Laboratory Information

During the lab portions of ENEE 759A, you will be exposed to a number of computational tools. This document provides insight on how to use these tools.

Contents

1 General Information 4
  1.1 Accessing the Necessary Software ........................................... 4
  1.2 Personal Access to Course Binaries ...................................... 5

2 Parallel C Reference Manual 6
  2.1 Introduction ................................................................. 6
  2.2 Machine Model ............................................................ 6
  2.3 Include Files ............................................................... 7
  2.4 Memory Allocation ....................................................... 7
  2.5 Distributed Arrays ....................................................... 7
  2.6 Thread Management ...................................................... 8
    2.6.1 Thread Spawning ..................................................... 8
    2.6.2 Thread Wait Queues ................................................ 9
  2.7 Synchronization ......................................................... 9
    2.7.1 Spin Locks ........................................................... 9
    2.7.2 Semaphores ......................................................... 10
    2.7.3 Condition Variables ............................................... 11
    2.7.4 Counting Semaphores ............................................. 11
    2.7.5 Barriers ............................................................. 11
    2.7.6 Reductions .......................................................... 12
  2.8 Message Passing Primitives ............................................ 13
4.2.1 Splot ......................................................... 38
4.2.2 Locality Array Utilities ................................. 38

5 Using the Network Simulator ................................. 40

6 Using the Directory Simulator ............................... 42

6.1 General Information ........................................ 42
6.2 Command-Line Interface to Directory Simulators ........ 42
  6.2.1 Examples .............................................. 43
6.3 The vsee Program ........................................... 44
  6.3.1 Example ................................................ 44
6.4 Files ........................................................ 44
1 General Information

This document contains information necessary to complete the homework assignments in ENEE 759A. Immediately following this introductory section, the document contains 4 parts:

- Section 2 describes the Parallel C programming language, which will be used for Homeworks 1, 2, and 4.
- Sections 3 and 4 describe NWO (the Alewife Simulator), and its statistics-gathering capabilities, which will be used for Homeworks 1, 2, and 4.
- Section 5 discusses use of a network simulator that will be used for Homework 4.
- Section 6 describes a cache-directory simulator that will be used for Homework 3.

This document is meant to serve as a stand-alone reference for the software tools you will be using in the course. Most of this document has been adapted from a handout prepared by Victor Lee from the MIT Alewife group. Much thanks goes to Victor for his efforts.

1.1 Accessing the Necessary Software

The programming portions of the homework assignments for the course will be performed using GLUE SUN workstations. The software for all the assignments can be found in the course directory: /software/stradivari4/class/759a. The first thing you should do is copy the environment files from the course directory into your home directory. There are four files that you will need: nwo, init.t, tea19.el, and .emacs. These files are available from /software/stradivari4/class/759a/envfiles. They are necessary to run the Parallel C compiler and the Alewife simulator, NWO. Follow the following directions:

- Create a “bin” sub-directory in your home directory (if it doesn’t already exist), and include it in your default path. Copy the file “nwo” into your “bin” sub-directory.
- Create a “t” sub-directory in your home directory (if it doesn’t already exist). Copy the file “init.t” into your “t” sub-directory.
- Create a “emacs” sub-directory in your home directory (if it doesn’t already exist). Copy the file “tea19.el” into your “emacs” sub-directory.
- If you do not already have a “.emacs” file, copy the “.emacs” file provided in the course directory into your home directory (at the topmost level). If you already have a “.emacs” file, append the contents of the “.emacs” file provided in the course directory to the end of your own “.emacs” file.
- Include the directory /software/stradivari4/class/759a/tools/sun4 into your default path.
• Add the line “setenv LD_LIBRARY_PATH /usr/local/X11R6.3/lib” in your “.cshrc” file.

Further instructions on the use of Parallel C and NWO will be provided in Sections 2 and 3.

1.2 Personal Access to Course Binaries

It is recommended that students access course software directly from the course directory, /software/stradivari4/class/759a. In some cases, students may want to access course software from a local machine (for instance to work at home or in another lab). This is not possible for Parallel C and NWO. These are extremely large pieces of software that are intimately tied to the SPARC ISA (i.e. not runnable on x86). To run Parallel C and NWO, students must use the GLUE SUN workstations.

The network simulator and the directory simulator, used in homeworks 3 and 4, are much smaller software packages and can be easily installed on other machines. You can find the network simulator in /software/stradivari4/class/759a/netwrap, and the directory simulator in /software/stradivari4/class/759a/dirsim. You are welcome to copy the “.c” source files from these directories and the “Makefile” to install the software on other machines.
2 Parallel C Reference Manual

2.1 Introduction

This section describes the library functions available to parallel programs written in C for the Alewife machine. There are two parts to the library. First is an Alewife parallel library developed specifically for the Alewife machine. Second is the shared-memory subset of the p4 library. All the assignments can be completed using the Alewife library only (and you are encouraged to do so). The p4 library is presented here for completeness.

The section first describes the Alewife-specific library. The second part will list the supported p4 functions and incompatibilities with p4, mainly having to do with the Unix-based model for forking threads. This section will conclude with several examples of code parallelized with this library.

2.2 Machine Model

Parallel C code is compiled and linked in much the same was as any C program. The extensions to C mostly take the form of support functions and a few extra syntactic keywords (such as \texttt{shared}).

Physical memory on Alewife is divided among the processing nodes. The memory on each processing node is logically divided into a local (or private) part and a shared part. The local (or private) part of memory on any given node is accessible only by the processor on that node. The shared part is accessible by any processor and is a fragment of a contiguous shared address space.

All globally declared variables in Parallel C reside by default in the local (or private) part of a node’s memory. Recall that the local part of memory is not shared between processors, so updating a globally declared variable in Parallel C on one node will not result in another node seeing the change. In contrast, variables declared with the \texttt{shared} keyword reside in the shared memory part and are accessible by all processors. To dynamically allocate data on the heap, you should use the \texttt{shmalloc()} function for shared data, and the \texttt{local malloc()} function for private data.

(Note that variables declared with the \texttt{shared} keyword should not be given initializers such as \texttt{shared int foo = 1000}; the initializer will be discarded by the Parallel C compiler.)

When a Parallel C program is loaded into the Alewife machine (for real or in the simulator), all the code and initialized data is loaded onto every processor. Initially, only processor zero is running code. If the application modifies globally declared private variables, it is possible to replicate them in the local part of the memories on the other processors for local (or private) access by other processors by calling \texttt{DISTRIBUTE DATA()}. Recall, globally declared variables in Parallel C reside by default only in the local (or private) part of a node’s memory. This function is typically called after sequential start-up code
is performed to initialize processor-local variables to useful values.

```c
void DISTRIBUTEDDATA()
{
    Copy all global variables on the local processor to all other processors.
}
```

Due to the potentially large number of global variables in the program (includes variables in all the linked library files), DISTRIBUTEDDATA can take about 100000 cycles to execute, which is a long time if you’re running on the simulator.

### 2.3 Include Files

To use the Alewife-specific parallel library, you need to include `<parallel.h>`. To use the p4 shared-memory monitor functions, you need to include `<p4.h>`. `<p4.h>` automatically includes `<parallel.h>`.

### 2.4 Memory Allocation

```c
char *shmalloc(unsigned nbytes)
{
    Allocate a memory block of size nbytes in the global address space. The memory block is allocated from the portion of global memory that is local to the calling processor.
}
```

```c
void shfree(char *ptr)
{
    Free memory block in the global address space pointed to by ptr. This function currently does not do anything.
}
```

### 2.5 Distributed Arrays

Like regular arrays, distributed arrays in Parallel C are logically contiguous in shared memory. However, distributed arrays are physically distributed across the memories of multiple nodes.

In a Parallel C program, distributed arrays have the same logical behavior as regular arrays, but are useful for large arrays than may not fit on one node, or to avoid contention on one node.

```c
void vm_init_pagemap(int n_pages, int pagesize, int nprocs, int pages_per_proc)
{
    Initialize the pagemap for a total of n_pages of memory, each of size pagesize. This needs to be called exactly once before any calls to vmAlloc(). The address space is
```
mapped across \texttt{nprocs} processors, and blocks of \texttt{pages\_per\_proc} consecutive pages are mapped to a single processor.

\texttt{nprocs} should be equal to the number of processors. \texttt{pagesize} is fixed by the compiler to be 256 bytes. (To change the page size, use \texttt{--DPAGESIZE=n}).

\texttt{char *vm\_alloc(\texttt{unsigned nbytes})}

Allocate a memory block of size \texttt{nbytes} in the global address space. The memory block is striped across the processing nodes.

Pointers to distributed arrays need to be declared as \texttt{darray}, \emph{e.g.}, \texttt{darray int pointer[]} declares \texttt{pointer} to be a pointer to a distributed array of integers. In general, \texttt{darray <pointer type>}

Restrictions: i) no pointer arithmetic on \texttt{darray} pointers. ii) arrays of structures - size of structure need to be a power of two bytes. This is to prevent a structure from spanning more than one node. Otherwise addressing fields of a structure will not work.

\section*{2.6 Thread Management}

\subsection*{2.6.1 Thread Spawning}

Currently, only producer-oriented thread spawning is supported. No thread migration is available, so threads have to be statically spawned on remote processors.

\texttt{void thread\_on(int pid, void f(), \ldots)}

Forks a thread on processor \texttt{pid} that calls the function \texttt{f} with the arguments in the rest of the argument list. Arguments are evaluated on the local processor and passed by value to the forked thread. Returns immediately without waiting for the forked thread to complete.

The following functions spawn threads across the entire machine and are useful for SPMD style computations.

\texttt{void do\_in\_parallel(void f(), \ldots)}

Forks a thread on each processor to call the function \texttt{f} with the arguments in the rest of the argument list. Arguments are evaluated on the local processor and passed by value to the forked threads. Returns after all threads have terminated.

\texttt{void do\_in\_parallel\_no\_synch(void f(), \ldots)}

Forks a thread on each processor to call the function \texttt{f} with the arguments in the rest of the argument list. Arguments are evaluated on the local processor and passed by value to the forked threads. Returns immediately without waiting for the forked threads to complete.
void mp_spin_barrier()
    Enforces a barrier among all threads spawned in the same call to do_in_parallel or do_in_parallel_no_synch. Note: this can only be called by threads spawned by the same call to do_in_parallel.

2.6.2 Thread Wait Queues

The following primitives are available for user-level scheduling of threads. An initialized wait queue will block any thread that calls BLOCK_ON_QUEUE on it. Threads blocked on a queue will be reenabled by a subsequent signal_queue call. After a queue has been signalled, it can no longer be used to block a thread until it is reinitialized.

waitq_p make_queue()
    Create and return an initialized wait queue.

void init_queue(waitq_p q)
    Initialize wait queue q.

waiters_present(waitq_p q)
    Returns TRUE if there are currently waiters blocked on q, FALSE otherwise.

void BLOCK_ON_QUEUE(waitq_p q)
    Block and deschedule self on q. Thread will remain blocked until the queue is signalled.

void signal_queue(waitq_p q)
    Wake up waiters currently blocked on q.

2.7 Synchronization

2.7.1 Spin Locks

The Alewife architecture includes full/empty bits that allow individual memory words (32 bits) to be locked and unlocked. The following spin lock primitives use a simple test_and_test_and_set algorithm to lock individual memory words. This implementation is very efficient in terms of instructions, but may suffer from bad performance under high contention. For more robust implementations, see Section 2.12.

unsigned spin_lock(unsigned addr)
    Locks and returns the contents of memory location addr. Note that the memory location must already have the full bit set; thus, it must have already been explicitly initialized at runtime.
void spin_unlock(unsigned val, unsigned addr)
    Sets the contents of memory location addr to val and unlocks the memory location.

2.7.2 Semaphores

sem_p make_semaphore()
    Create and return an unlocked semaphore.

sem_p make_locked_semaphore()
    Create and return a locked semaphore.

void SEMAPHORE_P(sem_p sem)
    Lock semaphore sem.

void SEMAPHORE_V(sem_p sem)
    Unlock semaphore sem.

int semaphore_conditional_p(sem_p sem)
    Returns TRUE if it succeeded in “P”-ing the semaphore, FALSE otherwise.

The following semaphore functions allow an integer to be associated with semaphore object. This integer can then be manipulated atomically. This has the advantage of allowing one to combine the manipulation of the lock bit together with the data.

sem_p_make_semaphore(int initval)
    Create and return an unlocked semaphore. The semaphore’s value is set to initval.

void init_semaphore(sem_p sem, int initval)
    Non-atomically initializes the semaphore’s value to initval.

int semaphore_take(sem_p sem)
    Lock semaphore sem and return its value.

void semaphore_put(sem_p sem, int val)
    Unlock semaphore sem and set its value to val.

int semaphore_conditional_take(sem_p sem, int *got_sem)
    Returns semaphore sem’s value and sets *got_sem to TRUE if successful in locking the semaphore. Returns 0 and sets *got_sem to FALSE otherwise.
2.7.3 Condition Variables

condvar_p make_condvar()
    Create and returns an initialized condition variable.

void init_condvar(condvar_p cv)
    Initialize the condition variable cv.

void condvar_wait(sem_p sem, condvar_p cv)
    Release the semaphore sem and block on condition variable cv; reacquire the semaphore when we unblock (wake up).

void condvar_broadcast(condvar_p cv)
    Wake up all threads currently blocked on condition variable cv.

2.7.4 Counting Semaphores

csem_p make_c_semaphore(int initval)
    Create and returns a counting semaphore initialized to initval.

void init_c_semaphore(csem_p cs, int initval)
    Initialize counting semaphore to initval.

void c_semaphore_p(csem_p cs)
    Wait until value of counting semaphore cs is positive, then decrements value of cs and returns.

void c_semaphore_v(csem_p cs)
    Increment value of counting semaphore cs.

void c_semaphore_wait(csem_p cs)
    Wait for counting semaphore cs to attain a positive value.

2.7.5 Barriers

The following are centralized implementations of barriers and thus are not scalable. For large numbers of processors, tree barriers should be used. (See Section 2.7.6).

sm_barrier_p make_sm_barrier(int n)
    Create and return a barrier initialized for n participants.

void init_sm_barrier(sm_barrier_p b, int n)
    Initialize barrier b for n participants.
void sm_barrier(sm_barrier_p b)
    Wait at barrier b for all participants to reach it.

2.7.6 Reductions

A reduction combines values using an associative operator, and returns the final result. For example, if the reduction operator is +, each thread participating in the reduction adds a number to a tally. The final tally of all the numbers is returned to all the participating threads. This is useful in computing dot products of vectors, for example. The reduction is implemented as a combining tree.

To use these reduction functions, the programmer will need some scheme for uniquely and contiguously numbering the participating threads from 0 upwards.

red_tree_p make_reduction_tree(int n, int fan-in)
    Create and return a centralized reduction tree for n threads. Each node in the tree has a fan-in of at most fan-in. A fan-in of 4 is recommended.

red_tree_p make_dist_reduction_tree(int n, int fan-in)
    Create and return a distributed reduction tree for n threads. The nodes of the tree are distributed among the machine nodes. A fan-in of 4 is recommended.

red_tree_p make_global_reduction_tree()
    Create and return a reduction tree for the entire machine. Since this is global, the mapping is pre-computed and is more efficient than make-dist-reduction-tree.

double REDUCE(int tid, red_tree_p tree, double op(), double val)
    Combine val with the operator op using the reduction tree tree, and wait for the final result before proceeding. tid must be unique among all participating threads. Returns the final result.

double REDUCE_ADD(int tid, red_tree_p tree, double val)
    Accumulates val using the reduction tree tree, and wait for the final result before proceeding. tid must be unique among all participating threads. Returns the final result.

void SM_TREE_BARRIER(int tid, red_tree_p tree)
    Wait at reduction tree tree with thread id tid. tid must be unique among all participating threads. Returns when all participating threads have arrived at the tree.
2.8 Message Passing Primitives

Alewife supports message-passing functions based on the Active Message model: upon receipt of a message, the receiving processor is trapped and the message handler is executed.

The following calls send an active message to a processor:

```c
void do_on(unsigned pid, handler (*handler_fn)(), ...)  
Send a message to processor pid to call handler handler_fn with the arguments in the rest of the argument list. Arguments are restricted to 32-bit values.

void do_on_dma(unsigned pid, handler (*handler_fn)(),..., dwords, dest_base, base, dwords)  
Send a message to processor pid to call dma_handler handler_fn with the arguments in the rest of the argument list, minus the last two arguments. Arguments are restricted to 32-bit values, and there must be an even number of them.

The last two arguments, base and dwords, specify a block of dwords double words starting at address base to be appended to the end of the message. This block will be automatically stored back at the destination processor starting at address dest_base (the third to last argument.)
```

Message handlers are declared as type `handler` and `dma_handler`.

```c
handler handler_fn(...)  
Defines the handler to be invoked on receipt of a message sent via `do_on`.

dma_handler handler_fn(...)  
Defines the dma_handler to be invoked on receipt of a message sent via `do_on_dma`. Call `wait_for_storeback()` within the handler to ensure that the block at the end of the message has been successfully copied to the specified destination.
```

Once a handler is invoked on a processor, it runs `atomically` until it exits. In other words, subsequent incoming messages to the processor do not interrupt the processor until the currently running handler exits.

Typically, a handler will only access data in a processor’s private memory. On occasion, it may be necessary to access data in shared memory from a handler. For various reasons beyond the scope of this document, it is highly risky to access shared memory from a handler that is executing atomically—doing so can lead to deadlock. Therefore, before accessing shared memory, a handler must transition out of atomic execution. To do this, the handler should execute the following function:

```c
void user_active_global()  
Transition out of atomic execution.
```
After calling `void user_active_global()`, it is safe for the handler to access shared memory. However, the currently running handler will lose its atomic execution properties; therefore, another incoming handler will interrupt the currently running handler. Any subsequent handlers that interrupt the current handler will execute atomically (unless they themselves call `void user_active_global()`), and the original handler will not re-execute until all other atomic handlers have completed.

See Section 2.11 for an example of a fetch-and-add operation implemented with active messages.

### 2.9 Useful Functions and Variables

**unsigned NPROCS**

Number of processors in machine (read-only, set by `(set-n-processors N)` in NWO).

**unsigned my_pid()**

Processor-id of the local processor (0 ... NPROCS-1).

**unsigned CReg->CycleCount**

Returns the current value of the hardware cycle counter (incremented once each cycle).

### 2.10 p4

The following p4 functions are supported:

- `char *p4_shmalloc(int n)`
- `VOID p4_shfree(char *p)`
- `int p4_moninit(p4_monitor_t *m, int i)`
- `VOID p4_menter(p4_monitor_t *m)`
- `VOID p4_mexit(p4_monitor_t *m)`
- `VOID p4_mcontinue(p4_monitor_t *m, int i)`
- `VOID p4_mdelay(p4_monitor_t *m, int i)`
- `VOID p4_lock_init(p4_lock_t *l)`
- `VOID p4_lock(p4_lock_t *l)`
- `VOID p4_unlock(p4_lock_t *l)`
- `int p4_getsub_init(p4_getsub_monitor_t *gs)`
- `VOID p4_getsub(p4_getsub_monitor_t *gs, int *s, int max, int nprocs)`
- `VOID p4_getsubs(p4_getsub_monitor_t *gs, int *s, int max, int nprocs, int stride)`

14
2.10.1 Incompatibilities with p4

Alewife’s C compiler allocates global variables (variables not within a function scope) in the local, non-shared memory of a processing node. Thus, global variables are private to a processor. Threads executing on the same physical processor share the same global variable.

When forking a thread, p4 assumes a Unix-style process fork model, where a forked process inherits a copy of all global variables at the time of the fork. In Alewife’s run-time system, a forked thread simply receives the arguments it was passed. This leads to a problem: p4 processes expect to inherit a private copy of all global variables; Alewife threads inherit whatever the values of the global variables happen to be on a particular node.

Thus, for programs written with the p4 process fork model in mind, a separate step is needed to copy the global variables to the destination processor before spawning a thread on that processor. (This works as long as we spawn exactly one thread on a processor.) One cumbersome method is to identify the variables in the program that expect to be copied upon a fork and explicitly copy them to the destination processor before forking the thread. (We know the destination processor since the current run-time system only allows spawning a thread on a specific processor.)

Since the most common method of spawning threads in p4 programs is to spawn a worker thread on each processor, the current solution is to spawn the worker threads via `do_in_parallel` and call `DISTRIBUTE_DATA()` prior to `do_in_parallel`. This is at best an interim workaround. The programmer still needs to be aware that threads on a processor still share the same copy of a global. The right solution is to allow global variables to be explicitly identified as private or shared, and for the program to treat them as so.

2.11 Examples
This program demonstrates how to call `do_in_parallel` and use its associated spin barrier.

```c
#include <stdio.h>
#include <parallel.h>

#define NBAR 4
#define NSPIN 4

void worker()
{
    int i;
    for (i = 0; i < NSPIN; i++) {
        mp_spin_barrier();
        if (my_pid() == 0)
            printf(" *** Spin barrier ***\n");
    }
}

main()
{
    int i;
    for (i = 0; i < NBAR; i++) {
        do_in_parallel(worker);
        printf("*** Barrier ***\n");
    }
}
```
This program demonstrates how to use the reduction functions, and how to pass arguments to threads spawned via `do_in_parallel`.

```c
#include <stdio.h>
#include <parallel.h>

double op(double x, double y)
{
    return MAX(x, y);
}

void worker(red_tree_p redtree, int *totals)
{
    int pid = my_pid();
    totals[pid] = (int)REDUCE(pid, redtree, op, (double)pid);
}

main()
{
    red_tree_p redtree;
    int *totals;

    redtree = make_global_reduction_tree();
    totals = (int *) shmalloc(sizeof(int) * NPROCS);
    do_in_parallel(worker, redtree, totals)
    return totals[0];
}
```
This program shows how to use the primitive message interface to implement fetch_and_add. A client calls \texttt{ipi\_faa}, which sends a message to a server. Due to restrictions on accessing global memory from message handlers, it is essential that access to the memory location containing the counter be restricted to the processor local to that location. In other words, no remotely cached copies of the counter should exist.

\begin{verbatim}
ipi\_faa(int *counter, int incr)
{
    handler faa\_request();
    int result;        /* stack allocated variables are in shared memory */

    lden(&result);     /* set full/empty bit of result to empty */

    /* send a message to the processor owning the counter */
    do\_on(ADDR2PID(counter), faa\_request, counter, incr, &result);

    /* poll for result by waiting for full/empty bit to become full */
    while (TRUE) {
        ldn(&result);     /* read result location */
        if (FE\_FULL) break; /* check state of full/empty bit */
        cswitch();        /* switch to next hardware context */
    }
    return result;
}

/* server side */
handler faa\_request(int *counter, int value, int *result\_loc)
{
    handler faa\_reply();
    int count;
    count = *counter;
    /* return current value of counter to client */
    user\_do\_on(ADDR2PID(result\_loc), faa\_reply, count, result\_loc);
    /* increment value of counter */
    *counter = count+value;
}

/* client side */
handler faa\_reply(int result, int *result\_loc)
{
    /* receive value of counter */
    *result\_loc = result;
}
\end{verbatim}
This program demonstrates the use of p4 monitor functions and how to distribute values of global variables via \texttt{DISTRIBUTE\_DATA}.

```c
#include <p4.h>

#define ALLOC(type) ((type *) shmalloc(sizeof(type)))

#define NITERS_I 4
#define NITERS_J 4

p4_monitor_t *mon;
p4_barrier_monitor_t *bar;
int *count;

void worker();

main(int argc, char **argv)
{
    p4_initenv(&argc, argv);

    mon = ALLOC(p4_monitor_t);
p4_moninit(mon, NPROCS);

    bar = ALLOC(p4_barrier_monitor_t);
p4_barrier_init(bar);

    count = ALLOC(int);

    printf("Distributing variables ...
");
    DISTRIBUTE\_DATA(); /* replicate global variables b4 forking */
    printf(" done.
");

    do_in_parallel(worker);

    return(*count); /* should equal NITERS\_I*NITERS\_J*NPROCS */
}

void worker()
{
    int i,j;

    printf("%d, t = %d: Executing worker\n", p4_get_my_id(), p4_clock());
    for (i = 0; i < NITERS\_I; i++) {
        for (j = 0; j < NITERS\_J; j++) {
            p4_menter(mon);
            *count += 1;
p4_mexit(mon);
        }
p4_barrier(bar, NPROCS);
    }
}

```
2.12 Scalable Synchronization Library

Recent research has designed scalable algorithms that perform well under high contention (at the price of a higher latency under low contention.) The scalable synchronization library implements some of these algorithms on Alewife. It also includes reactive synchronization algorithms that dynamically select the protocol to use to implement the synchronization operation. Alewife currently provides reactive algorithms for spin locks and fetch-and-op.

The reactive spin lock selects between using test-and-test-and-set with backoff and the MCS queue lock protocols. The reactive fetch-and-op selects between test-and-test-and-set, queuing, and combining tree protocols.

Note: Due to Alewife’s non-preemptive scheduling, a spin-waiting thread can hog the processor and run into deadlock. Thus, either the programmer should take care to avoid deadlock scenarios, or spawn off only as many threads as there are processors/hardware contexts to run the threads.

2.12.1 Spin Locks

lock_t *make_lock()
    Allocate a spin lock.
void init_lock(lock_t *L)
    Initialize spin lock *L.
void acquire_lock(lock_t *L)
    Acquire spin lock *L.
void release_lock(lock_t *L)
    Release spin lock *L.

In order to use these functions, you need to use:

#include <synch/locks/spin-backoff.h> for the test-and-set with backoff protocol.
#include <synch/locks/mcslock.h> for the MCS queue lock protocol.
#include <synch/locks/reactive.h> for the reactive spin lock algorithm.

For p4 users: In order to substitute the default p4 lock algorithm with any of the above algorithms, add #include <synch/p4.h> after #include <p4.h>, then include one of the three include files above. This redefines p4_lock_t, p4_lock_init, p4_lock and p4_unlock to one of the algorithms above.
2.12.2 Fetch-and-Add

```c
counter_t *make_counter()
    Allocate a counter.

void init_counter(counter_t *C, int initval)
    Initialize counter *C to value initval.

void set_counter(counter_t *C, int initval)
    Set the current value of counter *C to value initval.

int counter_value(counter_t *C)
    Return the current value of counter *C.

void fetch_and_add(counter_t *C, int increment, int dest)
    (Actually a C macro). Increments counter *C by increment, and assigns the pre-incremented value of the counter to dest.
```

In order to use these functions, you need to use:

```c
#include <synch/counters/spin-backoff.h> for the test-and-set lock based counter.
#include <synch/counters/queue.h> for the MCS queue lock based counter.
#include <synch/counters/gvw-ctree.h> for the combining tree protocol by Goodman, Vernon and Woest.
#include <synch/counters/reactive.h> for the reactive algorithm that selects among the preceding three protocols.
```

2.12.3 Spin Barriers

We do not have a reactive algorithm for spin barriers. However, we have several implementations of spin barriers that are much more efficient than the default blocking barrier implementations. The spin barriers use the following interface:

```c
bar_t *make_barrier(int nprocs)
    Create a barrier for nprocs processors.

void init_barrier(bar_t *b, int nprocs)
    Initialize barrier *b for nprocs processors.

void barrier(bar_t *b)
    Wait at barrier *b for all participants to arrive.
```
In order to use these functions, you need to use:

```c
#include <synch/barriers/dissemination> for a dissemination barrier.
#include <synch/barriers/tournament> for a tournament barrier.
#include <synch/barriers/mcstree> for a tree barrier by Mellor-Crummey and Scott.
#include <synch/barriers/mp-tree> for a message-passing-based tree barrier. This is the spin barrier associated with `do_in_parallel` (see Section 2.6.1). As a consequence, this algorithm can only be used by threads spawned by the same call to `do_in_parallel`.
```

For p4 users: In order to substitute the default non-scalable p4 barrier algorithm with any of the above algorithms, add `#include <synch/p4.h>` after `#include <p4.h>`, and include one of the four include files above. This redefines `p4_barrier_monitor_t`, `p4_barrier_init` and `p4_barrier` to one of the algorithms above.
3 NWO Users Manual

NWO is an architectural simulator for the Alewife Machine, a parallel computer built by the Alewife Group at the MIT Laboratory for Computer Science. This simulator allows users to develop, debug, and obtain measurements from software that can run on the Alewife Machine. A description of the Alewife machine architecture is beyond the scope of this document. You will hear more about Alewife in class, or you can acquire documentation and research papers by request.

NWO is a binary-compatible functional simulator of the Alewife machine. That is, a program written for the Alewife machine will be able to run on either the hardware or on NWO without recompilation. NWO can simulate about 3000 clock cycles of a single Alewife node per second. Furthermore, the functional simulator can simulate an entire Alewife machine with up to 512 processing nodes. This performance – combined with full support of the programming interface – allows NWO to be used to develop the software for the Alewife machine, including parallel C and parallel LISP compilers, the host interface, the runtime system, and benchmark applications. NWO provides a number of debugging and statistics functions that aid the analysis of programs and their performance on Alewife. Note that the simulator that preceded NWO was called ASIM, and you may come across references to ASIM in your readings; typically just think ‘NWO’ when you see ‘ASIM’.

3.1 Compiling a Parallel C Program

Before compiling a program or starting NWO, you must install (copy) four environment files into your home directory. If you have not done this, do so now by following the directions in Section 1.1.

A program for ENEE 759A typically consists of a set of Parallel C files. To compile them, run the alcc compiler front-end from the shell. It functions more or less just like gcc does for serial programs:

\%
 alcc -c module1.c
\%
 alcc -c module2.c
\%
 alcc -o prog module1.o module2.o

or

\%
 alcc -o prog proc.c

Using -c generates a .o file; linking with -o generates a program binary image (e.g. prog) and a debugging file (e.g. prog.debug).
3.2 Starting NWO

Emacs users may find it easiest to run NWO inside an emacs window. The “tea19.el” and “.emacs” files you were asked to copy into your home directory in Section 1.1 allow you to do this. Once you’ve copied these files, you can run NWO within emacs by typing `M-x run-nwo` in your emacs window.

You can also just type `nwo` from the shell prompt to start running NWO. NWO runs under the T language (a Lisp-like language most similar to Scheme), so the first thing you see when it starts is the T interpreter prompt. The entire simulator is written in T, and the interface to the simulator uses T syntax. T, and other Lisp-like languages, provide a very powerful software development environment in part due to the ability to load both compiled and uncompiled source code (which is interpreted) dynamically without ever leaving the T interpreter environment. If you don’t know T or Lisp, don’t panic—you will not be writing any T code in this class. In this section, we will describe only those T commands you will need in order to use the simulator.

Note that it can take awhile for a GLUE SPARCstation to load NWO, and it requires a lot of swap space on the local machine to run. In fact, since `alcc` also requires a lot of swap space, you may find that you have to exit from NWO (using `(exit)`) before you can run `alcc`. In general, try to find a machine in the GLUE cluster with as much physical memory and swap space as possible.

Commands for NWO are normally entered at the `>` prompt. If you make a mistake, and the T interpreter encounters a runtime error, a `>>` prompt may appear. This will occur recursively if you keep entering invalid T commands. To “go up” one level type `Control-D`.

To run your compiled Parallel C code on NWO, it must be loaded into the simulator by using the `ldap` (load Alewife program) function:

```plaintext
(ldap "filename")
```

(If you find that NWO’s working directory isn’t the one with your program in it, you can change it with `(set (working-directory) "pathname")`.)

If `(ldap)` executes with no errors you are now ready to execute your program:

```plaintext
(run <arg1> <arg2> ...)
```

where `<arg1>`, `<arg2>`, ... are the arguments to pass to `main()`.

3.3 Simulator Output

As the program runs, you will see output in the following format:

```
From pid:cid, t = cycle: message
```
When the program terminates, “Running exit program cleanup” messages will be generated by all the nodes, and NWO will display something along the lines of

Execution finished with status: 0. Execution time: 76764 cycles.

The current cycle value may be obtained from the T prompt by typing *the-system-time* to display the current system time. Note that the ‘execution time’ shown at program termination does not include cycles spent initializing the machine.

When you are finished using NWO, type (exit) to exit from the simulator.

The following sections attempt to give enough information to use the basic functionality of the sequential and parallel versions of NWO simulator. Since NWO is a very complex system, this document is intended to serve as a rough-and-ready introduction, not as an exhaustive compilation of NWO’s features.

### 3.4 Tweaking Parameters

- **(set-n-processors <N>)** sets the machine size using a default geometry.

- **(set-n-contexts <n>)** sets the number of contexts (defaults to 1).

- **(set-memory-bank-size <s>)** sets number of memory WORDS per node (default is 64K words = 256K bytes).

- **(set-local-memory-bank-size <s>)** sets number of local memory WORDS per node (default is 64K words = 256K bytes).

- **(print-settings)** displays current settings for the processor simulator.

Setting the memory bank size is particularly important. The default size set in the global init.t file is for a 1 MW (4 MB) shared memory on each node and a 512 KW (2 MB) local memory. For running NWO on machines with limited swap space, you will need to set these values to something much lower or you will run out of virtual memory, resulting in error messages such as

```
** Error: reference to non-existent memory
```

Setting the memory size lower will also improve NWO startup times even on machines configured with plenty of virtual memory. Toy programs will probably work fine if you use values such as

```
(set-local-memory-bank-size (* 192 1024)) ; 0.75 MB local memory
(set-memory-bank-size (* 16 1024)) ; 64 KB shared memory
```
If you set the memory sizes too low, you will get other error messages as various things run out of memory. For too small a local memory size you might see errors such as

** Error: Code size exceeds memory size

DEATH 0:1, t = 8707: Overflow in local_chunk
in procedure: bcopy

For too small a shared memory size you might see errors such as

DEATH 0:1, t = 4251: Overflow in static_chunk
in procedure: VECTORS

DEATH 0:1, t = 11268: Static Heap Overflow
in procedure: VECTORS

A useful place to reset the default local and shared memory bank sizes is in an init.t file in the current directory where NWO starts up. If NWO finds such a file, it will load it in before starting to run interactively.

### 3.5 NWO Debugging

The debugging facilities for NWO programs are minimal. However, some primitive facilities are provided:

- Setting breakpoints at machine instructions
- Tracing machine instructions
- Breaking at specific processor cycles
- Examining memory locations.

To break out of simulation mode, hit Control-C (or kill -INT the simulator process). This stops (‘breaks’) the simulation. To continue simulating after a break, execute (ret).

NWO does not have a debugger (in other words, there is no “gdb” or “dbx” equivalent for NWO). Therefore, you will rely heavily on printf and other print statements to provide debugging feedback from your simulated program. Here are a few suggestions for using these statements:

- A printf() statement can be executed from any node; however, printf() statements are quite slow since the simulator is simulating the rather complex functions
that make up the formatted printing routines. `write()` is much faster, but be aware that when a node uses `write()` the output goes via the network to the simulated front-end before being displayed, so it will lag somewhat behind the actual flow of execution.

- The `nwomsg(const char *msg)` function can be called from a Parallel C program. This ‘function’ compiles to a single no-op instruction (and is thus very fast) which is ignored by the Alewifé hardware, but traps in the NWO simulator to display the ‘msg’ string (prefixed by the node and context ID and a time stamp). The string to print must be a constant compile-time string. In other words, `nwomsg(const char *msg)` is useful in annotating what parts of your program have been reached during simulation, but cannot be used to look at the runtime contents of program variables.

### 3.5.1 Tracing and Setting Breakpoints

Since you are running on a (simulated) multiprocessor, you will have to enable tracing on specific processors before any of the tracing and breakpoint facilities will take effect. _Processors that do not have tracing enabled will not participate in any of the tracing or breaking activities described below._

### 3.5.2 Enabling/Disabling Tracing

- `(trace-processors)` enables tracing on all processors.
- `(trace-processor <n>)` enables tracing on processor `<n>`.
- `(untrace-processors)` disables tracing on all processors.
- `(untrace-processor <n>)` disables tracing on processor `<n>`.

To disassemble the processor code for specific code locations, use `trace-inst`; to view data, use `memory-peek`. (Note that memory addresses are byte addresses.)

- `(trace-inst <pid | processor-structure> [PC-address [count]])` disassembles the PC of node `<pid>`, giving the values of relevant registers. The optional argument `PC-address` gives the PC to print, and the optional `count` is the number of instructions to print after `PC-address`.
- `(memory-peek <pid> <addr>)` returns data at memory location address, as seen by processor `<pid>`. In many cases `<pid>` is 0.

### 3.5.3 Breakpoints

It is possible to set tracing and breakpoints on executions at a particular program counter or cycle. The following functions suffice:
(breakon-pc <pc>) sets a breakpoint on the given PC.
(unbreakon-pc) unsets the PC breakpoint.
(breakon-cycle <n>) sets the cycle breakpoint on a given cycle.
(unbreakon-cycle) unsets the cycle breakpoint.

Note: It is only possible to have ONE cycle breakpoint set at a time.

3.5.4 Miscellaneous Debug Functions

(shut-up!) deletes all breakpoints, stops all tracing.
(dump <pid>) dumps all of the processor registers on node <pid>.
(sump <pid>) dumps processor registers from frames 0 and 1 on node <pid>.
(pc->procedure-name <address>) returns the symbol name of the procedure corresponding to <address>.
(name->addr <name>) returns the address of the (static) object corresponding to <name>.

3.5.5 Debugging Network Messages

These are functions that allow network messages to be traced:

(spew-all) spew all messages.
(spew-on) turn spewage on output to screen which messages are currently spewed, as modified by the functions below.
(spew-off) don’t spew any messages.
(spew-only-mpi) only spew mpi messages.
(spew-only-prot) only spew protocol messages.
(spew-mpi-and-prot) only spew mpi and protocol messages.
(spew-msg op ...) spew only messages with the given major operands; no parameters means spew messages with all ops.
(unspew-msg op ...) stop spewing messages with the given ops. This doesn’t work if all messages are currently being spewed. Removing all ops from the list of spewed messages will stop all spewage.
(spew-node pid ...) spew only messages sent from nodes identified by pids; no parameters means spew messages from all nodes.

(unspew-node pid ...) stop spewing messages send from nodes identified by pids; if no pids remain on the list of spewed nodes, then NWO will resume spewing message from ALL nodes.

(spew-addr byte-addr ...) if spewing protocol messages, only spew messages with the indicated addresses; no parameters means if spewing protocol messages, spew messages for all addresses.

(unspew-addr byte-addr ...) stop spewing byte addresses; if no addresses remain on the list of spewed addresses, then NWO will resume spewing ALL addresses.

(spew-word-addr word-addr ...) same as above, except operates on word addresses.

(unspew-word-addr word-addr ...) stop spewing word addresses; if no addresses remain on the list of spewed addresses, then NWO will resume spewing ALL addresses.

These are functions that show the processor ↔ memory controller interface:

(spew pid ...) show the processor ↔ memory controller interface and memory controller state machines.

(unspew pids ...) stop showing the processor ↔ memory controller interface.

3.6 Debugging Example

Here are some examples of the use of the above functions:

> (shut-up!)
"ok boss"
> (breakon-cycle 7809)
7809
> (run)
From 0:1, t = 1895: Loading New Task
From 0:1, t = 1937: Running New Task
Hello, world!
From 1:1, t = 7500: Loading New Task
From 2:1, t = 7520: Loading New Task
From 1:1, t = 7542: Running New Task
From 3:1, t = 7560: Loading New Task
From 2:1, t = 7562: Running New Task
From 3:1, t = 7602: Running New Task
Broke on cycle 7809
>> (trace-inst 2)
3.7 Running NWO Without a Terminal

When simulations run for a long time, it is beneficial to run without a terminal. While it is possible to pipe a file directly to standard input (\texttt{nwo < input-file > output-file &}), this method can lead to two problems:

1. No input to the \texttt{nwo} process is possible after the simulation has been started.

2. If the \texttt{nwo} process attempts to access past the end of the input file (such a scenario might be caused by a simulation error that returns control to the REPL), \texttt{nwo} will continue to output the following message until \texttt{output-file} fills up its disk partition:
** Use ^Z or (STOP) to suspend, or (EXIT) to exit

An alternative method for running NWO uses the `tail` program to solve both of these problems. Run the simulation by executing `tail +0f input-file | nwo >& output-file &`. This command pipes the contents `input-file` to the `nwo` process, and keeps waiting for additional commands to be appended to the file. The premature end of file error no longer occurs, because `tail +0f` will never close its output (`nwo`'s input). Furthermore, additional commands may be sent to the process by appending to `input-file`. Appending can be done directly from a shell by typing `echo "new-command" >> input-file`. Also, sending a `kill -INT` to the `nwo` process (get the number from `ps ux`) acts the same as a control-C in the `nwo` terminal.

A side effect of this running method is that the `tail` and `nwo` processes will exist until they are explicitly killed. Don’t forget to clean up!

```
% echo > input-file
% tail +0f input-file | nwo >& output-file
% tail +0f input-file | nwo >& output-file &
[1] 21238 21239
% ps ux
USER    PID %CPU %MEM  SZ RSS TT STAT START TIME COMMAND
chaiken 21239 14.1  1.732716 516 p1 S 13:23 0:01 .../cnwop.dump ...
chaiken 21154  0.0   1.2 160 372 p1 S 12:59 0:02 -tcsh (tcsh)
chaiken 21238  0.0   0.5  44 160 p1 S 13:23 0:00 tail +0f input-file
chaiken 21246  0.0   1.5 212 456 p1 R 13:23 0:00 ps ux
% kill -INT 21239
% echo '(exit)' >> input-file
% ps ux
USER    PID %CPU %MEM  SZ RSS TT STAT START TIME COMMAND
chaiken 21247 15.4  1.6 216 472 p1 R 13:23 0:00 ps ux
chaiken 21154  0.0   1.3 160 380 p1 S 12:59 0:03 -tcsh (tcsh)
chaiken 21238  0.0   0.5  44 160 p1 S 13:23 0:00 tail +0f input-file
% kill 21238
[1] Terminated                tail +0f input-file |
    Done                    nwo >& output-file
%```
4 NWO Statistics

NWO has a wide range of statistics information that it can generate on the behavior of parallel programs. These statistics can be post-processed to get a clearer picture of what is going on in a parallel program—sequential sections, bottlenecks, etc.

4.1 Generating Statistics from NWO

There are two modes for using statistics: global mode and group mode. You can always see the state of statistics by running (stats?).

Many of the functions take an optional number of processor-ids as parameters. Example: (spew-on pid ...) means that (spew-on) (spew-on 7), and (spew-on 12 7 19 35 217) are all valid. In general, do not use the optional pid ... field in global mode for any functions except (spew-on) and (spew-off).

Use (stats-on) to turn statistics on and (stats-off) to turn them off.

In the global mode, all of the controllers update the same set of statistics (histograms, locality arrays and pointers). In the non-global mode, each controller updates its own set of statistics. Many functions have different behavior, depending on the statistics mode. (global-stats-on) and (global-stats-off) choose between global and non-global mode. The (stats?) function will print the current statistics mode.

At run time, (stats-on-nodes pid ...) and (stats-off-nodes pid ...) may be used to enable and disable statistics for individual controllers or for all controllers. These functions are independent of the current statistics mode.

(reset-stats) is run automatically at the beginning of every simulation. You may also use (reset-stats pid ...) at any time during a simulation. This function clears the statistics structures and enables or disables statistics, depending on the initial-stats mode. (initial-stats-on) causes (reset-stats) to enable statistics, (initial-stats-off) causes (reset-stats) to disable statistics. In global mode, (reset-stats) always resets the statistics for all controllers. In non-global mode, the optional [pids] arguments determine which nodes should have their statistics reset (and have their statistics enabled or disabled, depending on the initial-stats mode).

(spew-stats <filename> pid ...) outputs the statistics into text files. (See Section 4.1.1, for descriptions of these files.) In global mode, only one set of statistics will be spewed: <filename>.extensions. In non-global mode, one set of statistics is spewed for all specified nodes (all if the optional [pids] field is missing): <filename>.<pid>.extensions. Caution: spew-stats takes up a lot of disk space. Consider using dump-stats for permanent storage.

Dump the statistics to files using (dump-stats <filename>), where <filename> is any of the file specifiers accepted by T. In global mode, this function generates
a file `<filename>.dump`, which contains a binary image of the global statistics. In non-global mode, this function generates one binary file for each node, called `<filename>.<pid>.dump`.

When in non-global mode, `(coalesce-and-dump-stats <filename> pid ...) adds the statistics from all nodes, or a given set of nodes. It then dumps the statistics to `<filename>.dump. (coalesce-and-spew-stats pid ...) is similar, except it spews the statistics to text files.

Use `(retrieve-stats <filename>) to recover one or more previously dumped files. If `<filename>.dump exists, then this function will recover just one statistics image, and set the retrieved statistics mode to global. Otherwise, this function will look for `<filename>.<pid>.dump files, and recover them, setting the retrieved statistics mode to non-global. The `(stats?) function also describes the state of the retrieved statistics.

(dump-retrieved pid ...), (spew-retrieved pid ...), (coalesce-and-dump-retrieved pid ...), and (coalesce-and-spew-retrieved pid ...) do the same functions for retrieved statistics as the (*-stats) functions do for normal statistics.

4.1.1 Statistics file types

`<filename>.cachedlylat:`
    histogram of latencies for transactions that should have been cache hits, but required network messages due to (rare) interference

`<filename>.txnbufdlylat:`
    histogram of latencies for transactions that should have been transaction buffer hits, but required network messages due to (rare) interference

`<filename>.locallat:`
    histogram of latencies for transactions that had to go to local memory but (except for rare scenarios) did not need network messages

`<filename>.locdlylat:`
    histogram of latencies for transactions that had to go to local memory and (except for rare scenarios) did need network messages

`<filename>.remotelat:`
    histogram of latencies for transactions that had to go to a remote memory, except for rare scenarios

`<filename>.cachelat:`
    latency of request with cache hit, except for cachedlylat scenarios, in which case this is the same as accesslat
<filename>.txnbuflat:
latency of request with transaction buffer hit, except for txnbufdlylat
scenarios, in which case this is the same as accesslat

<filename>.greenlat:
time from the initial processor request until the last time that the data
becomes ready for access before the actual access

<filename>.{locgreenlat,locdlygreenlat,remotegreenlat}:
greenlat broken down for local, local + network, and remote

<filename>.wov:
window of vulnerability histogram

<filename>.{locwov,locdlywov, remotewov}:
wov broken down for local, local + network, and remote

<filename>.accesslat:
access phase for miss transactions
(locallat, locdlylat, and remotelat)

<filename>.{locaccesslat,locdlyaccesslat,remoteaccesslat}:
accesslat broken down for local, local + network, and remote

<filename>.busies:
histogram of the number of busies received by each transaction

<filename>.locality:
locality array for all global accesses

<filename>.misslocality:
locality array for global misses (locallat, locdlylat, remotelat)

<filename>.counters: text file with counters

<table>
<thead>
<tr>
<th>name</th>
<th>number of...</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATS-N-LOCAL-LO =</td>
<td>accesses to system local memory</td>
</tr>
<tr>
<td>STATS-N-LOCAL-HI =</td>
<td>accesses to user local memory</td>
</tr>
<tr>
<td>STATS-N-LOCAL-GLOBAL =</td>
<td>accesses to local global memory</td>
</tr>
<tr>
<td>STATS-N-REMOTE-GLOBAL =</td>
<td>accesses to remote global memory</td>
</tr>
<tr>
<td>STATS-N-IFETCH-LOCAL-LO =</td>
<td>ifetches to system local memory</td>
</tr>
<tr>
<td>STATS-N-IFETCH-LOCAL-HI =</td>
<td>ifetches to user local memory</td>
</tr>
<tr>
<td>STATS-N-IFETCH-LOCAL-GLOBAL =</td>
<td>ifetches to local global memory</td>
</tr>
<tr>
<td>STATS-N-IFETCH-REMOTE-GLOBAL =</td>
<td>ifetches to remote global memory</td>
</tr>
<tr>
<td>STATS-N-BUSIES =</td>
<td>total busies</td>
</tr>
</tbody>
</table>
STATS-N-CACHE-HIT = cache hits
STATS-N-TXNBUF-HIT = transaction buffer hits
STATS-N-LOCAL-NO-NET = all local transactions, no network
STATS-N-LOCAL-LO-NO-NET = just lower local accesses to memory
STATS-N-LOCAL-HI-NO-NET = just upper local accesses to memory
STATS-N-LOCAL-WITH-NET = local transactions, with network
STATS-N-REMOTE-MISS = remote transactions
STATS-N-CANT-FIND = lost transactions (simulation artifact)
STATS-N-LOCAL-FE-OPT = full/empty optimizations for local
STATS-N-LOCDLY-FE-OPT = full/empty optimizations for locdly
STATS-N-REMOTE-FE-OPT = full/empty optimizations for remote

The full-empty optimization counters account for the case when the controller decides that data is local, read-only, and does not need to write the fe-bits.

The group mode is almost exactly the same as global mode, except that you can get statistics for individual controllers or groups of controllers. To specify a controller group, use (stats-group <pid-i> ...). This will establish ’(<pid-i> <pid-j> ...) as a statistics group. Use (stats-unigroup <pid-i> ...) to delete a controller group. (stats-on <pid-i> ...) and (stats-off <pid-i> ...) may be used to turn statistics on and off during simulation, independent of the rest of the rest of the controllers in each group. At the end of the simulation, dump statistics for a group by specifying any of the pid’s in the group: (dump-stats <filename> <pid-n>)

Statistics may be turned on or off for ifetch accesses using the functions (stats-ifetch-on) and (stats-ifetch-off). The default is off. When statistics are off for ifetch accesses, STATS-N-IFETCH-REMOTE-GLOBAL contains a count of all ifetch accesses.

Statistics may be turned on or off for local accesses using the functions (stats-local-on) and (stats-local-off). The default is on.

Note: The transaction type counters are incremented at access time, so they should match the greenlat, wow, and accesslat histogram totals exactly. If there are transactions that are in the window of vulnerability or access phase when dump-stats is run, the locallat, locdlylat, and remotelat histograms may have more entries than the counters indicate.

4.1.2 Examples

> (stats?)
controller statistics off
no retrieved statistics
> (set-n-processors 2)

2 processors, 1 mesh dimension, 8 windows, 1 context
262144 bytes of global memory per node,
524288 bytes of local memory per node
5 directory pointers

> (stats-on)
STATISTICS-ON
> (stats?)

global controller statistics on
upon reset, statistics are on
no retrieved statistics
> (run 1)
From 0:1, t = 1742: Starting Boot Sequence
...
From 0:1, t = 5611: Finished Execution
From 0, t = 5742: Halt 4
1
> (dump-stats "global-stats")
#T
> (stats?)

global controller statistics on
upon reset, statistics are on
no retrieved statistics
> (retrieve-stats "global-stats")

retrieved 1 statistics object from "global-stats"
global controller statistics on
upon reset, statistics are on
retrieved statistics in global mode
> (spew-retrieved "global-stats")
1
> (global-stats-off)
GLOBAL-STATISTICS-OFF
> (stats?)

non-global controller statistics on
upon reset, statistics are on
retrieved statistics in global mode
> (run 1)
From 0:1, t = 1742: Starting Boot Sequence
...
From 0, t = 5742: Halt 4
1
> (stats?)

non-global controller statistics on
upon reset, statistics are on
retrieved statistics in global mode
> (dump-stats "non-global-stats")
()
> (stats?)

non-global controller statistics on
upon reset, statistics are on
retrieved statistics in global mode
> (retrieve-stats "non-global-stats")

retrieved 2 statistics objects from "non-global-stats.*"
non-global controller statistics on
upon reset, statistics are on
2 retrieved statistics in non-global mode
> (coalesce-and-dump-retrieved "coalesced-stats")
#T
> (retrieve-stats "coalesced-stats")

retrieved 1 statistics object from "coalesced-stats"
non-global controller statistics on
upon reset, statistics are on
retrieved statistics in global mode
> (dump-stats "stats-for-processor-1" 1)
()
> (retrieve-stats "stats-for-processor-1.001")

retrieved 1 statistics object from "stats-for-processor-1.001"
non-global controller statistics on
upon reset, statistics are on
retrieved statistics in global mode
> (retrieve-stats "stats-for-processor-1")

** Error: no statistics correspond to "stats-for-processor-1"
>>

4.2 Statistics Post-Processing Tools for NWO

There are several post-processing tools that can be used to aid in understanding of statistics output by NWO. Most of them are based on `splot`, a PostScript- and X-based plotting utility. In many cases, files are directly output by the simulator in `splot` format. In other cases, statistics take the form of text files that can be read directly. There are, however, two types of statistics output by the simulator that are not immediately useful. These include task traces and locality arrays.
Locality arrays are used to collect statistics about memory accesses. They are interpreted with respect to distance arrays, which give the distance (in network hops) between processors in a network. distance takes a locality array and a distance array and computes a locality plot. hotspot computes a plot of accesses to individual memories. p-hotspot computes a plot of accesses from individual processors.

### 4.2.1 Splot

splot is a plot utility for Encapsulated PostScript and X. It can be used to display statistics output from the cache / memory / network simulator, as well as output from parprof, distance and hotspot (see below).

splot is called as follows:

```
splot [-x] [infile]
```

splot takes a plot command file infile, or accepts input commands from the standard input, and writes Encapsulated PostScript formatted plots to standard output. Plots can also be previewed in the X environment.

A command of the form:

```
splot infile > outfile.ps
```

reads a plot description from infile and writes out a PostScript file to outfile.ps. To preview a graph in an X window, type splot -x infile. For a far more extensive description of splot, use man -M "/software/stradivari4/class/759a/splot-1.56/man" splot.

### 4.2.2 Locality Array Utilities

A number of statistics files are output by the simulator in the form of locality arrays. A locality array gives the access patterns from each processor to each memory module. Element (X,Y) gives the number of accesses from Processor X to Memory Module Y.

A locality array is more meaningful when associated with a distance array. Distance arrays, all of which reside in the directory "/software/stradivari4/class/759a/locality," give the distance (in network hops) from each processor to each network module. Element (X,Y) gives the number of network hops between Processor X and Memory Module Y. Distance arrays are provided for 1, 2, 3, and 4-dimensional meshes, for up to 1024 processors. For an <n>-dimensional mesh containing <p> processors, the associated distance array is in the file named "/software/stradivari4/class/759a/locality/locality.<n>.<p>".
• **distance** takes a locality array and a distance array, computing a locality plot on its standard output. It is used thus:

   distance <locality array file> <distance array file>

   The locality plot is simply a count of the number of accesses that traversed 1, 2, etc. network hops, up to the longest distance in the network.

• **hotspot** takes a locality array and a distance array, computing an access count plot on its standard output. It is used thus:

   hotspot <locality array file> <distance array file>

   The access count plot simply displays the number of accesses to each memory module in the system for the given run.

• **p-hotspot** takes a locality array and a distance array, computing an access count plot on its standard output. It is used thus:

   p-hotspot <locality array file> <distance array file>

   The access count plot simply displays the number of memory accesses from each processor in the system for the given run.
5 Using the Network Simulator

The network simulator is called *netwrap*. It can simulate various $k$-ary $n$-cube networks. Its current configuration limits the dimension $n$ to a maximum of 4. The simulator accepts the network parameters such as radix $k$, number of processors $N$ (the number of dimensions $n$ is calculated using $n = \log_k N$), and workload parameters such as message rate, message size, and locality. It outputs network statistics such as network channel utilization and average latency. The network simulator assumes infinite buffering at the switching nodes. It assumes unidirectional channels, and the network edges are connected to form a torus. For example, a network switch in a 2-D torus would look like:

![Switch Diagram](image)

The corresponding network might look like:
The simulator is run as follows. The arguments and their typical values are described in the following table. Statistics such as network channel utilization and average latency are sent to standard output.

```
prompt% netwrap procs pkts steps req-rt msg-size rad-k det-file loc > stat-file
```

<table>
<thead>
<tr>
<th>argument</th>
<th>description</th>
<th>typical value</th>
</tr>
</thead>
<tbody>
<tr>
<td>procs</td>
<td>number of processors</td>
<td>64</td>
</tr>
<tr>
<td>pkts</td>
<td>TOTAL number of packets to simulate</td>
<td>1000</td>
</tr>
<tr>
<td>steps</td>
<td>number of time steps to simulate</td>
<td>99999999</td>
</tr>
<tr>
<td>req-rt</td>
<td>message probability from a node</td>
<td>0.05</td>
</tr>
<tr>
<td>msg-size</td>
<td>average message size</td>
<td>6</td>
</tr>
<tr>
<td>rad-k</td>
<td>network radix $k$</td>
<td>8</td>
</tr>
<tr>
<td>det-file</td>
<td>file for detailed statistics</td>
<td>/dev/null</td>
</tr>
<tr>
<td>loc</td>
<td>locality parameter</td>
<td>1.0</td>
</tr>
<tr>
<td>stat-file</td>
<td>statistics file</td>
<td>netwrap.out</td>
</tr>
</tbody>
</table>

The simulator performs a cycle-by-cycle simulation of the network till the specified number of packers are generated. Each cycle every node generates a packet with probability $req-rt$ of constant size (equal to $msg-size$). The message destinations are randomly chosen from among all the nodes in the machine. However, when the locality parameter $loc$ is less than one, the simulator generates a randomly chosen destination from among a smaller subcube of nodes centered at the source node. The number of processors in this subcube is smaller than the total number of processors by the fraction $loc$.

To fire up a simulation with the typical arguments shown in the above table (corresponding to a 2 dimensional torus with 64 processors) type:

```
prompt% netwrap 64 1000 99999999 0.05 6 8 /dev/null 1.0 > netwrap.out
```
6 Using the Directory Simulator

6.1 General Information

This handout documents data and programs to be used in cache and directory simulations. The 64 processor address trace is available in brief64.ref. Each record in the trace contains the identifier of one of 64 processors, a type code (shared read, shared write, test-and-set, ...), and the address associated with this code. The trace was gathered using T-Mul-T (written by David Kranz) on a Speech recognition program (Viterbi search algorithm) written by Kirk Johnson.

One of the more difficult (and tedious) problems with trace-driven simulation involves manipulating very large quantities of data. Even though this trace is relatively short compared to other multiprocessor address traces, it is still 10 Mbytes long.

6.2 Command-Line Interface to Directory Simulators

The directory simulator has a command-line interface. Use dirsim to run simulations. The parameters for simulations are specified via command line switches as follows:

-s $l$ Number of lines (blocks) in each processor’s cache.
-b $b$ Size of each cache block. This is the unit of cache coherence.
-c $n_p$ Number of processors. Use 64 for this parameter.
-d $s$ Coherence directory scheme. Full-map = 64. Limited = 1, 2, 3, or 4.
          Single link chain = s. Double link chain = d.
-f <filename> Output file. Default output file is statZZZ.out.

The directory simulator output must be post-processed to calculate the probabilities of different types of cache events. The simulator output is post-processed with the awk program, a UNIX hack that was written by Aho, Weinberger, and Kernighan. awk reads a script file (written by the authors of the directory simulator) and an output file from the directory simulator. There is one script file for each type of directory protocol. awk outputs event probabilities and other information that may be used to calculate the system’s average network request rate and average network message size.

Watch out! The directory simulator appends to the file specified with the -f parameter (instead of just wiping out the file and rewriting it on each run). This feature reduces the chance that data from extremely long simulations will be lost. Before you run awk, you might want to make sure that there is only one set of results in the directory simulator output file.
6.2.1 Examples

1. The commands below run the directory simulator for a 64 processor system, with a 64K processor cache (4096 sets, 1 block per set, 16 bytes per block) and a 2 pointer limited directory. Note that the simulator command assumes you have links from your local directory to the trace and awk files in the course directory. You can do this by saying `ln -s /software/stradivari4/class/759a/trace/brief64.ref brief64.ref` and `ln -s /software/stradivari4/class/759a/awk/eventdirn.awk eventdirn.awk`.

   ```
   prompt% dirsim -s 4096 -b 16 -c 64 -d 2 -f dir2.out <brief64.ref
   ...
   prompt% awk -f eventdirn.awk dir2.out
   ...
   ```

2. The commands below run the directory simulator for a 64 processor system, with a 32K processor cache (2048 sets by 1 block per set by 16 bytes per block) and a single link chain directory. Note that the simulator command assumes you have a link from your local directory to the awk file in the course directory. You can do this by saying `ln -s /software/stradivari4/class/759a/awk/eventschain.awk eventschain.awk`.

   ```
   prompt% dirsim -s 2048 -b 16 -c 64 -d s -f schain.out <brief64.ref
   ...
   prompt% awk -f eventschain.awk schain.out
   ...
   ```
6.3 The vsee Program

The vsee program may be used to display trace data in human-readable format. Each record of the trace (as described) is printed to stdout. vsee requires one parameter: the number of records to be listed. The processor identifiers and the addresses are printed in hexadecimal, and the opcodes are printed according to the following scheme:

\[ r = \text{read}; \ m = \text{modify}; \ im = \text{interlocked modify}; \ w = \text{write}; \ fa = \text{fetch and add (usually signifies atomic synchronization operation)} \]

\[ b = \text{byte}; \ w = \text{word}; \ l = \text{long}; \ p = \text{private}; \]

The above are combined to indicate operation types; e.g., wp = write a word to private memory.

The other interesting instruction is ifetch = instruction fetch

The filler opcode indicates information from the tracing mechanism that does not fit into this format, but may be useful in some other context.

Operations named by their hex code are not currently in use.

6.3.1 Example

The command below lists 100 records of the brief64 trace to stdout.

\[ \text{prompt}\% \ vsee \ 100 \ < \ \text{brief64.ref} \]

6.4 Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brief64.ref</td>
<td>truncated speech trace</td>
</tr>
<tr>
<td>dirsim</td>
<td>directory simulator</td>
</tr>
<tr>
<td>eventdir.awk</td>
<td>awk script for reading dirsim output for limited/full map</td>
</tr>
<tr>
<td>eventdchain.awk</td>
<td>awk script for reading dirsim output for double link chain</td>
</tr>
<tr>
<td>eventschain.awk</td>
<td>awk script for reading dirsim output for single link chain</td>
</tr>
<tr>
<td>vsee</td>
<td>program used to display trace data</td>
</tr>
</tbody>
</table>