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.