Parallel Programming in Split-C

David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy,
Steven Lumetta, Thorsten von Eicken, and Katherine Yelick
Computer Science Division
University of California, Berkeley *

Abstract

We introduce the Split-C language, a parallel extension of C intended for high performance programming on distributed memory multiprocessors, and demonstrate the use of the language in optimizing parallel programs. Split-C provides a global address space with a clear concept of locality and unusual assignment operators. These are used as tools to reduce the frequency and cost of remote access. The language allows a mixture of shared memory, message passing, and data parallel programming styles while providing efficient access to the underlying machine. We demonstrate the basic language concepts using regular and irregular parallel programs and give performance results for various stages of program optimization.

1 Overview

Split-C is a parallel extension of the C programming language that supports efficient access to a global address space on current distributed memory multiprocessors. It retains the "small language" character of C and supports careful engineering and optimization of programs by providing a simple, predictable cost model. This is in stark contrast to languages that rely on extensive program transformation at compile time to obtain performance on parallel machines. Split-C programs do what the programmer specifies; the compiler takes care of addressing and communication, as well as code generation. Thus, the ability to exploit parallelism or locality is not limited by the compiler's recognition capability, nor is there need to second guess the compiler transformations while optimizing the program. The language provides a small set of global access primitives and simple parallel storage layout declarations. These seem to capture most of the useful elements of shared memory, message passing, and data parallel programming in a common, familiar context. Split-C is currently implemented on the Thinking Machines Corp. CM-5, building from GCC and Active Messages[17] and implementations are underway for architectures with more aggressive support for global access. It has been used extensively as a teaching tool in parallel computing courses and hosts a wide variety of applications. Split-C may also be viewed as a compilation target for higher level parallel languages.

This paper describes the central concepts in Split-C and illustrates how these are used in the process of optimizing parallel programs. We begin with a brief overview of the language as a whole and examine each concept individually in the following sections. The presentation interweaves the example use, the optimization techniques, and the language definition concept by concept.

1.1 Split-C in a nutshell

Control Model: Split-C follows an SPMD (single program, multiple data) model, where each of `PROCS` processors begin execution at the same point in a common code image. The processors may each follow distinct flow of control and join together at rendezvous points, such as `barrier()`. Processors are distinguished by the value of the special constant, `MYPROC`.

Global Address Space: Any processor may access any location in a global address space, but each processor owns a specific region of the global address space. The local region contains the processor's stack for automatic variables, static or external variables, and a portion of the heap. There is also a spread heap allocated uniformly across processors.

Global pointers: Two kinds of pointers are provided, reflecting the cost difference between local and global accesses. *Global pointers* reference the entire address space, while standard pointers reference only the portion owned by the accessing processor.

Split-phase Assignment: A split-phase assignment operator (`:=`) allows computation and communication to be overlapped. The request to *get* a value from a location (or to *put* a value into a location) is separated from the completion of the operation.

*Send e-mail to: `Split-C@boing.CS.Berkeley.EDU`

Signaling Store: A more unusual assignment operator (`:-`) signals the processor that owns the updated location that the store has occurred. This provides an essential element of message driven and data parallel execution that shared memory models generally ignore.

Bulk Transfer: Any of the assignment operators can be used to transfer entire records, *i.e.*, `structs`. Library routines are provided to transfer entire arrays. In many cases, overlapping computation and communication becomes more attractive with larger transfer units.

Spread Arrays: Parallel computation on arrays is supported through a simple extension of array declarations. The approach is quite different from that of HPF and its precursors because there is no separate layout declaration. Furthermore, the duality in C between arrays and pointers is carried forward to spread arrays through a second form of global pointer, called a *spread pointer*.

1.2 Organization

Section 2 describes a non-trivial application, called EM3D, that operates on an irregular, linked data structure. Section 3 gives a simple parallel solution to EM3D, begins a sequence of optimizations on the program, showing how unnecessary remote accesses can be eliminated. Sections 4 and 5 show how to make the remaining remote accesses more efficient using split-phase assignments and signaling stores. Section 6 discusses bulk transfers and applies them to EM3D. Section 7 illustrates the use of spread arrays and how various array layouts are achieved. Section 8 explores how disparate programming models can be unified in the Split-C context, and Section 9 summarizes our findings.

2 An Example Irregular Application

To illustrate the novel aspects of Split-C for parallel programs, we use a small, but rather tricky example application, EM3D, that models the propagation of electromagnetic waves through objects in three dimensions [13]. A preprocessing step casts this into a simple computation on an irregular bipartite graph containing nodes representing electric and magnetic field values.

In EM3D, an object is divided into a grid of convex polyhedral cells (typically nonorthogonal hexahedra). From this primary grid, a dual grid is defined by using the barycenters of the primary grid cells as the vertices of the dual grid. Figure 1 shows a single primary grid cell (the lighter cell) and one of its overlapping dual

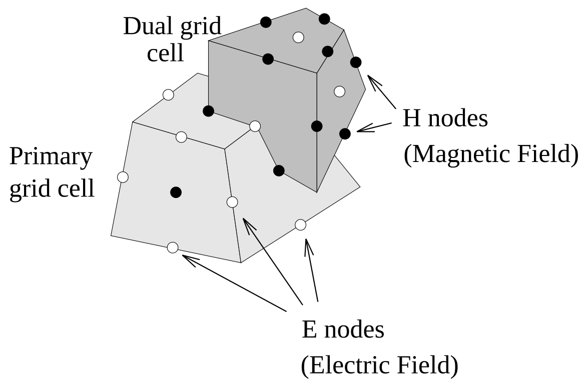

Figure 1: *EM3D grid cells.*

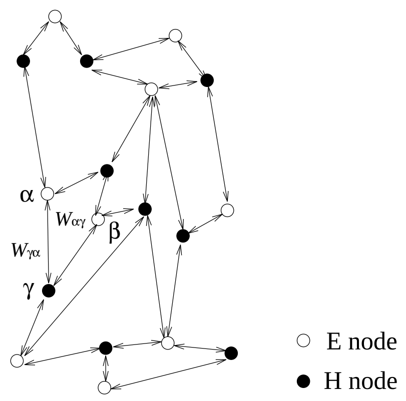

Figure 2: *Bipartite graph data structure in EM3D.*

grid cells. The electric field is projected onto each edge in the primary grid; this value is represented in Figure 1 by a white dot, an *E node*, at the center of the edge. Similarly, the magnetic field is projected onto each edge in the dual grid, represented by a black dot, an *H node*, in the figure.

The computation consists of a series of "leapfrog" integration steps: on alternate half time steps, changes in the electric field are calculated as a linear function of the neighboring magnetic field values and *vice versa*. Specifically, the value of each E node is updated by a weighted sum of neighboring H nodes, and then H nodes are similarly updated using the E nodes. Thus, the dependencies between E and H nodes form a bipartite graph. A simple example graph is shown in Figure 2; a more realistic problem would involve a non-planar graph of roughly a million nodes with degree between ten and thirty. Edge labels (weights) represent the coefficients of the linear functions, for example, $W_{\gamma\alpha}$ is the weight used for computing α's contribution to γ's value. Because the grids are static, these weights are constant values, which are calculated in a preprocessing step [13].

```c
typedef struct node_t {
  double        value;      /* Field value */
  int           edge_count;
  double        *coeffs;    /* Edge weights */
  double        *(*values); /* Dependency list */
  struct node_t *next;
} graph_node;

void compute_E()
{
  graph_node *n;
  int i;

  for (n = e_nodes; n != NULL; n = n->next)
    for (i = 0; i < n->edge_count; i++)
      n->value = n->value
             - *(n->values[i]) * (n->coeffs[i]);
}
```

Program 1: *Sequential EM3D, showing the graph node structure and E node computation.*

A sequential C implementation for the kernel of the algorithm is shown in Program 1. Each E node consists of a structure containing the value at the grid point, a pointer to an array of weights (`coeffs`), and an array of pointers to neighboring H node values. In addition, the E nodes are linked together by the `next` field, creating the complete list `e_nodes`. E nodes are updated by iterating over `e_nodes` and, for each node, gathering the values of the adjacent H nodes and subtracting off the weighted sum. The H node representation and computation are analogous.

Before discussing the parallel Split-C implementation of EM3D, consider how one might optimize it for a sequential machine. On a vector processor, one would focus on the gather, vector multiply, and vector sum. On a high-end workstation one can optimize the loop, but since the graph is very large, the real gain would come from minimizing cache misses that occur on accessing `n->values[i]`. This is done by rearranging the `e_nodes` list into chunks, where the nodes in a chunk share many H nodes. This idea of rearranging a data structure to improve the access pattern is also central to an efficient parallel implementation.

3 Global Pointers

Split-C provides a global address space and allows objects anywhere in that space to be referenced through global pointers. An object referenced by a global pointer is entirely owned by a single processor.[1] A global pointer can be dereferenced in the same manner as a standard C pointer, although the time to dereference a global pointer is considerably greater than that for a local pointer. In this section, we illustrate the use of the Split-C global address space on EM3D and explain the language extension in detail.

[1] The term *object* corresponds basic C objects, rather than objects in the sense of object-oriented languages.

```c
 1 typedef struct node_t {
 2   double        value;
 3   int           edge_count;
 4   double        *coeffs;
 5   double        * global (*values);
 6   struct node_t *next;
 7 } graph_node;
 8
 9 void all_compute_E()
10 {
11   graph_node *n;
12   int i;
13
14   for (n = e_nodes; n != NULL; n = n->next)
15     for (i = 0; i < n->edge_count; i++)
16       n->value = n->value
17              - *(n->values[i]) * (n->coeffs[i]);
18
19    barrier();
20 }
```

Program 2: *EM3D written using global pointers. Each processor executes this code on the E nodes it owns. The only differences between this Split-C kernel and the sequential C kernel are: insertion of the type qualifier* `global` *to the list of* `value` *pointers and addition of the* `barrier()` *at the end of the loop.*

3.1 EM3D using global pointers

The first step in parallelizing EM3D is to recognize that the large kernel graph must be spread over the machine. Thus, the structure describing a node is modified so that `values` refers to an array of global pointers. This is done by adding the type qualifier `global` in line 5 of Program 2. The new global graph data structure is illustrated in Figure 3. In the computational step, each of the processors performs the update for a portion of the `e_nodes` list. The simplest approach is to have each processor update the nodes that it owns, *i.e.*, *owner computes*. This algorithmic choice is reflected in the declaration of the data structure by retaining the `next` field as a standard pointer (see line 6). Each processor has the root of a list of nodes in the global graph that are local to it. All processors enter the electric field computation, update the values of their local E nodes in parallel, and synchronize at the end of the half step before computing the values of the H nodes. The only change to the kernel is the addition of `barrier()` in line 19 of Program 2.

Having established a parallel version of the program, how might we optimize its performance on a multiprocessor? Split-C defines a straight-forward cost model: accesses that are remote to the request-

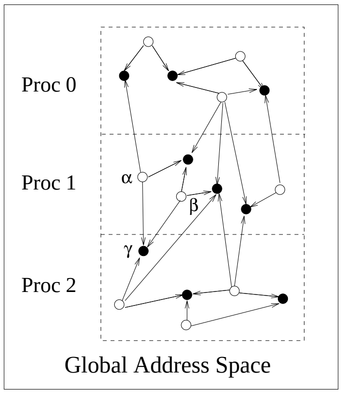

Figure 3: *An EM3D graph in the global address space with three processors. With this partitioning, processor 1 owns nodes α and β and processor 2 owns node γ. The edges are directed for the electric field computation phase.*

ing processor are more expensive than accesses that are owned by that processor. Therefore, we want to reorganize the global kernel graph into chunks, so that as few edges as possible cross processor regions[2]. Additionally, each processor should be responsible for roughly the same amount of work. For a given machine, we could estimate the cost of the kernel loop on a processor for a given layout as $L + XR$, where L is the number of edges to local nodes, R is the number of edges that cross to other processors, and X is the relative cost of a remote access. On the CM-5, the local accesses and floating point multiply-add cost roughly $3\mu s$ and a remote access costs roughly $14\mu s$. There are numerous techniques for partitioning graphs to obtain an even balance of nodes and a minimum number of remote edges, *e.g.*, [11, 14]. Thus, for an optimized program there would be a separate initialization step to reorganize the global graph using a cost model of the computational kernel. Load balancing techniques are beyond the scope of this paper, but the global access capabilities of Split-C would be useful in expressing such algorithms.

3.2 Language definition: Global Pointers

Global pointers provide access to the global address space from any processor.

Declaration: A global pointer is declared by appending the qualifier `global` to the pointer type declaration (*e.g.*, `int *global g;` or `int *global garray[10];`). The type qualifier `global` can be used with any pointer type (except a pointer to a function), and global pointers can be declared anywhere that standard pointers can be declared.

Construction: A global pointer may be constructed using the function `toglobal`, which takes a processor number and a local pointer. It may also be constructed by casting a local pointer to a global pointer. In this case, the global pointer points to the same object as the local pointer on the processor performing the cast.

Deconstruction: Semantically, a global pointer has a component for each of the two dimensions in the global address space: a processor number and a local pointer on that processor. These values can be extracted using the `toproc` and `tolocal` functions. Casting a global pointer to a local pointer has the same effect as `tolocal`: it extracts the local pointer part and discards the processor number.

Dereference: Global pointers may be dereferenced in the same manner as normal pointers, although the cost is higher.

Arithmetic: Arithmetic on global pointers reflects the view that an object is owned entirely by one processor: arithmetic is performed on the local pointer part while the processor number remains unchanged. Thus incrementing a global pointer will refer to the next object on the same processor[3].

Cost model: The representation of global pointers is typically larger than that of a local pointer. Arithmetic on global pointers may be slightly more expensive than arithmetic on local pointers. Dereferencing a global pointer is significantly more expensive than dereferencing a local pointer. A local/remote check is involved, and if the object is remote, a dereference incurs the additional cost of communication.

The current Split-C implementation represents global pointers by a processor number and local address. Other representations are possible on machines with hardware support for a global address space. This may change the magnitude of various costs, but not the relative cost model.

3.3 Performance study

The performance of our EM3D implementation could be characterized against a benchmark mesh with a specific load balancing algorithm. However, it is more illuminating to work with synthetic versions of

[2] Some of the optimizations we use here to demonstrate Split-C features are built into parallel Grid libraries like the Parti system [1].

[3] There is another useful view of the "next" object: the corresponding object on the "next" processor. This concept is captured by spread pointers, discussed in Section 7.

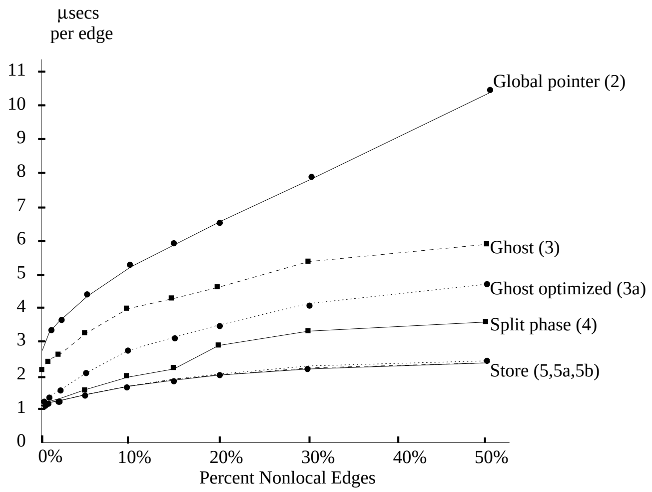

Figure 4: *Performance obtained on several versions of EM3D using a synthetic kernel graph with 320,000 nodes of degree 20 on 64 processors. The corresponding program number is in parentheses next to each curve. The y axis shows the average number of microseconds per edge. Because there are two floating point operations per edge and 64 processors, 1 µsec per edge corresponds to 128 Mflops.*

the graph, so that the fraction of remote edges is easily controlled. Our synthetic graph has 5,000 E and H nodes on each processor, each of which is connected to twenty of the other kind of nodes at random. We vary the fraction of edges that connect to nodes on other processors to reflect a range of possible meshes.

Figure 4 gives the performance results for a number of implementations of EM3D on a 64 processor CM-5 without vector units. The x axis is the percentage of remote edges. The y axis shows the average time spent processing a single graph edge, i.e., per execution of lines 16–17 in Program 2. Each curve is labeled with the feature and program number used to produce it. The top curve shows the performance of Program 2, the first parallel version using global pointers. The other curves reflect optimizations discussed below. For Program 2, 2.7 μs are required per graph edge when all of the edges are local. As the number of remote edges is increased, performance degrades linearly, as expected.

3.4 Eliminating redundant global accesses

Reorganizing the EM3D graph in the global address space does not necessarily minimize the the number of remote accesses, because some remote accesses may be redundant. For example, in Figure 3, two nodes, α and β on processor 1 reference a common node γ

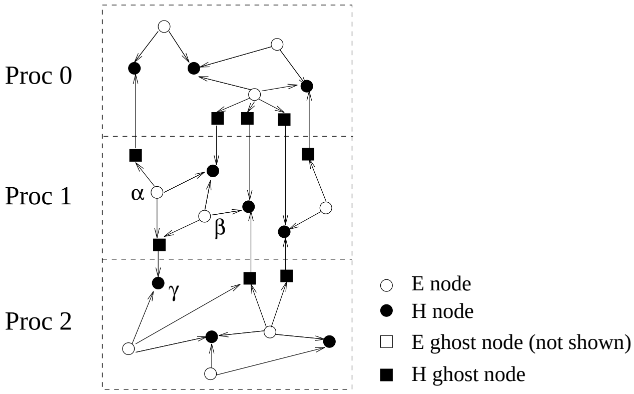

Figure 5: *EM3D graph modified to include ghost nodes. Local storage sites are introduced in order to eliminate redundant remote accesses.*

on processor 2. Eliminating the redundant references requires a more substantial change to the global graph data structure. For each remote node accessed by a processor, a local "ghost node" is created with room to hold a value and a global pointer to the remote node. Figure 5 shows the graph obtained by introducing the ghost nodes. The ghost nodes act as temporary storage sites, or caches, for values of dependent nodes that are remote in the global address space.

The resulting EM3D program is shown in Program 3. A new structure is defined for the ghost nodes and a new loop (lines 21–22) is added to read all the remote values into the local ghost nodes. Notice that the node struct has returned to precisely what it was in the sequential version. This means that the update loop (lines 24–27) is the same as the sequential version, accessing only local pointers. The Program 3 curve in Figure 4 shows the performance improvement.

In practice, the performance of parallel programs is often limited by that of its sequential kernels. For example, a factor of two in EM3D can be obtained by carefully coding the inner loop using software pipelining. The performance curve for this optimized version is called Program 3a in Figure 4. The ability to maintain the investment in carefully sequential engineered software is an important issue often overlooked in parallel languages and novel parallel architectures.

The shape of both Program 3 curves in Figure 4 is very different from our initial version. The execution time per edge increases only slightly beyond the point where 30% of the edges refer to remote nodes. The reason is that as the fraction of remote edges in the synthetic kernel graph increases, the probability that there will be multiple references to a remote node increases as well. Thus, the number of remote nodes referenced remains roughly constant, beyond some threshold. In other words, with an increasing number of remote edges, there are approximately the same number of ghost nodes; more nodes will depend upon these ghost nodes instead of depending upon other local nodes. The graphs obtained from real meshes exhibit a similar phenomenon.

4 Split-Phase Access

Once the redundant remote accesses have been eliminated, we want to perform the remaining remote accesses as efficiently as possible. The global read operations in line 22 of Program 3 are unnecessarily inefficient. Operationally, a request is sent to the processor owning the object and the contents of the object are returned. Both directions involve transfers across the communication network with substantial latency. The processor is simply waiting during much of the remote access. We do not need to wait for each individual access, we simply need to ensure that they have all completed before we enter the update loop. Thus, it makes sense to issue the requests one right after the other and only wait at the end. In essence, the remote requests are pipelined through the communication network.

Split-C supports overlapping communication and computation using split-phase assignments. The processor can initiate global memory operations by using a new assignment operator `:=`, do some computation, and then wait for the outstanding operations to complete using a `sync()` operation. The initiation is separated from the completion detection and therefore the accesses are called split-phase.

4.1 Split-phase access in EM3D

We can use split-phase accesses instead of blocking reads to improve the performance of EM3D, where we fill the values in the ghost nodes. We replace `=` by `:=` in line 8 of Program 4 and use the `sync` operation to ensure the completion of all the global accesses before starting the compute phase. By pipelining global accesses, we hide the latency of all but the last global access and obtain better performance, as indicated by the Program 4 curve in Figure 4.

4.2 Language definition: Split-Phase Access

GET: The get operation is specified by a split-phase assignment of the form: `l := g` where `l` is a local l-value and `g` is a dereference of a global l-value. The right hand side may contain an arbitrary global pointer expression (including spread array references

```
 1 typedef struct node_t {
 2   double        value;
 3   int           edge_count;
 4   double        *coeffs;
 5   double        *(*values);
 6   struct node_t *next;
 7 } graph_node;
 8
 9 typedef struct ghost_node_t {
10   double              value;
11   double *global      actual_node;
12   struct ghost_node_t *next;
13 } ghost_node;
14
15 void all_compute_E()
16 {
17   graph_node *n;
18   ghost_node *g;
19   int i;
20
21   for (g = h_ghost_nodes; g != NULL; g = g->next)
22     g->value = *(g->actual_node);
23
24   for (n = e_nodes; n != NULL; n = n->next)
25     for (i = 0; i < n->edge_count; i++)
26       n->value = n->value
27              - *(n->values[i]) * (n->coeffs[i]);
28
29   barrier();
30 }
```

Program 3: *EM3D code with ghost nodes. Remote values are read once into local storage. The main computation loop manipulates only local pointers.*

discussed below), but the final operation is to dereference the global pointer. Get initiates a transfer from the global address into the local address, but does not wait for its completion.

PUT: The put operation is specified by a split-phase assignment of the form: `g := e` where `g` is a global l-value and `e` is an arbitrary expression. The value of the right hand side is computed (this may involve global accesses) producing a local r-value. Put initiates a transfer of the value into the location specified by expression `g`, but does not wait for its completion.

SYNC: The `sync()` operation waits for the completion of the previously issued gets and puts. It synchronizes, or joins, the thread of control on the processor with the remote accesses issued into the network. The target of a split-phase assignment is undefined until a sync has been executed and is undefined if the source of the assignment is modified before executing the sync.

By separating the completion detection from the issuing of the request, split-phase accesses allow the communication to be masked by useful work. The EM3D code above overlaps a split-phase access with other split-phase accesses, which essentially pipelines the transfers. The other typical use of split-phase accesses is to overlap global accesses with local computation. This is tantamount to prefetching.

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 [7]. The ordering of puts is defined only between `sync` operations.

5 Signaling Stores

The discussion above emphasizes the "local computation" view of pulling portions of a global data structure to the processor. In many applications there is a well understood global computation view, allowing information to be *pushed* to where it will be needed next. This occurs, for example in stencil calculations where the boundary regions must be exchanged between steps. It occurs also in global communication operations, such as transpose, and in message driven programs. Split-C allows the programmer to reason at the global level by specifying clearly how the global address space is partitioned over the processors. What is remote to one processor is local to a specific other processor. The `:-` assignment operator, called *store*, stores a value into a global location and signals the processor that owns the location that the store has occurred. It exposes the efficiency of one-way communication in those cases where the communication pattern is well understood.

5.1 Using stores in EM3D

While it may seem that the store operation would primarily benefit regular applications, we will show that it is useful even in our irregular EM3D problem. In the previous version, each processor traversed the "boundary" of its portion of the global graph getting the values it needed from other processors. Alternatively, a processor could traverse its boundary and store values to the processors that need them.

The EM3D kernel using *stores* is given in Program 5. Each processor maintains a list of "store entry" cells that map local nodes to ghost nodes on other processors. The list of store entry cells acts as an anti-dependence list and are indicated as dashed

```
void all_compute_E()
{
  graph_node *n;
  ghost_node *g;
  int i;

  for (g = h_ghost_nodes; g != NULL; g = g->next)
    g->value  := *(g->actual_node);

   sync();

  for (n = e_nodes; n != NULL; n = n->next)
    for (i = 0; i < n->edge_count; i++)
      n->value = n->value
                 - *(n->values[i]) * (n->coeffs[i]);

  barrier();
}
```

Program 4: *EM3D with pipelined communication*

lines in Figure 6. The `all_store_sync()` operation on line 19 ensures that all the store operations are complete before the ghost node values are used. Note also that the barrier at the end the routine in Program 4 has been eliminated since the `all_store_sync()` enforces the synchronization. The curve labeled "Store" in Figure 4 demonstrates the performance improvement with this optimization. (There are actually three overlapping curves for reasons discussed below.)

Observe that this version of EM3D is essentially data parallel, or bulk synchronous, execution on an irregular data structure. It alternates between a phase of purely local computation on each node in the graph and a phase of global communication. The only synchronization is detecting that the communication phase is complete.

A further optimization comes from the following observation: for each processor, we know not only where (on what other processors) data will be stored, but how many stores are expected from other processors. The `all_store_sync()` operation guarantees globally that all stores have been completed. This is done by a global sum of the number of bytes issued minus the number received. This incurs communication overhead, and prevents processors from working ahead on their computation until all other processors are ready. A local operation `store_sync(x)` waits only until `x` bytes have been stored locally. A new version of EM3D, referred to as Program 5a, is formed by replacing `all_store_sync` by `store_sync` in line 19 of Program 5. This improves performance slightly, but the new curve in Figure 4 is nearly indistinguishable from the basic store version–it is one of the three curves labeled "Store."

```
typedef struct ghost_node_t {
  double value;
} ghost_node;

typedef struct store_entry_t {
  double *global ghost_value;
  double *local_value;
} store_entry;

void all_compute_E()
{
  graph_node *n;
  store_entry *s;
  int i;

  for (s = h_store_list; s != NULL; s = g->next)
    s->ghost_value :- *(s->local_value);

  all_store_sync();

  for (n = e_nodes; n != NULL; n = n->next)
    for (i = 0; i < n->edge_count; i++)
      n->value = n->value
              - *(n->values[i]) * (n->coeffs[i]);
}
```

Program 5: *Using the store operation to further optimize the main routine.*

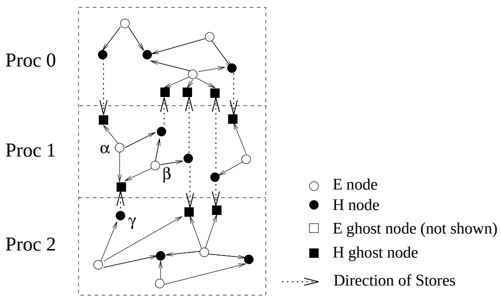

Figure 6: *EM3D graph modified to use stores*

5.2 Language definition: Signaling Stores

STORE: The store operation is specified by an assignment of the form: `g :- e` where `g` is a global l-value and `e` is an arbitrary expression. The value of the right hand side is computed producing a local r-value. Store initiates a transfer of the value into the location specified by expression `g`, but does not wait for its completion.

ALL_STORE_SYNC: The `all_store_sync` is a form of global barrier that returns when all previously issued stores have completed.

STORE_SYNC(N) The `store_sync` function waits until n bytes have been stored (using `:-`) into the local region of the address space. It does not indicate which data has been deposited, so the higher level program protocol must avoid potential confusion, for example by detecting all the stores of a given program phase.

The completion detection for stores is independent from that of reads, writes, gets and puts. In the current implementation, each processor maintains a byte count for the stores that it issues and for the stores it receives. The `all_store_sync` is realized by a global operation that determines when the sum of the bytes received equals the sum of that issued and resets all counters. The `store_sync(n)` checks that the receive counter is equal or greater than `n` and decrements both counters by `n`. This allows the two forms of completion detection to be mixed; with either approach, the counters are all zero at the end of a meaningful communication phase.

6 Bulk Data Operations

The C language allows arbitrary data elements or structures to be copied using the standard assignment statement. This concept of bulk transfer is potentially very important for parallel programs, since global operations frequently manipulate larger units of information. Split-C provides the natural extension of the bulk transfers to the new assignment operators. An entire remote structure can be accessed by a read, write, get, put, or store in a single assignment. Unfortunately, C does not define such bulk transfers on arrays, so Split-C provides a set of functions: `bulk_read`, `bulk_get`, and so on. Many parallel machines provide hardware support for bulk transfers. Even machines like the CM-5, which support only small messages in hardware,[4] can benefit from bulk transfers because more of the packet payload is utilized for user data.

Again, there is a small performance improvement with the bulk store version of EM3D (called Program 5b), but the difference is not visible in the three store curves in Figure 4. The overhead of cache misses incurred when copying the data into the buffer costs nearly as much time as the decrease in message count saves, with the final times being only about 1% faster than those of the previous version.

We have arrived a highly structured version of EM3D through a sequence of optimizations. Depending on the performance goals and desired readability, one could choose to stop at an intermediate stage. Having arrived at this final stage, one might consider how to translate it into traditional message passing style. It is clear how to generate the sends, but generating the receives without introducing deadlock is much trickier, especially if receives must happen in

[4] On the CM-5, each Split-C message can contain 16 bytes of user data. Four bytes of the 20 byte CM-5 network packet is used for header information.

the order data arrives. The advantage of the Split-C model is that the sender, rather than receiver, specifies where data is to be stored, and data need not be copied between message buffers and the program data structures.

7 Spread Arrays

In this section we shift emphasis from irregular, pointer-based data structures to regular, multidimensional arrays, which are traditionally associated with scientific computing. Split-C provides a simple extension to the C array declaration to specify *spread arrays*, which are spread over the entire machine. The declaration also specifies the layout of the array. The two dimensional address space, associated cost model, and split phase assignments of Split-C carry over to arrays, as each processor may access any array element, but "owns" a well defined portion of the array index space.

Most sequential languages support multidimensional arrays by specifying a canonical linear order, *e.g.*, 1-origin column-major in Fortran and 0-origin row-major in C. The compiler translates multidimensional index expressions into a simple address calculation. In C, for example, accessing `A[i][j]` is the same a dereferencing the pointer `A + i*n + j`. Many parallel languages eliminate the canonical layout and instead provide a variety of layout directives. Typically these involve mapping the array index space onto a logical processor grid of one or more dimensions and mapping the processor grid onto a collection of processors. The underlying storage layout and index calculation can become quite complex and may require the use of run-time "shape" tables, rather than simple arithmetic. Split-C retains the concept of a canonical storage layout, but extends the standard layout to spread data across processors in a straight-forward manner.

7.1 "Regular" EM1D

To illustrate a typical use of spread arrays, Program 6 shows a regular 1D analog of our EM3D kernel. The declarations of `E` and `H` contain a *spreader* (`::`), indicating that the elements are spread across the processors. This corresponds to a cyclic layout of `n` elements, starting with element 0 on processor 0. Consecutive elements are on consecutive processors at the same address, except that the address is incremented when the processor number wraps back to zero. The loop construct, `for_my_1d`, is a simple macro that iteratively binds `i` to the indexes from 0 to `n-2` that are owned by the executing processor under the

```
 1 void all_compute_E(int n,
 2                    double E[n]::,
 3                    double H[n]::)
 4 {
 5   int i;
 6   for_my_1d(i,n-1)
 7     if (i != 0)
 8       E[i] = w1*H[i-1] + w2*H[i] + w3*H[i+1];
 9   barrier();
10 }
```

Program 6: *A simple computation on a spread array, declared with a cyclic layout.*

canonical layout, *i.e.*, processor p computes element p, then $p +$ `PROCS` and so on. Observe, that if $n =$ `PROCS` this is simply an array with one element per processor.

In optimizing this program, one would observe that with a cyclic layout two remote references are made in line 8, so it would be more efficient to use a blocked layout. Any standard data type can be spread across the processors in a wrapped fashion. In particular, by adding a dimension to the right of the spreader, *e.g.*, `E[m]::[b]`, we assign elements to processors in blocks. If `m` is chosen equal to `PROCS`, this corresponds to a blocked layout. If `m` is greater than `PROCS`, this is a block-cyclic layout. The loop statement `for_my_1d(i,m)` would be used to iterate over the local blocks. One may also choose to enlarge the block to include ghost elements at the boundaries and perform the various optimization described for EM3D.

7.2 Language definition: Spread Arrays

Declaration: A spread array is declared by inserting a single spreader to the right of an array dimensions.

Addressing: All dimensions to the left of the spreader are spread across processors, while dimensions to the right define the per processor subarrays. The spread dimensions are linearized in row major order and laid out in a wrapped fashion starting with processor zero.

Spread pointers: A second form of global pointer, qualified by the keyword `spread`, provides pointer arithmetic that is identical to indexing on spread arrays.

Figure 7 shows some declarations (with element types omitted for brevity) and their corresponding layouts. The declaration of `X` produces a row-oriented layout by spreading only the first dimension. Because Split-C inherits C's row-major array representation, a column layout is not as simple. However, the effect can be obtained by spreading both dimensions and rounding the trailing dimension up to a multiple of `PROCS`.

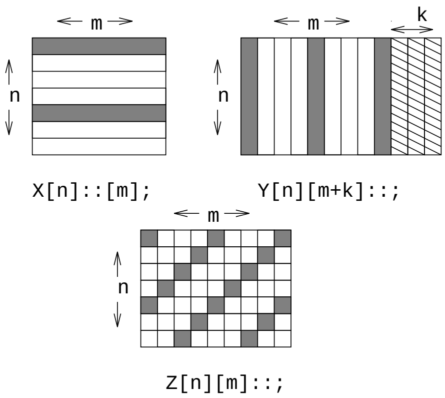

Figure 7: *Spread array declarations and layouts, with processor zero's elements highlighted. Assumes* `n` *is 7,* `m` *is 9,* `k` *is 3, and there are 4 processors.*

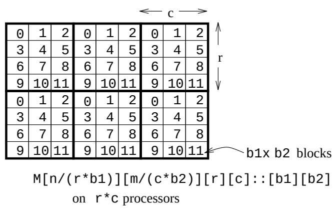

Figure 8: *A declaration for a blocked/cyclic layout in both dimensions. Each block shows the number of the processor that owns it. Shown for* `n`*=8,* `m`*=9,* `r`*=4,* `c`*=3, where there are 12 processors.*

The declaration of `Y` shows such a matrix, assuming that `k` is the computed constant such that `m + k` is a multiple of `PROCS`. The `Z` array is blocked, but has its elements scattered across processors.

To see how spread arrays can achieve other layouts, consider a generic declaration of the form

`M[n/(r*b1)][m/(c*b2)][r][c]::[b1][b2].`

Logically, we may think of this as a two dimensional matrix of size `n` by `m`. Physically, it has blocks of size `b1` by `b2`. A special property holds if we choose `r*c` to be a multiple of `PROCS`: the dimensions `r` and `c` act as an `r` by `c` processor grid. The layout becomes blocked-cyclic in both dimensions. Figure 8 shows a particular case of this for 12 processors, where `r` is 4 and `c` is 3. The number in each block is the number of the processor that owns that block. In each column there are 8 blocks spread across only 4 distinct processors; in each row there are 9 blocks spread across 3 distinct processors.

Since the data layout is well-defined, programmers can write their own control macros (like the `for_my_1d` macro) that iterate over arbitrary dimensions in arbitrary orders. They can also encode subspace iterations, such as iterations over lower triangular, or diagonal elements, or all elements to the left of an owned element. Spread arrays are sometimes declared to have a fixed number of elements per processor, for example, the declaration `int A[PROCS]` will have a single element for each processor.

The relationship between arrays and pointers in C is carried over to spread arrays in Split-C *spread pointers*. Spread pointers are declared with the word `spread`, *e.g.*`int *spread p`, and are identical to global pointers, except with respect to pointer arithmetic. Global pointers index the memory dimension and spread pointers index in a wrapped fashion in the processor dimension.

7.3 Matrix Multiply in Split-C

In many cases it is critical that blocking be employed to reduce the frequency of remote operations, but not so critical how the blocks are actually laid out. For example, in matrix multiplication a block size of b will reduce the number of remote references by a factor of b. However, the three matrices may have very different aspect ratios and not map well onto the same processor grid. In the blocked matrix multiply shown in Program 7, `C` is declared (in line 2) as a `n` by `m` matrix of `b` by `b` blocks and a blocked inner-product algorithm is used. The call to `matrix_mult` in line 17 invokes a local matrix multiply, which is written to operate on conventional C arrays. Observe that the layouts for the three arrays may be completely different, depending on the aspect ratios, but all use the same blocking factor. The iterator `for_my_2d` is used to bind `i` and `j` to the appropriate blocks of `C` under the owner-computes rule.

Figure 9 shows the performance of four of matrix multiply versions. The lowest curve, labeled "Unblocked" is for a standard matrix multiply on square matrices up to size 256×256. The performance curves for the blocked multiply are shown using $\frac{n}{8} \times \frac{n}{8}$ blocks: "Unopt" gives the results for a straightforward C local multiply routine with full compiler optimizations, whereas "Blocked" uses an optimized assembly language routine that pays careful attention to the local cache size and the floating point pipeline. A final performance curve in Figure 9 uses a clever systolic algorithm, Cannon's algorithm, which involves first skewing the blocks within a square processor grid and then cyclic shifts of the blocks at each step, *i.e.*, neighbor communication on the processor grid. All remote accesses are bulk stores and the communication is completely balanced. It peaks at 413 MFlops which on a per processor basis far exceeds published LINPACK performance numbers for the Sparc. This

```
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;
  /* Local copies of blocks */
  double la[b][b], lb[b][b];

  for_my_2D(i,j,l,n,m) {
    double (*lc)[b] = tolocal(C[i][j]);

    for (k=0;k<r;k++) {
      bulk_get(la, A[i][k], b*b*sizeof(double));
      bulk_get(lb, B[k][j], b*b*sizeof(double));
      sync();
      matrix_mult(b,b,b,lc,la,lb);
    }
  }
  barrier();
}
```

Program 7: *Blocked matrix multiply.*

comparison suggests that the ability to use highly optimized sequential routines on local data within Split-C programs is as important as the ability to implement sophisticated global algorithms with a carefully tuned layout.

8 Fusion of Programming Models

Traditionally, different programming models, *e.g.*, shared memory, message passing, or data parallel, were supported by distinct languages on vastly different architectures. Split-C supports these models by programming conventions, rather than enforcing them through language constraints.

Split-C borrows heavily from shared memory models in providing several threads of control within a global address space[10, 12]. Virtues of this approach include: allowing familiar languages to be used with modest enhancements[6, 3, 2], making global data structures explicit, rather than being implicit in the pattern of sends and receives, and allowing for powerful linked data structures. This was illustrated for the EM3D problem above; applications that demonstrate irregularity in both time and space[4, 18] also benefit from these features.

Split-C differs from previous shared memory languages by providing a rich set of memory operations, not simply read and write. It does not rely on novel architectural features, nor does it assume communication has enormous overhead, thereby making bulk operations the only reasonable form of communication [16, 9]. These differences arise because of differences in the implementation assumptions. Split-C

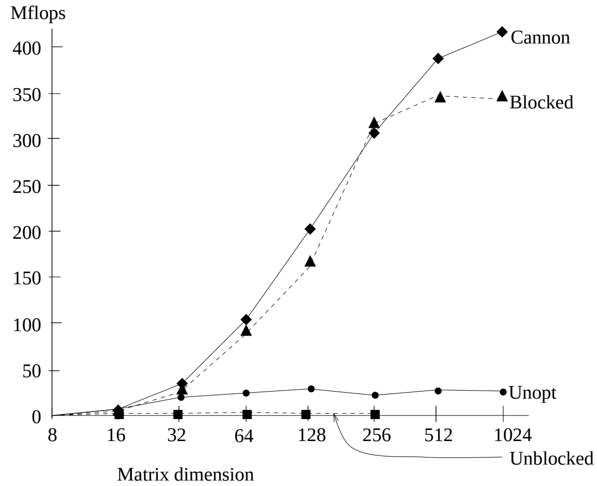

Figure 9: *Performance of multiple versions of matrix multiply on 64 Sparc processor CM-5.*

is targeted toward distributed memory multiprocessors with fast, flexible network hardware, including the Thinking Machines CM-5, Meiko CS-2, Cray T3D and others. Split-C maintains a clear concept of locality, reflecting the fundamental costs on these machines.

Optimizing global operations on regular data structures is encouraged by defining a simple storage layout for global matrices. In some cases, the way to minimize the number of remote accesses is to program the layout to ensure communication balance[5]. For example, an n-point FFT can be performed on p processors with a single remap communication step if $p^2 \leq n$ [5]. In other cases, *e.g.*, blocked matrix multiplication, the particular assignment of blocks to processors less important than load balance and block size.

The approach to global matrices in Split-C stems from the work on data parallel languages, especially HPF [8] and C* [15]. A key design choice was to avoid run-time shapes or dope vectors, because these are inconsistent with C and with the philosophy of least surprises. Split-C does not have the ease of portability of the HPF proposal or of higher level parallel languages. Some HPF layouts are harder to express in Split-C but some Split-C layouts are very hard to express in HPF. The major point of difference is that in Split-C, the programmer has full control over data layout, and sophisticated compiler support is not needed to obtain performance.

9 Summary

This paper has introduced Split-C, a new parallel extension to C which is designed to allow optimization of powerful algorithms for emerging large-scale multiprocessors. It supports a global address space with a clear notion of local and global access and a simple data layout strategy to allow programmers to minimize the frequency of remote access. It provides a rich set of assignment operators to optimize the remote accesses that do occur and to integrate co-ordination among processors with the flow of data. With these simple primitives, it captures many of the best aspects of shared memory, message passing, and data parallel programming avoiding many of the drawbacks of each. It retains a "small language" philosophy, so program performance is closely related to the program text. The languages concepts and optimization techniques have been illustrated by a simple example, which we believe will prove challenging for other available language systems.

Acknowledgements

We would like to thank Ken Stanley for optimizing the local matrix multiply code, and Randy Wang and Su-Lin Wu for writing the original EM3D code.

This work was supported in part by the National Science Foundation as a PFF (number CCR-9253705), RIA (number CCR-9210260), Graduate Research Fellowship, and Infrastructure Grant (number CDA-8722788), by Lawrence Livermore National Laboratory, by the Advanced Research Projects Agency of the DOD monitored by ONR under contract DABT63-92-C-0026, by the Semiconductor Research Consortium 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.

References

[1] H. Berryman, J. Saltz, and J. Scroggs. Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Multiprocessors. *Concurrency: Practice and Experience*, pages 159–178, June 1991.

[2] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J.Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, 1987.

[3] E. Brooks. PCP: A Parallel Extension of C that is 99% Fat Free. Technical Report UCRL-99673, LLNL, 1988.

[4] S. Chakrabarti and K. Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *Principles and Practice of Parallel Programming*, San Diego, CA, 1993.

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles and Practice of Parallel Programming*, May 1993.

[6] F. Darema, D. George, V. Norton, and G. Pfister. A Single-Program-Multiple Data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[7] M. Dubois and C. Scheurich. Synchronization, Coherence, and Event Ordering in Multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[8] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, Jan. 1993.

[9] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimziations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1991 International Conference on Supercomputing*, 1991.

[10] B. A. C. Inc. TC2000 Technical Product Summary. 1989.

[11] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[12] D. Lenoski, J. Laundon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory Based Cache Coherance Protocol for the DASH Multiprocessor. In *In Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, 1990.

[13] N. K. Madsen. Divergence Preserving Discrete Surface Integral Methods for Maxwell's Curl Equations Using Non-Orthogonal Unstructured Grids. Technical Report 92.04, RIACS, February 1992.

[14] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *Siam J. Matrix Anal. Appl.*, 11(3):430–452, July 1990.

[15] J. Rose and G. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing, Vol. 2*, pages 2–16, May 1987.

[16] A. Skjellum. Zipcode: A Portable Communication Layer for High Performance Multicomputing – Practice and Experience. Unpublished draft, March 1991.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, 1992.

[18] C.-P. Wen and K. Yelick. Parallel Timing Simulation on a Distributed Memory Multiprocessor. In *International Conference on CAD*, Santa Clara, California, November 1993. To appear.