Software Cache Coherent Shared Memory under Split-C

CS258, Spring 1999
Final Report

Ronny Krashinsky (ronny@cs.berkeley.edu)
Erik Machnicki (machnick@cs.berkeley.edu)

Abstract

Split-C is a single program, multiple data (SPMD) programming language. Split-C provides the ability to use a shared address space abstraction on a distributed shared memory (DSM) system, such as a network of workstations (NOW). The existing Split-C compiler translates all references to shared variables owned by another processor into network transactions. We implement SWCC-Split-C, a modified Split-C compiler which automatically caches remote accesses to avoid redundant network transactions. Our cache coherence system is implemented entirely in software. It uses a directory system and a simple MSI coherence protocol to provide cache coherence with blocks of fine granularity. We demonstrate significant improvements for programs with redundant remote memory accesses, and programs with spatial locality in remote accesses. Our SWCC-Split-C compiler is ideally suited to irregular applications with access patterns which are difficult to predict.

1 Introduction

As application and data demands increase, parallel computing is becoming increasingly pervasive. However, Despite the growing demand for parallel applications, programming parallel computers remains quite difficult and error prone.

There are two main paradigms for programming parallel systems, message-passing and shared address space. Message passing paradigms, such as MPI, provide the advantage of portability. They also give the application programmer more control in optimizing communication patterns, since all communication operations are explicit calls to the message-passing library.

Shared address space programming is conceptually simpler, and hides much of the detail of communication from the programmer. However, for just this reason, it is difficult for the programmer to examine and optimize the communication patterns. It is thus very important for the system to provide an efficient execution of the program, either through compiler or runtime system optimizations.

One approach to providing an efficient shared address space system has been to build shared memory multiprocessors. Usually bus or directory based, these systems use caching to provide efficient access to memory. All addresses are treated as shared and hardware protocols are run to keep the cache contents coherent.

However, this approach to parallel computing requires large investments in expensive SMPs. An alternative approach is to link together commodity workstations in a network of workstations (NOW). A NOW is a natural fit for parallel computing using message passing. To support a shared address space programming model on a system with distributed memory, some software layer must provide the illusion of a shared address space.

There are two main approaches to providing a shared address space in a distributed memory environment. The first approach is to simulate the hardware solution, caching remote values and using some sort of directory protocol to maintain coherence across processors. This approach has been implemented in several systems such as Shasta [11] and Tempest [9].

The Split-C [3,4] programming environment takes a different approach, and provides no caching. Each variable is identified as local or global by the programmer, and global variables consist of a processor number and a local address on that processor. Accesses to global variables in the user program are translated by the Split-C compiler into function calls to the Split-C library. At run time, the Split-C library function checks if the global variable is local to the processor; if so a simple memory access is performed. If the global access is to a remote memory location, a message is sent to the owning processor to complete the transaction. This makes the Split-C system much simpler than one with automatic and coherent replication. Unfortunately, it places a much greater demand on the application programmer to provide efficient data distribution and access.

It is unclear which type of software distributed shared memory (DSM) system is more appropriate. If redundant remote accesses are uncommon, or if the programmer can easily "cache" the data within the application, the extra overhead associated with maintaining caches and directories to provide coherency may outweigh the advantage of caching the remote accesses.

To examine this issue, we modified Split-C to provide coherent caching of remote data. We implement a directory structure that tracks the usage of blocks of memory and maintains coherency. We then compare the performance of regular Split-C with SWCC-Split-C, our Software Cache Coherent version of Split-C.

The next section discusses the overall framework and design of our shared memory system. Section 3 describes the applications we used to measure the performance of the various Split-C implementations and gives the results of our experiments. In section 4, we further examine the results in relation to other systems and suggest future areas for exploration.

2 Design

The Split-C compiler translates all reference to global variables into function calls to the Split-C library. In implementing a software cache coherent Split-C system, we modified the Split-C library functions, but did not change the interface to these functions. Thus our SWCC-Split-C library works with any compiler which targets the Split-C library interface.

2.0 Notation

When servicing an access to a global variable, there are three types of nodes which may be involved; *local node*, *home node*, and *remote node*. The relationship between these nodes is pictured in Figure 1 below. The local node is the node on which the user program has requested access to the variable. The home node is the node which owns the data; the local node may send a request for the data to the home node. A remote node is one which has copies of the variable which the local node is requesting access to. The home node may send various requests to one or more remote nodes in the process of servicing a request from the local node. Note that although these nodes are separated logically, it is possible for a node to serve more than one role in a given transaction.

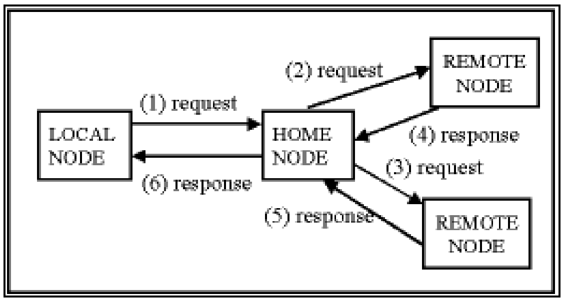

Figure 1: Nodes Involved in a Shared Memory Transaction

2.1 Address Blocks

Split-C provides a two dimensional shared address space constructed from the local memory of each processor. A *global address* consists of a *processor number* and a *local address*. The local address part of the global address is the virtual memory address of the variable, as seen by the owning processor. For example, a global address of 8:0x0012 points to local memory location 0x0012 on processor 8.

To implement caching, we divide the address space into blocks. The size of the blocks (*block size*) is a constant which is set when building the SWCC-Split-C library. It would be possible to allow the user to define the block size when compiling a program, but this would impact the performance of the library functions since the block size would not be a constant when building the SWCC-Split-C library.

Coherence is maintained at the block level. When the user program accesses a global variable, the lower bits of the address are dropped to determine the *block address*. All addresses associated with the directory structure and coherence protocols are block addresses.

2.2 Directory Structure

To maintain coherence in the software caches, we employ a directory structure to manage the state of each shared block. Each cache block has a home node, which is the processor that is responsible for servicing requests for that block. The processor number of a global address gives the home node for the block containing that global address.

The directory structure consists of a *directory hash table* of pointers to *directory entries*, as shown in Figure 2 below. The hash table is a two dimensional array indexed by processor number and the lower bits of block address. A lookup consists of finding the directory entry pointed to by the hash table and checking that the full block address matches that entry. If not, the correct entry is found by following a linked list of directory entries starting with the one pointed to by the hash table. The hash table has one row per processor, but the number of block addr bits used to index into the table is chosen when building the SWCC-Split-C library. This could also be chosen by the programmer, but isn't for the same reason that block size is defined when building the library.

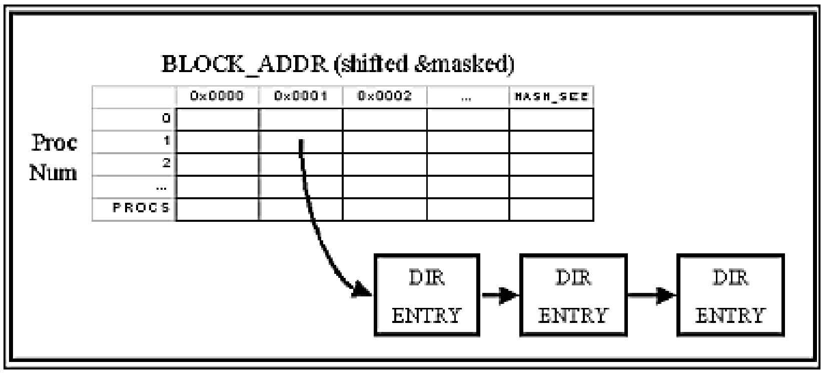

Figure 2: Directory Hash Table Structure

Each directory entry consists of:

- *block addr*
- *state*
- *data* (array of block size bytes)
- *next* (pointer to next directory entry in linked list)
- *user vector* (used and maintained only by the home node)

The possible values for the state of a block are:

- *invalid*:
  The block is not cached at this node. The data is not current.
- *shared*:
  One or more processors has a read-only copy of the block. The data is current and can be read from (but not written).
- *modified*:
  - If the processor is not the home node:
    The processor has exclusive access to the block. The data is current and can be read or written.
  - If the processor is the home node:
    The user vector indicates which node has exclusive access to the block
    (this may or may not be the home node).
- *read-busy*:
  The processor is the home node, and there is currently an outstanding read request.
- *write-busy*:
  The processor is the home node, and there is currently an outstanding write request.

A directory entry is created for every shared block which a program accesses. Also, the home node creates a directory entry for a block when it is accessed by another node. Unlike many hardware or hardware/software directory schemes [1,5,9], our implementation maintains directory entries for both remote and local data; this is because we have no hardware or operating system support for detecting access violations. The only difference between a directory entry for a local block and one for a remote block is that the user vector is maintained only for local blocks.

When a directory entry is created on the home node, the data is copied from local memory into the directory entry data field. After this, all accesses to global variables in that directory entry must use the directory entry and not the home node's local memory to maintain coherence. Once a directory entry is created the data is never written back to the local memory of the home node. This is to ensure that private (non-global) data of the home node which may happen to reside in the directory entry is not corrupted. An alternative could be to use the local memory of the home node instead of making a copy in the directory entry and also maintain a byte mask to track which bytes in the directory entry are actually global [8]. Another alternative could be to partition the address space into private and shared sections [2,10]; in this case there is no need to make a copy of the data in the directory entry or to keep track of the byte mask.

Note that it is possible to discard a directory entry which is in the shared or invalid state if the node is not the home node. We have a flag which determines whether or not to discard directory entries as they become invalidated, but in our current implementation we do not discard the entries.

2.3 Optimizations to Lookup

Previous implementations of software cache coherence have found that the overhead of access control checks can be extremely important, and must be optimized for an access hit[10,11]. In our implementation we try to minimize the overhead for checks which result in a hit. A standard software cache hit consists of these steps:

- Calculate the directory hash table index based on the processor number and local address
- Load the address of the directory entry
- Load the block address field of the directory entry
- Check that it matches the block address of the global variable
- Load the state of the directory entry
- Check the state of the entry
- Perform the memory access

In the standard Split-C implementation, accesses to global variables for which the local node is also the home node require very little overhead; after a simple check, the memory access can proceed. We are able to optimize for a subclass of these transactions. It may be the case that although a region of memory is declared global, the home node is actually the only node which ever accesses the memory. This could occur for example in the data structures associated with an "owner-computes" arrangement, in which each processor performs computations on the portion of the global data which it owns, and communicates with other processors only as necessary. If there is a separate result data structure, or if only a small portion of each processor's data is actually shared with other processors, it is very likely that the associated software cache blocks will only be accessed by the home node. Furthermore, Owner-computes parallel programming is especially common in Split-C programs due to the semantics of the language [3,4].

In order to optimize for this case, we realize that it is a waste of time and space to create and access directory entries for blocks which are only used by the home node. To avoid this, we initialize all of the home node's entries in the directory hash table to NULL. Then, an access to a global variable which is owned by the requesting node and has not been accessed by any other node consists of these steps:

- Check that the node is the home node for the global variable
- Calculate the directory hash table index

- Load the entry from the directory hash table
- Check that the entry is NULL
- Perform the memory access

Note that as soon as a memory block is accessed by another processor, the pointer in the directory hash table will no longer be NULL, and all accesses to global variables in that block must go to the directory entry even if the local node is the home node. A disadvantage of this optimization is that extra checks are performed when the data is local but the block has been accessed by another processor.

2.4 Protocols

The coherence is maintained using a three-state invalidate protocol. The simplified state diagram is given below. The state diagram has been divided into local node, home node, and remote node, although it is important to realize that a single node may serve more than one role in a given transaction. Some details involving NACK's and retries, and non-FIFO network order are not shown in the diagrams.

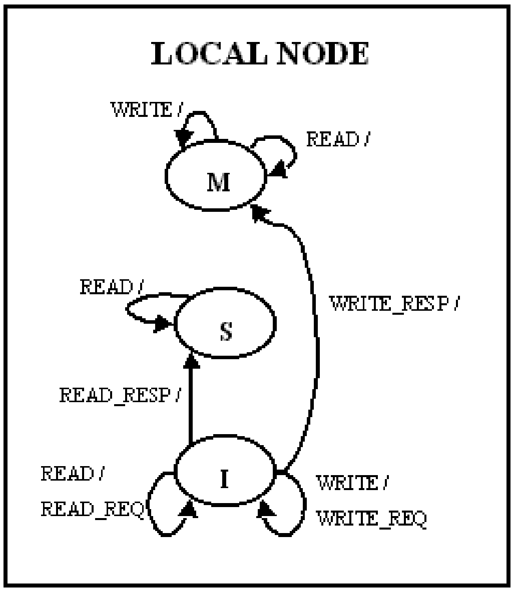

Figure 3: Local node State Transition Diagram

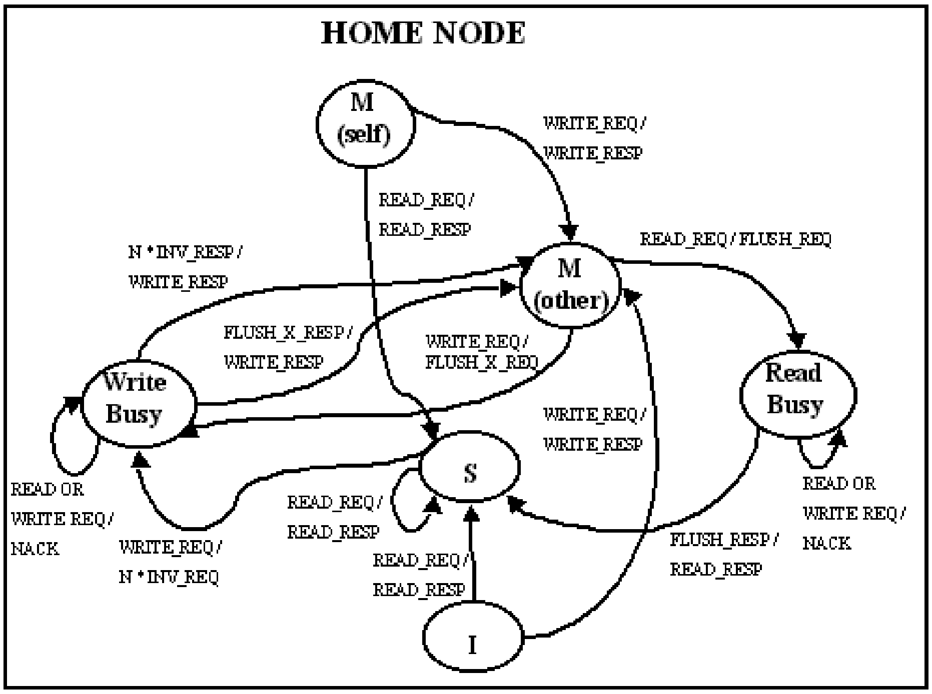

Figure 4: Home node State Transition Diagram

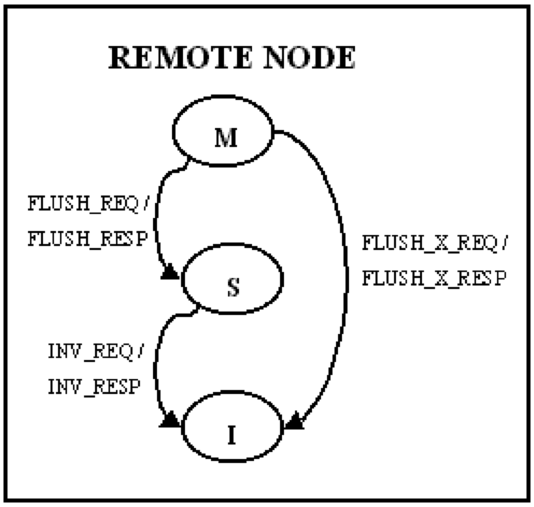

Figure 5: Remote node State Transition Diagram

Coherency is maintained using the directory and messages. When the user program running on a node attempts to read or write a global variable, the local node first checks if the corresponding directory entry is in an appropriate state to complete the transaction. If not, a request is sent to the home node. The home node does the necessary work to preserve coherency; for example it may need to send invalidate or flush requests to remote nodes. Once the home node finishes the necessary protocol transactions, the block is returned to the local node.

Our messages are implemented using Active Messages[7]. With Active Messages (AM), a node issues an AM call to a request handler function on a remote node which in turn processes the request and issues an AM call to a reply handler function on the requesting processor. However, in the interest of simplicity and time, we did not follow the semantics of AM exactly. Currently, we issue more AM requests from within request and reply handlers. The specification of AM says that in order to insure that the system is deadlock free, this should not be done. We did not experience any known problems with our method, but for a production system, more care would have to be taken to avoid deadlock.

In preserving the coherence and consistency model of Split-C, we take an approach similar to the SGI Origin2000 system. The serialization requirement for coherence is provided by ensuring that all accesses to global variables go through the home node. Additionally, the home node uses busy states to ensure that the requests are processed in the order in which it accepts them. If it receives an access request for a block which is still being processed, the home node sends a NACK and the requesting node must retry the request. Write completion is guaranteed because the local node waits for the response from the home node before it completes the request and returns control to the user program. Write atomicity is ensured because an invalidation-based protocol is used in which the home node doesn't reply to the local node until all invalidation acknowledgments have been received from remote nodes.

Split-C also allows the programmer to use a more relaxed consistency model via the "split-phase" transactions "get" and "put" which are the asynchronous analogies to "read" and "write". For these accesses, control returns to the programmer after the request has been issued, and completion isn't guaranteed until the user program issues an explicit synchronization command. Our implementation supports this model by returning to the user program after a request has been sent to the home node, without waiting for the response. In this case, memory consistency is only guaranteed at the explicit synchronization points.

2.5 Other Design Points

Due to Split-C's split-phase transactions, it is possible for a node to attempt to access a block while a previous request is still outstanding for that block. One approach to dealing with this situation is to keep track of pending transactions to avoid sending a redundant request to the home node [10]. In the interest of simplicity, we don't do this, and instead always send a request to the home node. Usually in this case the home node will be in a busy state servicing the previous request, and will NACK the second request for the block. An optimization which we perform in this scenario is that when the local node receives the NACK, it re-checks the state of the directory entry to determine if it already magically has the block in the desired state (due to completion of the previous request). If so, it completes the memory transaction instead of sending a retry request to the home node.

Active Message handlers execute atomically with respect to the user program, but there are still possible race conditions. While a SWCC-Split-C library function is handling a request from the user program it is possible that it will be interrupted by an Active Message. This can lead to a race condition if the library function is handling a write request from the user program and it determines that the state of the directory entry is modified, but before it can update the data, an Active Message handler is invoked which changes the state of the directory entry. In order to avoid this situation, the SWCC-Split-C library function which handles write requests from the user program sets a *write lock flag* to be the block address of the global variable it will modify before it checks the state of the entry. Now, if an AM handler will change the state of a directory entry it first checks if the write lock flag matches the block address of the directory entry it will change. If so, it aborts the transaction by sending a NACK to the requesting node which will then have to retry the request.

The NOW doesn't guarantee that messages are delivered in FIFO order. This can lead to anomalous situations in which a node receives a request for a directory entry which doesn't make sense based on the state of the entry. For example, if the home node issues a write response to a node followed by a flush request, the node may receive the flush request first, and it won't have the directory entry in the modified state. In such situations, the anomaly is detected, and the node responds with a NACK so that the request will be retried.

In the situations where the local node must send a request for a block to the home node, it is possible to optimize the transaction if the local node happens to also be the home node. We don't optimize for this situation, and the node will just send a message to itself. This makes the protocol much simpler since an access by the home node is handled in the same way as an access by any other node.

Split-C provides a family of bulk memory access functions for efficiency. The standard Split-C implementation divides these bulk transactions up into medium-sized Active Messages (8192 bytes in the AM implementation we are using). In our SWCC implementation, we divide these bulk transactions up into directory block size transactions in the interest of simplicity. This design puts us at a disadvantage if block size is small compared to the medium-sized AM.

Split-C provides a special "store" operation which is an asynchronous write; the difference between a store and a put is that the requesting node isn't notified of the completion of a store operation. We take the semantics of store to mean that the home node will access the variable next, and should obtain the directory entry in a modified state, rather than the local node as with a write or a put. Thus, to handle a store, the local node issues a request to the home node which essentially tells the home node to write the data contained in the message.

Split-C provides the ability to convert freely between local and global pointers. Local pointers simply access memory with a load or store, while global pointers invoke the library functions. This functionality is used when the programmer knows that a global variable lives on a certain processor, and wishes to access it without the overhead of a library call.
We must restrict the use of conversions from global to local pointers. The local memory is in general not consistent with the data in a directory entry. In certain cases, it may be ok to use local pointers, for example if the programmer is sure that a directory entry has not been created for the block in question, but great care must be taken in doing this.

2.6 Examples

2.6.1 Example Write

Consider the code fragment below, executed on processor 3:

```
remote_ptr = toglobal(4, ADDR);
*remote_ptr = 5;
```

This will execute a write to location ADDR on processor 4.
In this case, processor 3 is the local node and processor 4 is the home node.

First processor 3 will find the directory entry via its directory hash table. If it doesn't find the directory entry it creates a new one and sets the state to invalid.
Next processor 3 sets the write lock flag to ADDR as described above, and then it checks the state of the entry.

- If the state is modified it is a SWCC hit and the data is written to the directory entry.
- If the state is shared or invalid, processor 3 will send a write request to processor 4 and wait for a response.

When processor 4 receives the write request, it finds the directory entry in its directory hash table, and checks the state.

- If the state is invalid (there is no entry), it creates a new directory entry, reads the data from its local memory into the directory block data field, sets processor 3 as the owner in the user vector, sets the state of the entry to modified, and sends a responds to processor 3 with the data.
- If the state is shared, processor 4 determines which nodes have copies of the directory entry and sends invalidate requests to these remote nodes (however, it will never send an invalidate request to itself or to processor 3). Once it has received responses for all of these requests, it sets processor 3 as the owner in the user vector, sets the state of the entry to modified, and responds to processor 3 with the data.
- If the state is modified, processor 4 determines the owner of the block based on the user vector. If it is the owner, it sets processor 3 as the owner, and responds to processor 3 with the data. Otherwise, it sends an exclusive flush request to the owner of the block. Once it receives a response, it sets processor 3 as the owner, and responds to processor 3 with the data.
- If the state is read-busy or write-busy, processor 4 sends a NACK to processor 3, and processor 3 must retry the request.

When processor 3 receives the response from processor 4, it updates the directory entry with the data sent by processor 4, then writes the data from the user program's write request to the directory entry, changes the state of the directory entry to modified, and returns to the user program.

2.6.2 Example Read

Consider the code fragment below, executed on processor 3:

```
remote_ptr = toglobal(4, ADDR);
local_var = *remote_ptr;
```

This will execute a read of location ADDR on processor 4.
In this case, processor 3 is the local node and processor 4 is the home node.

First processor 3 will find the directory entry via its directory hash table. If it doesn't find the directory entry it creates a new one and sets the state to invalid.
Next processor 3 checks the state of the entry.

- If the state is modified or shared it is a SWCC hit and the data is read from the directory entry.
- If the state is invalid, processor 3 will send a read request to processor 4 and wait for a response.

When processor 4 receives the read request, it finds the directory entry in its directory hash table, and checks the state.

- If the state is invalid (there is no entry), it creates a new directory entry, reads the data from its local memory into the directory block data field, sets processor 3 as a user in the user vector, sets the state of the entry to shared, and sends a response to processor 3 with the data.

- If the state is shared, processor 4 sets processor 3 as a user in the user vector, and sends a response to processor 3.
- If the state is modified, processor 4 determines the owner of the block based on the user vector. If it is the owner, it sets processor 3 as a user, sets the state of the entry to shared, and responds to processor 3 with the data. Otherwise, it sends a flush request to the owner of the block. Once it receives a response, it sets processor 3 as a user, and responds to processor 3 with the data.
- If the state is read-busy or write-busy, processor 4 sends a NACK to processor 3, and processor 3 must retry the request.

When processor 3 receives the response from processor 4, it updates the directory entry with the data sent by processor 4, then reads the data which was requested by the user program, changes the state of the directory entry to shared, and returns to the user program.

3 Results

To evaluate the performance of SWCC-Split-C compared to Split-C, we selected a small number of applications to perform comparisons on. Below we present micro-benchmarks, matrix multiply, and EM3D.

All tests were run on a NOW of 167 MHz UltraSPARC-I nodes.

Terms:

There are several versions of the cached Split-C implementation, which differ by block size. These are denoted by swcc_(*block size*), where *block size* is the number of bytes in a software cache block. For example, *swcc_64* refers to the version of cached Split-C using 64 byte blocks. The original, uncached version of Split-C is referred to simply as *sc*.

3.1 Micro-benchmarks

We use a variety of microbenchmarks to test read and write latencies for different scenarios. At the highest level, a read/write is classified as to a private variable (accessed using a standard load/store) or a shared variable (accessed through a call to the Split-C library). Accesses to shared variables are further classified based on whether the read/write is to a local block (i.e. the requesting node is also the home node) or a remote block (the requesting node is not the home node). The accesses are further classified by what state the block is in on the requesting node and what state the block is in at the home.

The resulting latencies are shown below in Figures 6 and 7. The microbenchmarks access 32 bit integer variables, and the SWCC block size is 8 bytes. The standard Split-C access times for local and remote variables can be compared to the SWCC-Split-C times for the various flavors of each kind of access. Note that the microbenchmarks are designed to give average access times without distinguishing between processor hardware cache hits and misses.

As expected, the software cache coherence protocols add some overhead to shared variable accesses which are local; these latencies are about 2 to 5 times that of standard Split-C. Additionally, an access to a variable which another processor has obtained in the modified state is several orders of magnitude longer because the block must be fetched with a network transaction. When a local shared variable is accessed whose block has not been accessed by any other node (the state is invalid, i.e. not present), our previously described optimizations allow us to satisfy the request in the nodes local memory after a single check in directory hash table. This has a significant performance advantage, and the access takes less than half the time of one which must go to a directory entry. In this case we also don't have to create a new directory entry the first time the variable is accessed, which can take as long as 50us.

The real win comes when a shared variable is accessed whose home is on a remote processor but that is in an appropriate state in the local directory. In this case, SWCC-Split-C satisfies the request locally, while standard Split-C must conduct a network transaction which takes two orders of magnitude longer.

When SWCC-Split-C must go to the home node to satisfy a request, the transaction takes about 50us if the home node has the block in an appropriate state to return to the local node without conducting network transactions with remote nodes. One reason this time is significantly longer than the 30us for standard Split-C is that our implementation uses medium-sized Active Messages to transfer the data block, while standard Split-C uses a short AM to send the variable. Medium-sized Active Messages are designed for send large blocks of data and optimizing for throughput, while short Active Messages optimize for latency [7].

When the home node must send invalidate or flush messages to remote nodes, the latency is about twice as long, depending on the types of messages which must be sent.

					SC time (ns)	SWCC time (ns)
READ						
	PRIVATE				~10	~10
	SHARED					
		LOCAL			110	
			(local state)			
			M (SELF)			470
			M (OTHER)			95,000
			S			350
			I (not present)			160
		REMOTE			31,000	
			(local state)			
			M			260
			S			300
			I			
				(home state)		
				M (SELF)		50,000
				M (OTHER)		100,000
				S		50,000
				I (not present)		53,000

Figure 6: Read Micro-Benchmark Results

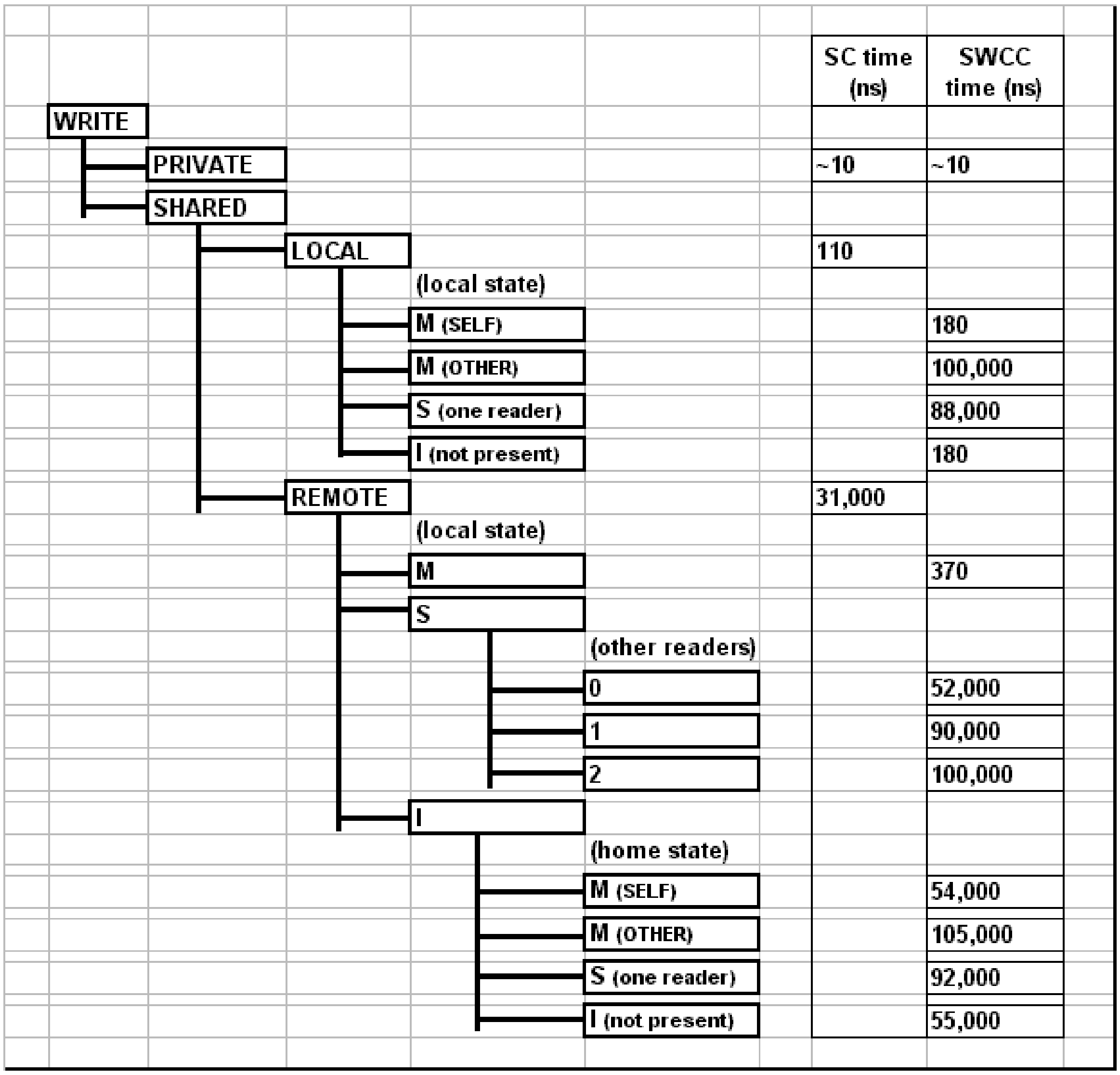

Figure 7: Write Micro-Benchmark Results

3.2 Matrix Multiply

This application does a matrix multiply: C = A x B. Each matrix is made up of 64bit doubles, and is spread across the processors; each processor is responsible for computing its own portion of the result matrix. There are 3 versions ranging from the most naive implementation to a highly optimized, blocked implementation. We show the performance advantage of software cache coherence for the naive version, and evaluate the advantage of caching as the application becomes more optimized. The varying levels of optimization are described below.

3.2.1 Versions

Naive

This is the simplest implementation of matrix multiply. It does stagger the starting point for the vector vector dot products that makes up the inner loop, but it ignores locality, both within a processor and across processors. The Split-C kernel of the naive matrix multiply is given below:

```
for_my_2D(i,j,l,n,m) {
 double sum = C[i][j];
 kk = MYPROC%r;
 for (k=0;k < r;k++) {
  sum = sum + A[i][kk]*B[kk][j];
  kk++;
  if (kk==r) kk=0;
 }
 C[i][j] = sum;
}
```

Blocked

Here, the multiply is blocked to take advantage of locality within the data. Note that it is not coded for locality across processors. The code uses a bulk split-phase read (bulk_get) to optimize the memory transaction. "matrix_mult" is a call to a local matrix multiply written in c. For the SWCC version, we read the data back into a global array after doing the local matrix multiply. The Split-C kernel of the blocked matrix multiply is given below:

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

Optimized Blocked

This version of the matrix multiply is identical to the previous version except the local matrix multiply has been hand optimized at the level of assembly code.

3.2.2 Algorithm Analysis

In a square matrix multiplication of dimension d, there are $2d^3$ floating point operations, $2d^3$ reads, and d^3 writes.

Thus, the MFLOP rate is approximately $2d^3/(2d^3*Tf + 2d^3*Trd + d^3*Twr) = 1/(Tf + Trd + Twr/2)$, where Tf, Trd, and Twr are the average times for a floating point operation, a read, and a write respectively.

The data set size for the reads is $2d^2$, so the number of redundant reads is $(2d^3 - 2d^2)$.
For a blocked matrix multiply with square blocks of size b, the number of redundant reads is reduced by a factor of b.

3.2.2 Performance Results

Scaling the Number of Processors

Figure 8 below shows the result of the naive matrix multiplication algorithm on 128x128 square matrices with a varying number of processors. The peak rate for this algorithm using a sequential c version on a single processor is about 3.5 MFLOPS. The standard Split-C version is completely dominated by remote memory accesses. The lowest line in the plot shows the expected performance for a matrix multiplication dominated by a 30us read access time, and the standard Split-C version performs only slightly better than this.

The SWCC-Split-C versions take advantage of the redundant memory references, and all perform significantly better than the standard Split-C version. With increasing block size, the performance of the SWCC versions improves due to spatial locality as expected; this effect is shown in Table 1. The maximum rate achieved by the SWCC versions is about 1 MFLOP per processor. This is significantly lower than the sequential version due to the cost of the initial remote fetch of a block, and the protocol overhead after the block is cached locally.

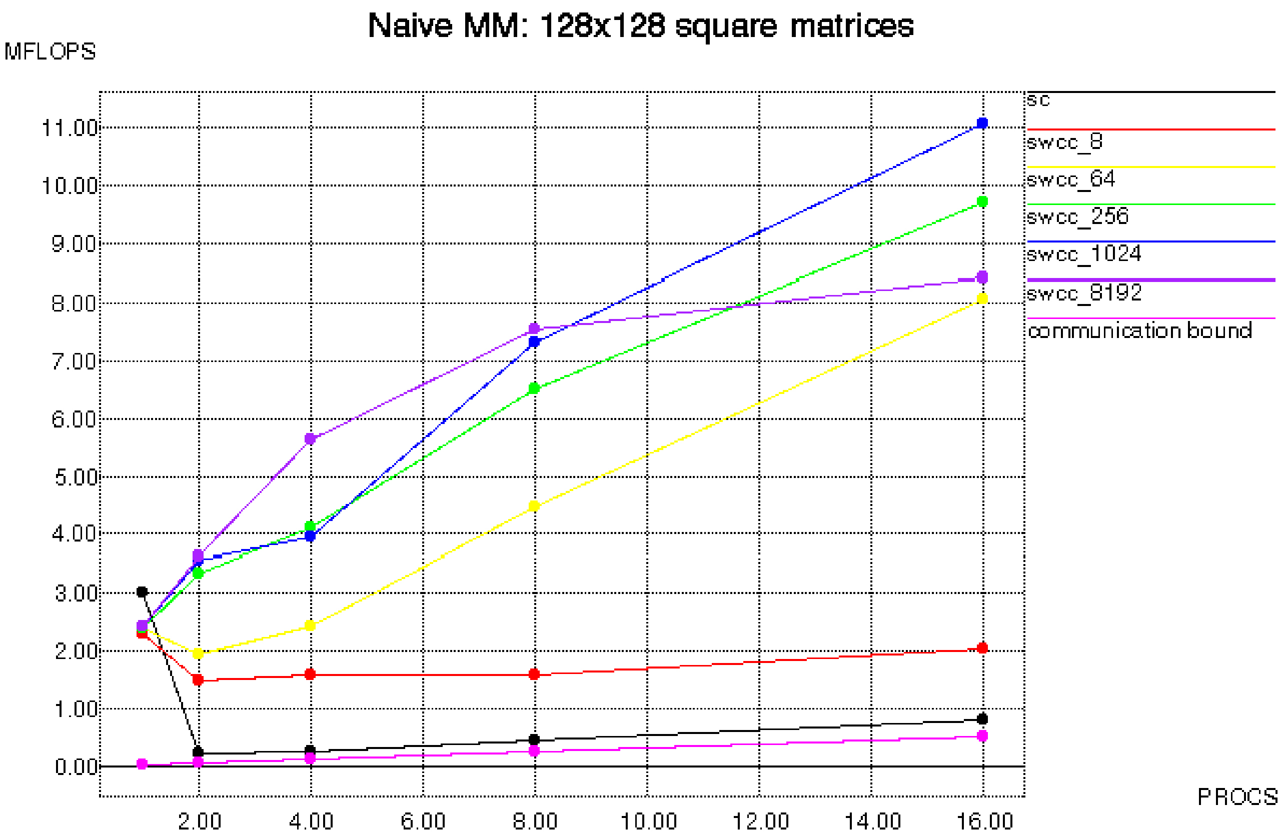

Figure 8: Naive Matrix Multiply, 128x128 square matrices, aggregate MFLOPS vs. number of processors.

	shared reads / processor	%hits	# of misses
swcc_8	229377	97.28	2048
swcc_256	229377	99.91	65
swcc_8192	229377	99.99	3

Table 1: Naive Matrix Multiply, 128x128 square matrices, 8 processors, read statistics

Scaling the Matrix Size

Next we examine scaling the matrix dimensions with a fixed number of processors. Figure 9 below shows the performance results of using 8 processors to multiply square matrices of varying dimensions.

For a naive matrix multiplication of dimension d, the proportion of the total memory reads which are redundant is proportional to d. Thus, as the matrix size is increased, the SWCC read hit percentage goes up as shown in Table 2 below. This behavior causes the local hit time to dominate the performance rather than the remote miss time. In Figure 9, the standard Split-C version achieves no performance improvement because it does not benefit from redundant memory accesses. The SWCC versions do take advantage of the redundant memory reads, and they all improve with increasing matrix dimensions.

We can calculate the predicted ideal MFLOP rate by assuming all reads and writes are SWCC hits. Based on the data from the Micro-Benchmarks, reads take approximately 350ns, and writes take about 180ns. Matrix multiply using standard Split-C on a single processor achieves a rate of 3 MFLOPS, so we estimate that floating point operations along with additional overhead cost 333ns. The ideal rate should then be 1/(Tf + Trd + Twr/2), which is 1.3 MFLOPS per processor. The SWCC versions in Figure 9 approach 1 MFLOP per processor which is a reasonable margin of error from the calculated ideal rate.

We don't know why performance drops off with the largest matrix dimension for the 64 and 256 byte block size versions. There are more directory entries needed with larger matrices; for example with a 256 byte block size, each processor creates 2056 directory entries for 256x256 matrix multiply compared to 8200 directory entries for 512x512 matrix multiply. However, the directory hash table was always big enough to have every entry pointed to directly by the hash table (instead of through a linked list). Still, with more directory entries, the entry pointers will cause more conflicts in the processors L1 hardware cache, which will impact performance. Also, each processor will have to service more block requests from other processors, which will take cycles away from the local program and likewise hurt performance. Or there might be problems with network traffic due to the increased number of messages.

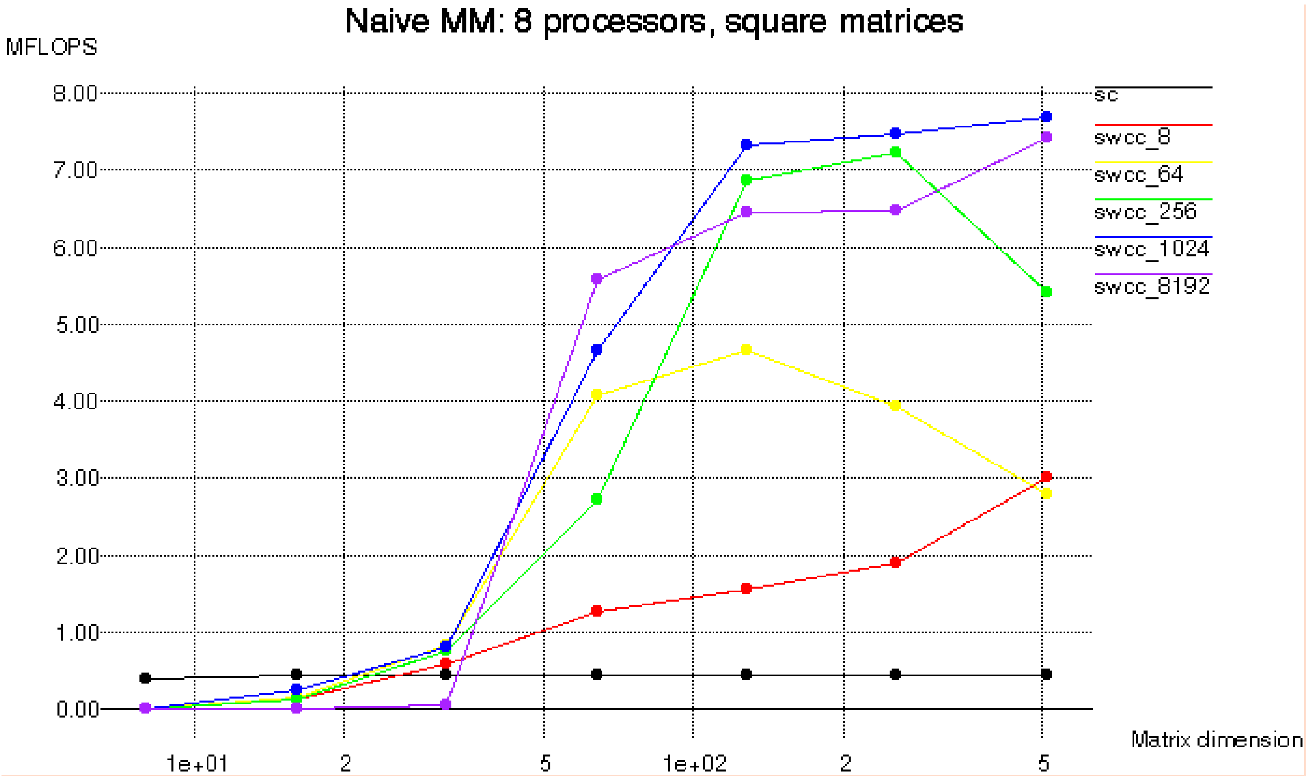

Figure 9: Naive Matrix Multiply, square matrices, 8 processors, aggregate MFLOPS vs. matrix dimension.

	shared reads / processor	**%remote**	**%hits**
8x8	136	41.18	86.03
16x16	1056	42.42	98.67
32x32	8320	43.08	99.58
64x64	66048	43.41	99.82
128x128	526336	43.58	99.91
256x256	4202496	43.66	99.96
512x512	33587200	43.71	99.98

Table 2: Naive Matrix Multiply, 8 processors, swcc_256 stats

Varying the Algorithm

Figure 10 below shows the performance of increasingly optimized various versions of matrix multiply. Note that the versions on the x-axis of the plot are arbitrary, there is no numerical order to the progression. Also note that the y-axis is a log scale.

For the blocked versions, one thing to note is that the blocks are read with a Split-C bulk_get operation. Standard Split-C divides bulk transactions into medium sized Active Messages, but our implementation of SWCC-Split-C divides them up into block size transactions. For the implementation of Active Messages which we use, the medium AM size is 8192, so swcc_8192 is the only really useful basis for comparison.

As shown before, the SWCC performs an order of magnitude better than standard Split-C for the naive version. For blocked matrix multiply with block size b*b, the redundant references are reduced by a factor of b; this affect is shown in Table 3 by the decreasing hit ratio. Due to this decrease, SWCC-Split-C has less opportunity to out-perform Split-C, although swcc_8192 does do better in several cases.

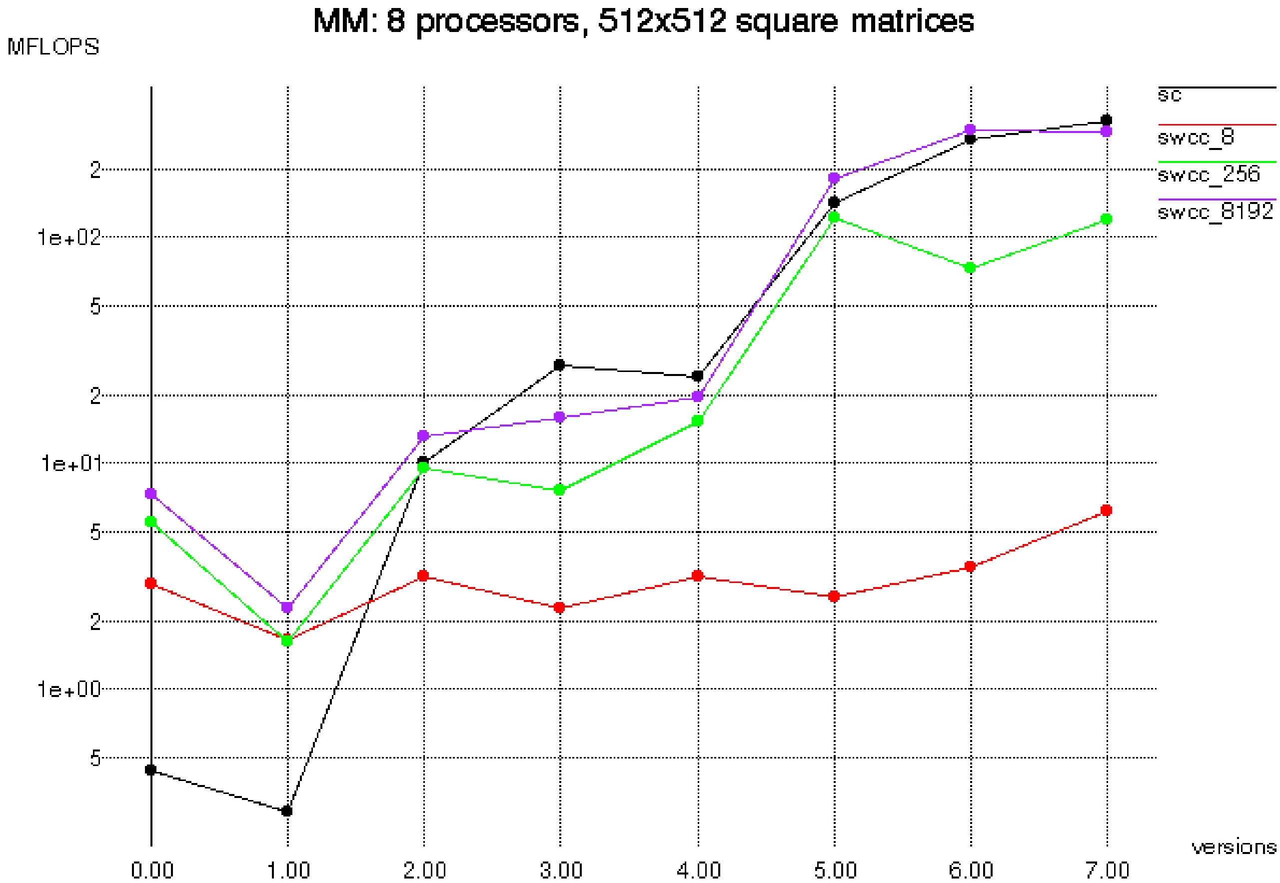

Figure 10: Matrix Multiplication, 512x512 element matrix, 8 processors, various versions:

Version 0 = naive
Version 1 = blocked, 1x1 element blocks (512x512 blocked matrix)
Version 2 = blocked, 4x4 element blocks (128x128 blocked matrix)
Version 3 = blocked, 16x16 element blocks (32x32 blocked matrix)
Version 4 = blocked, 64x64 element blocks (8x8 blocked matrix)
Version 5 = optimized blocked, 16x16 element blocks (32x32 blocked matrix)
Version 6 = optimized blocked, 64x64 element blocks (8x8 blocked matrix)

Version 7 = optimized blocked, 128x128 element blocks (4x4 blocked matrix)

	shared reads / processor	%remote	%hits
naive	33587200	43.71	99.98
blocked 1x1	33554432	43.75	99.97
blocked 4x4	786432	43.75	98.92
blocked 16x16	73728	43.75	88.45
blocked 64x64	16512	43.75	50.05
blocked 128x128	8208	62.50	31.30

Table 3: Matrix Multiplication, 512x512 matrix, 8 processors, swcc_256 statistics

3.3 EM3D

This application models the interaction of electric and magnetic fields on a 3-d object. The program used for benchmarking is actually a simplified version, using a randomly generated graph instead of an actual object to generate the fields and node dependencies. However, it is an accurate representation, as the EM3D application is in practice quite irregular. This makes it difficult for the application programmer to program for locality, possibly making a cached system more desirable.

The em3d application takes several parameters:

- *number of nodes* - number of e (electric field) nodes and number of h (magnetic field) nodes on EACH processor
- *degree of nodes* - average number of nodes that a node depends on
- *remote probability* - the probability that a given dependence will be remote (owned by another processor)
- *distance span* - the number of processors "away" that a node with remote dependencies can depend on. For example, a distance span of 3 means that processor 0's remote dependencies may depend on proc 1, 2, 3, n, n -1, and n -2.
- *number of iterations* - how many times the main loop is executed

Note that the number of nodes denotes how many nodes each processor has. Thus, the computation scales as the number of processors scales.

The application first builds a random graph based on the parameters specified. Each processor is allocated number of nodes e nodes and number of nodes h nodes. For each node, a dependence count is randomly generated, with an average dependence count of degree of nodes. For each dependence, a node to be depended on is selected. First, the remote probability parameter is used to determine if the node to be depended on is remote or local. If it is remote, the dist_span parameter is used to determine on which processor the node should depend on. A random node is then selected from that processor. Note that h nodes only depend on e nodes, and e nodes only depend on h nodes, allowing the computation to be broken into two phases.

Once the graph is created, the main loop is entered:

```
for(numIters)
{
 compute_e_nodes(...);
 barrier();
 compute_h_nodes(...);
 barrier();
}
```

Here, each processor computes the new values for the nodes it owns. To do this, the dependence list of each node is traversed. Since the nodes may depend on remote nodes, this step may cause remote reads. Note that since each processor computes only its nodes new values, there are no remote writes in the main loop.

For EM3D measurements, each configuration was run 5 times and the average time/iteration was measured. For each run, the number of iterations was 10 and the dist_span was set to 1.

It should be noted that the parameters used may not necessarily be representative of real EM3D problem sets. However, the setup is still useful as a general model of how many parallel programs may communicate.

Scaling Remote Dependency Probability

We first examined the effect that the remote probability had on the applications. Figure 11 below shows the results of an EM3D test with 8 processors, in which each processor has 500 nodes and each node has an average of 40 dependencies. We set the distance span to 1 in order to increase the number of common dependencies which nodes on a processor share.

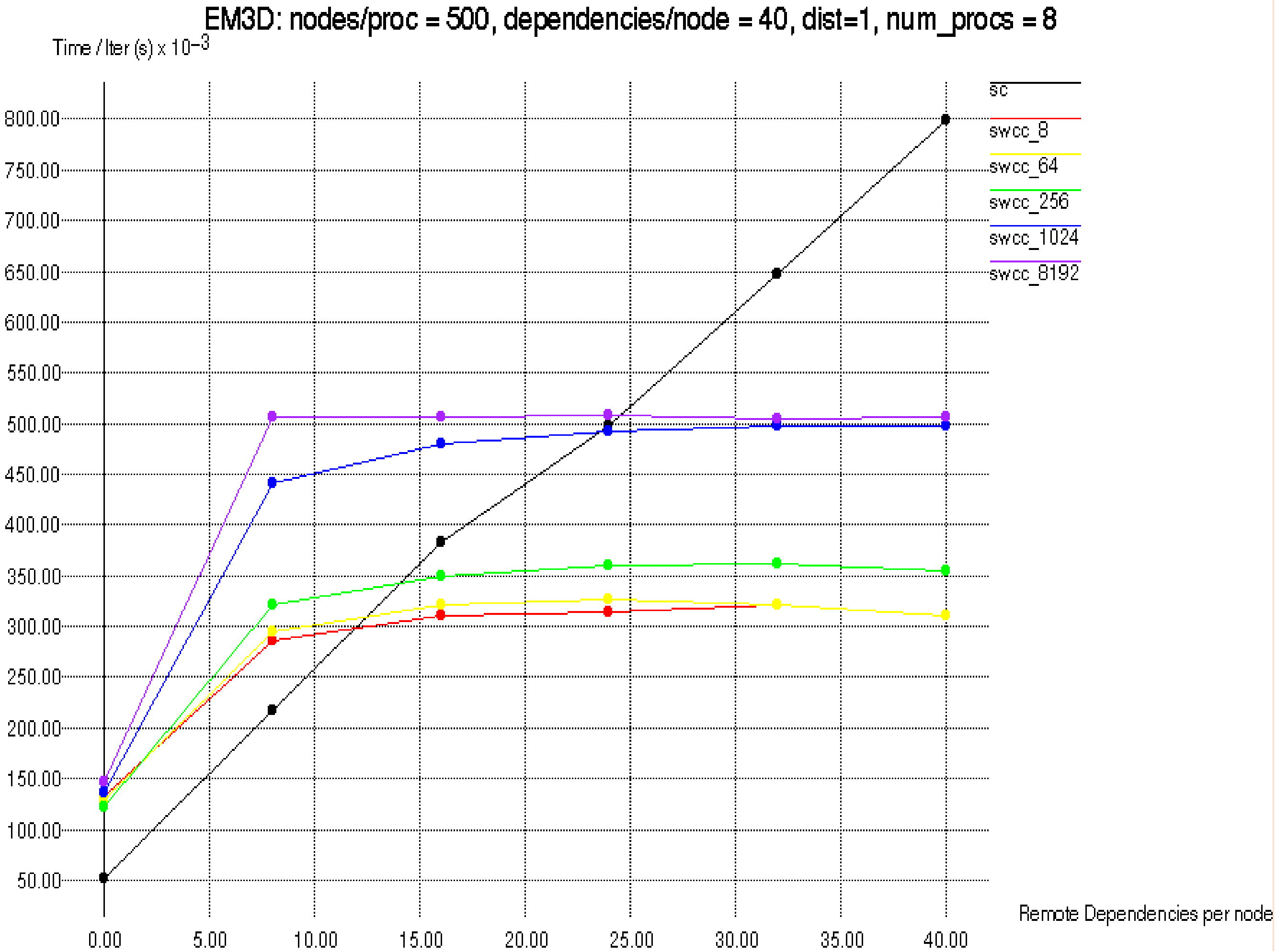

Figure 11: EM3D, Remote Dependencies vs. Time

	shared reads	**%remote**	**swcc_8 %hits**	**swcc_256 %hits**	**swcc_8192 %hits**
0	1160711	0.00	99.83	99.91	100.00
8	1160711	3.58	98.99	99.07	99.82
16	1160711	7.14	98.98	99.05	99.82
24	1160711	10.68	98.98	99.05	99.82
32	1160711	14.23	98.98	99.05	99.82
40	1160711	17.85	98.98	99.05	99.82

Table 4: EM3D, 8 procs, 500 nodes/proc, 40 dep/node, Processor 1 read statistics for various remote dependency averages

	unshared local read hit%	**unshared local write hit%**
swcc_8	31.13	15.99
swcc_64	29.70	15.26
swcc_256	23.26	11.95
swcc_1024	11.36	5.83
swcc_8192	0.02	0.01

Table 5: EM3D, 8 procs, 500 nodes/proc, 40 dep/node, Processor 1 Unshared Hit Statistics

We can see that the original Split-C version, sc, scales linearly with the number of remote dependencies. The various SWCC versions have more overhead when there are few remote dependencies. However, once there are more than about 8 remote dependencies per node, the time/iteration is almost constant. With more than 12 remote dependencies per node, the SWCC 8, 64, and 256 versions outperform standard Split-C, as do the 1024 and 8192 versions when there are more than 24 remote dependencies per node.

As the remote probability increases, the number of remote reads that each processor does increases; this effect is demonstrated in Table 4. This is why the running time of the standard Split-C version of EM3D scales linearly with the percentage of remote dependencies.

The advantage that the SWCC versions have is that, increasing the percentage of remote dependencies also increases the probability that two or more nodes on a given processor will depend on the same remote node. This means that once the value of the remote node has been obtained, future accesses will be software cache hits rather than remote memory accesses. Since each processor has a constant number of nodes, the collection of dependencies of the nodes owned by one processor will eventually include every node owned by its neighboring processors. Once this happens, increasing the remote probability will not increase the total number of nodes depended on by the collection of nodes owned by a single processor. This means that the number of remote memory transactions per iteration remains constant, which is why the curves in Figure 11 become flat.

One disadvantage of caching is that at the end of each iteration, a processor must send out invalidate Requests before it can update its nodes which have been read by other processors. A way to avoid this problem is to send updates instead of invalidates at the end of each iteration; this option was explored with Stache [9].

The reason that the SWCC 1024 and 8192 versions perform worse than those with smaller block sizes seems to be an interesting form of false sharing. As described previously, our protocol implementation optimizes shared variable accesses which are local to a processor and for which the software cache block has not been accessed by any other processor. The frequency of this type of access in the EM3D program is shown in Table 5. As the block size increases, it becomes more likely that shared variables which are only used by the owning processor will lie within blocks that are accessed by other processors. This will increase the latency for accessing these variables, and thereby impact performance. Better data layout could potentially alleviate this problem.

Scaling Number of Processors

We next look at how the application scales as the number of processors increases. Remember that the problem size scales with the processors, so the computation per processor remains constant. As the number of processors is increased, the amount of network transactions increases. We fix the percentage of remote dependencies as 24/40, and vary the number of processors from 2 to 16, as shown in Figure 12 below.

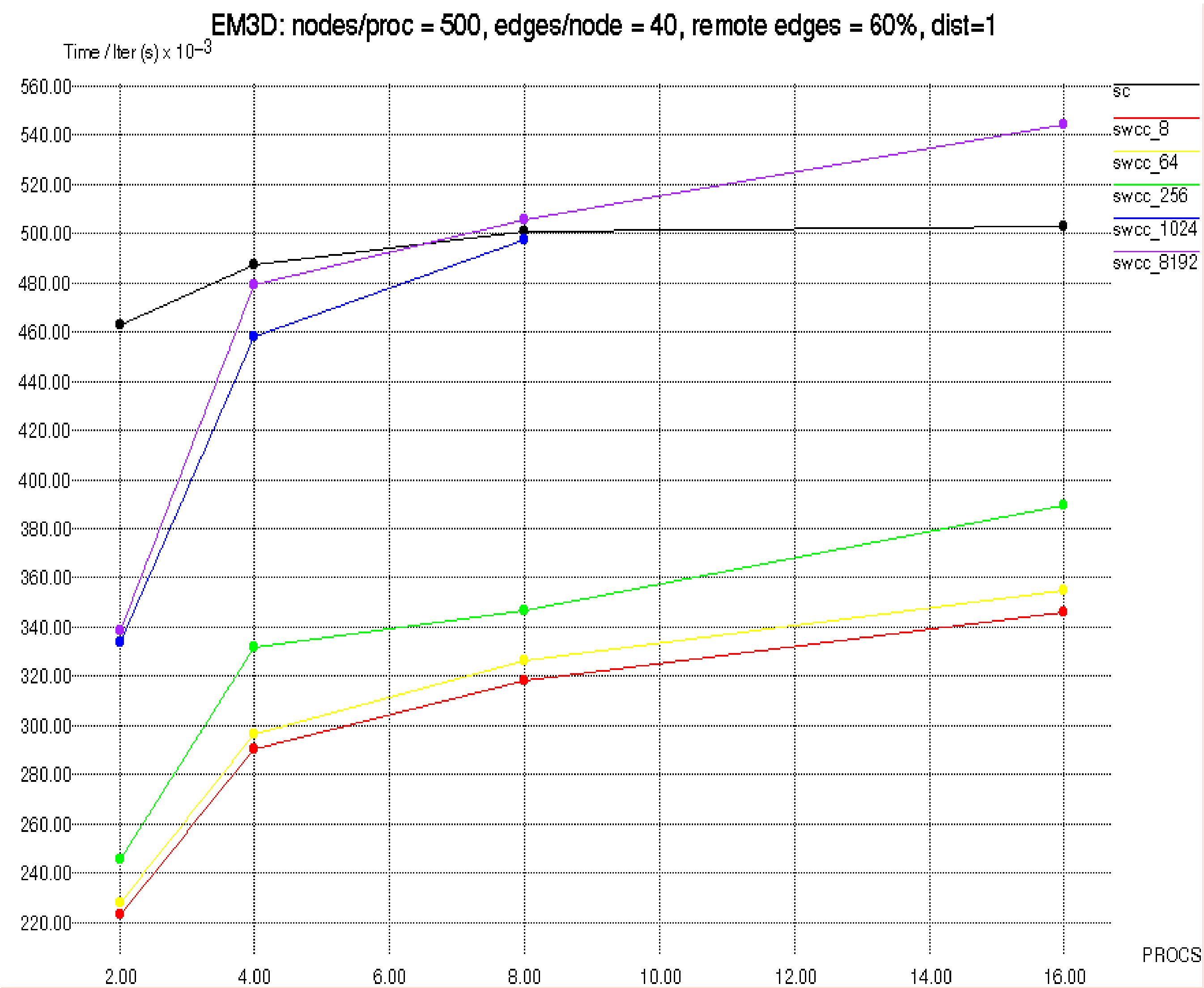

Figure 12: EM3D, Number of Processors vs. Time

The SWCC configurations with small block size outperform standard Split-C for each processor configuration tested. It is interesting that the standard Split-C implementation reaches a constant Time/Iter, while the SWCC versions continue to increase. This effect actually seems to be an artifact of our implementation. In our EM3D code, Processor 0 initializes all of the node values. This part of the application isn't timed, but it does cause Processor 0 to obtain all of the shared blocks in a modified state. Then during the first iteration of EM3D, there is contention at Processor 0 for the shared values which scales with the number of processors. Presumably, this one-time effect would diminish with more iterations of EM3D.

4 Future Work / Ideas

Although we found software cache coherence to help for several applications, it may not be useful in all cases. If the application is regular enough, the programmer may be able to handle the caching of remote values more efficiently. It would be ideal if caching could be provided as an optional feature. With our implementation, it would be simple to allow the user to choose whether or not to use caching on a per-variable basis. Global variables using automatic caching would invoke our SWCC-Split-C library calls, while others would just use the standard Split-C library calls.
Perhaps an even better possibility could be to use our SWCC as a tool for an optimizing compiler. Optimizations have been created for the Split-C compiler which will analyze a program to automatically "cache" global variables in local memory when the compiler detects repeated accesses [6]. However, the ability to evaluate the program at compile time is limited, and our software cache coherence library could be used as a fall-back when the compiler isn't able manage the caching statically. Ideally, the compiler would even determine which variables have a possibility of benefiting from software caching.

The semantics of the Split-C language provide the programmer with precise knowledge of where variables live, and which ones are local or remote. This environment limits the usefulness of software cache coherence since the programmer can easily control the data layout and access patterns. However, In many parallel programming languages, the user doesn't have such detailed knowledge, and software cache coherence becomes more important. An example is the Titanium programming language [12]. Titanium doesn't provide the programmer with the notion of local and remote as in Split-C. Additionally, there is a back-end for the Titanium compiler which targets the Split-C library, so it could be an ideal platform to further test our SWCC optimizations.

Several aspects of our MSI coherence protocol incur unnecessary network transactions and overhead. For example, if a block is being used in a producer-consumer relationship, the block will be in the shared state with one or more readers (consumers). Then the producer will obtain the block in the modified state by invalidating the consumers' copies. After the producer writes the new data, the consumers again get the block in the shared state, and the cycle repeats. A more ideal scenario would be for the producer to send out *updates* to the block, rather than invalidating the consumers copies. Another example of unnecessary network transactions occurs when a processor reads a variable, and then writes it soon after. In this scenario, the processor first obtains the block in the shared state, and then must send an additional request to upgrade it to the modified state. A more ideal sequence would be for the processor to do a *read exclusive*, obtaining the block in the modified state with the first access. It would be easy to add *update* and *read exclusive* functionality to our SWCC-Split-C library. We could either allow the user to specify which kind of transaction to perform, or let an optimizing pass of the compiler choose the best option. The Munin system [1,2] takes a slightly different approach, and allows the programmer to indicate the type of access pattern expected for each variable. This is another option to explore, or we could even attempt to determine the access pattern dynamically.

Because private variables can be mixed with shared variables in the address space provided by the

Split-C compiler, we were forced to make copies of shared blocks in the directory entries. Ideally, this could be avoided if we had a segmented address space, such as in Shasta [10,11]. Additionally, with more control over the memory layout we could better align data to benefit from the block size of our cache coherence system and to avoid false sharing. Another aspect of Shasta that a segmented address space could enable is variable coherence granularity. Shasta has the ability to choose a different block size for each page of shared data. With the ability to control memory layout, it should be possible to have variable coherence granularity with our SWCC implementation; we could further segment the address space into regions of different block size. This could potentially be a great benefit in choosing the best granularity for a data structure, and eliminating false sharing.

5 Conclusions

We have successfully implemented a software cache coherent Split-C compiler. Our SWCC-Split-C compiler uses a directory based scheme to automatically and coherently replicate blocks of shared data.

We have demonstrated significant performance improvements for programs with redundant remote accesses or spatial locality in remote accesses. For highly localized data access patterns, the overhead of maintaining the directory outweighs the advantage of reducing the number of network transactions.

The SWCC method of directory-based software cache coherence is ideally suited to irregular applications in which the access patterns are unpredictable. Caching simplifies the application programmer's job, as it alleviates the importance of initial data layout. The matrix multiply kernel shows that caching allows the simplest implementation to be efficient, reducing complexity and therefore reducing development and debugging time. Likewise with EM3D, automatic cache coherence removes the burden of caching from the programmer.

Since the protocol overhead is so critical, we may want to allow the application programmer, or preferably the compiler, to identify which blocks would benefit from caching and only use cache coherence with those blocks. This would provide the advantage of caching without incurring unnecessary overhead.

The performance gains are also highly dependent on the size of the cache blocks, and different applications perform best with different block sizes. For example, the matrix multiply kernel performs best with the largest block size while the EM3D application performs best with the smallest block size. For this reason, the granularity should be configurable, either at compile time or dynamically. Ideally, the granularity should even be variable within an application.

Parallel programming remains a challenging task. We feel that automatic cache coherence is a useful tool in lessening the burden on the parallel programmer.

6 References

[1] Carter, Bennett, Zwaenepoel. Implementation and Performance of Munin. *Operating Systems Review*, 25(5):152-64, 1991.

[2] Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and*

Distributed Computing, 29(2):219-27, 1995.

[3] Culler, Dusseau, Goldstein, Krishnamurthy, Lumetta, Luna, von Eicken, Yelick. Introduction to Split-C. 1995

[4] Culler, Dusseau, Goldstein, Krishnamurthy, Lumetta, von Eicken, Yelick. Parallel programming in Split-C. *Proceedings SUPERCOMPUTING '93*, pp 262-73, 1993.

[5] Kontothanassis, Scott. Software cache coherence for large scale multiprocessors. *Proceedings First IEEE Symposium on High-Performance Computer Architecture*, pp 286-95, 1995.

[6] Krishnamurthy, Yelick. Analyses and Optimizations for Shared Address Space Programs. *Journal of Parallel and Distributed Computing*, 38(2): 130-44, 1996.

[7] Mainwaring, Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. *UC Berkeley Computer Science Division Technical Report*, 1996.

[8] Miyamoto, Liblit. Themis: Enforcing Titanium Consistency on the NOW. *UC Berkeley, CS262 Semester Project Report*, 1997.

[9] Reinhardt, Larus, Wood. Tempest and Typhoon: User-level Shared-Memory. *Proceedings the 21st Annual International Symposium on Computer Architecture*, pp 325-36, 1994.

[10] Scales, Gharachorloo. Towards transparent and efficient software distributed shared memory. *Operating Systems Review*, 31(5):156-69, 1997.

[11] Scales, Gharachorloo, Thekkah. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. *SIGPLAN Notices*, 31(9):174-85, 1996.

[12] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, Aiken. Titanium: A High-Performance Java Dialect. *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.