may call normal C routines for local computations.) Most Split-C files will also include the standard control macros in `<split-c/control.h>`. The integer reduction under addition is one of the standard global communication operations in `<split-c/com.h>`. We will look more closely at how this can be implemented as we introduce more of the language.

Implementation note: Example programs with `gmake` files for the current release can be found in `/usr/cm5/local/src/split-c/examples`. The code in this tutorial is in the `tutorial` subdirectory there.

The example illustrates a bulk synchronous programming style that arises quite frequently in Split-C. In this style programs are typically constructed as a sequence of parallel phases. Often the phases alternate between purely local computation and global communication, as in this example. Notice also that the program works on any number of processors, even though the number of processors is exposed. The results will not be quite identical because of the initialization of the random number generator. By default, the compiler produces configuration independent code. By following a few simple conventions it is possible to optimize for the machine size, yet run on any configuration.

Another common style is to allow the threads to co-operate is a less structured fashion through operations on shared objects. Split-C supports both styles and a variety of others. The following sections focus on how the various forms of interaction between processors are supported in Split-C.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <split-c/split-c.h>
#include <split-c/control.h>
#include <split-c/com.h>

int hit()
{
  int const rand_max = 0xFFFFFF;
  double x = (double) (rand()&rand_max)/(rand_max);
  double y = (double) (rand()&rand_max)/(rand_max);
  if ((x*x+y*y) <= 1.0) return(1);
  else return(0);
}

splitc_main(int argc, char **argv)
{
  int i, total_hits, hits = 0;
  double pi;
  int trials, my_trials;
  if (argc != 2)
    trials = 1000000;
  else
    trials = atoi(argv[1]);

  my_trials = (trials + PROCS - 1 - MYPROC)/PROCS;

  srand(MYPROC*17);                          /* Different seed on each processor */
  for (i=0; i < my_trials; i++) hits += hit();
  barrier();

  total_hits = all_reduce_to_one_add(hits);
  on_one {
    pi = 4.0*total_hits/trials;
    printf("PI estimated at %f from %d trials on %d processors.\n",
                  pi, trials, PROCS);
  }
}
```

Figure 1: Example Split-C program computing and approximation to pi using a parallel Monte Carlo integration technique.

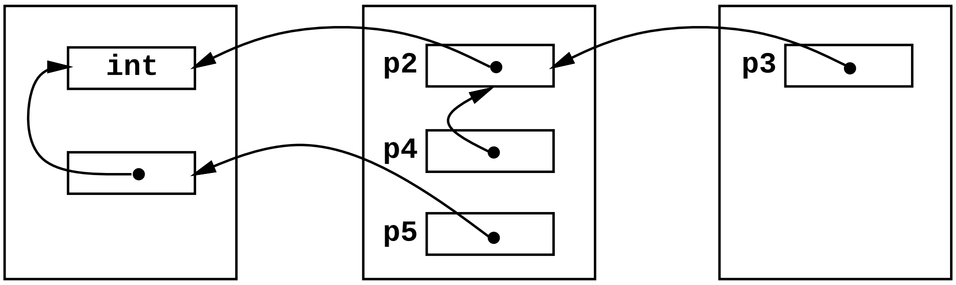

Figure 2: A schematic representation of one possible configuration of the global pointers discussed in the text. The large boxes represent local address spaces and the small boxes represent individual locations

4 Global Pointers

A global pointer is used to refer to an arbitrary object in the global address space. Global pointers of arbitrary type can be declared, but the object is owned entirely by a single processor.

4.1 Declaring global pointers

A global pointer is declared by appending the new keyword `global` to the pointer declaration. For example,

```
int *global P1;
```

declares `P1` to be a pointer to an integer in the global address space, *i.e.*, on any processor.

The `global` type qualifier can be used with any pointer type. Complicated pointer types are understood by reading from right to left, *i.e.*, `P1` is a global pointer to an integer. Several examples follow and are shown schematically in Figure 2.

```
typedef int *global gptr;
gptr P2;
int *global *global P3;
int *global *P4;
int **global P5;
```

`P2` has the same type as `P1`. `P3` is a global pointer that refers to a global pointer to an integer. `P4` is a local pointer to a global pointer to an integer. `P5` is a global pointer to a local pointer to an integer. Generally, this would need to be dereferenced on the processor that owns the local pointer. Alternatively, it could be coerced into a complete global pointer, as described below.

4.2 Constructing global pointers

Global pointers can be constructed by casting a local pointer into a global form. For example,

```
char *global P = (char *global) malloc(16);
```

allocates an character array of 16 bytes on the local processor and creates a global pointer to it. Similarly, the following produces a global pointer to a static variable `x`. [4]

```
static int x;
...

int *global G = (int *global) &x;
```

More generally, a global pointer can be formed from a processor number and a local pointer on that processor using `toglobal`, which has the following signature.

```
void *global toglobal(int proc, void *lptr);
```

Thus, the cast expression above is equivalent to the following.

```
int *global G = toglobal(MYPROC, &x);
```

The cast expression sets the processor part of the address to the pseudo-constant `MYPROC`, which is the processor number of the local processor.

4.3 Destructuring a global pointer

The inverse operation, destructuring a global pointer into its processor part and its local address part, can also be done with either a cast or more generally with library functions. The local address portion of a global pointer is most simply extracted by casting it to a local pointer.

```
int *l = (int *) G;
```

Alternatively, a library function, `tolocal`, is provided to extract the same information.

```
int *l = (int *) tolocal(G);
```

The processor portion of a global pointer is extracted with the function `toproc`.

[4] Currently, Split-C does not recognize a "private" storage class, as is common on shared-memory systems. In the interests of good programming style and compatability with future implementations of Split-C, it is advisable to exercise restraint in globalizing private local objects.

```
int p = toproc(G);
```

4.4 Using global pointers

Global pointers can be used in the same manner as standard pointers; they can be assigned, dereferenced, passed as parameters to functions, returned as values, declared as a field in a struct, and so on.[5] Arithmetic on global pointers reflects the assumption that a global pointer refers to an object that is entirely owned by a processor. The right-hand-side of the assignment to a global object can be an arbitrary expression of the correct type, and global objects can be used in the same manner as local objects within expressions. [6]

The following example copies a global string into a local string up to a specified length. Observe that the standard tests for null pointers work for global pointers. (A null global pointer always has a zero processor part.) Pointer arithmetic is meaningful for global pointers, as discussed below. This version performs a global access for every character. A more efficient version will be described in the optimization section below.

```
char *gstrn_read(char *dst, char *global src, int maxlen)
{
    int i;

    if(src == NULL)
        return 0;
    for(i = 0; i < maxlen; i++) {
        *dst = *src;
        if(*dst != 0)
            src++;
        dst++;
    }
    return dst;
}
```

The use of global pointers in structs, in function calls, as result values, and in expressions is illustrated in the following example which locates the node of a binary tree containing a target value. The `typedef` defines a global tree, where each node contains an integer value and global pointers to the left and right subtrees.

```
typedef struct global_tree *global gt_ptr;
typedef struct global_tree {
    int value;
    gt_ptr left;
    gt_ptr right;
} g_tree;
```

[5] One exception is that global pointers cannot refer to procedures.

[6] Some limitations of the current implementation are discussed in the appendix.

Declaration, assignment, and dereferencing such global objects is just as in C. The following assignment sets the `value` field of the left-left grandchild of `*t` to 3. Each of the pointer dereferences involves a global access.

```
gt_ptr t;

t->left->left->value = 3;
```

Finally, here is the recursive lookup function. For input, it requires a global pointer to the root of the tree and a target value. The assumption is that all the processors could be performing lookups simultaneously. To avoid contention for the root, it may be necessary to produce a multi-rooted tree by replicating the top few levels.

```
gt_ptr lookup(int target, gt_ptr root)
{
    int mid;

    if (root == NULL)
        return NULL;
    mid = root→value;
    if (mid == target)
        return root;
    else {
        if(mid < target)
            return lookup(target, root→left);
        else
            return lookup(target, root→right);
    }
}
```

4.5 Arithmetic on global pointers

Arithmetic on global pointers is dictated by the view that a global pointer refers to an object that is owned entirely by one processor. However, there are two useful views of the "next" element, in the sense of `++`. The form of global pointer declared by `global` supports the view that arithmetic on the pointer has the same effect as arithmetic on a standard pointer. Successive objects are on the same processor as the object referenced by the pointer. For example,

```
i = *(gptr + 4);
j = gptr[4];
```

both copy the fourth object beyond the one referred to by `gptr` from the processor that owns that object. As it makes no sense for standard pointers to wrap around off the top of the address space, arithmetic on `global` pointers that goes outside the local address space of the specified processor is meaningless. There is no wrap-around from one processor to another. The address space defined by `global` pointers is strictly two-dimensional; arithmetic in one dimension does not effect the other dimension. The view that successive objects are on successive processors is supported by `spread` pointers, discussed below.

The following example, which extends the standard `strncpy` function to copy a local string to a global string, demonstrates the use of local and `global` pointer arithmetic, as did the earlier `gstrn_read` example.

```
char *global gstrn_write(char *global dst, const char *src, int n)
{
    char *global g = dst;
    int i;

    for(i = 0; i < n; i++) {
        *(g++) = *src;
        if(*src ≠ 0)
            src++;
    }
    return dst;
}
```

4.6 Spread Pointers

A second form of global pointer, called a *spread pointer*, provides convenient access to collections of objects that are distributed across the processors. Spread pointers are declared using the keyword `spread` in place of `global`. The key difference is that successive objects are on successive processors. Local and global pointers can be cast into spread pointers and vice versa, as shown below.

```
int *global gptr;
int *spread  sptr;
sptr = (int *spread) gptr;
```

Address arithmetic on spread pointers defines a one dimensional address space over the entire machine, in units of the object size, with the processor portion varying fastest. Thus, the `++` operator increments a spread pointer to refer to the object on the next processor. If the processor part of the spread pointer is less than `PROCS` $-$ 1, then the processor part is incremented and the address remains the same. If the processor part equals `PROCS` $-$ 1, then the address part of the spread pointer is incremented by the object size and the processor part is reset to zero. In the above example, `gptr + 1` and `sptr + PROCS` refer to the same object.

4.7 Using spread pointers

The following example provides an illustration the use of spread pointers. We assume that a large hash table has been allocated across the processors and and several name-value pairs have been entered into it. (We discuss allocation and static declaration in the next section.) Any of the processors can lookup values in the hash table using the following `lookup` routine. The hash table is passed in as a spread pointer. A hash function is applied to the target name string to get an offset from the base of the hash table. Collisions are resolved by finding the next empty slot, and the last entry in the hash table is assumed to be NULL. The hash table may be much larger than the number of processors; the increments to the probe pointer simply traverse the array.

```
typedef struct {
    char *global name;
    char *global value;
} hash_entry;

char *global lookup(hash_entry *spread table, char *target)
{
    hash_entry *spread probe = table + hash(target);
    char *global probe_name = probe->name;
    while(probe_name) {
        if(g_strcmp(probe_name, (char *global) target) == 0)
            return probe->value;
        probe += 1;
        probe_name = probe->name;
    }
    return NULL;
}
```

5 Spread Arrays

Array indexing in C is defined using pointer arithmetic. In the previous section we saw that Split-C carries over this definition for indexing into arrays of objects that are all owned by one processor using arithmetic on global pointers. Analogously, indexing into arrays of objects that are distributed across processors is defined in Split-C using arithmetic on spread pointers.

5.1 Declaring spread arrays

An array can be declared as a *spread* array by inserting a pair of colons, `::`, called the *spreader*, in the array declaration. All dimensions to the left of the spreader are interpreted as spread across processors, where the per processor object is defined by the dimensions to the right of the spreader and the data type. For example, the simple reduction operation used in the calculation of π can be written with a spread array as follows. As indicated by its name, the function is entered by all of the processors with a local value and provides the sum of the values to processor 0.

```
static int reduce_buf[PROCS]::;

int all_reduce_to_one_add(int val)
{
	int i, sum = 0;

	reduce_buf[MYPROC] = val;
	barrier();
	on_one {
		for(i = 0; i < PROCS; i++)
			sum += reduce_buf[i];
	}
	return sum;
}
```

The spread array `reduce_buf` is statically allocated so that each processor owns a single integer. All processors deposit their contribution to the sum into the array (this is a local access using a distributed object) and processor 0 sums the entire array after all the deposits are complete. This serial implementation of reduction is clearly inefficent; a better approach would be to reduce the values as a tree. We will consider this later.

Spread arrays are either allocated as external variables (declared at the top level), or explicitly allocated in the heap using `all_spread_malloc`. They cannot be allocated on the stack. Thus, the following declaration of `A` could appear as an external variable or as a parameter to a procedure, since arrays in C are passed by pointer and require no storage allocation; it could not appear as a declaration within a procedure. If this declaration appears as an external variable, there are further restrictions on n and m as described below.

```
double A[n][m]::[b][b];
```

Spread array declarations fall into two other categories, depending on whether the amount of storage allocated per processor is machine-size independent or machine-size dependent. In the above definition, `A` is an $n \times m$ array of $b \times b$ blocks. It is spread across `PROCS` processors a block at a time. Thus, the total size of `A` is independent of the number of processors, but the extent of `A` on each processor, (namely $\lceil \frac{8nmb^2}{\texttt{PROCS}} \rceil$ bytes), depends on the number of processors. In contrast, the size of `rBodies` and `x` given below depend on the number of procesors, but the extent on each processor does not. Both are spread arrays, with `rBodies` having `m` bodies per processor, and `x` having a single integer per processor. [7]

```
body rBodies[PROCS]::[m];
int x[PROCS]::;
```

External spread arrays must be declared to be a multiple of `PROCS` in exactly one of the spread dimensions. This ensures that the load image produced by the linker is independent of the number of processors. Dynamically allocated spread arrays (discussed below) have no such restriction.

> *Implementation note:* The Split-C compiler, split-cc, provides a command line option (`-PROCS=` as an optimization to fix the number of processors in the compilation target. Under this option the restriction on the spread array declarations is lifted.

Our previous hash-table lookup example can be revised to use a one dimensional spread array of length `size` containing `hash_entry`.

```
char *global lookup(int size, hash_entry table[size]::, char *target)
{
    int h = hash(target, size);
    char *global name = table[h].name;
    while(name) {
        if(g_strcmp(name, (char *global) target) == 0)
            return table[h].value;
        name = table[++h].name;
    }
    return NULL;
}
```

The duality with spread pointers is essential. The pointer equivalent for a spread dimension in a spread array of some type is to the right of the spreader. Eliminating the last spread dimension yields a spread pointer to the basic per processor object. Indexing further into the object yields a `global` pointer.

[7] It is possible to coerce a standard pointer into a spread pointer, if the standard pointer refers to an external variable. This works because the external variable exists on all processors at the same local address. However, this is discouraged, since the same effect can be acheived by referring to a static spread array of extent `PROCS`. The latter declaration makes the intention clear and is likely to be more portable.

5.2 Dynamic allocation of spread objects

Distributed objects referenced by spread pointers can be created through dynamic allocation of storage on the "spread heap." The function `all_spread_malloc` must be entered by all processors to allocate a number of objects of a specified size across the processors. The result is a spread pointer to the base of the rectangle on processor 0. The number of objects and the object size are provided separately so that no individual object is split across processors. A spread pointer with the processor part equal to zero is said to be *normalized*, since it provides a consistent reference to such a rectangular region.

The following is one possible implementation of the `all_reduce_to_one_add` function using `all_spread_malloc`. In this case, all of the processors enter the together and return with a spread pointer to a rectangular window across the processors, one integer in height.

```
int all_reduce_to_one_add(int val)
{
    int i, sum = 0;
    int *spread w = all_spread_malloc(PROCS, sizeof(val));
    int *loc = tolocal(w);                          /* loc points to w on MYPROC */

    *loc = val;                                     /* All fill the spread array */
    barrier();

    on_one {                                        /* Sum across processors     */
      for(i = 0; i < PROCS; i++)
            sum += *(w++);
    }
    barrier();
    all_spread_free(w);
    return sum;                                     /* Processor 0 obtains result */
}
```

In this case, we have coerced a spread pointer to a local pointer by simply extracting the local part. The following would be an equivalent statement.

```
int *loc = (int *) &w[MYPROC];
```

The `++` on the use of `w` serves to iterate across the processors.

In the following, all the processors join in allocating a rectangle of total size equal to 100*`PROCS` doubles and each receives a normalized spread pointer. To traverse the sequence of objects allocated to a particular processor (not necessarily the processor executing the traversal), a spread pointer can be cast to a global pointer. To traverse objects allocated on the executing processor, the spread pointer can be cast to a local pointer. In the example, each processor produces a local pointer to the tenth element of their own portion.

```
splitc_main ()
{
    double *spread sptr = (double *spread) all_spread_malloc(100*PROCS, sizeof(double));
    double *lptr = (tolocal(sptr)) + 10;
}
```

5.3 Address arithmetic

The spreader position is part of the static type of the object. Address generation involves linearizing in row major order the subscripts to the left of the spreader and to the right. The left indexes locate a block in the entire machine, *i.e.*, a global address. The right indexes locate the offset within the block, *i.e.*, the local displacement relative to this global address.

Consider again the declaration of `A`.

```
double A[n][m]::[b][b];
```

If `A` is normalized, *i.e.*, the first element is on processor 0, then block of `A[i][j]` is on processor $(im + j) \bmod \texttt{PROCS}$ and its offset, relative to that of `A`, is $8b^2 \lfloor \frac{im+j}{\texttt{PROCS}} \rfloor$ bytes. In general, if S is a spread pointer to an object of size b, then $S + i$ refers to processor $(S.proc + i) \bmod \texttt{PROCS}$ and address $S.local + b\lfloor \frac{S.proc+i}{\texttt{PROCS}} \rfloor$.

As illustrated below, this approach provides quite a bit of power without mapping functions. The mapping is described entirely by the multidimensional structure and the spreader position.

5.4 Configuration independent use of spread arrays

We illustrate by example how spread arrays can be used in efficient, configuration independent programming. Our example is a global matrix multiplication. We start with the following code to perform the multiplication on a single processor: matrix `A` will be multiplied with matrix `B`, accumulating the result into matrix `C`. (This style of array declarations is quite familiar to Fortran programmers, but may raise a few eyebrows among C aficionados. It relies heavily on the GCC extensions, as does Split-C.)

```
void matrix_mult(int n, int r, int m,
                 double C[n][m],
                 double A[n][r],
                 double B[r][m])
{
        int i, j, k;

        for(i = 0; i < n; i++)
          for (j = 0; j < m; j++)
            for (k = 0; k < r; k++)
              C[i][j] = C[i][j] + A[i][k]*B[k][j];
}
```

We can produce a global matrix multiplication by simply spreading the matrices and the inner products across the processors, as follows:

```
#include <split-c/split-c.h>
#include <split-c/control.h>

void all_matrix_mult(int n, int r, int m,
                 double C[n][m]::,
                 double A[n][r]::,
                 double B[r][m]::)
{
    int i, j, k, tmp;

    for_my_2D(i, j, tmp, n, m)
        for (k = 0; k < r; k++)
            C[i][j] = C[i][j] + A[i][k]*B[k][j];
    barrier();
}
```

As indicated by the `all_` prefix this is entered by all the processors. The `control.h` macro `for_my_2D` serves to distribute work according to the "owner computes" rule. It iteratively binds `i` and `j` to the elements of the `n` by `m` domain of `C` which are assigned to the local processor, assuming `C` is a normalized spread pointer. (It also binds the middle parameter to the linear index from the base of the array, but our example discards that information.) Analogous macros for other dimensions are provided in `split-c/control.h`.

Knowledge of the owner-computes rule can be exploited to yield the slightly more efficient version below, which operates on local pointers where possible.

```
void all_matrix_mult(int n, int r, int m,
                     double C[n][m]::,
                     double A[n][r]::,
                     double B[r][m]::)
{
    int i, j, k, l;

    for_my_2D(i, j, l, n, m) {
        double *c = &C[i][j];

        for (k = 0; k < r; k++)
            *c = *c + A[i][k]*B[k][j];
    }
    barrier();
}
```

Notice that this routine operates correctly regardless of the layouts of the three matrices. In fact, since the three arrays have different dimensions, there need not be any simple relation among their various rows and columns. The processors need not all perform the same number of inner-products. If `m` and `r` happen to be multiples of `PROCS`, then the matrices are mapped to processors by columns.

However, this example is still not very efficient, because two global references are made for each multiply-add. To improve the computation to communication ratio, we need to obtain a local block of `A` and a local block of `B` and do a local matrix multiplication of blocks. This can be done using the data structures given above, but an elegant way to express the algorithm is to introduce additional dimensions to represent the blocks as sub-matrices. This means the compiler, rather than the programmer, does the messy address arithmetic. The result is as follows:

```
void all_matrix_mult(int n, int r, int m, int b,
                     double C[n][m]::[b][b],
                     double A[n][r]::[b][b],
                     double B[r][m]::[b][b])
{
    int i, j, k, l, ii, jj;
    double la[b][b], lb[b][b], (*lc)[b][b];

    for_my_2D(i, j, l, n, m) {
        lc = (double *([b][b])) C[i][j];
        for (k = 0; k < r; k++) {
            bulk_read(la, A[i][k], sizeof(**la)*b*b);
            bulk_read(lb, B[k][j], sizeof(**lb)*b*b);
            matrix_mult(b, b, b, *lc, la, lb);
        }
    }
    barrier();
}
```

Since C does not provide operators on arrays, we need to resort to subroutine calls for the sub-matrix reads and the multiplication. Fortunately we can use our local C version of matrix multiply to operate on the blocks, or some more highly optimized version of the local multiply.

We will return to this example later to show how to gain greater performance by overlapping remote references with computation. We should observe, however, that if the matrix dimensions happen to be multiples of `PROCS`, then all the processors read blocks from a small number of processors at each step. This can create contention for the processors holding the data. The problem can be prevented by having the processors start at a different `k` indexes, staggering the loop by `MYPROC`.

5.5 Configuration dependent use of spread arrays

We now consider an example where we want to exploit the machine configuration to obtain close to peak performance. The example computes the forces between a collection of bodies under a gravitational force law. Using units such that G=1, we use the following notation: (x_i, y_i) is the position of body$_i$, $d_{i,j}$ is the distance between body$_i$ and body$_j$ (forced to be at least 0.01 units), m_i is the mass of body$_i$, $a_{i,j}$ is the magnitude of the acceleration of body$_i$ caused by body$_j$, and $(a^x_{i,j}, a^y_{i,j})$ is the acceleration vector of body$_i$ caused by body$_j$. The acceleration is given by the equations:

$$
\begin{aligned}
d_{i,j} &= max(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, 0.01) && (1) \\
a_{i,j} &= \frac{m_j}{d^2_{i,j}} && (2) \\
(a^x_{i,j}, a^y_{i,j}) &= a_i(\frac{(x_j - x_i)}{d_{i,j}}, \frac{(y_j - y_i)}{d_{i,j}}) && (3) \\
& && (4)
\end{aligned}
$$

The program operates on a number of bodies that is a multiple of `PROCS`. (Presumably in the main program the number of bodies is padded in some manner to make this true.) Thus, the collection of bodies is described by a matrix where the first dimension is `PROCS` and the second is the number of bodies per processor. A body is represented by a struct containing its position co-ordinates, `x` and `y`, its velocity, `u` and `v`, and its mass, `m`.

The function `all_nbody` is entered concurrently on all processors with a two dimensional spread array of bodies. Each obtains a standard pointer to the local portion of the array and allocates a temporary array for use in pairwise force calculations. It iterates over each of the rows of bodies, making a local copy of the row and accumulating the pairwise interactions. The local copy is a significant optimization, since it is used repeatedly in the force calculations.

```
typedef struct {
      double x, y;                                        /* location */
      double u, v;                                        /* velocity */
      double m;                                           /* gravitational mass */
} body;


void all_n_body( int m, body g_bodies[PROCS]::[m])
{
      int ip, i, j;
      double dx, dy, dist, accel;
      body *l_bodies = (body *) g_bodies[MYPROC];
      body r_bodies[m];

      for(ip = 0; ip < PROCS; ip++) {
            for(i = 0; i < m; i++)
                  r_bodies[i] = g_bodies[ip][i];

            for(j = 0; j < m; j++) {
                  for(i = 0; i < m; i++) {
                        dx = r_bodies[i].x - l_bodies[j].x;
                        dy = r_bodies[i].y - l_bodies[j].y;
                        dist = max(sqrt(dx*dx + dy*dy),0.01);
                        accel = r_bodies[i].m/(dist*dist);
                        l_bodies[j].u += accel*(dx/dist);
                        l_bodies[j].v += accel*(dy/dist);
                  }
            }
      }
      barrier();
}
```

To operate on data structures spread over a logical processor grid, rather than a one dimensional array, define the arrays to be of the form

```
foo A[p][q]::[n][m];
```

where the product of `p` and `q` is `PROCS` and the block assigned to each processor is and `n` by `m` block of `foo`'s.

6 Bulk assignment

In operating on global objects it is frequently useful to be able to transfer the entire object. Split-C provides the same bulk assignment for structs as C, *i.e.*, the assignment of a struct copies all elements of the struct. This property is exploited in the N-body example shown earlier in the copy of a remote row of bodies into a local array:

```
typedef body struct {
    double x,y,u,v,c;
};
void all_nBody(int m, body gBodies[PROCS]::[m])
{
    ...
    body rBodies[m];

    for(ip=0; ip<PROCS; ip++) {
        for(i=0; i<m; i++)
            rBodies[i] = GBodies[ip][i];
        ...
    }
}
```

To support bulk copies of arbitrary memory regions, the Split-C library provides a family of bulk copy functions. Each of the basic memory operations has a corresponding bulk version. `bulk_read` and `bulk_write` are blocking, `bulk_get` and `bulk_put` are split-phase, and `bulk_store` is non-acknowledging. These functions all take a destination pointer, a source pointer and the size as arguments. The array copy above can be rewritten using `bulk_read` as follows:

```
    ...
    bulk_read(rBodies, GBodies[ip], sizeof(rBodies));
```

A copy initiated by the current owner can be written analogously using `bulk_write`. More extensive examples of the bulk operations are provided in the following sections, as we describe the use of basic memory operations and then optimizations on those examples.

The bulk operations on arrays would be more easily expressed using array assignment statement, rather than the library calls. Unfortunately, C does not provide notation for array assignment. (We are considering support for Fortran-90 style array triplet notation in Split-C, *e.g.*, `A[0:n:2] := B[2:n+2:2]`.)

7 Split-phase Assignment

Accessing a global object is usually a costly operation, because the object must be copied from a remote processor. Operationally, a request is sent to the processor owning the object and the contents of the object are sent back. Both directions involve transfers across the communication network with substantial latency. The processor is unused during much of the remote access. Thus, it is worthwhile to allow the processor to continue with computation until the global object is actually needed. This overlap of communication and computation is supported in Split-C by the split-phase assignment operator, "`:=`". The idea is that `:=` is the first-phase of the operation and starts the transfer. The second phase is to receive the data and update the local variable. To determine that the transfer is complete a `sync()` operation is performed. This synchronizes or joins the thread of control with the remote accesses.

7.1 Get and put

Syntactically, split-phase assignment are allowed to have an arbitrary mixture of global and local pointers. The simplest case is when one side of the assignment is a global pointer and the other is local. This correspond to two basic operations: *get* a global into a local and *put* a local into a global. In either case, the destination is undefined until after the `sync` and neither source nor destination should be modified until a `sync` is performed.

> *Implementation note:* The current implementation guarantees atomicity on a word basis, meaning that reading large object between a `get` and a `sync` will result in seeing either an old or new version of each word in the object. No guarantees are made on the order in which words are updated, and for portability reasons, users are discouraged from relying on the per word atomicity.

The following example initiates a get of `*Gx`, performs some other computation, initiates a get of `*Gy`, does some more computation, and then syncs the two outstanding gets before summing the local variables.

```
lx := *Gx;
...;
ly := *Gy;
...;
sync();
z = lx+ly;
```

The following performs two puts, perhaps with some computation in between. The put does not wait for the update to be performed. There is no ordering guaranteed between the updates to `Gx` and `Gy`. However, the updates before the sync (to `*Gx` and `*Gy`) are guaranteed to take place before the update after the sync (to `*Gz`).

```
*Gx := lx;
...;
*Gy := ly;
...;
sync();
*Gz := 2;
```

We can use gets, instead of global reads, to produce a more efficient version of our reduction example, as shown below. Here we issue gets on four different elements and then wait for all of them to come back. Issuing multiple outstanding requests in this fashion hides the latency of all but the last.

```
static int spread_buf[PROCS]::;

int all_reduce_to_one_add(int val)
{
    int i, res = 0;
    int v0, v1, v2, v3;

    spread_buf[MYPROC] = val;
    barrier();
    on_one {
        for(i = 0; i < PROCS%4; i++)
            res = res + spread_buf[i];
        for(i = i; i < PROCS; i += 4) {
            v0 := spread_buf[i];
            v1 := spread_buf[i+1];
            v2 := spread_buf[i+2];
            v3 := spread_buf[i+3];
            sync();
            res = res + v0 + v1 + v2 + v3;
        }
    }
    barrier();
    return res;
}
```

The following variation on the N-body example shown earlier copies the remote row of bodies into a local array using gets (only the copy loop is shown here):

```
void all_n_body(int m, body g_bodies[PROCS]::[m])
{
    int ip, i, j;
    double dx, dy, dist, accel;
    body *l_bodies = (body *) g_bodies[MYPROC];
    body r_bodies[m];

    for(ip = 0; ip < PROCS; ip++) {
        for(i = 0; i < m; i++)
            r_bodies[i] := g_bodies[ip][i];
        sync ();
        ...
    }
    barrier();
}
```

Notice that each get operation retrieves an entire body, which may be implemented as a single request for the block rather than as a collection of independent requests. The power of split-phase operations is enhanced by the ability to transfer large blocks. In fact, since the row of bodies is on one processor, it makes sense to get the whole row by replacing the loop with the following:

```
void all_n_body(int m, body g_bodies[PROCS]::[m])
{
    int ip, i, j;
    double dx, dy, dist, accel;
    body *l_bodies = (body *) g_bodies[MYPROC];
    body r_bodies[m];

    for(ip = 0; ip < PROCS; ip++) {
        bulk_get(&r_bodies[0], g_bodies[ip][0], m*sizeof(body));
        sync ();
        ...
    }
    barrier();
}
```

The blocked matrix multiply can be adapted to use gets as shown below. This code also staggers the `k` loop to avoid contention, as discussed in the previous section.

```
void all_mat_mult_blk(int n, int r, int m, int b,
                      double C[n][m]::[b][b],
                      double A[n][r]::[b][b],
                      double B[r][m]::[b][b])
{
  int i,j,k,l,kk;
  double la[b][b], lb[b][b];                    /* Local copies of the subblocks */

  for_my_2D(i,j,l,n,m) {
    double (*lc)[b] = tolocal(C[i][j]);
    kk = MYPROC%r;
    for (k=0;k<r;k++) {
      bulk_get (la, A[i][kk], b*b*sizeof(double));
      bulk_get (lb, B[kk][j], b*b*sizeof(double));
      sync();
      matrix_mult(b,b,b,lc,la,lb);
      kk++;
      if (kk==r) kk=0;
    }
  }
  barrier();
}
```

Another example of using get, an improvement on the previous lookup function does the string comparison while speculatively getting the next entry in the table.

```
char *global lookup(int size, hash_entry table[size]::, char *target)
{
    int h = hash(target, size);
    char *global name = table[h].name;
    char *global next_name;

    while(name) {
        next_name := table[h+1].name;
        if(g_strcmp(name, (char *global) target) == 0)
            return table[h].value;
        sync();
        name = next_name;
        h++;
    }
    return NULL;
}
```

To reiterate, `get` and `put` are split-phase versions of read and write. In a split assignment, if the global reference appears on the LHS, a `put` operation is performed; if the global references appear on the RHS, the corresponding number of `get` operations is performed. Syntactically, a split assignment may contain an arbitrary mixture of global and local pointers, but only the last reference will be performed as a split-phase operation. For example, assigning one global pointer to another global's value will require that the the get completes (as if it were a read) before the put is initiated. Like read and write, `get` and `put` involve two-way communication for the requester to determine when the operation is complete. However, by separating the completion detection from the issuing of the request, they allow the communication to be masked by useful work.

Reads and writes can be mixed with gets and puts, however, reads and writes do not wait for previous gets and puts to complete. A write operation waits for itself to complete, so if another operation (read, write, put, get, or store) follows, it is guaranteed that the previous write has been performed. The same is true for reads; any read waits for the value to be returned. In other words, only a single outstanding read or write is allowed from a given processor; this ensures that the completion order of reads and writes match their issue order. Only the synchronization operations, *e.g.*, `sync`, wait for operations other than themselves. Thus, if a single processor does a write followed by a get to the same location, the get will see the new value (assuming no other processors have modified it). On the other hand, a put followed by a read may see the old value.

7.2 Store

A weaker form of global operation is supported in Split-C by the `:-` assignment operator, called *store*, which simply stores a value into a global location. This is very useful in global data reorganization operations and also in situations akin to message passing where the computation is driven by data arrival. There is no acknowledgement of the store operation, so the corresponding overhead is eliminated, but there is no way for a processor to determine locally that the effects of its stores have taken place. However, the processor owning the location being stored is notified and there is a global notion of all the stores being done. The operation `all_store_sync()` synchronizes all processors and waits for all stores on all processors to complete. This completion detection for stores is independent from that of reads, writes, gets and puts.

There is no dual *retreive* operation for store, since any form of read requires a round trip, and there is no acknowledgement overhead to be saved. For this reason, only the LHS of a `:-` statement may include a global reference.

7.2.1 Global data movement

In the following example, A is a "blocked" two dimensional array with one row per processor. We want to copy it to a "cyclic" structure `B`, which is in row-major order. All processors scatter their row of A into B, then all are synchronized on the global completion.

```
void all_map_block_to_cyclic(int n, double B[PROCS*n]::, double A[PROCS]::[n])
{
    int i;
    int *a = &A[MYPROC][0];

    for(i = 0; i < n; i++)
        B[i+MYPROC*n] :- a[i];
    all_store_sync();
}
```

Notice that if n is a multiple of PROCS that all the processors start out by storing to processor 0. This contention might be avoided by staggering the loop as follows:

```
void all_map_block_to_cyclic(int n, double B[PROCS*n]::, double A[PROCS]::[n])
{
    int i, ii;
    int *a = &A[MYPROC][0];
    int i = MYPROC % n;

    for(ii = 0; ii < n; ii++) {
        B[i+MYPROC*n] :- a[i];
        if (++i == n) i = 0;
    }
    all_store_sync();
}
```

The reverse mapping is accomplished by the following:

```
void all_map_cyclic_to_blocked(int n, double B[PROCS]::[n], double A[PROCS*n]::)
{
    int i;
    for_my_1D(i,PROCS*n)
        B[i/n][i%n] :- A[i];
    all_store_sync();
}
```

The ability to change layouts for different phases of an algorithm is a powerful optimization technique. For example, an FFT on n points has no remote access for the first $\log \frac{n}{\texttt{PROCS}}$ stages under a cyclic mapping and no remote access for the last $\log \texttt{PROCS}$ stages under a blocked mapping. Thus, for $n \geq \texttt{PROCS}^2$ the only communication in the FFT is a cyclic to block remap and the bit-reversal map at the end. The following shows the the main structure for the FFT:

```
void all_fft(unsigned int n, complex X[n]::)
{
        unsigned int my_n = n/PROCS;
        complex (*spread T)[n/PROCS];
        complex TF[my_n/2];

        T = tospread(0, (complex *)all_spread_malloc(n, sizeof(complex)));

        /* X starts with cyclic mapping. Do log(n)-log(PROCS) local FFT steps */
        local_fft(my_n, my_n/2, (complex *)X, TF);

        /* Permute X from cyclic mapping to block mapping */
        all_remap_cyclic_to_blocked(n, X, (void *spread)T);

        /* T is now blocked mapping. Do log(PROCS) local FFT steps */
        local_fft(my_n, PROCS/2, (complex *)T, TF);
}
```

7.2.2 Data driven execution

Store also provides a natural means of representing data driven computation. Consider, for example, the problem of broadcasting a value to all the processors. One approach is to allow all the processors to simply read it. However, this is rather inefficient because the reads occur serially. (They are actually serviced by the processor owning the location.) An alternative approach is to fan-out the value by writing it to each processor. Again, a single processor could write the value to each of the processors, as follows:

```
int all_bcast(int val)
{
  int i;

  on_one {
    for (i=0; i<PROCS; i++ ) spread_buf[i] = val;
  }
  barrier();
  return(spread_buf[MYPROC]);
}
```

However, since completion is detected on each write (to ensure sequential consistency) the rate of the loop is determined by the full round-trip remote access time (about 17us on the CM-5). This is much improved using puts, since we wait for all the acknowledgements at the end. In the following the loop rate is less than 5us.

```
int all_bcast(int val)
{
   int i;

   on_one {
      for (i=0; i<PROCS; i++ ) spread_buf[i] := val;
      sync();
   }
   barrier();
   return(spread_buf[MYPROC]);
}
```

In both cases, we utilize a global barrier to inform all the processors that they have received the value. (Processor zero only enters the barrier when all the acknowledgements have returned.) However, using store the processors could detect when the stores have happened to them. This utilizes `store_sync` which waits until a certain number of bytes have been stored. (The store may have been initiated remotely or locally.) This approach eliminates the two-way communication. The barrier is needed, however, to reset the implicit store counter. These are adjusted automatically by `all_store_sync`.

```
int all_bcast(int val)
{
   int i;

   on_one {
      for (i=0; i<PROCS; i++ ) spread_buf[i] :- val;
   }
   store_sync(4);

   barrier();
   reset_store_sync();
   return(spread_buf[MYPROC]);
}
```

Of course, a better way to do broadcast is using a fan-out tree. This is shown in the following:

```
int all_bcast(int val)
{
   int left = 2*MYPROC+1;
   int right = 2*MYPROC+2;
   on_proc (0) {
      store_sync(4);
      val = spread_buf[MYPROC];
   }
   if (left < PROCS) spread_buf[left] :- val;
   if (right < PROCS) spread_buf[right] :- val;
   return(val);
}
```

Clearly, reduction also a tree-based operation. The determination of parent and number of children is a little more complicated, but the basic data driven style is the same and uses stores.

```
static int Tbuf[PROCS]::[2];

int all_reduce_to_one_addT(int val)
{
    int parent = (MYPROC - 1)/2;
    int parity = MYPROC & 1;
    int odd_child = 2*MYPROC + 1;
    int even_child = 2*MYPROC + 2;

    if (MYPROC == 0) {
        if(PROCS == 1)
            return val;
        else {
            if(PROCS == 2) {
                store_sync(4);
                return (val + Tbuf[0][1]);
            }
            else {
                store_sync(8);
                return (val + Tbuf[0][1] + Tbuf[0][0]);
            }
        }
    }
    else {
        if(odd_child ≥ PROCS)                                   /* no children */
            Tbuf[parent][parity] :- val;
        else {
            if(even_child ≥ PROCS) {
                store_sync(4);
                Tbuf[parent][parity] :- val + Tbuf[MYPROC][1];
            }
            else {
                store_sync(8);
                Tbuf[parent][parity] :- val + Tbuf[MYPROC][1]
                                            + Tbuf[MYPROC][0];
            }
        }
    }
    return val;
}
```

7.2.3 Message passing

Stores are also a useful means of representing message passing algorithms in a global address framework. As an example, we consider Cannon's matrix multiplication algorithm. This appears extensively in the systolic array literature also. It is pretty close to magic the first time you see it, but it is really pretty simple. If you think of the matrix as defining a logical grid, all communication occurs along rows and columns of the grid. Thus, this example will illustrate the basic machinery for simpler grid-based algorithms. Alternatively, you may find it easier to reason about the data structure itself.

First, the A matrix is skewed by shifting row i to the left by i places, assuming the zero origin. Matrix B is skewed in the other dimension, *i.e.*, column i of B is shifted up by i. Both of these shifts are done cyclicly. Now the amazing thing is that the multiply procedes by local multiplies and simple cyclic shifts of distance one. For each step, all of A shifts one to the left while all of B shifts up by one. The same algorithm works when the elements are matrices, so we get alternating global communication and local matrix operations. The following defines the two matrices to be $l \times l$ grids of conforming matrices:

```
#include <stdio.h>
#include <cm/cmmd.h>

#include <split-c/split-c.h>
#include <split-c/control.h>
#include <split-c/com.h>

inline void gswap(void *global *A, void *global *B)
{
    void *global T = *A;
    *A = *B;
    *B = T;
}

void all_canon_mm(int l, int n, int r, int m,
                  double (*spread A)[l]::[n][r],
                  double (*spread B)[l]::[r][m],
                  double C[l][l]::[n][m])
{
    double (*spread nA)[l]::[n][r];
    double (*spread nB)[l]::[r][m];

    nA = all_spread_malloc (l*l, n*r*sizeof(double));
    nB = all_spread_malloc (l*l, r*m*sizeof(double));
    {
      int numSteps;
      int x,y;
      int i,j,tmp;

      /* Initialize result matrix C */
      for_my_2D(i,j,tmp,l,l)
        zero_matrix(n,m,tolocal(C[i][j]));

      /* Continued on next page */
```

```
        /* Skew input matrixes */
        for_my_2D(i,j,tmp,l,l) {
            bulk_store(nA[i][(j-i+l)%l], tolocal(A[i][j]), n*r*sizeof(double));
            bulk_store(nB[(i-j+l)%l][j], tolocal(B[i][j]), r*m*sizeof(double));
            }
        gswap(&nA,&A);
        gswap(&nB,&B);
        all_store_sync();
        on_one printf("Initialized\n");

        /* Multiply and shift */
        for (numSteps=0; numSteps<l-1 ; numSteps++) {
            for_my_2D(i,j,tmp,l,l) {
                matrix_mult(n, r, m, tolocal(C[i][j]), tolocal(A[i][j]), tolocal(B[i][j]));
                bulk_store(nA[i][(j-1+l)%l], tolocal(A[i][j]), n*r*sizeof(double));
                bulk_store(nB[(i-1+l)%l][j], tolocal(B[i][j]), r*m*sizeof(double));
            }
            gswap(&nA,&A);
            gswap(&nB,&B);

            all_store_sync();

            on_one printf("Step\n");
        }

        /* Final Multiply */
        for_my_2D(i,j,tmp,l,l) {
            matrix_mult(n, r, m, tolocal(C[i][j]), tolocal(A[i][j]), tolocal(B[i][j]));
        }
        barrier();
    }
}
```

Notice that two copies of the A and B matrices are required. One is sent while the other is received. We flip pointers at each step.

All of the global memory operations implicitly maintain a set of counters for completion detection. The library interface makes these explicit so that abstraction can be constructed that do not introduce in the enclosing program.

8 Synchronization

Concurrent operation on linked data structures generally requires mutual exclusion on portions of the data structure. Control oriented exclusion approaches, such as critical sections, ensure that only one process is in the section of code that would update the data structure. This clearly does not scale well, so it is more attractive to enforce synchronization through atomic operations on the objects themselves. If done correctly, several processes may be able to modify portions of a shared data structure concurrently.

Split-C provides, as part of the library support, a set of primitives for building powerful synchronization models. This section describes some of the basic primitives.

8.1 Executing Code Atomically

In many cases, the programmer may want a simple operation performed on a global object to be atomic with respect to the operations performed simultaneously by other processors. For example, in the following code segment, processors 0 and 1 simultaneously update the member of the spread array `x` which resides on processor 4.

```
#include <split-c/split-c.h>

static int x[PROCS]::;

splitc_main()
{
    int *global i = (int *global) (x + 4);

    (*(int *) x) = 0;                                    /* Initialize x to 0 */
    barrier();
    if(MYPROC == 0)
        *i = *i + 2;
    else
        if(MYPROC == 1)
            *i = *i + 3;
}
```

The programmer would like the updates to `i` to occur atomically. That is, `*i` should contain 5 at the end of the code segment. However, it is possible for both processor 0 and processor 1 to read the value of `*i` as 0 and to then update `*i` such that `*i` is equal to either 2 or 3 at the end of the code.

A simple solution to this to define a procedure `add()` and to use it as an *atomic procedure*. *Atomic procedures* are executed atomically on the processor owning the global object (in this case, `*i`).

The following code shows how to use atomic operations to implement the previous example:

```
#include <split-c/split-c.h>
#include <split-c/atomic.h>

static int x[PROCS]::;

void add(int *a, int b)
{
    *a += b;
}

splitc_main()
{
    int *global i = (int *global) (x + 4);

    (*(int *) x) = 0;                                  /* Initialize x to 0 */
    barrier();
    if(MYPROC == 0)
        atomic(add, i, 2);
    else
        if(MYPROC == 1)
            atomic(add, i, 3);
}
```

Atomic procedures should be as short and efficient as possible. Performing system calls within atomic procedures is highly discouraged. Furthermore, atomic operations may not be nested and global pointers cannot be dereferenced within atomic procedures. Note that if a local pointer is dereferenced, then this occurs on the processor executing the atomic procedure. The atomic procedure used in the above example cannot return a value; its only purpose is for its side-effects. Atomic procedures can take up to two words of arguments in addition to the local pointer. These two words can be any combination of integers and doubles.

In order to execute a function atomically and return a value, `atomic_i` or `atomic_d` is used. If an atomic procedure returns a value, then the first argument in the formal parameter list must be the number of the calling processor, `int proc`. These procedures can also take two additional words at the end of the argument list.

The macros `atomic_return_i` and `atomic_return_d` are used to return an integer or double value back to the calling processor. These macros reference the processor number, `proc`, as declared in the formal parameter list.

All of the common locking operations, (test-and-set, fetch-and-op, compare-and-swap, etc.) can be built from these atomic operations. Many are supplied in the atomic library. The following code segment implements the `fetch_and_add()` atomic operation and demonstrates how it might be used:

```
#include <split-c/split-c.h>
#include <split-c/atomic.h>

static int x[PROCS]::;

void fetch_and_add_atomic(int proc, void *return_val, int *addr,
                                     int incr_val)
{
    int tmp = *addr;

    *addr += incr_val;
    atomic_return_i(tmp);
}


int fetch_and_add(int *global addr, int incr_val)
{
    return atomic_i(fetch_and_add_atomic, addr, incr_val);
}

splitc_main()
{
    int *global i = (int *global) (x + 4);
    int old_val;

    (*(int *) x) = 0;                                   /* Initialize x to 0 */
    barrier();
    if(MYPROC == 0) {
        old_val = fetch_and_add(i, 2);
        if (old_val == 0)
            /* ... */ ;
    }
    else
        if(MYPROC == 1) {
            old_val = fetch_and_add(i, 3);
            if (old_val == 0)
                /* ... */ ;
        }
}
```

If all atomic changes must be complete before a new section of the program may execute, the function `all_atomic_sync` is called to guarantee such completion. All processors must call the function–there is currently no means to guarantee completion locally (although one could call `atomic_i` or `atomic_d` in every case and thus be assured that all calls had completed). If there is no need for synchronization of the calls, one may call `atomic_no_sync`, which allows three words of arguments in addition to the global pointer.

To use more than two extra words of arguments, it is necessary to call the somewhat slower `atomic_long` procedure, which accepts up to four words of arguments. There is currently no support for more than four words or for more than two words with a return value.

9 Optimizing Split-C Programs

Several of the important optimization techniques for Split-C programs have been mentioned in previous parts of this document. These include the usual goals of balancing the computational load and minimizing the ratio of communication to computation. Blocked algorithms are the primary example of the latter. The general philosophy behind Split-C is that it should be possible to incorporate highly optimized local routines (written in C, Fortran, or anything else) into an effective global Split-C program. In addition to these guidelines, we have seen that where communication occurs, split-phase operations provide a means for obtaining high bandwidth and masking latency with useful computation. Stores eliminate two-way communication. Bulk transfers are generally more efficient than many small transfers.

One of the more subtle optimizations is elimination of global pointer arithmetic by conversion to local pointers. For example, the following is an optimized version of the tree summation routine:

```
static int Tbuf[PROCS]::[2];

int all_reduce_to_one_addT3(int val)
{
    int parent = (MYPROC - 1)/2;
    int odd_child = 2*MYPROC + 1;
    int even_child = 2*MYPROC + 2;
    int parity = MYPROC & 1;
    int *buf = tolocal(&Tbuf[MYPROC][0]);
    int *global dest;

    if(MYPROC == 0) {
        if(PROCS == 1)
            return val;
        else {
            if(PROCS == 2) {
                store_sync(4);
                return (val + buf[1]);
            }
            else {
                store_sync(8);
                return (val + buf[1] + buf[0]);
            }
        }
    }
    /* For all processors but 0 */
    dest = &Tbuf[parent][parity];
    if(odd_child ≥ PROCS)                                    /* no children */
        *dest :- val;
    else {
        if(even_child ≥ PROCS) {
            store_sync(4);
            *dest :- val + buf[1];
        }
        else {
            store_sync(8);
            *dest :- val + buf[1] + buf[0];
        }
    }
    return val;
}
```

10 Library extensions

This section describes the various libraries in `libsplit-c` that augment Split-C in the same way that `libc` augments C. Almost all of the language extensions are library routines. The syntactic extensions, including new pointer type qualifiers and new assignment operators, are translated internally to uses of an intrinsic library. This allows a very modular construction of the Split-C system and allows for a very simple interface to conventional C and other languages.

10.1 Special variables

The following pseudo constants are defined in `<split-c/split-c.h>`.

`PROCS` Number of processors. This is a constant if the `-PROCS=` compiler option is used. The default compilation assumes that `PROCS` is a power of two. This can be overridden by the option `-PROCS`, but the library must also be recompiled.

`MYPROC` Local processor number, from 0 to `PROCS` $-$ 1.

`PROCSLOG` log2(PROCS) or -1 if not a power of two.

`PROCSMAX` Constant defining the largest value allowed for `PROCS`.

10.2 Barriers

`barrier()` implements barrier synchronization of all processors.

```
void barrier(void);
```

10.3 Global pointers

Casts between local, global and spread pointers.

```
int   toproc (void *global gptr);
void *tolocal(void *global gptr);

void *global toglobal(int proc, void *lptr);
void *spread tospread(int proc, void *lptr);
```

10.4 Read/Write

The functions below provide blocking access to global objects where not available as `=` assignments involving global pointers.

```
void bulk_read (void *dst,        void *global src, int len);
void bulk_write(void *global dst, void        *src, int len);
```

10.5 Get/Put

The functions below provide split-phase access to global objects where not available as `:=` assigments involving global pointers. `sync` waits for all previous gets and puts to complete. This applies to `:=` assignments and explicit calls. `Is_sync` checks for completion and returns the status; it does not wait for completion.

```
void bulk_get(void        *dst, void *global get, int len);
void bulk_put(void *global dst, void        *src, int len);


void sync(void);
bool is_sync(void);
```

The standard get/put operations assume an implicit synchronization counter. In building a variety of abstractions, such as a read-ahead stream, it is important to isolate the request and completion detection from the program control flow. This can be done by explicitly specifying a variable to use as a counter with the following routines. Since there is no syntactic support for manipulating counters (such as, for example, a `reset_counter` routine), the primitive data types are supported in the library.[8]

```
void bulk_get_ctr(void        *dst, void *global src, int len, Counter *ctr);
void bulk_put_ctr(void *global dst, void        *src, int len, Counter *ctr);


void sync_ctr   (Counter *ctr);
bool is_sync_ctr(Counter *ctr);


void __c_get_ctr (char  *dst, char  *global src, Counter *ctr);
void __sh_get_ctr(short *dst, short *global src, Counter *ctr);
void __i_get_ctr (int *dst, int   *global src, Counter *ctr);
void __f_get_ctr (float *dst, float *global src, Counter *ctr);
void __d_get_ctr (double *dst, double *global src, Counter *ctr);
void __ll_get_ctr (long long *dst, long long *global src, Counter *ctr);


void __c_put_ctr (char   *global dst, char  val, Counter *ctr);
void __sh_put_ctr(short  *global dst, short val, Counter *ctr);
```

[8] It has been proposed that a counter could be associated with a block of statements in a manner similar to a "with" block in Pascal.

```
void __i_put_ctr (int    *global dst, int   val, Counter *ctr);
void __f_put_ctr (float  *global dst, float val, Counter *ctr);
void __d_put_ctr (double *global dst, double val, Counter *ctr);
void __ll_put_ctr (long long *global dst, long long val, Counter *ctr);
```

10.6 Store

The functions below provide non-acknowledged stores to global objects. They extend `:-` assigments involving global pointers on the left-hand side. `all_store_sync` is executed on all processors and defines a store epoch. It waits for all stores issued in the epoch to be received, *i.e.*, the sum of the store initiations on all processors equals the sum of the store receives. `store_sync` is executed on a single processor and waits until the specified number of bytes have been store into the local address space of that processor. `is_store_sync` is similar, but tests rather than waits.

```
void bulk_store(void *global dst, void *src, int len);


void all_store_sync(void);
void store_sync(int bytes_received);
bool is_store_sync(int bytes_received);


void bulk_store_ctr(void *global gptr, char *lptr, int len, Counter *rbytes, Counter *sbytes);


void all_store_sync_ctr(Counter *rbytes, Counter *sbytes)
void store_sync_ctr   (Counter expecting, Counter *rbytes, Counter *sbytes);
bool is_store_sync_ctr(Counter expecting, Counter *rbytes, Counter *sbytes);


void __c_store_ctr (void *global dst, char  val,
Counter *rbytes, Counter *sbytes);
void __sh_store_ctr(void *global dst, short val,
Counter *rbytes, Counter *sbytes);
void __i_store_ctr (void *global dst, int   val,
Counter *rbytes, Counter *sbytes);
void __f_store_ctr (void *global dst, float val,
Counter *rbytes, Counter *sbytes);
void __d_store_ctr (void *global dst, double    val,
Counter *rbytes, Counter *sbytes);
void __ll_store_ctr(void *global dst, long long val,
Counter *rbytes, Counter *sbytes);
```

10.7 Storage management

The following functions provide allocation and deallocation of storage on the spread heap.

```
void *spread all_spread_malloc(int count, int object_size);
void all_spread_free(void *spread sptr);
```

10.8 Global communication

Global communication operations are included in `<split-c/com.h>`. They include collective reduction over all processors to one processor or all processors, scan over all processors, and broadcast from a processor to all processors. Note that each routine begins with `all`, indicating that each processor must call the routine.

Collective reduction routines use the prefix `all_reduce_to_` followed by the option `one_` to reduce to either a single processor (processor 0) or `all_` to reduce to all processors. Next is the operand type: '`u`' specifies unsigned int, '`d`' specifies double-precision floating-point, and no addition specifies an int. Finally, the suffix determines the operation to be performed: `add`, `mult`, `max`, `min`, `or`, `xor`, or `and`. The logical operators are of course incompatible with floating-point values.

For example, to reduce a double-precision floating-point number by summation from all processors to processor 0, one must call

```
double all_reduce_to_one_dadd(double value)
```

on each processor and use the result only on processor 0. For those unfamiliar with the terminology, this operation would add together the set of `PROCS` numbers (one from each processor) and return the result to processor 0.

Scan is the same as reduce, except that the prefix `all_scan_` is used instead. With scans, each processor receives the result of applying the given operation to all values from processors with `MYPROC` equal to or less than its own (hence the scans are inclusive). To scan using exclusive or on integers, one would call

```
int all_scan_xor(int value)
```

Broadcast routines begin with `all_bcast_` and allow only the selection of operand type. In the current implementation, broadcast is only supported from processor 0. Four types exist: '`i`' for an integer, '`u`' for an unsigned integer, '`d`' for a double-precision floating-point number, and '`p`' for a pointer.

10.9 I/O

Split-C uses I/O functions that are provided by the system. For CMMD, CMMD I/O routines can be used. For the Paragon, the OSF/1 routines can be used. For CMAM, `printf` and `scanf` are supported.

10.10 Timing

`get_seconds()` returns the value of a timer that is measured in seconds. To time an interval in your program, call `get_seconds()` before and after the interval. Then take the difference between the return values.

```
double get_seconds();
```

10.11 Strings

The C operations on strings have been extended to handle global strings. These are included as <split-c/string.h>. Before calling a global string routine which writes (or puts) to another processor, the destination processor must call `gstr_init()`. This requirement will be removed in a future release.

10.11.1 String copy

General string copy from global to global string. Calls either `gstr_read` or `gstr_write`, or both if both strings are remote.

```
g_strcpy (char *global dst, char *global src);
g_strncpy(char *global dst, char *global src);
```

Blocking string copy.

```
gstr_read (char *dst, char *global src);
gstr_write(char *global dst, char *src);
gstrn_read(char *dst, char *global src, int maxlen);
gstrn_write(char *global dst, char *src, int maxlen);
```

Split-phase string copy. Use `sync` to detect completion.

```
gstr_get(char *dst, char *global src);
gstr_put(char *global dst, char *src);
gstrn_get(char *dst, char *global src, int maxlen);
gstrn_put(char *global dst, char *src, int maxlen);
```

Split-phase string copy with explicit counter.

```
gstr_get_ctr(char *dst, char *global src, Counter *flag);
gstr_put_ctr(char *global dst, char *src, Counter *flag);
gstrn_get_ctr(char *dst, char *global src, int maxlen, Counter *flag);
```

```
gstrn_put_ctr(char *global dst, char *src, int maxlen, Counter *flag);
```

10.11.2 String concatenation

General string concatenation of global onto global string. Calls either `gstrcat_read` or `gstrcat_write`, or both if both strings are remote.

```
g_strcat (char *global dst, char *global src);
g_strncat(char *global dst, char *global src);
```

Blocking string concatenation.

```
gstrcat_read (char *dst, char *global src);
gstrcat_write(char *global dst, char *src);
gstrncat_read(char *dst, char *global src, int maxlen);
gstrncat_write(char *global dst, char *src, int maxlen);
```

Split-phase string concatenation. Use `sync` to detect completion.

```
gstrcat_get(char *dst, char *global src);
gstrcat_put(char *global dst, char *src);
gstrncat_get(char *dst, char *global src, int maxlen);
gstrncat_put(char *global dst, char *src, int maxlen);
```

Split-phase string concatenation with explicit counter.

```
gstrcat_get_ctr(char *dst, char *global src, Counter *flag);
gstrcat_put_ctr(char *global dst, char *src, Counter *flag);
gstrncat_get_ctr(char *dst, char *global src, int maxlen, Counter *flag);
gstrncat_put_ctr(char *global dst, char *src, int maxlen, Counter *flag);
```

10.11.3 Miscellaneous

All following functions are blocking.

```
g_strlen(char *global s);
```

```
g_index (char *global s, char c);
g_rindex(char *global s, char c);
```

```
g_strchr (char *global s, char c);
g_strrchr(char *global s, char c);
```

```
g_strcmp (char *global s1, char *global s2);
g_strncmp(char *global s1, char *global s2, int maxlen);

g_strcasecmp (char *global s1, char *global s2);

g_strdup(char *global s);

g_strpbrk(char *global s1, char *global s2);
g_strspn(char *global s1, char *global s2);
g_strcspn(char *global s1, char *global s2);
g_strstr(char *global s1, char *global s2);
```

10.12 Atomic operations

A set of basic data synchronization primitives are provided in `<split-c/atomic.h>`.

```
int exchange(int *global loc, int val);
int test_and_set(int *global loc);
int cmp_and_swap(int *global loc, int test_val, int new_val);
int fetch_and_add(int *global loc, int val);
```

These primitives are implemented on top of a set of atomic routines. `atomic()` can take up to two additional words as arguments. They can be any combination of integers and doubles. `atomic_i()` and `atomic_d()` return an integer or double value, and can also take two additional words as arguements. `atomic_return_i()` and `atomic_return_d()` are used within atomic functions to return the integer or double value.

```
void atomic(void (*fun)(), void *global gptr, ...);
int atomic_i(void (*fun)(), void  *global gptr, ...);
double atomic_d(void (*fun)(), void *global gptr, ...);

void atomic_return_i(int val);
void atomic_return_d(int val);
```

10.13 Split-cc intrinsics

All of the global operations generated by the split-cc compiler are constructed by introducing calls to inlinable routines. This includes read/write, get/put, and store for all the basic data types. The complete set of routines used internally by split-cc are included here. Some of these are provided as user routines also. Others are used internally, as indicated by the `__`.

We anticipate that developers of other languages may want to build on the same communication substrate as Split-C. Furthermore, debugging or instrumentation can be incorporated into Split-C programs with changing the actual compiler by revising this library. Finally, interfacing with conventional C is trivial where the C program operates only on local data. It can operate on global data as well, using the split-cc intrinsics.

The internal representation of global pointers is the following.

```
typedef long long ___globPtrType;

typedef struct {
Processor pid;
void * addr;
} ___splitPtrAsStruct;

typedef union {
___splitPtrAsStruct asStruct;
void *global asLong;
} ___splitPtrAsUnion;


int   toproc (void *global gptr);
void *tolocal(void *global gptr);


void *global toglobal(int proc, void *lptr);
void *spread  tospread (int proc, void *lptr);


void bulk_read (void *dst,        void *global src, int len);
void bulk_write(void *global dst, void        *src, int len);


char      __c_read  (char      *global src);
short     __sh_read (short     *global src);
int       __i_read  (int       *global src);
float     __f_read  (float     *global src);
double    __d_read  (double    *global src);
long long __ll_read (long long *global src);


char      __c_write (char      *global dst, char      val);
short     __sh_write(short     *global dst, short     val);
int       __i_write (int       *global dst, int       val);
float     __f_write (float     *global dst, float     val);
double    __d_write (double    *global dst, double    val);
long long __ll_write(long long *global dst, long long val);
```

```
void bulk_get(void        *dst, void *global get, int len);
void bulk_put(void *global dst, void        *src, int len);

void sync(void);
bool is_sync(void);

void __c_get (char  *dst, char      *global src);
void __sh_get(short *dst, short     *global src);
void __i_get (int  *dst, int       *global src);
void __f_get (float *dst, float     *global src);
void __d_get (double *dst, double    *global src);
void __ll_get(long long *dst, long long *global src);

void __c_put (char      *global dst, char      val);
void __sh_put(short     *global dst, short     val);
void __i_put (int       *global dst, int       val);
void __f_put (float     *global dst, float     val);
void __d_put (double    *global dst, double    val);
void __ll_put(long long *global dst, long long val);

void bulk_store(void *global dst, char *src, int len);

void all_store_sync(void);
void store_sync(int received);
bool is_store_sync(int received);

void __c_store (void *global dst, char      val);
void __sh_store(void *global dst, short     val);
void __i_store (void *global dst, int       val);
void __f_store (void *global dst, float     val);
void __d_store (void *global dst, double    val);
void __ll_store(void *global dst, long long val);
```

11 Appendix: Open Issues and Inadequacies

In the distributed memory view, local external variables define a spread array of size `PROCS`. We have chosen to suppress this observation and encourage the use of the equivalent spread-array declaration. One reason for this is the compatibility with a traditional shared memory view, which would be either that there are `PROCS` private variables or one shared variable.

As mentioned in the discussion of spread arrays, it would be convenient to have bulk array assignment operations. This should probably be done using the tuple syntax of Fortran 90.

In some algorithms, it would be useful to have distinct sets of `put/get/sync` operations that are synchronized within the set, but not across sets. This is currently possible through the use of explicit `put/get` counters in the code. An alternate solution, based on encapsulating code sections with a default `get/put` counter, is being considered.

11.1 Restrictions on global operations

While Split-C allows all legal pointer operations to be performed on global or distributed pointers, the current version prohibits the combination of an assignment operators and a dereferenced global or distributed pointer. It also prohibits `++` and `--` as operators to a dereferenced global/distributed pointer.

For instance,

```
int *global gptr;
int darray[128]::[2];
int i;

gptr = darray[15];
*(gptr++) += 5; /* ILLEGAL */
darray[3][1] *= 7; /* ILLEGAL */
(*gptr)++; /* ILLEGAL */
```

Note that it is the `+=` operation on the dereferenced gptr, `*gptr`, not the `++` on the pointer itself, that makes the first statement illegal. Instead these actions can be accomplished with the following:

```
i = *gptr;
*(gptr++) = i + 5;

i = darray[3][1];
darray[3][1] = i*7;

i = *gptr;
*gptr = i + 1;
```

References

[1] Brian W. Kerninghan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.