Implementing an Efficient Portable Global Memory Layer on Distributed Memory Multiprocessors

Steve Luna

ssl@cs.berkeley.edu

Report No. UCB/CSD 94/#810, May 1994
Computer Science Division -- EECS
University of California, Berkeley, CA 94720

Abstract - Libsplit-C is a library of communication and global storage primitives for distributed memory multiprocessors. The library has been implemented on the CMAM and CMAML active message layers for the CM5, the HPAM active message layer for an FDDI network of HP workstations, the NX message passing library for the Paragon, and the MPLp message passing library for the SP1. This paper describes the issues involved in porting this library to different message layers, and it shows that the library can be implemented efficiently on these message layers. This paper presents microbenchmarks and performance of applications for all the implementations. It also discusses the primitives exported by the library and the library's relationship to languages.[1] [2]

1. This paper was submitted as the thesis for the author's M.S. degree.

2. This research was supported by the NSF Presidential Faculty Fellowship CCR-9253705, the NSF Infrastructure Grant CDA-8727788, the Eugene C. Gee and Mona Fay Gee Scholarship, and by donations from Intel, IBM, and HP.

1.0 Introduction

In this document, we investigate the implementation of an efficient global memory and global communication layer on a spectrum of modern distributed memory multiprocessors. To make the investigation concrete, we focus on Libsplit-C, which serves as a substrate for the parallel language Split-C.

There are several goals for Libsplit-C. First, it should provide a foundation on which parallel languages can be built. Second, it should be portable to a variety of architectures, including the Thinking Machines CM5, Intel Paragon, IBM SP1, and networks of workstations. Finally, it should provide the highest performance possible to languages on which it will be built.

Currently, Split-C is the only language that has been built on Libsplit-C, but the primitives that it exports would be useful to many languages. It has functions that do blocking and non-blocking reads and writes of remote memory. It has low latency communication for small transfers and high bandwidth communication for large ones. It provides global communication and synchronization, atomic functions, and the ability to allocate global memory in a coordinated fashion.

Although the initial implementation of Libsplit-C only ran on the CM5,we have recently re-engineered it so that it is portable to a variety of message layers[3,4]. It is now running on CMAM, CMAML, HPAM, and NX. CMAM and CMAML are two different versions of active messages for the CM5. HPAM is a version of active messages for an FDDI network of HP workstations. NX is the message passing layer that comes with the Intel Paragon. In the future, this library will be ported to many more message layers, and it will serve as a substrate for a single version of Split-C that can run on all the message layers.

Related work includes SAM, Nexus, and distributed memory systems[1,5]. SAM provides a global address space at the level of user defined types; whereas, Libsplit-C provides a flat global address space. SAM does caching but avoids consistency problems by having a no overwrite policy. Libsplit-C does not do caching because the overhead of caching conflicts with the goal of making Libsplit-C as efficient as possible. Nexus has a flat global address space, but it is multithreadid. Libsplit-C was not made multithreadid because it was felt this would add too much overhead. Distributed memory systems, such as Munin, have higher overheads because they are implemented in the operating system. They also transfer entire pages, which increases the possibility of false sharing.

This paper is mainly focused on two issues: 1) porting the library to multiple message layers and 2) implementing the library efficiently. As a quick overview to Split-C and Libsplit-C, section 2 describes the Split-C compilation process, and section 3 describes the primitives that Libsplit-C exports. Section 4 describes the issues involved in porting Libsplit-C to different message layers. Section 5 presents the performance of several message layers and the performance of Libsplit-C on these message layers. Section 6 presents the performance of applications on three architectures. Section 7 is the conclusion.

2.0 Relationship between Split-C and Libsplit-C

Figure 1 shows Libsplit-C's relationship to other software. Libsplit-C is designed to be a compiler target. Its interface is independent of the message layer that is being used. Compilers, such as Split-C, can achieve portability by targeting the library. Libsplit-C is retargeted to a new message layer for every new machine. For the CM5, Libsplit-C was implemented on two different active message layers, CMAM and CMAML. For the network of HP workstations with Medusa FDDI cards, it was implemented on the active message layer HPAM. For the Paragon, it was implemented on the message passing library, NX.

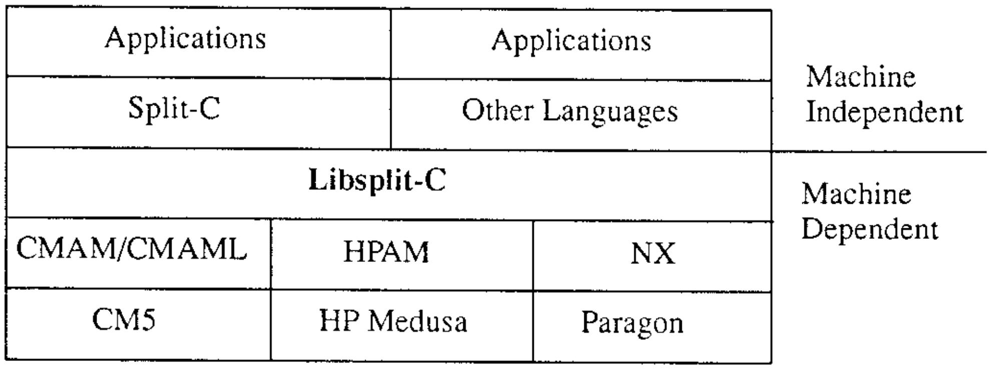

FIGURE 1. Libsplit-C is retargeted for every machine and exports a portable interface.

The Split-C compiler translates a Split-C program into a C program with Libsplit-C calls. The Split-C compiler is responsible for type checking and translating the Split-C syntax into types and routines that are defined in the library. The library is responsible for defining types, operations on these types, and communication and global storage primitives.

2.1 Types and Primitives Defined in Libsplit-C

Libsplit-C defines two types: a global pointer type and a counter. The global pointer type is used to point to addresses that can exist on any of the memories in a distributed memory machine. A global pointer is implemented as a concatenation of a processor id and a local address, but this implementation is not exposed because future machines may provide hardware support for global pointers. The library exports functions to manipulate global pointers. There are functions to set or return the processor id or local offset, to increment and decrement global pointers, and to index global pointers that point to arrays (called spread pointers). Counters are used to record the completion of non-blocking events. Split-C uses instances of counters that are defined in the library. To record the completion of non-blocking communication on a fine granularity, counters can be defined in a program and Libsplit-C primitives can be called directly with these counters as arguments. To check for the completion of the communication, there are Libsplit-C primitives that take counters as arguments. On all the current implementations, counters are implemented as integers, but this implementation is not exposed because future machines (e.g. Cray T3D) will provide hardware support for detecting completion.[1]

The communication and global storage primitives provided by Libsplit-C include global memory, bulk transfer, global communication, global storage, and atomic procedures. Global memory and bulk transfer procedures take global pointers and counters as arguments. The global pointers are used as the source or destination address of remote memory. The counters are used to record completion. Global memory operations transfer intrinsic data types (i.e. char, short, int, float, double). Bulk transfer moves sequences of bytes. Both global memory and bulk transfer provide primitives to do blocking *reads* and *writes* and non-blocking *gets*, *puts*, and *stores*. Global communication functions consist of barrier synchronization and other global communication functions like broadcast. Global storage provides a special malloc for dynamically allocating global memory. Atomic procedures provide support for manipulating remote memory atomically.

2.2 Split-C Syntax

Split-C adds to C type modifiers for creating two new types of pointers. The `global` type modifier turns a pointer into a global pointer and the `spread` type modifier turns a pointer into a spread pointer. Both pointers point to an object on any processor. The difference between them is how pointer arithmetic is performed. Incrementing a global pointer will always increment the local address; whereas, incrementing a spread pointer will increment the processor id until it is incremented past the last processor id. Then, the processor id is reset to 0 and the local address is incremented.

Split-C also adds to C two new assignment operators. The operator `:=` specifies that an non-blocking assignment (i.e. get or put) should take place. Completion of these events can be assured by calling the Libsplit-C procedure, `sync()`. The operator `:-` specifies that a *store* should be used. *Store* differs from *put* because it does not have an acknowledgment. A processor can wait until a certain number of bytes have been stored to it by calling the Libsplit-C procedure, `store_sync()`. Completion of all stores can be assured by calling the Libsplit-C procedure, `all_store_sync()`, on all processors.

2.3 Translation Process

The Split-C compiler validates all Split-C statements. It must ensure that the right hand side of a store is local, and it must also do type checking.

Once the compiler has validated the program, it translates declarations of global or spread pointers into declarations of the Libsplit-C global pointer type. The compiler translates assignment statements into calls to the appropriate calls to global memory procedures.

1. The T3D will at least require a new a implementation of counters, and it may require a new abstraction if counters prove to be too different from the hardware.

Figure 2 illustrates the translation of a simple Split-C program into a C plus Libsplit-C program. The compiler translates declarations of global pointers into declarations of global pointer types. It translates the assignment statement into the appropriate Libsplit-C procedure. Also, because we wanted the compiler to be able to use the library while still achieving high performance, many of the library routines are inlined. This allows the compiler to perform optimizations across library boundaries.

```
void foo()
{
   int *global gptr;
   *gptr :- 5;
}
```

```
void foo()
{
  ___globPtrType gptr;
  __i_store(gptr, 5);
}
```

```
void foo()
{
  ___globPtrType gptr;
  __i_store_ctr(gptr, 5, &recvBytes,
&sentBytes);
}
```

```
void foo()
{
  ___globPtrType gptr;
   {
     Processor pid = toproc(gptr);
     int *addr     = tolocal(gptr);

     if(pid == MYPROC) {
         *(&sentBytes) += sizeof(int);
         *addr = val;
         *(&recvBytes) += sizeof(int);
     } else {
         *(&sentBytes) += sizeof(int);
        am_request3(pid, __i_store_handler, addr,
val, (&recvBytes));
     }
   }
}
```

FIGURE 2. Translation of a Split-C procedure.

3.0 The Libsplit-C Interface

Libsplit-C can be divided into five major components: global memory, bulk transfer, global communication, global storage, and atomic procedures. Global memory consists of the functions available for performing *reads*, *writes*, *gets*, *puts*, and *stores* to intrinsic data types (i.e. char, short, int, float, double) on any processor's memory. Bulk transfer allows large sequences of bytes to be moved between memories. Global communication consists of barrier synchronization and other global communication functions, like broadcast. Global storage provides a special malloc for dynamically allocating global memory. The atomic procedures allow atomic references to remote memory and allow arbitrary procedures to be executed atomically on other nodes.

3.1 Global Memory

Global memory procedures move intrinsic data types between memories. For each of the data types, five types of operations are available: *read*, *write*, *get*, *put*, and *store*. *Read* and *write* are blocking; *get* and *put* are non-blocking. *Store* is a *put* that does not require an acknowledgment. *Store* is an optimization that can be made for networks that are reliable, such as the CM5 or the Paragon. Global memory is optimized for low latency. For high bandwidth, the bulk transfer should be used.

When a global memory operation is executed, two things happen. The data is transferred between the virtual address spaces of two processors, and the event is recorded. As discussed in section 2.1, the completion of non-blocking communication is recorded using counters. The procedures that the Split-C compiler inlines use instances of counters that are defined in the library. The procedure, `sync()`, will wait for the completion of all gets and puts.[1] The procedure, `is_sync()`, will return 1 if all *gets* and *puts* have completed. To detect completion of a single *get* or *put*, as opposed to all *gets* and *puts*, there are *get* and *put* procedures that take a counter as an argument. A Split-C program can define a counter and this counter can be used as an argument to one of these *get* or *put* procedures. `Sync_-ctr()` and `is_sync_ctr()` can be used to wait or test for the completion of one of these *get* or *put* procedures.

The mechanism for determining whether a *store* is complete is different from that for *gets* and *puts*. Like *gets* and *puts*, the *store* procedures that are inlined by the library use instances of counters that are defined by the library. But, because there is no acknowledgment of a *store*, the sender cannot wait or test for the completion of a *store*. Instead, the receiver must call `store_sync()` or `is_store_sync()` with a number of bytes as an argument. `Store_sync()` will wait until that many bytes have been received. `Is_store_-sync()` will return true if that many bytes has been received. Another procedure, `all_store_sync()`, must be called by all processors and will wait until all *stores* have completed. To detect completion of *stores* on a fine granularity, procedures for issuing *stores* and waiting or testing for completion are provided. These procedures take two counters as arguments. The reason for having two counters will be explained in the section 4.1 which discusses implementation.

These are the procedures that the Split-C compiler inserts into the code. They are divided into three groups: those that initiate a global memory operation and use counters defined in the library, those that initiate a global memory operation and take counters as arguments, and those that are used for detecting completion. These procedures initiate a global memory operation and use counters defined in the library.

```
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);
```

1. Except those that were issued with an explicit counter.

```
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 __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 __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);
```

These are the procedures that initiate a global memory operation and take counters as arguments.

```
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);
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);

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);
```

These are the procedures that are used to wait or test for completion.

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

```
void all_store_sync(void);
void store_sync(int bytes_expected);
bool is_store_sync(int bytes_expected);

void sync_ctr(Counter *ctr);
bool is_sync_ctr(Counter *ctr);
void store_sync(int bytes_expected, Counter *rbytes,
                Counter *sbytes);
bool is_store_sync(int bytes_expected, Counter *rbytes,
                   Counter *sbytes);
```

3.2 Bulk Transfer

The procedures for bulk transfer are optimized for high bandwidth. These procedures transfer a series of bytes from the virtual address space of one processor to the virtual address space of another. They also record the completion of the transfer after it has finished.

There are again five operations: *read*, *write*, *get*, *put*, and *store*. These procedures are analogous to the global memory procedures. *Read* and *write* are blocking. *Get* and *put* are non-blocking. *Store* is a *put* that does not require an acknowledgment. These procedures manipulate counters in the same manner that the global memory procedures do. Each of these procedures has a form which uses implicit counters, and a form which takes one or more explicit counters as arguments. Waiting or testing for completion is done using the same procedures that are used for global memory. However, unlike global memory routines, the bulk transfer routines are not inlined.

These are the bulk transfer procedures.

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

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

3.3 Global Communication

These functions require all the processors to participate (i.e. all processors must call the function). Barrier, broadcast, reduce, and scan are the functions provided. Barrier synchronizes the processors by ensuring that no processors leaves the barrier until all have entered it. Broadcast sends a message from processor 0 to all other processors. Reduce computes the following function for the operations: add, multiply, max, min, logical or, logical xor, and logical and.

$$a_0 \oplus \ldots \oplus a_i \oplus \ldots \oplus a_n, \text{where } a_i \text{ is the argument from processor } i$$

For a scan, processor i computes the following function for the operations: add, multiply, max, min, logical or, logical xor, and logical and.

$$a_0 \oplus \ldots \oplus a_i\text{, where } a_i \text{ is the value from processor } i$$

These are the global communication functions.

```
int all_reduce_to_one_add(int val);
unsigned int all_reduce_to_one_uadd(unsigned int val);
float all_reduce_to_one_fadd(float val);
double all_reduce_to_one_dadd(double val);

int all_reduce_to_one_mult(int val);
unsigned int all_reduce_to_one_umult(unsigned int val);
float all_reduce_to_one_fmult(float val);
double all_reduce_to_one_dmult(double val);

int all_reduce_to_one_max(int val);
unsigned int all_reduce_to_one_umax(unsigned int val);
float all_reduce_to_one_fmax(float val);
double all_reduce_to_one_dmax(double val);

int all_reduce_to_one_min(int val);
unsigned int all_reduce_to_one_umin(unsigned int val);
float all_reduce_to_one_fmin(float val);
double all_reduce_to_one_dmin(double val);

int all_reduce_to_one_or(int val);
unsigned int all_reduce_to_one_uor( unsigned int val);
float all_reduce_to_one_fmax(float val);
double all_reduce_to_one_dmax(double val);

int all_reduce_to_one_min(int val);
unsigned int all_reduce_to_one_umin(unsigned int val);
float all_reduce_to_one_fmin(float val);
double all_reduce_to_one_dmin(double val);

int all_reduce_to_one_or(int val);
unsigned int all_reduce_to_one_uor( unsigned int val);

int all_reduce_to_one_xor(int val);
unsigned int all_reduce_to_one_uxor(unsigned int val);

int all_reduce_to_one_and(int val);
unsigned int all_reduce_to_one_uand(unsigned int val);


int all_bcast(int val);
int all_bcast_i(int val);
unsigned int all_bcast_u(unsigned int val);
float all_bcast_f(float val);
double all_bcast_d(double val);


int all_scan_add(int val);
unsigned int all_scan_uadd(unsigned int val);
float all_scan_fadd(float val);
double all_scan_dadd(double val);

int all_scan_mult(int val);
unsigned int all_scan_umult(unsigned int val);
float all_scan_fmult(float val);
double all_scan_dmult(double val);
```

```
int all_scan_max(int val);
unsigned int all_scan_umax(unsigned int val);
float all_scan_fmax(float val);
double all_scan_dmax(double val);

int all_scan_min(int val);
unsigned int all_scan_umin(unsigned int val);
float all_scan_fmin(float val);
double all_scan_dmin(double val);

int all_scan_or(int val);
unsigned int all_scan_uor(unsigned int val);

int all_scan_xor(int val);
unsigned int all_scan_uxor(unsigned int val);

int all_scan_and(int val);
unsigned int all_scan_uand(unsigned int val);
```

3.4 Global Heap

Libsplit-C exports a function, `all_spread_malloc()`, which dynamically allocates global memory. Global memory includes the data segment and a portion of the heap. This portion of the heap has the same starting address on every processor. Local memory, by contrast, contains the stack, static and external variables, and the portion of the heap that is allocated by `malloc()`. Note that besides the memory starting at the same address on every processor, there is no difference between global and local memory. A global pointer can point to local memory and a local pointer can point to global memory.

The reason global memory needs to have the same starting address on every processor is so that a pointer to an array that is spread across the processors can be described by a processor id and a local offset. If the data started at different addresses on different nodes, an additional level of indirection would be required.

In addition to `all_spread_malloc()`, `all_spread_free()` is also provided. This function allows dynamically allocated data to be freed. Here are the prototypes of the global heap functions.

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

`Spread_malloc()` is no longer supported in Libsplit-C. The reasons for this are discussed in section 4.4.2.

3.5 Atomic Procedures

Libsplit-C provides several primitive atomic procedures: `test_and_set()`, `fetch_and_add()`, `exchange()`, and `cmp_and_swap()`. These operations are built on a more general facility that allows an application to define its own atomic procedures. Here are the prototypes of the atomic procedures.

```
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);

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);
```

The RPC functions are no longer supported in Libsplit-C because atomic provides the same functionality.

4.0 Implementing Libsplit-C

The focus of this paper is implementing Libsplit-C efficiently on multiple message layers. This section will discuss the issues involved in porting the different components of Libsplit-C to three different message layers. Section five will discuss the efficiency of the library.

The three implementations that will be discussed are the CM5, the FDDI network of HP workstations, and the Paragon. On the CM5, there are two message layers that Libsplit-C has been ported to, CMAM and CMAML. Except for global communication, these interfaces are almost identical so only CMAM will be discussed. On the network of HP workstations, Libsplit-C was built on the active message layer, HPAM. On the Paragon, Libsplit-C was built on the message passing software, NX, that comes with the machine.

4.1 Global Memory

The implementations of global memory on CMAM and HPAM have demonstrated to us that global memory can be implemented easily and efficiently on active messages. Given an active message layer, porting the Libsplit-C global memory to a new architecture is trivial. The major issues involved in porting CMAM to HPAM involved the calling convention of the processors and whether or not an unreliable network should be exposed to Libsplit-C. However, the implementation of Libsplit-C on the Paragon demonstrated that message passing does not support global memory without adding a layer of software on top of it.

4.1.1 The CM5

Active messages can be used to implement global memory easily and efficiently. CMAM provides support for sending a five word message. When a message arrives at the remote node, the first word is used as the address of a handler. The handler is invoked with the remaining four words as arguments.

Figure 3 depicts the implementation of a *get*. The four columns represent the global memory of four processors. The processor on which the *get* is invoked increments the counter and then sends an active message containing the address of the source, the destination, and the counter to the remote node. The handler on the remote node loads the data and sends it along with the address of the destination and counter in a reply message. When the reply message arrives, the handler stores the data into the destination address and then decrements the counter.

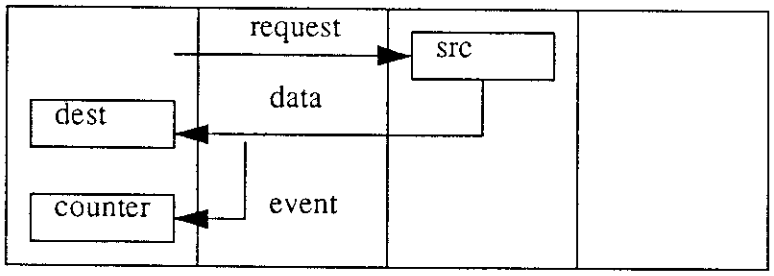

FIGURE 3. The implementation of get and read on active messages.

Figure 4 depicts the implementation of *put* on active messages. The processor on which the *put* is invoked increments the counter, loads the data, and sends it along with the address of the destination and the counter to the remote node. The handler on the remote node stores the data and sends the address of the counter in a reply message. When the reply message arrives, the handler decrements the counter.

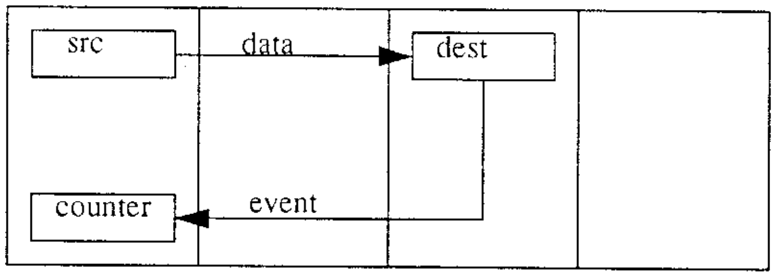

FIGURE 4. The implementation of put and write on active messages.

Because the counters record the number of *gets* or *puts* that still have not completed, procedures that test or wait for completion can be implemented easily. `Sync()` and `sync_ctr()` wait until a counter becomes zero. `Is_sync()` and `is_sync_ctr()` return true if a counter is zero.

Read and *write* can be implemented easily using *get* and *put*. For *read*, a *get* is issued, and for *write*, a *put* is issued. Then, the *read* or *write* waits until the *get* or *put* completes.

Figure 5 depicts the implementation of *store* on active messages. As mentioned in section 3.1, a *store* takes two counters as arguments. The processor on which the *store* is invoked increments the local counter by the size of the data. Then it loads the data and sends it along with the address of the destination and the remote counter to the remote node. The handler on the remote node stores the data and increments the remote counter by the size of data.

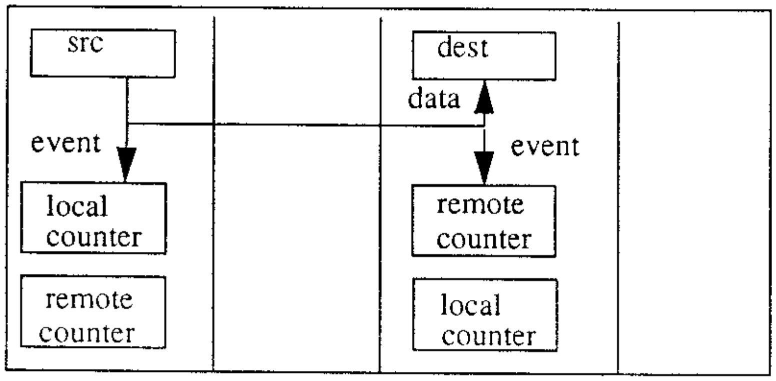

FIGURE 5. The implementation of store on active messages.

The procedures which wait or test for completion of *stores* can be implemented using these counters. `store_sync()` is implemented by waiting until the local counter has been incremented by the number of bytes that are expected. Once it has, that many bytes are subtracted from both the local and the remote counter. When the processors execute `all_-store_sync()` each processor subtracts its local and remote counters, and then every processor's difference is summed. Once this value has reached zero, the same number of bytes have been received as sent so all *stores* have completed.

4.1.2 The HPs

Implementing global memory on HPAM was easy because HPAM supports an active message interface similar to CMAM's[4]. However, there were several issues that arose during the port to HPAM. First, there is an architectural reason why HPAM supports the same interface as CMAM. Second, because the calling convention of PA-RISC differs from the calling convention of the Sparc, there is an opportunity for optimization. Finally, there was the issue of dealing with the unreliable network on which HPAM was built.

Both CMAM and HPAM send a five word message where the first word is the address of a handler and the next four words are arguments to the handler. CMAM restricts the number of arguments that a handler can have to four words because the CM5 network interface limits message sizes to a few words. HPAM also restricts the number of arguments to four words, but it does so for a different reason. The HP Precision Architecture specifies that the first four arguments to a procedure will be passed in registers. By restricting the number of arguments to four, the HPs can load a message from the network interface and then call the handler without having to store arguments to the stack.

The rationale behind HPAM's five word limit is more fundamental than the rationale behind CMAM's limit. The CM5's restriction is an artifact of the CM5's network interface and will change with the next implementation of the CM5. The Intel Paragon supports larger message sizes. FDDI supports message sizes of up to 4KB, and ATM supports message sizes of 12 words. However, many processors, including the i860 on which the Paragon is based, support a calling convention which passes the first four arguments in registers. Active messages on the Paragon will probably support the same interface as HPAM and CMAM.

There are ways that the implementation on HPAM could have been optimized. HP PA passes arguments of type float and double in floating point registers; whereas on the Sparc, they are passed in integer registers. On the CM5, there was no need to provide any procedure to invoke an active message other than the one that took four arguments in integer registers. But, on the HPs, if one of the arguments is a float or a double, it must be moved from the floating point registers to the stack and then from the stack to integer registers. It would be more efficient to provide an active message function which took one of its arguments in floating point registers. The extra cost of moving a double to memory and back into registers was measured at 40ns on a 99MHz HP 735.

Unlike the CM5 network, the network that HPAM is built on is unreliable. The experience of implementing Libsplit-C on the HPs provided us insight into whether or not this unreliability should be exposed to the library. In the initial version of HPAM, the cost of checking to see if packets had been dropped was significantly higher than the cost to poll the network. (In both CMAM and HPAM, the program must poll the network. The poll procedure checks if new messages have arrived and extracts them if any exist.) Because Libsplit-C provides procedures which wait or test for completion, Libsplit-C could check for dropped packets during these procedures. Since a program that waits or tests for completion is probably waiting for messages anyway, this is a good time to pay the overhead of checking for dropped packets.

The problem with checking for dropped packets during these procedures is that poll has become exposed to application programmers in the form of a procedure called splitc_poll. This procedure was originally provided as a way to increase performance. By making sure that processors check the network frequently, an application can decrease the latency of messages. However, now that it has become exposed, it is possible for applications to write loops that do not wait or test for completion but instead loop on some condition and call splitc_poll inside the loop. Unless poll checks for retransmission, deadlock is possible.

Because checking for retransmission in Libsplit-C would have left the possibility of deadlock, a second version of HPAM was designed that checked for dropped packets infrequently. This version of the poll procedure adds minimal overhead and checks for dropped packets.

4.1.3 The Paragon

The implementation of Libsplit-C on the Paragon was built on message passing instead of active messages. Porting to the Paragon provided us with experience implementing Libsplit-C on a message layer other than active messages. It also filled a short term goal of getting Split-C running on the Paragon during the period before active messages was available.

There are several obstacles to building get, put, and store on message passing. First, some active operation is required on the receiving side. For get and put, a reply message must be generated, and for store, a counter must be incremented. Because of this inherent problem with building shared memory on message passing, any solution will have to post at least one receive in advance, and it will have to periodically check to see if the receive has completed. If it has, it will have to perform some active operation.

Another problem with implementing put or store on message passing is that the destination address is not encoded in the send; it is encoded in the receive. In Libsplit-C, the remote node has no way of knowing for which address it should post a receive. Because of this problem, data will have to be buffered temporarily. Puts and stores cannot write directly to their target address because the receiving node will have to look at the message to determine this address.

One solution is to have each node post three receives, each with a different tag. One tag would be for gets, another for puts, and a third for stores. Poll could check if any of the receives had completed. If it had, it could handle each message separately. There are two problems with this solution. First, it does not handle atomic procedures. A fourth tag must be used to represent atomic procedures. Second, with NX it is advantageous to only have one type of receive outstanding. The cost of checking whether a receive has completed is five microseconds. If three different types of receives were used, the cost of a poll would be 15 microseconds.

The solution that was chosen was to build an active message layer, NXAM, on top of NX. It posts some receives of the same type in advance. A poll procedure checks if the receives have completed, and if they have, it uses the first word of the message as the address of a handler, and the remaining four words as arguments to the handler. This solution requires that only one type of receive needs to be checked on each call to poll. Atomic procedures are implemented in the same way that they are implemented in CMAM and HPAM.

It is interesting to note that NX actually provides active message-like functionality. It is possible to perform a send which executes a handler on the remote node. The problem is that the cost of this operation is twice the cost of a normal send/receive. With the message processor turned off, the round trip latency of an NX active message is about one millisecond, whereas the round trip latency is about half a millisecond for send/receive.

4.1.4 Concluding Remarks on Global Memory

Building global memory on top of active messages is easy. However, message passing does not provide enough flexibility to implement global memory without building another layer of software on top of it. Building an active message like interface on top of message passing is the simplest way to implement global memory. This also provides atomic procedures at no extra cost.

The experience of porting global memory to HPAM provided us with two insights. First, because global memory is supposed to provide the lowest possible latency, its implementation is closely tied to the calling convention of the processor. On the sending side, data should be moved directly from the registers to the network interface. On the receiving side, data should be moved directly from the network interface to the registers. Data should not have to pass through memory. Second, unreliable networks should not be exposed to Libsplit-C. Once splitc_poll has been exposed to the programmer, there is no clear dichotomy between Libsplit-C procedures that should do retransmission and those that should not.

In all the current implementations, counters are implemented as integers, but counters have been defined as an opaque type. Future architectures (e.g. T3D) will provide support for global memory. They will also provide ways to determine if non-blocking events have completed. By making the counters an opaque type, Libsplit-C reserves the ability to record the completion of events in some manner other than counters.

4.2 Bulk Transfer

Because of the large difference in maximum packet size, the CM5 and the HPs have different implementations of bulk transfer. On the CM5, overhead is paid at the beginning of a message so that the full payload of subsequent messages can be used to transfer data. On the HPs, packet size is large enough that this overhead is unnecessary. On the Paragon, overhead is paid at the beginning of the transfer so that a receive with the correct address can be posted on the remote node before the data arrives. Posting this receive avoids an extra copy.

4.2.1 The CM5

CMAM requires that a segment be opened on the receiving node before a bulk transfer can be initiated. Opening this segment is required because of two artifacts of the CM5's architecture. First, because packets can become unordered in the CM5's network, a record of how many bytes remain needs to be kept on the receiving node. Otherwise, the receiver would have no way of determining that the transfer had completed. Second, because packet sizes are small, as much of the payload as possible should be used to transfer data. Transferring information that is used to record completion (the address of a handler) should not be sent with every message because this would restrict bandwidth unnecessarily. It is better to put this information in a segment on the receiving node so that the final message will be able to invoke the handler with the address of the counter.

If the CM5's network was ordered, or if the maximum message size was larger, a segment would not be required. If the network was ordered, the first packet could contain the address for its data. The final packet would contain the address of a handler so that completion could be recorded. If the maximum packet size was larger, fragmentation could be pushed up to Libsplit-C. Each packet could contain the address of a handler. For a *store*, the handler would increment the counter by the number of bytes transferred. For a *get* or a *put*, it would send a reply message which would decrement the counter on the node which initiated the *get* or *put*. If packet sizes are large, the cost of transferring the address of the handler and counter with each packet is negligible.

For a *get*, the cost of setting up a segment is negligible because no communication needs to occur. But, for a *put* or a *store*, a full round trip latency must be paid before the bulk transfer can start. For both the *put* and the *store*, the initiating node must send an active message to the remote node. The handler of this message allocates a segment and sends a reply message containing the segment id. The initiating node waits until this id has arrived. Once it has, it can transfer data to the remote address. Completion is recorded by a handler which is called after the transfer is complete. For a *put*, this handler sends a reply message which decrements a counter on the initiating node. For a *store*, this handler increments a counter by the number of bytes that have been stored.

4.2.2 The HPs

The network on which HPAM is built is both ordered and allows 4KB message sizes. But because the network is unreliable, it is not good enough to have the last packet record completion. If a packet before the last packet is dropped, completion of the event can be recorded before all of the fragments have actually arrived. Pushing fragmentation up to Libsplit-C is an acceptable solution though. It eliminates having to set up a segment before data is transferred.

HPAM provides a bulk transfer procedure that takes a remote node, a source address, a destination address, a length, a handler, and the address of a counter. It copies length bytes from the source address into a packet, sends this packet to the remote node, copies the data to the destination address, and then calls the handler with the address of the counter as an argument.

For a *put*, Libsplit-C increments the counter of *puts* that are outstanding by the number of fragments into which the bulk transfer must be broken. The initiating node sends these fragments to the remote node. After a fragment is copied to the destination address on the remote node, a handler is invoked which sends a reply message with the address of the counter to the initiating node. When the reply message arrives, a handler is invoked which decrements the counter. Only after all fragments have been copied to the remote node will completion for all the *puts* in this bulk transfer be recorded. Note that although a reply message is generated for each of the fragments, a reply must be generated anyway because the network is unreliable. The message which records the completion for a fragment is piggybacked on the acknowledgment.

A store is similar to a *put*, except that when the handler on the remote node is invoked, the counter is incremented by the number of bytes that were in the fragment. Note that for a *store*, each message generates an acknowledgment even though no information is required by Libsplit-C. This acknowledgment is needed because the network is unreliable.

For a *get*, the counter of *gets* outstanding is incremented by the number of fragments. For each fragment, a message is sent to the remote node. The handler for this message replies with an HPAM bulk transfer message. After the data has been copied to the destination address, the handler for this message decrements the counter of *gets* outstanding. Only after all data has been received will the completion for all the *gets* in this transfer be recorded.

4.2.3 The Paragon

The Paragon implementation must pay the extra overhead of sending a message for every *bulk put* and *bulk store* because it is built on message passing. If a receive is posted before data arrives, the receiving node can copy data directly into the destination address. If the receive is posted after the data arrives, the receiver must buffer the data until the receive is posted. This buffering adds the overhead of an extra copy and will decrease bandwidth.

For a *put* or a *store*, a NXAM message is sent to the remote node. The handler for this message posts a receive for the destination address of the actual store. Posting this receive allows the NX layer to avoid an extra copy of the data. Immediately after sending the message to the remote node, the node on which the *put* or *store* was initiated sends the data. Note because the sending node does not wait for an acknowledgment, the posting of the receive and arrival of the data could become unordered. If they become unordered, the message passing software will buffer the data and the put or store will still function correctly. In the common case, the messages will be ordered and the copy will be avoided.

4.2.4 Concluding Remarks on Bulk Transfer

Having both an unordered network and a small maximum packet size forces the implementation to set up a segment on the receiving node before the data is transferred. For a put or a store, this translates into the cost of one round trip latency. If a network allows a large maximum packet size, segments are not required. If a network is both ordered and reliable, segments are not required.

In order to get maximum bandwidth out of an implementation built on message passing, a receive must be posted on the receiving node before the message arrives. For a *put* or a *store*, this translates into the overhead of sending an additional message.

4.3 Global Communication

The CM5 provides hardware support for global communication. The implementation of Libsplit-C on CMAML uses this hardware for all global communication. CMAM only uses the hardware for barrier synchronization, but because this barrier is fast, it influenced the way other global communication functions were implemented. On the HPs and the Paragon, there is no hardware support for global communication. The resulting implementation is optimized for machines that have to build all global communication from point to point messages.

4.3.1 The CM5

The implementation of Libsplit-C on CMAML uses the control network on the CM5 for all global communication functions. The implementation that is built on CMAM only uses it for broadcast, but because the broadcast is fast (3 microseconds), it is used as a building block for the other global communication functions.

On CMAM, broadcast and reduce are implemented using stores. A tree of processors is created statically using the following mapping.

$$parent = \frac{MYPROC - 1}{2},\ leftchild = 2MYPROC + 1,\ rightchild = 2MYPROC + 2$$

Figure 6 shows the tree for eight processors. To implement broadcast, processors receive values from their parent and send them to their children. To implement reduce, processors receive values from their children, perform an operation on the values, and send the result to their parent.

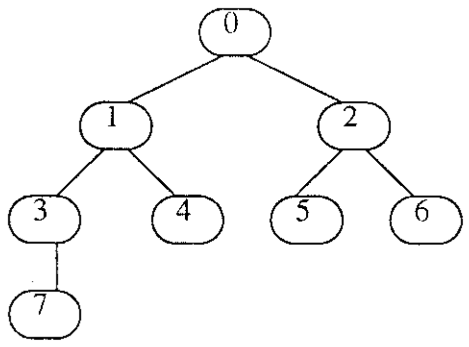

FIGURE 6. To do a broadcast the nodes propagate values down the tree. To do a reduce they propagate values up the tree. To do a barrier, they propagate values up and then down.

The major issue involved in implementing broadcast and reduce this way is flow control. Either an unlimited amount of buffer space must be required at every node or flow control must be used. For a broadcast, if flow control is not used, a parent can send to its children before they have sent the value of the previous broadcast to their children.

Because the CM5 provides support for fast (3 microseconds) barrier synchronization, these barriers are used to control flow. Before every global operation is performed, a barrier is executed. This ensures that the previous global operation has completed; thereby, guaranteeing that the buffers can be used for the next global operation. On a 4 node CM5, the overhead of the barrier is about 10% of the cost of a reduce or a broadcast, and on a 64 node CM5 it is less than 5%.

Scans are implemented in a manner that is similar, but not identical, to broadcast and reduce. The tree for scans is different, and scan is a two phase operation. Instead of just passing values up or down a tree, processors first send values up the tree, and then they pass values down the tree. The overhead of a barrier in a scan is even less than the overhead for a reduce or a broadcast because scan is a more expensive operation.

The implementation of global communication on the Cray T3D and the Meiko will probably be similar to the implementation on CMAM. Both machines provide support for a fast barrier that is executed by the hardware.

4.3.2 The HPs

On the HPs, barrier is a much more expensive operation because hardware support for barrier is not provided. Because the barrier can be twice as expensive as a reduce or a broadcast, another method is used for flow control.

Barrier is implemented using the same tree that is used for broadcast and reduce. All processors wait for a signal from each of their children. Once all signals are received, they pass a signal on to their parent. When processor 0 receives all its signals, it broadcasts a signal down the tree. Once a processor has received the signal from its parent and passed the signal to all its children, it is free to return from the barrier. This implementation ensures that no processor leaves the barrier until all have entered it.

Note, this implementation is basically a reduce followed by a broadcast. The cost of the barrier is about twice the cost of a reduce or a broadcast. The cost of these operations would have been tripled if flow control was managed using barrier.

A more efficient method of doing flow control is to treat the tree as a static dataflow graph. For a broadcast, a processor waits for a value from its parent and acknowledgments from its children. Once these have been received, it sends the value to its children. After the value has been sent, the buffer space that is used to hold the value can be reused for the next broadcast. The processor sends an acknowledgment to its parent signifying that it is ready for the next value. The implementation of reduce is similar.

Treating the tree as a static dataflow graph allows the operations to be pipelined. Log(p) operations, where p is the number of processors, can be in progress at any given time. This was not possible in the CM5 implementation because the barrier at the beginning of each function prevents more than one function from being in progress at a time. The disadvantage to this method is that an acknowledgment is required for every message up or down the tree.

Scan cannot be pipelined easily because it is a two phase operation. The leaves of the scan tree must initiate the scan by passing values up the tree. Then, they must wait for the result to be passed back down the tree. Because the leaves both initiate and end the scan, multiple operations cannot be pipelined. After the leaves have returned, all the nodes have completed the scan already. On the other hand, because scan is a two phase operation, flow control is not required. Only one scan can be in progress at a time.

An issue that arose with this implementation of scan is that there is a dependency hazard between the result of a global function and a read from another processor. For example, consider the following code fragment.

```
int g_buckets[PROCS]::[RADIX];
int *buckets;
buckets = g_buckets[MYPROC];

for (i = 0; i < RADIX; i++) {
   if (MYPROC == 0 && i > 0)
      buckets[i] += g_buckets[PROCS - 1][i - 1];
   buckets[i] = all_scan_add(buckets[i]);
}
```

This code worked on the CM5 because that implementation of scan does a barrier before it returns. The barrier synchronizes the nodes enough to ensure that processor PROCS-1 will have stored the result of the scan before processor 0 reads g_buckets[PROCS-1][i-1]. The implementation on the HPs does not synchronize processors before the function returns because barrier is expensive. This means processor 0 can read g_buckets[PROCS-1][i-1] before processor PROCS-1 has returned from the scan.

This is not a problem with the implementation. It is actually a bug in the code fragment. Even if a barrier were inserted at the end of the scan, the problem would still exist. The barrier does not guarantee that all processors leave the barrier together, it only guarantees that no processor leaves the barrier until all have entered it. It is possible for processor 0 to leave the barrier and read from processor PROCS-1 before the barrier signal has propagated to processor PROCS-1. This will cause processor 0 to get the wrong value because processor PROCS-1 has not left the barrier and has not stored the result from all_scan_add.

For the code fragment to be correct, there would have to be a barrier after the assignment statement. This would ensure that the result of the function was stored on all processors before the read took place. This hazard could have been hidden if the functions were procedures that took a reference instead of functions that returned a value. The procedures could store the result in the reference and then execute a barrier. Although this would make the global communication interface simpler, it would double the cost of every scan on architectures that did not have hardware support for a fast barrier.

4.3.3 The Paragon

Like the HPs, the Paragon does not provide hardware support for global communication functions. The issues involved in implementing global communication on the Paragon are the same as those involved in implementing it on the HPs.

Note, NX provides global communication functions. It would have been nice to use their global communication functions instead of having to implement our own. However, there are two problems with this. First their global communication functions have ghastly performance (2ms for a barrier). Second, there was no way to compose the NX global communication functions with the message layer that we built on top of NX. Our message layer requires that the receive queue be polled in order for messages to be serviced. Since NX does not poll our receive queue, a deadlock could ensue.

4.3.4 Concluding Remarks on Global Communication

If the hardware does not provide support for global communication functions, they must be built from point to point messages. A method of flow control is needed to ensure that multiple invocations of a global function do not interfere with each other. Executing a barrier at the beginning of every global operation is an efficient way of controlling flow if barriers are implemented efficiently in hardware. If hardware support for barriers is not provided, techniques from static dataflow can be used.

Because adding a barrier at the end of a function does not guarantee that all processors store the result of the function at the same time, a dependency hazard between the result of a global communication function and a read of the result is exposed to the programmer.

4.4 Global Heap

Split-C differentiates between global and local memory. Global memory includes a portion of the heap and the data segment. Local memory includes the stack, static and external variables, and the portion of the heap that is allocated by malloc. The important property of global memory is that the address of variables that are allocated in global memory start at the same address on every node. This ensures that a global pointer that points to a variable in global memory will point to the same global variable on other nodes. For example, the following code fragment is an incorrect use of a global pointer.

```
foo()
{
   int *global gptr;
   int flag;
   gptr = toglobal((MYPROC+1)%PROCS, &flag);
   *gptr = 1;
}
```

Because the address of stack variables are dependent on the calls that a processor executes at run time, there is no guarantee that these variables will start at the same address on all processors. Even if foo() is called by all processors, there is no guarantee that the stack frame allocated to foo starts at the same address on all processors.

The proper way to dynamically allocate a flag on every processor is to allocate an array on the global heap.

```
foo()
{
   int *spread sptr;
   sptr = all_spread_malloc(PROCS, sizeof(int));
   sptr[(MYPROC+1)%PROCS] = 1;
}
```

4.4.1 All_spread_malloc

`All_spread_malloc()` is a Libsplit-C function that returns data starting at the same address on every node. On the CM5, the implementation of `all_spread_malloc()` was easy because a call to `sbrk()` on any of the nodes will become synchronized through the host. (`sbrk()` is a Unix function that extends the heap by a given number of bytes.) The host keeps track of the top of the heap for all the nodes and when it extends the heap, it extends it for all the nodes. If `all_spread_malloc()` needs more memory for its free pool, it performs an `sbrk()`. It is guaranteed that the memory it receives from the `sbrk()` will start at the same address on all processors.

For machines that do not provide support for a global heap (e.g. the HPs and the Paragon), `all_spread_malloc()` can be written using `sbrk()`. Because `malloc()` relies on it being the only function to use `sbrk()`, `malloc()` must also be rewritten.

In the implementation of `malloc()` and `all_spread_malloc()` for the HPs and the Paragon, both functions have their own free pools. When an `sbrk()` needs to be called from within `all_spread_malloc()`, a global reduction is performed in order to determine the maximum of all the top of heaps. Then all processors `sbrk()` to that value and put the fragmented data on the local free pool. After that, all processors can `sbrk()` space from the free pool because it will start at the same address.

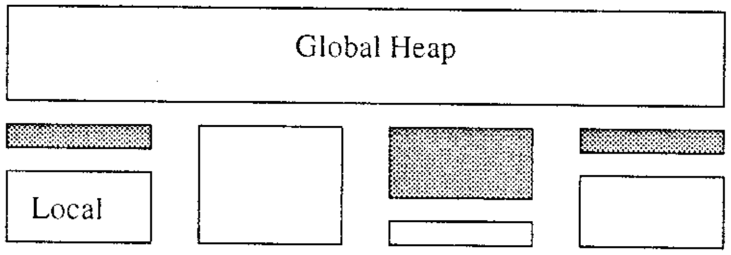

FIGURE 7. The when a global sbrk needs to be performed, the local heaps are brought to the same level and the fragments (in grey) are put on the local free pools. Then the global sbrk is performed.

4.4.2 Spread_malloc

The initial implementation of Libsplit-C on the CM5 supported a function called `spread_malloc()`. It allowed any processor to autonomously allocate global memory on all the processors. This was feasible on the CM5 because any processor can autonomously extend the heap. But, on the implementation that is built on `sbrk()`, it is difficult to provide a `spread_malloc()`.

If a processor wants to allocate memory on other processors, it must have a way of sending a message to all other processors. The most efficient way to do this is to form a broadcast tree. But, supporting autonomous global communication would require an active message layer that allows a send, not just a reply, from within a handler. CMAM, CMAML, and HPAM do not support this. The other way to do the communication is to have the processor send a message to all other processors, one by one. This is not scalable. Also, it will also be necessary to stop processors from calling `sbrk()` while information about the heap is being collected by the processor doing the `spread_malloc()`.

Because implementing a scalable `spread_malloc()` would have been difficult, it was removed from Libsplit-C. The decision to remove it was made easier by the fact that none of the applications that we examined use `spread_malloc()`. They use `all_spread_malloc()`.

4.5 Atomic Procedures

On the CM5 and the HPs, atomic procedures were trivial to implement because active message handlers are atomic procedures. On the Paragon, an active message layer, NXAM, was built on the message passing library, NX. NXAM was built to provide support for global memory operations, but it also supports atomic procedures.

5.0 Performance of Libsplit-C

This section describes the efficiency of Libsplit-C on the different message layers. It compares the cost of Libsplit-C operations with the cost of message layer operations. It describes overhead and latency for global memory, bandwidth for bulk transfer, and execution time for global communication functions. Also, it reports the performance of Split-C pointer operations. It shows numbers for the CM5, the HPs, and the Paragon.

The same mechanism was used to measure all events. In order to account for the overhead of the timers, the event was repeated inside a loop and the loop was timed. The time to execute the loop without the event was also measured and these two numbers were subtracted. The result was divided by the number of iterations to get the cost of a single event. In order to account for context switches, cold-start cache misses, etc, the previous measurement was repeated several times. The first time was thrown out. The next three times were used to get an approximation to the mean time. If any of the remaining measurements were greater than three times the mean, they were assumed to be corrupted by a context switch and were thrown out. At least thirty iterations were always measured, but enough iterations were measured so that the confidence interval was at most 10% of the mean.

5.1 Global Memory

One goal of active messages is to provide a mechanism for sending messages with the lowest overhead and latency possible. Libsplit-C provides portable global memory primitives but at the cost of some overhead. In the process of designing and implementing Libsplit-C, we tried to minimize the overhead that is caused by adding another layer on top of active messages. This section shows that the extra overhead that Libsplit-C adds to active messages is small.

The comparison between active messages and Libsplit-C was made by comparing three different measurements of active messages with the five Libsplit-C global memory operations. A read or a write operation spends time in the message layer that is equivalent to the round trip time of an active message. A get or a put operation only pays the overhead of sending and receiving a message. The latency through the network is masked by pipelining several operations. A store operation only pays the overhead of sending a message because there is no acknowledgment.

The round trip time of an active message was measured by sending a message to a remote node and then waiting for a reply. The send and receive overhead was measured by sending N messages to a remote node and then waiting for N responses to be received. N was chosen so that the overhead of sending and receiving messages would be greater than the network latency. The send overhead was measured by sending a message and not waiting for a response.

5.1.1 Overhead and Latency on the CM5 and the HPs

Figure 8 shows the cost of the five Libsplit-C global memory operations (e.g. store, get, put, read, and write) for the implementations that are built on CMAM, CMAML, and HPAM. Also included is the overhead of sending an active message (AM send overhead), the overhead of sending and receiving an active message (AM send+recv overhead), and the round trip latency for an active message (AM latency).

For CMAM and CMAML, the cost of a get or a put is approximately twice the cost of a store. This makes sense if the cost of sending an active message is about the same as the cost of receiving one. On HPAM, however, the cost of a get or put is closer to five times the cost of a store. This is because there is a bug in the Medusa card that the HPs use for their network interface. This bug causes several status packets to be sent whenever a message is sent. The sender has to receive these status packets and this overhead makes the send and receive overhead greater than twice the send overhead.

Note that the send overhead on HPAM is slightly misleading. On the HPs, packets are not reliable so every send must reply. This means that a program that executes a store will eventually have to pay the cost of receiving the reply and the extra status packets. AM send overhead is actually equal to the AM send+recv overhead, but in figure 8 it is reported as only the cost of the send.

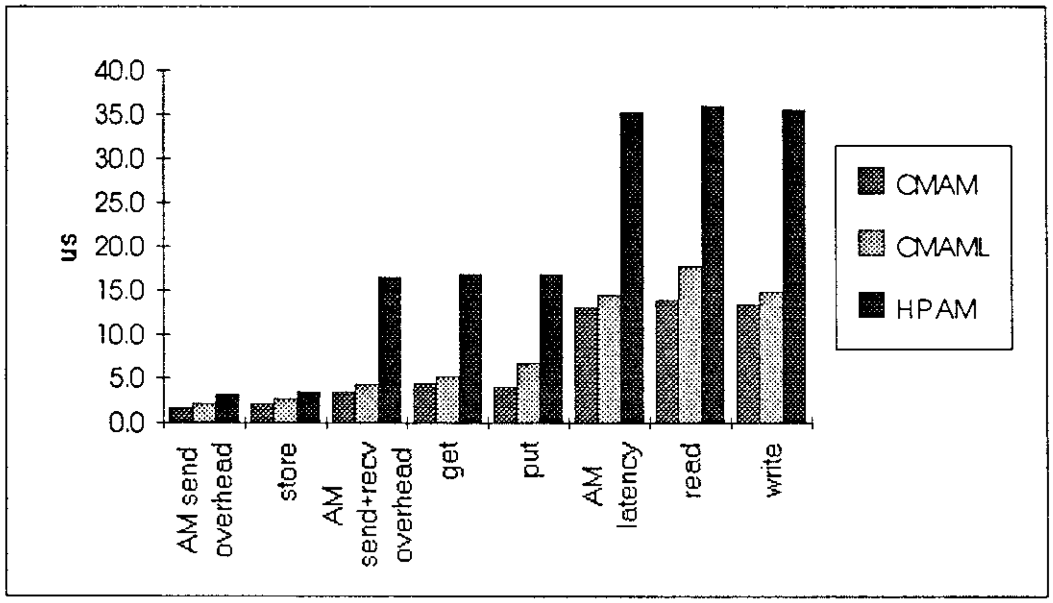

FIGURE 8. Overhead and latency of active messages and Libsplit-C on CMAM, CMAML, and HPAM.

5.1.2 Overhead and Latency on the Paragon

For the Paragon, there are two overheads to consider. The overhead introduced by the active message layer (NXAM) that is built on NX, and the overhead introduced by Libsplit-C. NX and NXAM send overhead are the times to send an NX message or a NXAM active message. NX and NXAM send and receive overhead are the times to send and receive an NX or NXAM message. NX and NXAM latency are the round trip times of an NX or NXAM message. These times are for OSF version 1.1.

The Paragon has a second processor on each node which can be used to send messages. The faster of the times is for the message processor turned on, and the slower is for this processor turned off. The times for the message processor turned off are reported because only recently has it become possible to turn this processor on. Some sites have not done so yet.

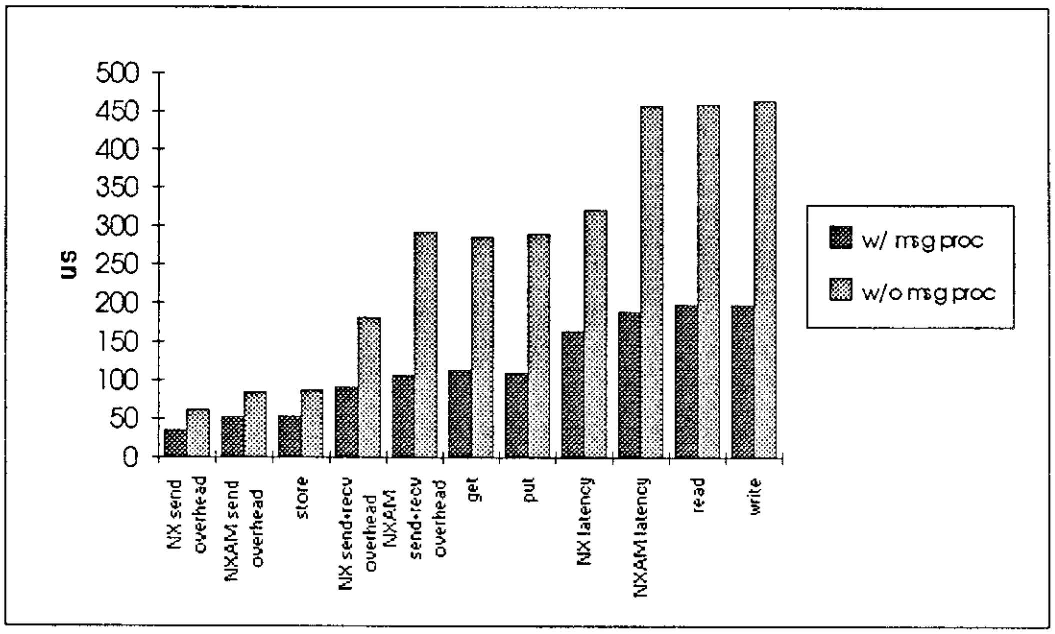

FIGURE 9. Overhead and latency of NX and Libsplit-C on the Paragon.

5.2 Bulk Transfer

In addition to low overhead and low latency, the second goal of active messages is to provide a mechanism for sending data at the highest possible bandwidth. Libsplit-C provides portable bulk transfer but at some reduction of bandwidth. This section presents the amount of bandwidth that is lost by using Libsplit-C instead of active messages.

The comparison between active messages and Libsplit-C was made by comparing the bandwidth that can be achieved for active messages (and NX) with the bandwidth of the five Libsplit-C bulk transfer operations.

Figure 10 shows the bandwidth of the four active message layers and NX. These measurements do not include the cost of opening segments for CMAM and CMAML, nor do they include the cost of posting a receive in advance on the Paragon. The CM5 reaches a peak bandwidth of about 10 MB/sec, and it gets half of peak bandwidth for a message size of about 50 bytes. The Paragon reaches a peak bandwidth of 38MB/sec with the message processor and a peak of 29 MB/sec without it. It gets half of peak for a message size of about 4KB. The HPs reach a peak bandwidth of 39 MB/sec for transfers under 4KB and half of peak bandwidth for a message size of about 150 bytes. The bandwidth on the HPs drops considerably once the message size becomes larger than 4KB because the network interface for the HPs contains enough memory to allocate a 4KB buffer for every send. 39 MB/sec is the rate at which data can be stored into the network interface. For transfers larger than 4KB, the data rate starts to drop to the FDDI bandwidth of 12 MB/sec.

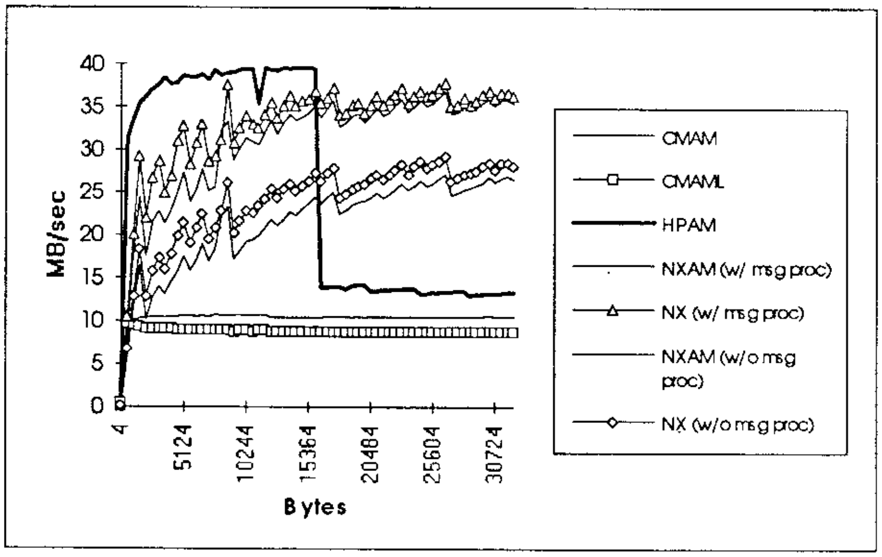

FIGURE 10. The bandwidth of four active message layers and NX.

Figures 11 through 15 show the bandwidth of the five Libsplit-C bulk transfer operations.

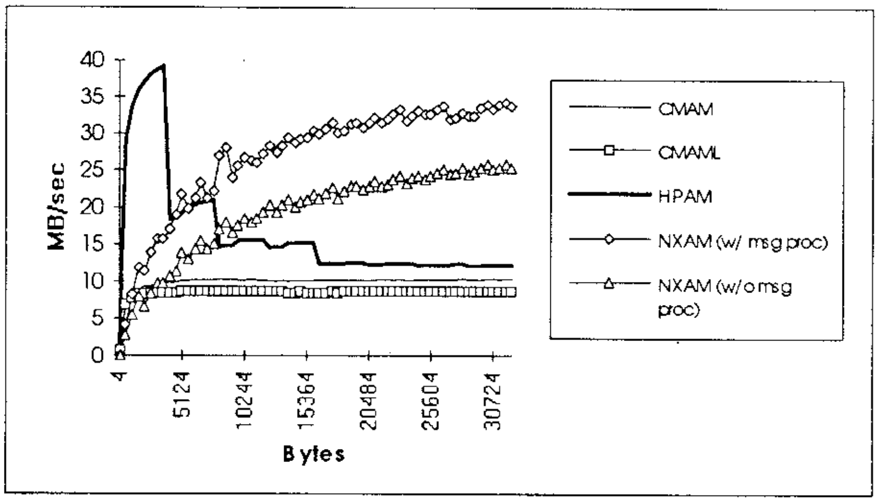

FIGURE 11. Bandwidth of bulk store.

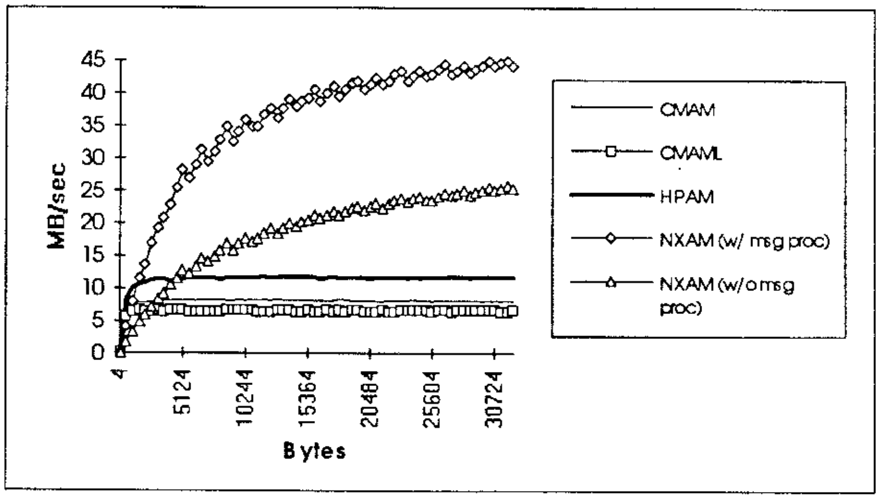

FIGURE 12. Bandwidth of bulk get.

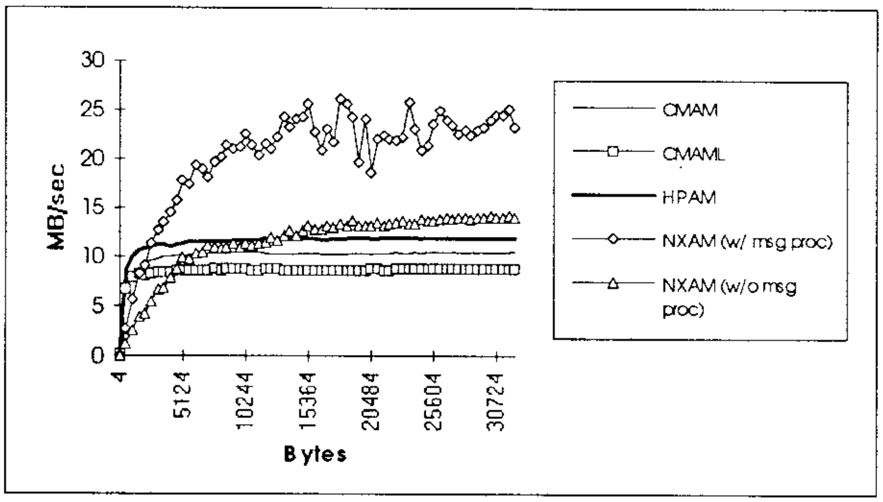

FIGURE 13. Bandwidth of bulk put.

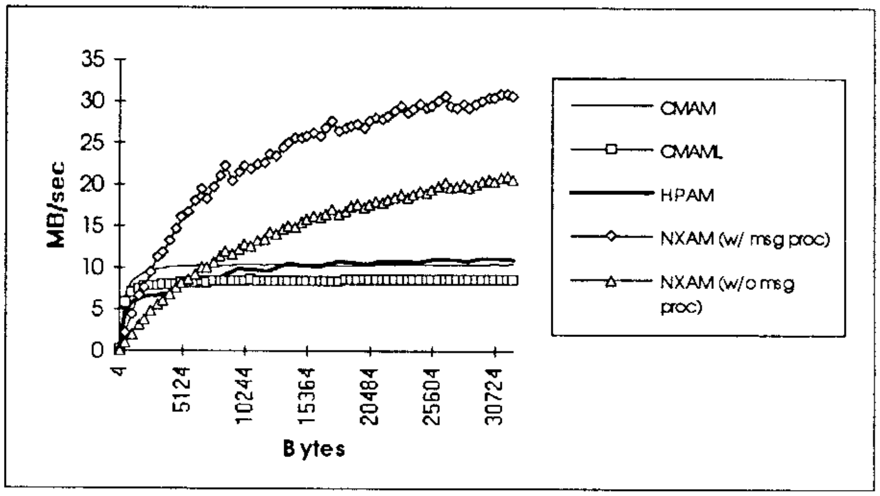

FIGURE 14. Bandwidth of bulk read.

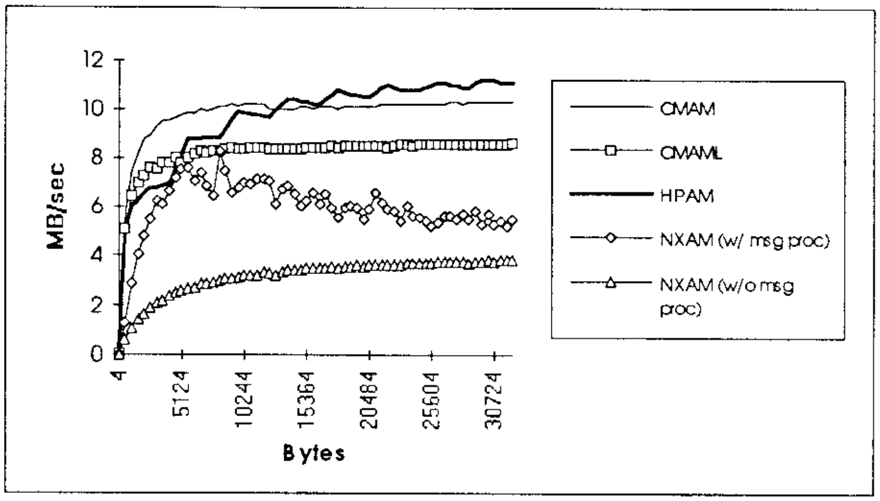

FIGURE 15. Bandwidth of write.

5.3 Global Communication

The cost of global communication is a function of the number of processors. Because we only had four HP workstations with medusa cards, we were only able to measure their performance for four processors. CM5 performance was measured on 4 nodes to compare with the HPs, and on 64 nodes because a 64 node CM5 was available. The performance of the Paragon differs by an order of magnitude, so it is displayed separately.

5.3.1 Global Communication on the CM5 and the HPs

For global communication functions, CMAML is the fastest because it uses the CM5's control network. Barrier on CMAM also uses this network. CMAM for other operations and HPAM are slower because they must perform each of these functions by sending messages. The times for both the CM5 and the HPs are for four processors.

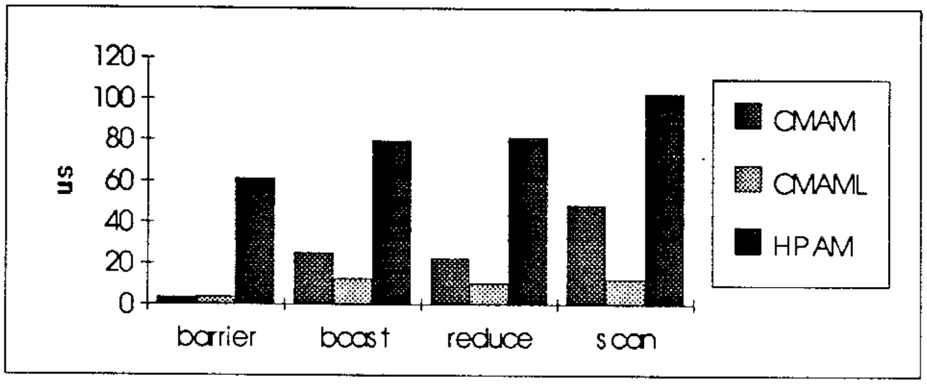

FIGURE 16. Cost of global communication for four processors on CMAM, CMAML, and HPAM.

5.3.2 Global Communication on the CM5

The times for broadcast, reduce, and scan are higher for CMAM on a 64 node machine. The times for CMAML remain about the same because CMAML uses the control network.

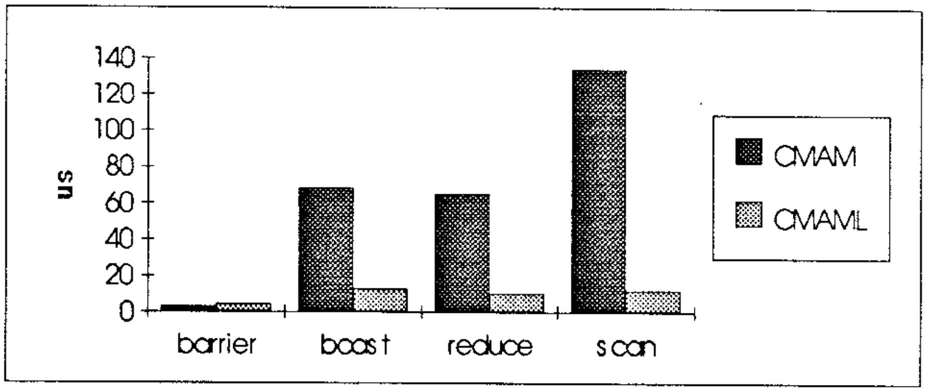

FIGURE 17. Cost of global communication for 64 processors on CMAM and CMAML.

5.3.3 Global Communication on the Paragon

Figure 18 shows two sets of times for the Paragon. Barrier is the cost of global communication using the same implementation that was used on the CM5. For this version, flow control is handled with barriers. The dataflow version uses static dataflow techniques. It avoids the cost of the barriers and it can be pipelined. Barrier could not be optimized so its cost is the same for both versions.

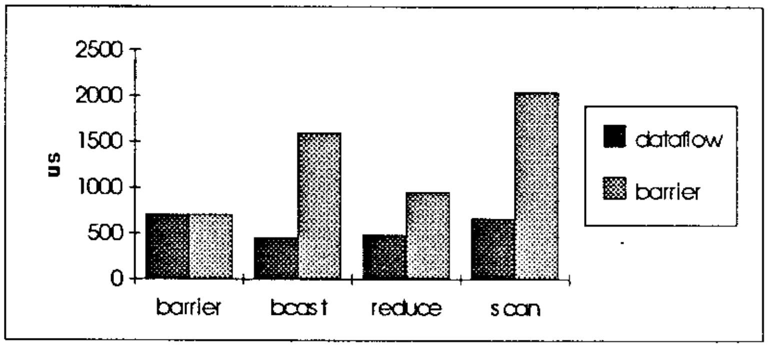

FIGURE 18. Cost of global communication for eight processors on the Paragon.

5.4 Overhead of Split-C Pointer Operations

The previous sections have shown the overhead of using Libsplit-C operations versus active messages or message passing. This section shows the overhead of several Split-C pointer operations.

Sptr[i] is the cost of indexing a spread array. For_my_1d is a macro that iterates through all the local elements of a one dimensional spread array. The cost of one iteration of the macro is shown. For_my_2D iterates through all the local elements of a two dimensional spread array, and the cost is for one iteration. Sptr++ is the cost of incrementing a spread pointer.

The cost of the Split-C pointer operations is a rough guide of the relative scalar performance of the processors on the different machines. The CM5 is built with 32 MHz Sparc2 processors. The HPs are built with 99 MHz PA RISC processors. The Paragon is built with 50 MHz i860 processors.

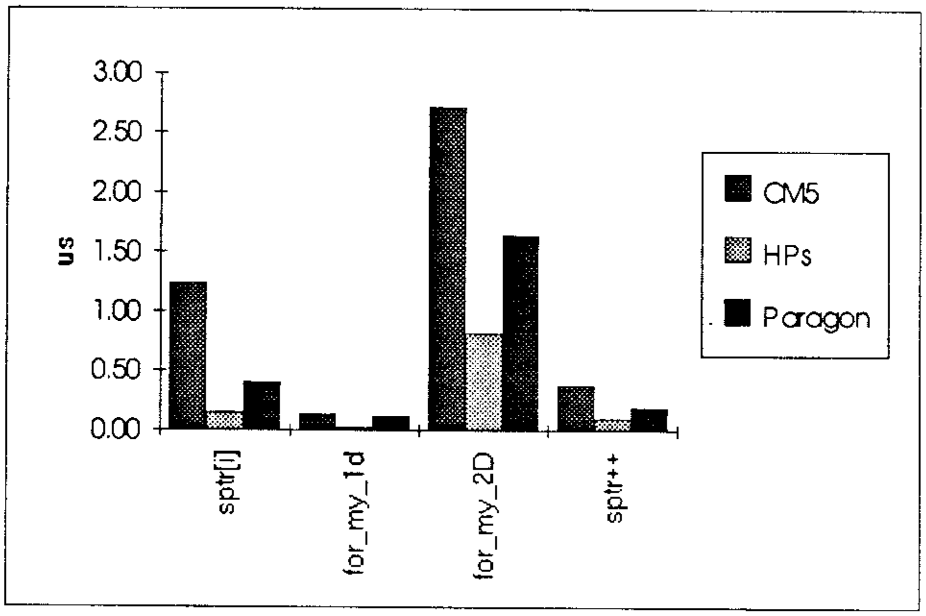

FIGURE 19. Overhead of Libsplit-C array, loop, and pointer operations.

6.0 Performance Comparison of Applications on Three Platforms

This section continues to investigate the efficiency of Libsplit-C by describing the performance of applications on the different implementations. The applications, radix sort and matrix multiply, were measured for different problem sizes on four HPs, four nodes of the CM5, and four nodes of the Paragon. For the CM5 and the Paragon, scaled speedup curves were also generated. For the HPs, we were unable to generate speedup curves because we only had four machines available. Also, because our Paragon is only 8 nodes, the SDSC Paragon was used to generate the speedup curves. The SDSC Paragon is running NX (version 1.2) and is slower than ours. The performance of global memory and bulk transfer on the SDSC Paragon is described in Appendix A. Our Paragon is running NX (version 1.1).

6.1 Matrix Multiply

Figure 20 shows the performance of matrix multiply for different problem sizes. Two square matrices were multiplied. 64x64 means that each processor contained a 64x64 element square of each matrix. The performance on the HPs drops from 87 to 29 MFLOPS when going from 128x128 to 256x256. This happens because the HPs have a 256KB cache. For small problem sizes, the matrices fit in cache.

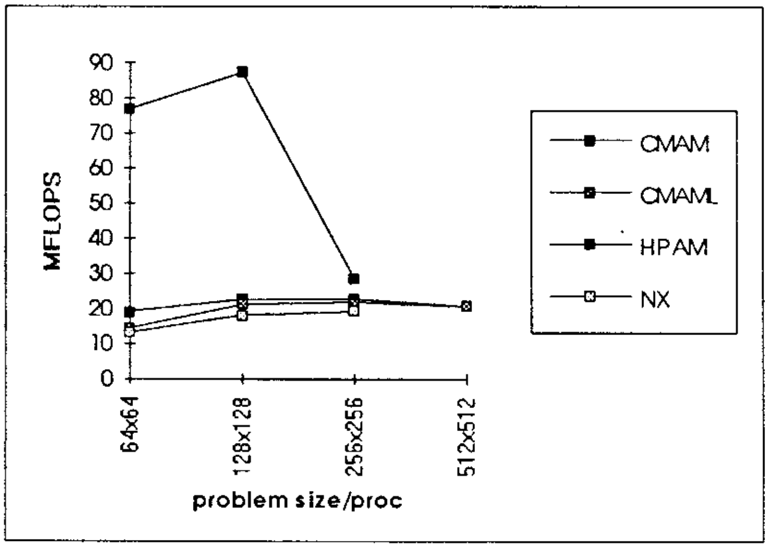

FIGURE 20. The performance of matrix multiply for different problem sizes on four processors.

TABLE 1. The performance of matrix multiply for different problem sizes (in MFLOPS).

	CMAM	CMAML	HPAM	NX
64x64	19	14	77	13
128x128	23	21	87	18
256x256	23	22	29	19
512x512	21	21		

Figure 21 shows the scaled speedup of matrix multiply on the CM5 and on the SDSC Paragon. For this problem, two square matrices were multiplied, and each processor contained a 128x128 element square of each matrix.

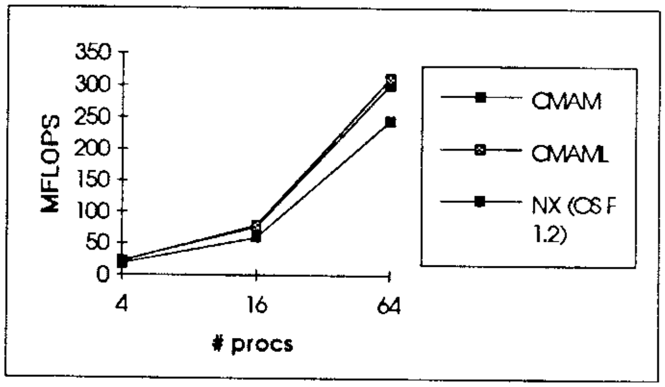

FIGURE 21. Scaled speedup of matrix multiply.

TABLE 2. Scaled speedup of matrix multiply (in MFLOPS).

# PROCS	CMAM	CMAML	NX (1.2)
4	23	21	18
16	76	79	59
64	301	311	245

6.2 Radix Sort

The radix sort operates on 32 bit integers. It has three stages: fill, scan, coalesce. Fill histograms the keys locally. Scan computes the global histograms using the global communication function, all_scan_add. Coalesce moves the keys to their correct positions. The fill time is always small. The scan time is a function of the radix and is constant with the number of keys per processor. Coalesce time is a function of the number of keys per processor and the number of passes that have to be performed.

For the CM5 and the HPs, global communication is fast enough that a 16 bit radix could be used. For a 16 bit radix, 65,536 scans must be performed for each pass, and two passes are required. Because global communication on the Paragon is much more expensive, an eight bit radix was used. For an eight bit radix, 256 scans must be performed for each pass, and four passes are required. The two extra passes increase the coalesce time on the Paragon, but this is advantageous because the global communication time is so slow.

Figures 22 through 25 show the performance of radix sort on CMAM, CMAML, HPAM, and NX. Each of these figures is for times on four processors.

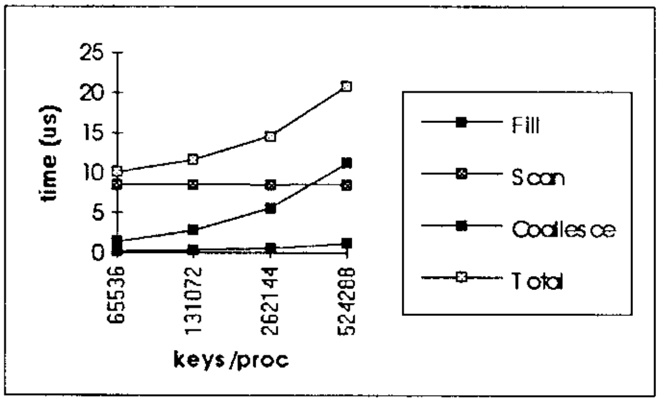

FIGURE 22. Performance of radix sort on CMAM for different problem sizes.

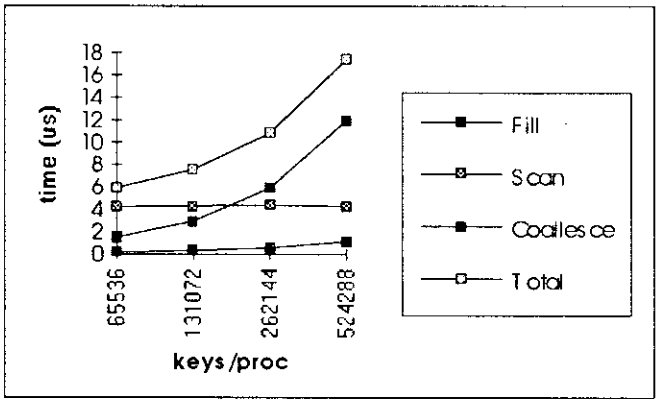

FIGURE 23. Performance of radix sort on CMAML for different problem sizes.

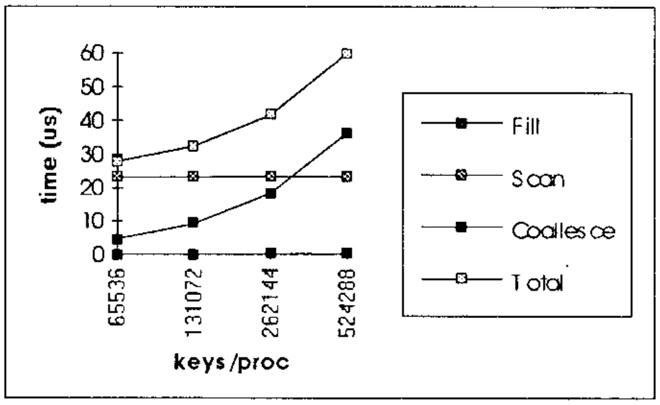

FIGURE 24. Performance of radix sort on HPAM for different problem sizes.

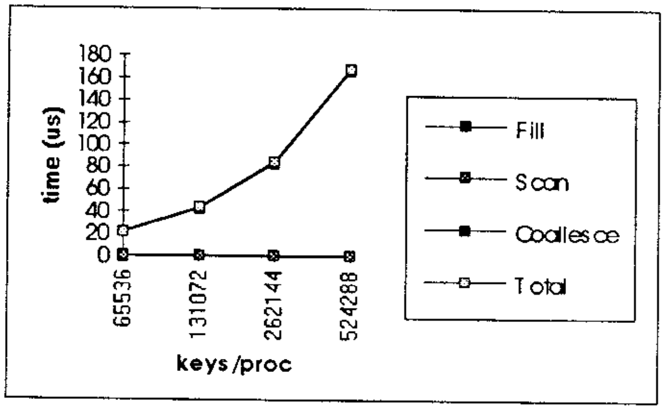

FIGURE 25. Performance of radix sort on NX for different problem sizes.

Figure 26 shows the scaled speedup of radix sort. For the times in this figure, 524,288 keys per processor were used. On the CM5, a radix of size 16 was used, and on the Paragon, a radix of size eight was used. The times for the Paragon on this figure differ from those in figure 25 because these times are for the Paragon at SDSC running NX version 1.2. For the performance of the SDSC Paragon on global memory and global communication, see appendix A.

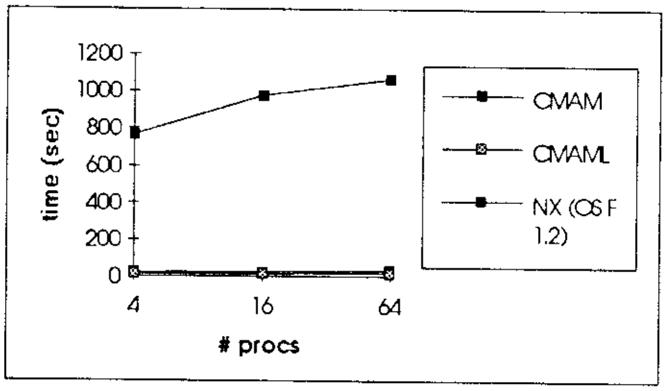

FIGURE 26. Scaled speedup of radix sort.

TABLE 3. Scaled speedup of radix sort. (Times in seconds.)

	4	**16**	**64**
CMAM	21	16	762
CMAML	26	17	977
NX (1.2)	32	17	1063

7.0 Conclusion

The experience of porting Libsplit-C to a different platforms has helped us determine how portable are global memory, bulk transfer, global communication, the global heap, and atomic procedures. It has also demonstrated that Libsplit-C can be implemented on a variety of architectures with little overhead to the message layer. Finally, Split-C can be used to write programs that are portable across distributed memory machines.

Global memory and atomic procedures were implemented easily on top of active messages. The main issue was to use the calling convention of the processor so that Libsplit-C always passes arguments in registers instead of passing them through memory. The easiest way to implement global memory on message passing was to build active messages on message passing.

For bulk transfer, a network that is both unordered and has a small maximum packet size forces the implementation to have segments. Segments cost a round trip latency for a store or a put. A network with a large maximum packet size, or an ordered and reliable network, does not need segments. In order to achieve maximum bandwidth on message passing, an additional send is required to post a receive in advance on the remote node.

For global communication functions, if the hardware doesn't provide support for them, the functions must be built from point to point messages. Implementations that are built using messages must control flow. Architectures that support barrier in hardware can control flow using barriers. Architectures that must build barrier from messages can use techniques from static dataflow. In architectures that do not provide support for a fast barrier, using static dataflow instead of barriers increases performance by 2 to 3.5 times.

Architectural support for a global heap is not required. All_spread_malloc can be implemented on multiple platforms using sbrk. However, spread_malloc is much more difficult to implement because it requires autonomous global communication.

On the CM5, Libsplit-C adds less than 25% overhead and latency to active messages. On the HPs, it adds less than 25%. On the Paragon, it adds less than 6% to the active message layer. The active message layer adds 13 to 35% overhead to the message passing layer.

The experience with matrix multiply and radix has shown us that an application that is written in Split-C can be run efficiently on multiple machines with a minimum of tuning.

8.0 Acknowledgments

I would like to thank David Culler for his guidance and advice, Kathy Yelick for her helpful comments and suggestions, and Arvind Krishnamurthy for his help with the Split-C compiler. I would like to thank Rich Martin for his help in understanding the HPAM layer, and all the others who provided me with ideas and insight.

I would like to thank the organizations that provided us with the NSF Presidential Faculty Fellowship CCR-9253705, the NSF Infrastructure Grant CDA-8727788, and the Eugene

C. Gee and Mona Fay Gee Scholarship. I would also like to thank Intel for its donation of an Intel Paragon, and IBM for its donation of an SP1. I would like to thank LANL for time on its CM5, and SDSC for time on its Paragon.

Appendix A: Performance of SDSC Paragon

Figure 27 shows the overhead and latency of NX, NXAM, and the five global memory operations on the SDSC Paragon running OSF 1.2. The methods used to gather these numbers are the same as those described in section 5.1.

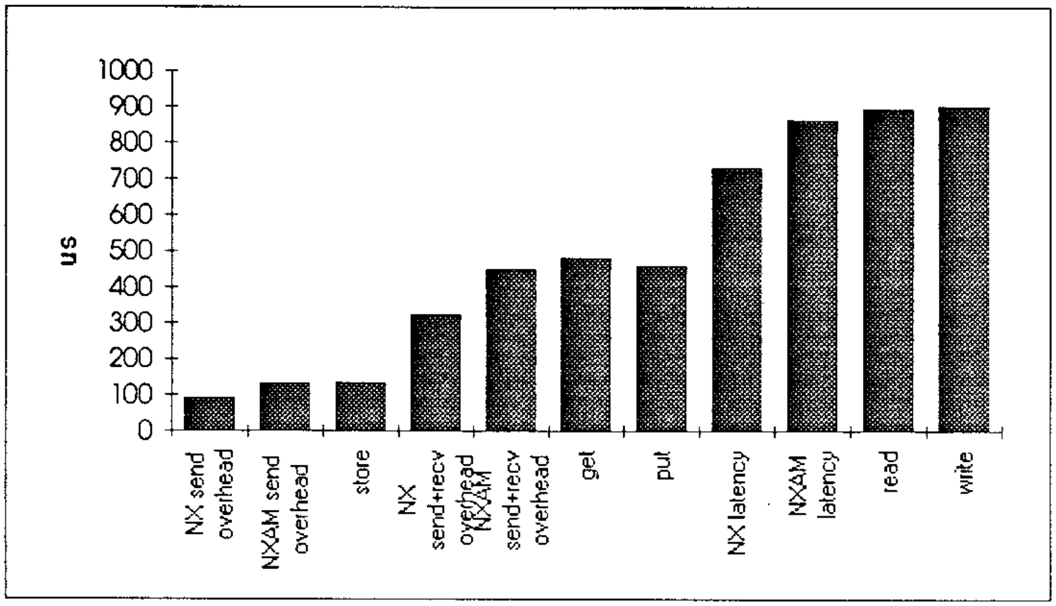

FIGURE 27. Overhead and latency of NX, NXAM, and global memory on Paragon running OSF 1.2.

Figure 28 shows the bandwidth of NX, NXAM, and the five bulk transfer operations. The methods used to gather these numbers are the same as those described in section 5.2.

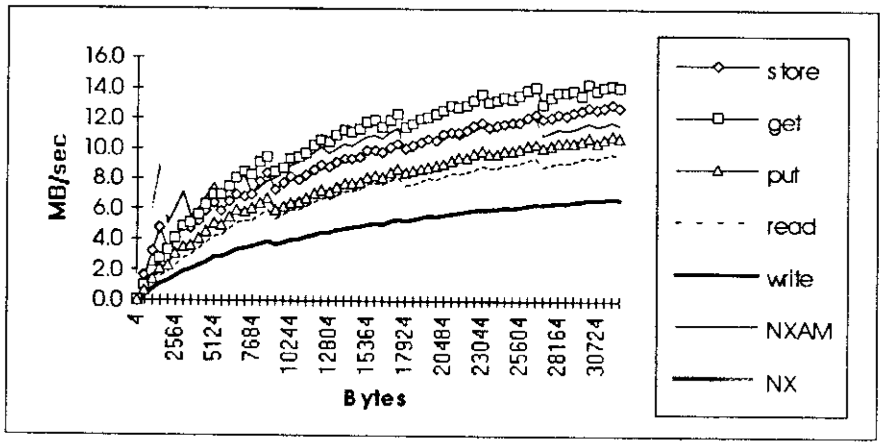

FIGURE 28. Bandwidth of NX, NXAM, and global memory on Paragon running OSF 1.2.

Appendix B: Performance of IBM SP-1

After this thesis was written, Libsplit-C was ported to the IBM SP-1. This implementation of Libsplit-C was built on the message passing layer, MPLp, and is almost identical to the implementation on NX. This port took about one day.

Here are the performance numbers for overhead and latency, bandwidth, global communication, and the applications. Latency and overhead are shown for the message passing library, MPLp, the active message layer that is built on the message library, and the five Libsplit-C global memory operations. The overhead and latency is about twice the overhead and latency for the HPs.

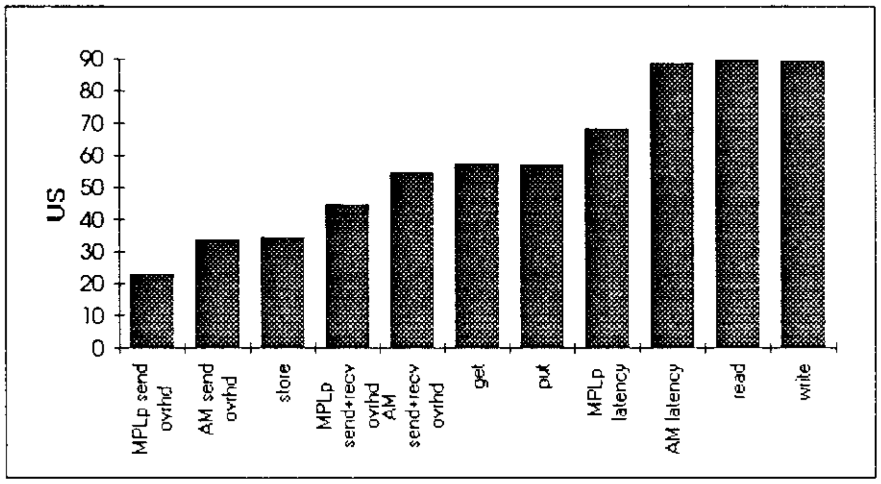

FIGURE 29. Overhead and latency on the SP-1.

Bandwidth is shown for the message library, the active message layer, and the five Libsplit-C bulk transfer operations. Two graphs are shown. The first one shows message sizes of up to 4KB. The second shows message sizes of up to 16KB. The maximum bandwidth is slightly less than the peak bandwidth achieved on the CM5.

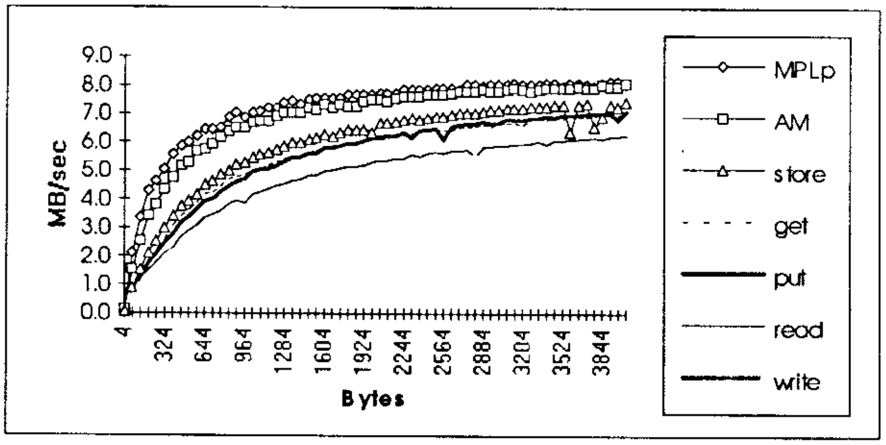

FIGURE 30. Bandwidth for message sizes up to 4KB.

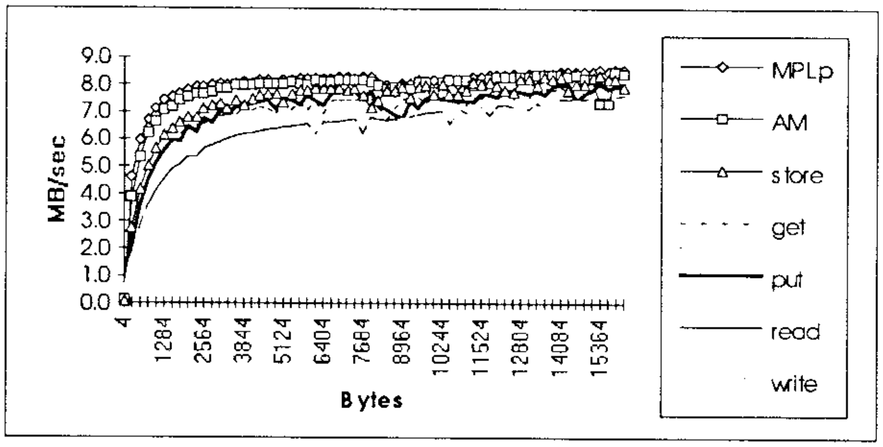

FIGURE 31. Bandwidth for message sizes up to 16KB.

Figure 32 shows the performance of global communication.

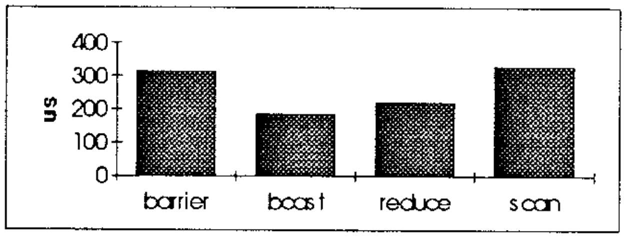

FIGURE 32. Performance of global communication.

Figures 33 and 34 show the performance of matrix multiply and radix sort for different problem sizes. For radix sort, a 16 bit radix was used. Because only an eight node SP-1 was available, speedup curves could not be generated for the SP-1.

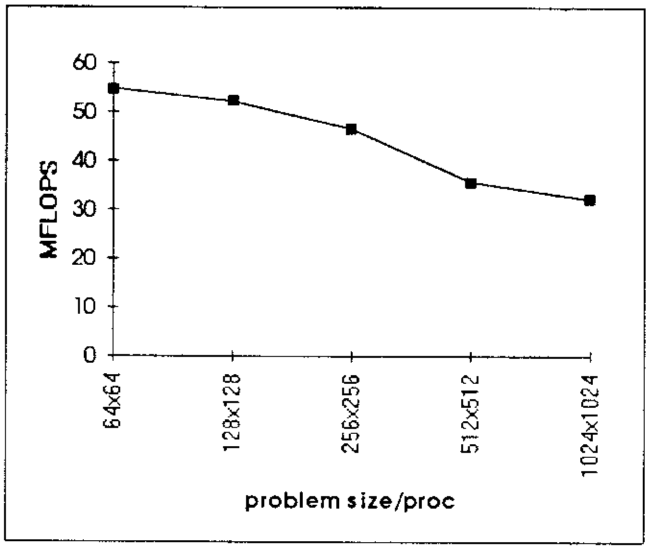

FIGURE 33. Performance of matrix multiply for different problem sizes.

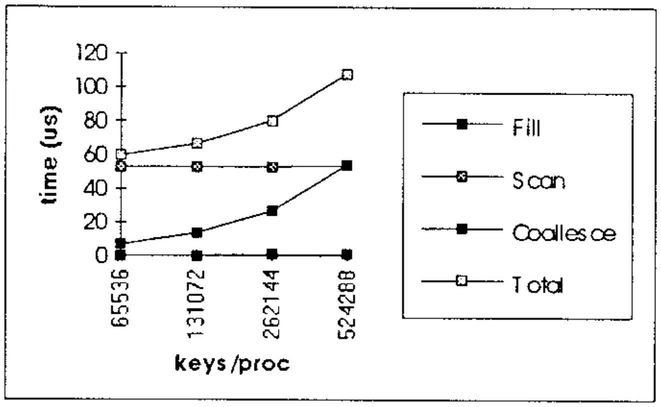

FIGURE 34. Performance of radix sort for different problem sizes.

Appendix C: Charts in Numerical Form

This appendix has tables with the numerical values of the graphs that appear in this paper.

TABLE 4. Overhead and latency of Libsplit-C on CM5 and HPs.

	CMAM	CMAML	HPAM
AM send overhead	1.6	2.1	3.2
store	2.1	2.7	3.4
AM send + recv overhead	3.4	4.3	16.5
get	4.4	5.2	16.9
put	4.1	6.8	16.9
AM latency	13.1	14.5	35.2
read	14.0	17.8	35.9
write	13.4	14.9	35.5

TABLE 5. Overhead and latency of Libsplit-C on Paragon.

	w/ msg. proc.	w/o msg. proc.
NX send overhead	35	61
NXAM send over-head	52	84
store	53	87
NX send + recv overhead	91	182
NXAM send + recv over-head	107	292
get	114	286
put	109	290
NX latency	164	321
NXAM latency	189	458

TABLE 5. Overhead and latency of Libsplit-C on Paragon.

	w/ msg. proc.	w/o msg. proc.
read	198	460
write	198	464

TABLE 6. Overhead and latency on the SP-1.

Op.	Time (us)
MPLp send overhead	23
AM send overhead	33.7
store	34.5
MPLp send + recv overhead	44.8
AM send + recv overhead	54.6
get	57.5
put	57
MPLp latency	68.3
AM latency	88.6
read	89.6
write	89.2

TABLE 7. Overhead of Libsplit-C pointer operations.

	CM5	HPs	Paragon
sptr[i]	1.24	0.15	0.41
for_my_1d	0.13	0.03	0.12
for_my_2D	2.27	0.82	1.65
sptr++	0.38	0.10	0.19

TABLE 8. Performance of global communication for four nodes on CM5 and HPs.

	CMAM	CMAML	HPAM
barrier	3.4	3.6	61.3
broadcast	25.2	12.5	79.5
reduce	22.3	10.5	81.1
scan	48.1	12.1	102.4

TABLE 9. Performance of global communication on 64 nodes of CM5.

	CMAM	CMAML
barrier	3.2	4.4
broadcast	68.3	13.0
reduce	65.3	10.4
scan	134.1	12.2

TABLE 10. Performance of global communication on eight nodes of Paragon.

	dataflow	barrier
barrier	705	705
broadcast	453	1597
reduce	486	951
scan	661	2039

TABLE 11. Performance of global communication on eight nodes of SP-1.

Op.	Time (us)
barrier	314
broadcast	187
reduce	221
scan	329

Appendix D: Instructions for Porting Libsplit-C to a New Message Layer

There are two steps to implementing Libsplit-C on a new message layer. First, you must determine how to implement the different components of Libsplit-C efficiently on the new message layer. Then, you must add a directory, and add and change certain files in Libsplit-C. The details of what files to change and add are described in the file PORTING.

For global memory, an active message layer is the most portable interface. The one thing to consider when implementing a new message layer on the active message interface is that this interface should match the calling convention of the processor. Floats should be passed in registers instead of going through memory. If you are implementing Libsplit-C on message passing, there is an active message layer, nxam, for the Paragon that is built on message passing. Another one, sp1am, exists for the SP-1.

For bulk transfer, you may or may not have to set up a segment before you send a message. If packet sizes are large, or if the network is ordered and reliable, the message layer will not need segments. Code from the CMAM implementation can be used as a starting point if segments are needed. Code from the HPAM implementation can be used as a starting point if segments are not needed. If you are implementing Libsplit-C on message passing, you will want to send a message to the remote node which posts a receive before doing a put or a store. This will save the message layer a copy. See the implementation on the Paragon or on the SP1 for an example.

For global communication, you should use hardware implementations if they exist. If barrier is the only function that exists, the others must be constructed using point to point messages. But, the barrier can be used for flow control. See the CMAM implementation for an example. If barrier is not supported in hardware, all global communication functions must be constructed using point to point links, and flow control must be down using techniques from static dataflow. See the implementation on HPAM for an example.

The global heap should be portable to any implementation that supports the Unix system call `sbrk()`. However, a problem will arise if the message layer calls `malloc()`. `All_spread_malloc()` performs a reduction across the processors in order to determine the maximum heap value across the processors. If the message layer calls malloc while this reduction is in progress, one of the nodes may call `sbrk()` in which case the value for the top of the heap may be incorrect. The implementation that is built on PVM fixes this problem by ensuring that the free pool will always have enough extra space just in case malloc is called from within `all_spread_malloc()`.

Atomic procedures are supported easily once an active message interface exists. Because the easiest way to implement global memory is to use active messages, atomic procedures should come for free.

References

[1] J. Bennett, J. B. Carter, W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. Proceedings of the Second ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming. March 1990.

[2] D. E. Culler et. al. Introduction to Split-C.

[3] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. Proceedings of the 19th International Symposium on Computer Architecture, May 1992.

[4] R. P. Martin. HPAM: An Active Messages Layer for a Network of HP Workstations.

[5] D. J. Scales, M. S. Lam. A Flexible Shared Memory System for Distributed Memory Machines.