Pathfinder System Programming Guide

Document Number: 210113
Version Number: 1.0.2
Origination Date: January 13, 2021
Modification Date: June 23, 2021

Lucata Corp.
270 West 39th St., Suite 1302
New York, NY 10018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This document provides an overview for programming the Pathfinder system, with an emphasis on testing applications in a provided simulation environment (`emusim`). It is expected that programmers will initially develop, validate, and optimize programs using the simulation environment prior to executing on Pathfinder hardware.

The goal of `emusim` is to model the Pathfinder system and permit emulation and characterization of programs running on this system. It provides debugging capabilities and insight into execution characteristics that are not yet available on the hardware.

The simulation environment will also provide estimated timing performance; we continue to test and validate these timing models against the actual hardware and evolve the simulator as necessary.

1.2 System Overview

This section introduces the Pathfinder system organization and program execution with threads.

1.2.1 Pathfinder System Organization

A Pathfinder system is a shared memory multiprocessor consisting of a number of nodes and an IO system connected by an interconnection network. Each node in the system contains multiples of each element: Stationary Cores (SCs), Gossamer Cores (GCs), Memory Side Processors (MSPs), and Serial RapidIO (SRIO) network ports. The GCs, MSPs, and SRIO network ports are implemented on an FPGA that communicates with the SCs over PCIEexpress. The FPGA also has a ring-based network for on-chip communication. Figure 1.1 illustrates a single node within the system.

The Serial RapidIO (SRIO) network provides communication paths between the GCs and MSPs of a node to any other node in the system. For example, a GC on node 1 would be able to transmit information to an MSP on node 4 over the SRIO network.

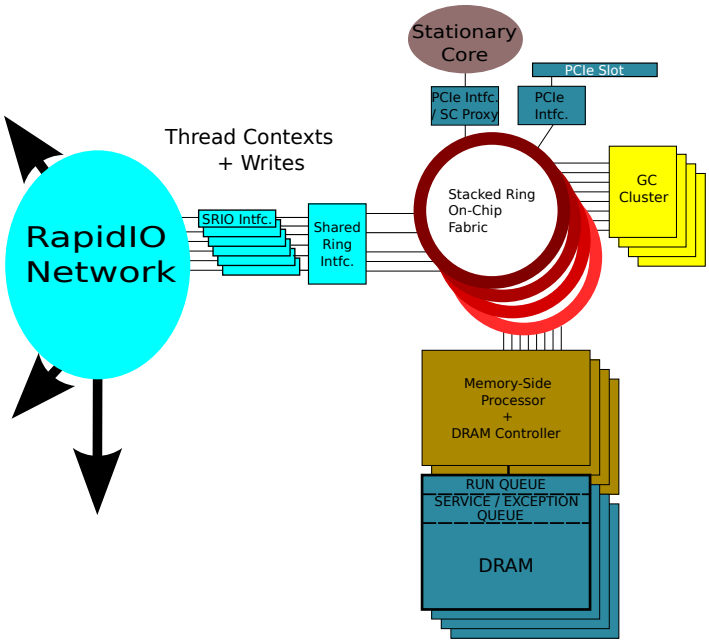


Figure 1.1: Pathfinder Node Architecture

The Stationary Cores are COTS processors run Linux and provide specific services for the node.

1.2.2 Execution Model

The Pathfinder system architecture implements a Partitioned Global Address Space (PGAS). In a PGAS, all memory in the system has unique addresses that are visible to everywhere in the system. However, the range of addresses need not be contiguous, and the memory may be distributed across various partitions with different latencies and bandwidths available depending on the proximity between a device requesting a memory access and the memory it accesses.

Programs executing on the Gossamer Cores are known as *Threads*. A thread has a lightweight context that the Pathfinder system hardware automatically transfers from node to node so that all of the memory references it makes occur at the node where it is executing. In other words, if the thread attempts access to a memory location that is part of the memory of a node other than the one where it is currently executing, hardware suspends its execution, transmits its context to the node containing the referenced memory, and restarts the thread on that node. This process is known as a migration. Once the migration occurs, the actual memory reference always occurs locally. Since the bandwidth required to migrate a thread context is only slightly more than the bandwidth consumed transmitting a memory reference across the network, this strategy is performance neutral when only a single reference occurs at a node, and is a significant win any time multiple references are made to memory at that node. Since, in practice, most algorithms usually make at least a few such references between migrations, significantly less bandwidth is consumed overall than in a conventional computer.

Programs executing on the Stationary Cores are known as *Stationary Threads*. Stationary threads are the type of programs typically run on conventional computers. They always execute entirely on the same core where they were started, although they may create threads to perform functions

on their behalf. For Pathfinder systems, stationary threads are used primarily for control and for performing system services. Generally, the main work done on the system will be performed by groups of threads executing on the Gossamer Cores. The stationary threads in the system are primarily intended to perform the following functions, although they are not limited to these:

- Run the Operating System (typically Linux)
- Perform input/output operations and run I/O Drivers
- Perform exception handling for most exceptions that occur on Gossamer cores
- Initialize and boot the system
- Perform various system services for threads running on Gossamer Cores

As noted previously, Stationary Cores are not capable of reading memory except on their own node, and have limited capability to write memory on nodes other than their own. When such memory accesses are required, they typically create threads to perform remote memory accesses on their behalf.

In general, user programs are launched by the Stationary Cores and run exclusively on the Gossamer Cores, with the Stationary Cores used only to handle exceptions, I/O, etc. User programs do not run on the SCs. The simulator does not explicitly model the SCs.

1.3 Overview of the Simulation Environment

The simulation environment (`emusim`) provides a simulator to emulate application execution on the Pathfinder system as well as a profiling tool to view the results of simulation. The simulator can run in functional simulation or performance simulation modes. The functional simulation runs applications with a simplified system model to allow quicker simulation for debugging and correctness testing. The performance mode provides timing and statistics for application performance starting at a desired location (future development will allow the user to stop a timing simulation as well) to predict application performance on the actual Pathfinder hardware. The profiling tool can be utilized to see how system resources are used over time.

1.4 Organization

This document is organized as follows:

- Chapter 2 describes the thread creation and control routines.
- Chapter 3 describes the memory configuration, data distribution and data allocation routines.
- Chapter 4 describes Pathfinder architecture-specific operations such as atomic memory operations and system queries.
- Chapter 5 provides a summary of available libraries.
- Chapter 6 provides a set of example programs to illustrate program development and analysis.
- Chapter 7 describes the simulation environment and the statistics that can be collected.

- Chapter 8 describes the profiling tools that can be used to analyze the simulator output.

1.5 Version History

Table 1.1 gives the version history for this document.

Version	Date	Changes
1.0.2	23-Jun-2021	Minor bug fixes; update profiling graphs due to profiler bugfixes. Remove intrinsic functions that no longer exist.
1.0.1	24-Mar-2021	Revamp cover page and add logo. Update headers/footers.
1.0.0	1-Feb-2021	Initial release of this guide for the Pathfinder system.

Table 1.1: Version History.

Chapter 2

Thread Creation and Control

2.1 Overview

Execution in Pathfinder systems is performed by one or more *threads* that run in parallel on the same or multiple Gossamer cores. The Cilk programming language has been chosen to support thread spawning and synchronization using the Cilk functions shown in Table 2.1. Note that the Pathfinder architecture supports thread spawning and synchronization directly in hardware and does not use the Cilk software runtime. This reduces the overhead involved in managing threads.

Table 2.1: Cilk Support for Spawning and Synchronization

Attribute	**Functionality**
cilk_spawn	Indicates that the function can run in parallel with the caller. Typically causes a new thread to be spawned to execute the function.
cilk_sync	Indicates that the current function cannot continue in parallel with its children and must wait for all children to complete.
cilk_for	Replaces a normal for loop to indicate that the loop iterations can execute in parallel.

2.2 Cilk Functions

2.2.1 cilk_spawn

```
[var = ] cilk_spawn function_name(arguments)
```

Functionality: The `cilk_spawn` keyword modifies a function call statement to tell the runtime system that the function may (but is not required to) run in parallel with the caller (referred to as the parent). This typically results in a new thread being spawned to execute the function. If a thread cannot be spawned, the function is executed as a traditional function call.

The compiler supports the idea of migration during spawning; that is, the programmer can specify that a new thread be started on the node associated with a particular memory address (as opposed to spawning on the local node). This memory address, or "hint", can be provided via `cilk_migrate_hint` (Section 2.2.2) or `cilk_spawn_at` (Section 2.2.3). The child thread's stack will be allocated in the remote node's memory, which can aid in reducing total migrations and distributing work across the system.

Notes:

1. A `cilk_spawn` expression must be the only expression on the right side of an assignment and cannot be part of a larger expression. For example, the following is disallowed and will cause the compiler to issue an error diagnostic:

   ```
   var = var2 + cilk_spawn func(args);
   ```

2. If a child thread is spawned, the parent continues execution in parallel with the child thread. There is an implicit `cilk_sync` at the end of every function and every `try` block that contains a `cilk_spawn`. Otherwise, it is up to the programmer to insert a `cilk_sync` before any code that requires all children to have completed. For example, a `cilk_sync` is required before the parent attempts to access a value returned from a function invoked with a `cilk_spawn`.

3. When a `cilk_spawn` is encountered, the hardware determines if there are sufficient resources to spawn a new thread. If not, it will execute the function as a call. The Pathfinder system can execute about 1500 threads per node and more threads than that can be created. However, the number of threads that are executing may differ depending on the program's execution characteristics such as the number of migrations and/or remote memory operations issued and the workload distribution. The simulator statistics files will indicate the number of successfully spawned threads

2.2.2 cilk_migrate_hint

```
cilk_migrate_hint(hint);
```

Functionality:

This built-in provides a locality hint for the next `cilk_spawn`. The child will be spawned on the node that `hint` points to. If `hint` is null, the spawn will occur locally.

Notes:

1. See `cilk_spawn_at` for a more convenient syntax for providing spawn hints.

2.2.3 cilk_spawn_at

```
cilk_spawn_at(hint) function_name(arguments)
```

Functionality:

Combines `cilk_spawn` and `cilk_migrate_hint` into a single command to spawn a thread at a particular memory location. The function will be spawned on the node that `hint` points to. If `hint` is null, the spawn will occur locally.

Notes:

1. Because `cilk_spawn_at` is implemented as a macro which emits a `cilk_migrate_hint` followed by a `cilk_spawn`, it cannot be used in contexts where a single statement is expected. For example, this `for` loop will fail to compile without braces.

   ```
   // Not supported, need to add {}
   for (long i=0; i<N; i++)
       cilk_spawn_at(&T[n]) foo(a, i);
   ```

2. Similarly, assigning the result of `cilk_spawn_at` to a variable is not supported. Use `cilk_migrate_hint` (Section 2.2.2) directly in these cases.

2.2.4 cilk_sync

```
cilk_sync
```

Functionality: The cilk_sync statement applies to a task block, which, in most cases, is a function. It indicates that the current function cannot continue in parallel with its spawned children, and must wait. After all of the children complete, the current function can continue.

Notes:

There is an implicit `cilk_sync` at the end of every function and every `try` block that contains a `cilk_spawn`. Otherwise, it is up to the programmer to insert a `cilk_sync` before any code that requires all children to have completed. For example, a `cilk_sync` is required before the parent attempts to access a value returned from a function invoked with a `cilk_spawn`.

2.2.5 cilk_for

```
cilk_for (declaration and initialization; conditional expression; increment expression)
body
```

Functionality: The cilk_for loop is a replacement for the normal C/C++ for loop in order to permit loop iterations to run in parallel.

Notes:

The cilk_for statement divides the loop into chunks with each containing one or more loop iterations. During execution of the loop a thread is spawned to execute each chunk of the loop with an implicit `cilk_sync` at the end of the loop to wait for all threads to complete.

The maximum number of iterations in each chunk is the grainsize. The current default grainsize is 16 iterations per thread and the grainsize can be set using the cilk grainsize pragma as shown in the following example:

```
#pragma cilk grainsize = 4
```

Note that because the loop iterations may be executed in parallel, copies of the local variables are made for each thread but any non-local variables are treated as shared variables that can be accessed by any of the threads.

The `cilk_for` is best used for spawning threads locally as it has not yet been optimized for non-local spawns.

Notes:

1. The grain size argument must be an integral constant expression. Support for a broader range of grainsize expressions may come in a later release.
2. The iteration variable may not be declared before the loop.

2.3 Priority, Flow Control, and Control of Parallelism

The Pathfinder system does not prioritize the execution order of threads in the system. As each thread arrives to run at a given node, it is placed into a FIFO based run queue to await execution.

Each node, and thus the system as a whole, has physical limitations on the amount of storage space available for threads. To control the total number of threads alive in the system and prevent running out of space for threads, a hardware based credit mechanism is implemented. In this scheme, each node is provided with a base number of credits; when a spawn occurs, this value is decremented and when a thread dies, this value is incremented. Should a thread attempt to spawn when no credits are available, the spawn will fail and the software will jump to a function call instead.

Currently, credits are not redistributed while a program executes, but this support may be implemented in the future. As threads typically die where they were born, this is not a major impediment.

The minimum number of credits will at least be the number of threads able to actively execute in the system.

Chapter 3

Data Allocation and Distribution

3.1 Overview

This section provides an overview of data allocation and distribution on Pathfinder systems. Because execution moves to the data in these systems, the way in which the data is distributed in memory helps to define the execution parallelism available to the algorithm.

In Pathfinder, data can be defined in a single node's memory or can be distributed across multiple nodes in the system. Table 3.1 summarizes the various functions used to support dynamic memory allocation and free. Table 3.2 summarizes the `replicated` attribute and functions used to access replicated data.

NOTE: The names of many functions in this section are subject to change in the future.

Table 3.1: Support for Dynamic Memory Allocation

Function	Functionality
Data allocation on single node	
`malloc()`	Allocate data structure on local node
`mw_localmalloc()`	Allocate data structure local to (on the same node as) some other data structure
Data allocation on multiple nodes	
`mw_malloc1dlong()`	Allocate a 1D array of longs striped across multiple nodes with a single element on each node
`mw_malloc2d()`	Allocate data structure striped across multiple nodes
Free previously allocated data	
`free()`	Free data allocated by `malloc`
`mw_free()`	Free data allocated by `mw_malloc1dlong` or `mw_malloc2d`
`mw_localfree()`	Free data allocated by `mw_localmalloc`
Access data allocated with `mw_malloc2d`	
`mw_arrayindex()`	Access data allocated by `mw_malloc2d` without causing a migration

Table 3.2: Support for Replicated Data

Attribute/Function	Functionality
Static Replicated Data	
`replicated`	Attribute placed in front of a variable definition to indicate that the variable should be replicated on each node
Dynamic Replicated Data	
`mw_mallocrepl()`	Dynamically allocate a replicated block on each node
Replicated Data Initialization	
`mw_replicated_init()`	Initialize a replicated variable to the same value on each node
`mw_replicated_init_multiple()`	Initialize each instance of a replicated variable to a different value as returned by the user-defined `init_func()` as a function of the node id
`mw_replicated_init_generic()`	Initialize each instance of a replicated variable using the user-defined `init_func()`
Replicated Data Access	
`mw_get_nth()`	Get the instance of a replicated data structure on node n
`mw_get_localto()`	Get the instance of a replicated data structure local to (on the same node as) some other data structure

3.2 Local Dynamic Data Allocation and Free

Local dynamic data allocation utilizes the standard C library functions `malloc` and `free` as defined in `stdlib.h`.

3.2.1 malloc

```
void* malloc( size_t Size)
```

Return Value: Returns a void* to the node's local heap

Arguments:

- Size: the size in bytes to be allocated

Functionality: The `malloc` function specifies the total block size (bytes) to be allocated on the local heap of the node on which the thread is currently executing and returns a pointer to the requested block of memory.

Notes:

Example:

3.2.2 free

```
void free(void* EltPtr)
```

Return Value:

Arguments:

- EltPtr: pointer to a block of memory previously allocated with malloc

Functionality: The `free` function deallocates the memory on the local heap previously allocated by malloc.

Notes:

Example:

3.3 Distributed Dynamic Data Allocation and Free

These functions are used to dynamically allocate data structures on a non-local node or distributed across multiple nodes in the system. These functions can be accessed by including the library header file `memoryweb.h`.

3.3.1 mw_localmalloc

```
void * mw_localmalloc(size_t eltsize, void * localpointer)
```

Return Value: Returns a void * to a block of eltsize memory located on the node that contains localpointer.

Arguments:

- eltsize: the size in bytes to be allocated
- localpointer: pointer to a variable on the node where you want the allocation.

Functionality: The `mw_localmalloc` allocates a chunk of at least eltsize on the same node that localpointer points to. A localpointer of zero will cause an exception.

Notes:

Example:

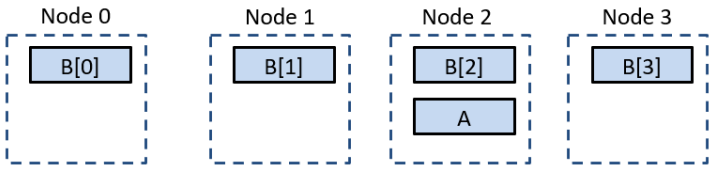


Figure 3.1: Allocating `A` co-located with `B[2]` using `mw_localmalloc()`

The following code demonstrates how to use the local malloc to allocate a variable `A` co-located (on the same node as) `B[2]` as shown in Figure 3.1:

```
long * A = (long *) mw_localmalloc(sizeof(long), &B[2]);
```

3.3.2 mw_malloc1dlong

```
long * mw_malloc1dlong(size_t numelements)
```

Return Value: Returns a long * to an array of numelements longs

Arguments:

- numelements: number of 64-bit elements to be allocated

Functionality: The `mw_malloc1dlong` allocates an array of `numelements` longs striped across the nodes round robin. This array may wrap if `numelements` is larger than the number of nodes. Arrays allocated using this function are accessed using array notation. Since each sequential element is on a different node, accessing sequential elements of this array will cause a migration on each access. This distribution is best used for cases where the array is not access sequentially.

Notes: Works ONLY for 64-bit types (e.g. long). For types of other sizes, you must use `mw_malloc2d`.

Example:

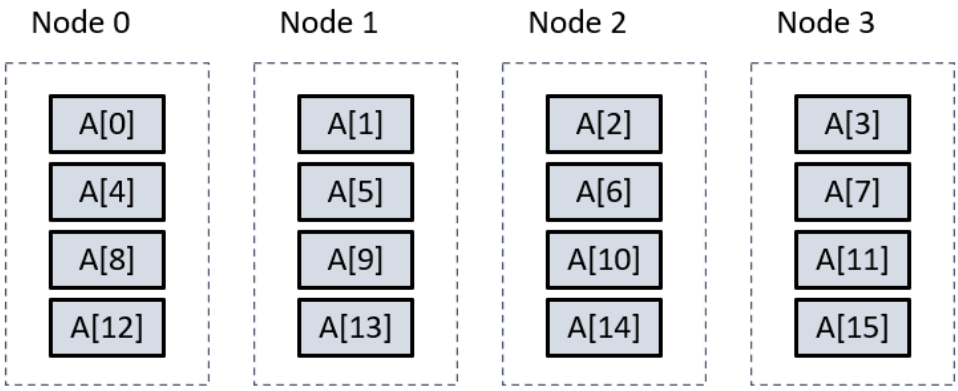


Figure 3.2: Round robin array distribution for array of longs.

The following code demonstrates how to create and access an array `A` of size 16 longs with one element on each node as shown in Figure 3.2:

```
#define N 16
long * A = mw_malloc1dlong(N);
for (long i=0; i<N; i++)
       A[i] = i;
```

3.3.3 mw_malloc2d

```
void * mw_malloc2d(size_t numblocks, size_t blocksize)
```

Return Value: Returns a void * to an array of `numblocks` elements, each of which is a pointer to a block of memory of size `blocksize`. Will return NULL if either input argument is 0.

Arguments:

- numblocks: the number of blocks to be allocated
- blocksize: the size in bytes to be allocated for each block

Functionality: The `mw_malloc2d` function allocates a distributed array of numblocks pointers striped across nodes round robin. Each points to a co-located memory block of blocksize.

Notes:

- Returns an array of pointers, so must be cast to `<type>` **
- Because it is a distributed array, accessing elements can cause migrations
- See `mw_arrayindex` for a more complete discussion of when migrations may occur and approaches to avoid them.

Example:

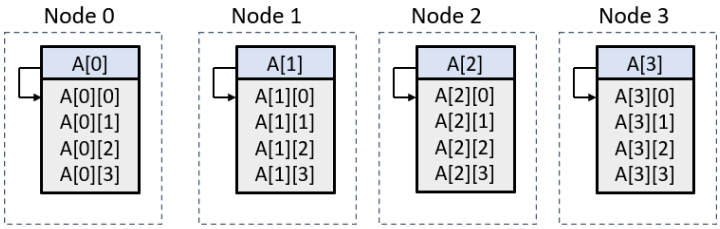


Figure 3.3: 2D array distribution on 4 nodes using `mw_malloc2d()`

The following code demonstrates how to create and access a 2D array, `A`, of longs. This example has 16 elements with 4 consecutive elements on each of 4 nodes as shown in Figure 3.3:

```
#define N 16
long epn = N/NUM_NODESS();
long ** A = (long **) mw_malloc2d(NUM_NODES(), epn * sizeof(long));
for (long i=0; i<N; i++)
       A[i/epn][i%epn] = i;
```

Note that this loop will access the first 4 elements of the array locally, then migrate to the next node to access that node's elements, and so on.

3.3.4 mw_free

```
void mw_free(void *allocedpointer)
```

Return Value: None

Arguments:

- allocedpointer: pointer to allocated memory

Functionality: The `mw_free` frees data allocated by `mw_malloc1dlong`, `mw_malloc2d`, or `mw_mallocstripe`

Notes:

3.3.5 mw_localfree

```
void mw_localfree(void *allocedpointer)
```

Return Value: None

Arguments:

- allocedpointer: pointer to allocated memory

Functionality: The `mw_localfree` frees data allocated by `mw_localmalloc`.

Notes:

3.3.6 mw_arrayindex

```
void * mw_arrayindex(void * array2d, unsigned long i,
   unsigned long numblocks, size_t blocksize)
```

Return Value: Returns a `void *` to the address of `array2d[i][0]`.

Arguments:

- array2d: array allocated with `mw_malloc2d`
- i: index for first dimension
- numblocks: numblocks parameter passed to `mw_malloc2d` when allocating `array2d`
- blocksize: blocksize parameter passed to `mw_malloc2d` when allocating `array2d`

Functionality: The `mw_arrayindex` takes as input a 2D array allocated with `mw_malloc2d` along with an index, `i`. It then calculates and returns the address of `array2d[i][0]`.

Notes:

This function is used for computing an address for `array2d[i][j]` without migrating to `array2d[i]` to load the pointer. It uses knowledge of how `mw_malloc2d` lays data out in memory to compute the address of `array2d[i][j]` directly. This function would be used to compute the address for a remote memory operation in order to avoid migration during the address computation. Note that this can be an expensive calculation and should be used at an outer level whenever possible, not as part of the inner loop computation.

Example: The following code demonstrates how to use `mw_arrayindex` to access a 2D array as shown in Figure 3.3:

```
#define N 32
long epn = N/NUM_NODES();
long** A = (long **) mw_malloc2d(NUM_NODES(), epn * sizeof(long));

// Migrating loop
for (long i=0; i<NUM_NODES(); i++)
    for (long j=0; j<epn; j++) {
        A[i][j] = 0;


// Loop using remote updates
for (long i=0; i<NUM_NODES(); i++) {
    long * Ai = (long *) mw_arrayindex(A, i, NUM_NODS(), epn * sizeof(long));
    for (long j=0; j<epn; j++)
        REMOTE_ADD((Ai+j), 1);
}
```

In the first loop, the thread will migrate to each element of `A[i][j]` in order to initialize it to 0. In the second loop, we want to use a remote update where the thread does not migrate. The compiler can calculate the location of `A[i]` directly based on the address in `A`. However, because `A[i]` is a pointer, it would migrate to `A[i]` and load that address in order to compute the location of `A[i][j]`. The `mw_arrayindex` function uses knowledge of the data layout to compute the start of the block pointed to by A[i], thus avoiding migrations in the computation of the location of A[i][j].

3.4 Replicated Data

These functions are used to allocate and access replicated data. When a variable is marked as `replicated`, a copy of the data element is allocated on each node at the same address. The address of a replicated data element is configured such that the local instance (i.e. the instance located on the node where the thread is executing) is always accessed. Therefore, specialized functions are needed to initialize and access specific instances of a replicated data structure. These functions can be accessed by including the library header file `memoryweb.h`.

3.4.1 replicated

```
replicated <var type> <var name> [= <intial value>];
```

Functionality: The `replicated` keyword is used to indicate a global replicated variable. A copy of this variable is placed on each node. A replicated variable always accesses the local copy on the node where the thread is executing. Replicated variables should only be used for data where it does not matter which instance of the variable is used, such as for constants or accumulating local data.

Notes: Updating a replicated variable changes only the LOCAL instance of that variable. Therefore different instances can have different values. In order to access a specific instance you must use specialized access functions.

Example:

```
struct stats {
    double max;
    double min;
};
replicated struct stats s;
replicated double PI = 3.14;
```

In this example, the stats struct, `s`, is replicated on each node in the system. `PI` is also replicated on each node in the system, with each copy initialized to 3.14. Whenever either `s` or `PI` is referenced in the program, the thread accesses the value on the local node.

3.4.2 mw_mallocrepl

```
long * mw_mallocrepl(size_t blocksize)
```

Return Value: Returns a pointer to replicated block of size `blocksize`.

Arguments:

- blocksize: the size in bytes of the block to be allocated at each node

Functionality: The `mw_mallocrepl` allocates a block of size `blocksize` on each node. It returns a replicated pointer so that accesses using that pointer will always access the block on the local node.

Notes: This is similar to using the `replicated` keyword but is used when the size of the data structure is not known at compile time and therefore must be dynamically allocated.

Example:

The following code demonstrates how to dynamically allocate a replicated structure at each node.

```
#include "memoryweb.h"
#include "stdio.h"
#include "stdlib.h"

typedef struct info {
    long node;
    long index;
} info;

replicated info *rinfo;

// Initialize the info struct on each node
void fill_struct(void *ptr, long node)
{
    info *gto = (info *)ptr;
    gto->node = node;
    gto->index = NODE_ID();
}

// Allocated a replicated info struct on each node
info *alloc_struct()
{
    info *mr = (info *)mw_mallocrepl(sizeof(info));
    mw_replicated_init_generic(mr, fill_struct);

    return mr;
}

int main(int argc, char *argv[])
{
    // Array with one element on each node
    long *A = mw_malloc1dlong(NUM_NODES());

    info *tmp_info = alloc_struct();
    mw_replicated_init((long *)&rinfo, (long)tmp_info);

    // Print info from instance of info struct on each node
    for (long i = 0; i < NUM_NODES(); i++) {
        info *local_info = mw_get_nth(rinfo, i);
        MIGRATE(&A[i]);
        printf("i=%d node=%d: LOCAL_INFO node %d index %d, RINFO node %d index %d\n",
               i, NODE_ID(), local_info->node, local_info->index, rinfo->node, rinfo->index);
    }

    return 0;
}
```

Figure 3.4: Simple Replicated Structure

3.4.3 mw_replicated_init

```
void mw_replicated_init(long * repl_addr, long value)
```

Return Value: None

Arguments:

- repl_addr: pointer to the replicated variable to be initialized
- value: initial value

Functionality: This function takes as input a pointer to a replicated variable and an initial value. It initializes each instance of the replicated variable to the initial value.

Notes:

Example: The following code demonstrates how to initialize each instance of a replicated variable to the SAME value (3.14 in this case):

```
replicated long PI;
int main()
{
  mw_replicated_init(&PI, 3.14);
  ...
}
```

3.4.4 mw_replicated_init_multiple

```
void mw_replicated_init_multiple(long * repl_addr, long (*init_func)(long))
```

Return Value:

Arguments:

- repl_addr: pointer to the replicated variable to be initialized
- init_func: user-defined function that takes the node id as input and returns the initial value for the replicated variable at that node

Functionality: This function takes as input a pointer to a replicated variable and user-define function `init_func(long nid)`. It initializes the instance of the replicated variable on node n to the value that is returned by the function when n is passed as the argument.

Notes:

Example: The following code snippet demonstrates how to initialize each instance of a replicated variable to a different value. In this example, each instance of B is initialized to (node ID * 5). So, for example, the instance on node 0 is initialized to 0 and the instance on node 5 is initialized to 25.

```
replicated long B;

long init_func(long nid) {
    return nid * 5;
}

int main()
{
    mw_replicated_init_multiple(&B, init_func);
    ...
}
```

3.4.5 mw_replicated_init_generic

```
void mw_replicated_init_generic(void * repl_addr, void (*init_func)(void *, long))
```

Return Value:

Arguments:

- repl_addr: pointer to the replicated variable to be initialized
- init_func: user-defined function that takes the absolute address of the replicated data structure on the node with the ID given by the second argument.

Functionality: This function initializes a replicated data structure by calling `init_func` on each node. The parameter `init_func` should take two arguments. The first is an absolute address of the replicated data structure on the node with the ID given by the second argument.

Notes:

Example: The following code snippet demonstrates how to initialize each instance of a replicated variable to a different value.

```
#include <memoryweb.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
replicated long foo;

replicated
struct info
{
    long x;
    long y;
} info;

void init_info(void *m, long node)
{
    struct info * i = (struct info*) m;
    i->x = node;
    i->y = 5*node + 4;
    return;
}

int main()
{
    long ** array = mw_malloc2d(NUM_NODES(), sizeof(long));
```

```
    mw_replicated_init_generic(&info, init_info);
    long i;
    for(i = 0; i != NUM_NODES(); i++)
    {
    MIGRATE(array[i]);
    printf("info.x = %ld and info.y = %ld on node %ld\n", info.x, info.y, i);
    }
    printf("info.x = %ld and info.y = %ld on node 5\n",
       ((struct info*)mw_get_localto(&info, array[5]))->x,
       ((struct info*)mw_get_localto(&info, array[5]))->y);
    return 0;
}
```

3.4.6 mw_get_nth

```
void * mw_get_nth(void * repl_addr, long n)
```

Return Value: Returns a pointer to the nth instance of the ptr data structure.

Arguments:

- repl_addr: pointer to the replicated data structure
- n: index indicating which instance of the data structure

Functionality: This function takes as input a pointer to a replicated data structure and returns a pointer to the nth instance of that data structure.

Notes:

Example:

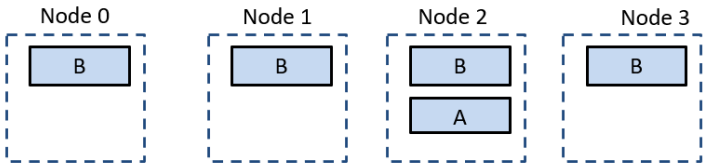


Figure 3.5: Data to illustrate use of `mw_get_nth` and `mw_get_localto`.

The following code demonstrates how to use the `mw_get_nth` function. It returns a pointer to the instance of the replicated variable `B` on node 1 as shown in Figure 3.5.

```
replicated long B;
long * Bi = mw_get_nth(&B, 1);
```

3.4.7 mw_get_localto

```
void * mw_get_localto(void * repl_addr, void * localpointer)
```

Return Value: Returns a void * to the instance of the replicated variable located on the same node as the address referenced by localpointer.

Arguments:

- repl_addr: a pointer to a replicated data structure.
- localpointer: pointer to a second data structure

Functionality: The function returns a pointer to the instance of the replicated data structure that is located on the same node as the address referenced by localpointer.

Notes:

Example: The following code demonstrates how to use the `mw_get_localto` function to access a specific instance of a replicated data structure as shown in Figure 3.5. It returns a pointer to the instance of B that is located on the same node as A.

```
replicated long B;
...
long * localB = (long *) mw_get_localto(&B, &A);
```

Chapter 4

Architecture Specific Operations

Intrinsic functions are constructs that allow the programmer to directly access particular instructions provided by the machine. The following tables list the available intrinsic functions. These functions can be accessed by including the library header file `memoryweb.h`.

4.1 Atomic Operations

A full set of atomic operations on 64-bit data are provided as shown in Table 4.1. The atomics ensure that an operation completes without the possibility of another thread accessing the data element during the operation. These operations take as input a pointer to the memory location to be atomically updated and an additional argument. There are three "flavors" of atomic operations that specify what result is to be written to memory and what result is to be returned by the function, specified as follows:

- **M**: the result of the operation is written to the memory location and returned.
- **S**: the second argument is written to the memory location and the result of the operation is returned.
- **MS**: the original value at the memory location is returned, and the result is written to the memory location.

For example, `ATOMIC_ADDM(&A, 2)` loads the value of A from memory, adds 2, writes the result to A in memory, and returns the result. Executing `ATOMIC_ADDS(&A, 2)` loads the value of A from memory, adds 2, returns the result, and writes 2 to A in memory. Finally, `ATOMIC_ADDMS(&A, 2)` loads the value of A in from memory, adds 2, writes the result to A in memory and returns the original value of A loaded from memory.

IMPORTANT NOTE: the first argument MUST be a pointer to a 64-bit data type. The atomic instructions load and store 64-bit words, so calling the functions on a data-type smaller than 64-bits is not supported.

Table 4.1: Atomic Arithmetic Operation Intrinsics

long ATOMIC_ADDM(long *, long)
long ATOMIC_ANDM(long *, long)
long ATOMIC_ORM(long *, long)
long ATOMIC_XORM(long *, long)
long ATOMIC_MAXM(long *, long)
long ATOMIC_MINM(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is returned and also written to the location specified by the first argument. The entire function is performed atomically.

long ATOMIC_ADDS(long *, long)
long ATOMIC_ANDS(long *, long)
long ATOMIC_ORS(long *, long)
long ATOMIC_XORS(long *, long)
long ATOMIC_MAXS(long *, long)
long ATOMIC_MINS(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is returned and the value of the second argument is written to memory at the location specified by the first argument. The entire function is performed atomically.

long ATOMIC_ADDMS(long *, long)
long ATOMIC_ANDMS(long *, long)
long ATOMIC_ORMS(long *, long)
long ATOMIC_XORMS(long *, long)
long ATOMIC_MAXMS(long *, long)
long ATOMIC_MINMS(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is written to memory and the original value at the memory location addressed by the first argument is returned. The entire function is performed atomically.

4.2 Remote Updates

Remote updates are atomic updates to 64-bit data in memory that do NOT cause a thread migration. Instead, they send to the remote memory location a packet that contains data and the operation to be performed. Remote updates to the same address from the same thread are guaranteed to be performed in order. Remote updates to the same address from different threads or to different addresses from the same thread may occur in any order. Remote updates do NOT return a result; however they send an acknowledgement (ACK) that indicates that the update has been accepted by the memory for processing. A thread is not allowed to migrate until all ACKs have returned. This ensures that (a) the ACK can "find" the thread and (b) that the thread cannot access the data before the remote update has completed. An explicit FENCE operation can also be used to block the thread until all ACKs have returned. The full set of remote update intrinsics are shown in Table 4.2.

IMPORTANT NOTE: the first argument MUST be a pointer to a 64-bit data type. The remote instructions load and store 64-bit words, so calling the functions on a data-type smaller than 64-bits is not supported.

Table 4.2: Remote Update Intrinsics

void REMOTE_ADD(long *, long)
void REMOTE_AND(long *, long)
void REMOTE_OR(long *, long)
void REMOTE_XOR(long *, long)
void REMOTE_MAX(long *, long)
void REMOTE_MIN(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is written to memory at the location specified by the first argument. A remote packet is generated to travel to the specified memory location and perform the function atomically.

void FENCE(void)
The thread performing the FENCE stalls until there are no outstanding acknowledges from previously executed remote memory operations. This instruction will otherwise act as a NOOP (No Operation) and makes no changes to the state of the thread.

4.3 Swaps

The architecture provides two types of atomic swap as detailed in Table 4.3.

IMPORTANT NOTE: the first argument MUST be a pointer to a 64-bit data type. The swap instructions load and store 64-bit words, so calling the functions with a data-type smaller than 64-bits is not supported.

Table 4.3: Atomic Swap Intrinsics

long ATOMIC_SWAP(long *, long)
This function writes the second argument to the location addressed by the first argument. The original value in memory is returned. The entire operation is performed atomically.

long ATOMIC_CAS(long *, long, long)
This function compares the third argument to the value addressed by the first argument. If equal, the second argument is written to memory. The original memory value is returned regardless of the result of comparison. The entire operation is performed atomically.

4.4 Specialized Operations

Two specialized operations are provided as shown in Table 4.4.

IMPORTANT NOTE: the second argument to `POPCNT` MUST be a pointer to a 64-bit data type. The swap instructions load and store 64-bit words so calling the functions with a data-type smaller than 64-bits is not supported.

Table 4.4: Specialized Intrinsic Operations

long POPCNT(long, long *)
Computes the popcount: The 64 bit word in the second argument is read and the number of 1 bits in the word counted. This value is added to the first argument and the result is returned.

unsigned long PRIORITY(unsigned long)
Computes the 6 bit priority encode on the argument, returning the result. The priority encode is the bit position of the highest number bit that is nonzero.

4.5 Thread Management

A set of intrinsics provide thread management capabilities as detailed in Table 4.5.

Table 4.5: Thread Management

void RESIZE()
This intrinsic resizes the thread to carry only the currently live registers. It is used by the compiler and programmer before a possible migration to reduce the thread size and thus the bandwidth requirements.

void RESCHEDULE()
This intrinsic places the thread at the end of the Run Queue to allow a new thread to be scheduled in the core. It can be used as a part of a busy wait loop (e.g. while waiting for a lock to become available). **This should not be used if you hold a lock.**

4.6 System Information

Table 4.6 lists the set of intrinsics that are used to provide system specific information.

Table 4.6: System Query Intrinsics

unsigned long CLOCK(void)
Returns a 64-bit real time counter for the clock on the current node. See below for additional information on using this intrinsic for timing comparisons.

unsigned long THREAD_ID(void)
Returns a special value called a thread ID that can be used to uniquely identify a thread for debugging.

unsigned long NODE_ID(void)
Returns the current node ID where the thread is executing.

unsigned long NUM_NODES(void)
Returns the total number of nodes in the system.

unsigned long BYTES_PER_NODE(void)
Returns the total number of bytes of memory in each node.

unsigned long GCS_PER_NODE(void)
Returns the number of Gossamer Cores (GCs) on the node.

The `CLOCK()` intrinsic returns a 64-bit real time counter that can be used for timing comparisons in the simulator and on the hardware. When using `CLOCK()` for timing comparisons, it is important to read the start and stop values from the same node since counters on different nodes may vary. Given start and stop counter values, the clock rate can be used to calculate elapsed time.

Section 7.3 has information on using the `starttiming()` function on the simulator, which is treated as a NOOP on the hardware. Alternatively, the `time.h` functions, such as `clock_gettime()`, are supported and use the SC clock. However, the time involved in doing a system call to the SC to get the time may be significant for small programs, so `CLOCK()` is more accurate. `CLOCK()` is the preferred mechanism for timing on the hardware.

It is recommended that the variables used to store the `CLOCK()` value are declared as `volatile`. Otherwise, the compiler may perform optimizations on those variable, and the results of such program may be invalid.

Chapter 5

Toolchain and Libraries

A brief note regarding this section: Lucata was previously known as Emu Technology and some of our libraries continue to maintain the Emu name. We expect to update naming in a future release of the toolchain and this guide.

5.1 Toolchain

The toolchain and libraries for the Pathfinder system support program development in C/C++/Cilk. The current toolchain is based on the OpenCilk implementation of Cilk which is currently developed and maintained by MIT (cilk.mit.edu).

The `emu-cc` program in the toolchain installation manages the compilation of programs for the Pathfinder system. The interface to `emu-cc` is similar to that of GCC. Note that this toolchain is a cross-compiler in that it runs on x86 systems and generates executable programs for the Pathfinder system.

In this guide, the default extension for executables intended to run on the Pathfinder is `.mwx`.

5.1.1 C Support

The Pathfinder toolchain supports the C language and has ported much of the musl C library. The toolchain currently supports the most common functionality from the following categories:

- stdio.h
- math.h
- string.h
- stdlib.h
- rand.h
- libgen.h
- stdarg.h

- search.h
- assert.h

The following are not supported and/or known to fail. In particular, pthread functionality is almost completely unsupported at this time:

- pthread.h
- wctype.h
- wchar.h
- stdio.h wide character conversions e.g. btowc
- sys.h: most of these are not yet tested
- spawn.h: not supported
- semphores.h: not supported
- unistd.h

NOTE: The `memalign()` function is currently known to be broken; when called a runtime service will be called instead and the program will exit.

Testing of the C library is an ongoing process and new features are added as needed.

5.1.2 C++ Support

The toolchain currently supports basic C++, such as classes and objects, inheritance, polymorphism, and templates, along with the most common functionality from the C++ standard library including:

- Containers: array, deque, forward_list, list, map, queue set, stack, unordered_map, unordered_set, vector
- General: algorithm, chrono, iterator, tuple
- Language support: limits, new, typeinfo
- Numerics: valarray, numeric, ratio, random
- Strings
- Streams: IO and file streams

Additional work is still needed in the following areas **S**KK: PLEASE REVIEW:

- C++ exception handling (try/catch/throw) is not yet supported. It is turned off via a compiler flag in the current toolchain.
- C++ containers (e.g. vector, list, etc.) are single threaded and allocated on a single node.

5.2 GNU Multiple-precision Library

The GNU Multiple Precision Arithmetic (GMP) library is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. The signed integer arithmetic portion of the GMP library is provided with the Pathfinder toolchain.

5.3 Emu C utilities library

5.3.1 Overview

Parallelism on the Pathfinder system is essential to achieving good performance. Since Cilk creates threads at the level of a function, implementing patterns such as recursive spawn trees or a two-level remote spawn require a lot of boilerplate code and intermediary functions. The `emu_c_utils` library implements most of these common patterns efficiently as library calls.

The programmer provides a pointer to a function that handles a single array slice, and the implementation spawns threads across the entire system to process each slice in parallel. Local variables can be made available to each worker function using the varargs calling convention.

The parallel apply functions accept a grain size argument. In order to avoid overwhelming the system with too many threads, it is usually best to spawn just enough threads to saturate the GC's on each node. The `LOCAL_GRAIN()` function will calculate a grain size from the array length to spawn exactly enough threads to saturate a single node. `GLOBAL_GRAIN()` will do the same for the entire system. `GLOBAL_GRAIN_MIN()` and `LOCAL_GRAIN_MIN()` accept an additional grain size argument to avoid spawning too many threads for a small array.

Table 5.1: Overview of utility functions

Function	Functionality
Working with local arrays	
`emu_local_for()`	Applies a function to a range in parallel.
`emu_local_for_set_long()`	Initializes an array of `long`s to a single value in parallel.
`emu_local_for_copy_long()`	Copies an array of `long`s in parallel.
`emu_sort_local()`	Sorts a range in parallel with a custom comparator (like qsort).
`emu_memcpy()`	Copies a range of bytes (like memcpy).
`LOCAL_GRAIN_MIN()`	Chooses a grain size based on the array size.
Working with distributed (striped) arrays	
`emu_1d_array_apply()`	Implements a distributed parallel for over a `malloc1dlong()` array
`emu_1d_array_reduce_sum()`	Implements a distributed parallel reduction over a `malloc1dlong()` array
`GLOBAL_GRAIN_MIN()`	Chooses a grain size based on the array size.
Working with distributed (chunked) arrays	
`emu_chunked_array_replicated_new()`	Allocates and initializes an `emu_chunked_array`, returning a replicated pointer to the data structure.
`emu_chunked_array_replicated_free()`	Frees a pointer allocated with `emu_chunked_array_replicated_new`.
`emu_chunked_array_replicated_init()`	Initializes an `emu_chunked_array` struct.
`emu_chunked_array_replicated_deinit()`	Deallocates the array associated with a `emu_chunked_array` struct.
`emu_chunked_array_index()`	Returns a pointer to the `i`th element within the array.
`emu_chunked_array_size()`	Returns the number of elements in the array.
`emu_chunked_array_apply()`	Implements a distributed parallel for over an `emu_chunked_array`.
`emu_chunked_array_set_long()`	Initializes an array of longs stored within an `emu_chunked_array` to a single value in parallel.
`emu_chunked_array_reduce_sum()`	Implements a distributed parallel reduction over an `emu_chunked_array`.
`emu_chunked_array_from_local()`	Implements a scatter operation from a local array to an `emu_chunked_array`.
`emu_chunked_array_to_local()`	Implements a gather operation from an `emu_chunked_array` to a local array.
Timing hooks	
`hooks_region_begin()`	Marks the beginning of a region of interest.
`hooks_region_end()`	Marks the end of a region of interest.
`hooks_set_attr_u64()`	Records an unsigned integer value for the current region.
`hooks_set_attr_i64()`	Records a signed integer value for the current region.
`hooks_set_attr_f64()`	Records a double-precision floating point value for the current region.
`hooks_set_attr_str()`	Records a string value for the current region.
`hooks_set_active_region()`	Sets which region will be active during the current execution

5.3.2 Working with local arrays

These functions constitute an alternative to `cilk_for`, implemented without compiler support.

5.3.2.1 emu_local_for

```
void
emu_local_for(
    long begin,
    long end,
    long grain,
    void (*worker)(long begin, long end, va_list args),
    ...
);
```

Arguments:

- begin: beginning of the iteration space (usually 0)
- end: end of the iteration space (usually array length)
- grain: Minimum number of elements to assign to each thread
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function should go from `begin` to `end` with a stride of 1.
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Applies a function to a range in parallel. These loops can be replaced with `cilk_for` once it is working.

Notes:

- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
long * x = malloc(n * sizeof(long));

void worker(long begin, long end, va_list args)
{
    long * x = va_arg(args, long*);
    long b = va_arg(args, long);
```

```
    for (long i = begin; i < end; ++i) {
        x[i] += b;
    }
}
emu_local_for(0, n, LOCAL_GRAIN_MIN(n, 64), worker, x, b);
```

5.3.2.2 emu_local_for_set_long

```
void
emu_local_for_set_long(long * array, long n, long value);
```

Arguments:

- dst: Pointer to array
- n: Array length
- value: Value to set

Functionality: Sets each value of `array` to `value` in parallel.

Notes:

- Initializing striped arrays with this function will be inefficient, use `emu_1d_array_apply` for that instead.

Example:

```
long n = 1024;
long * a = malloc(n * sizeof(long));
// Initialize elements of a to zero
emu_local_for_set_long(a, n, 0);
```

5.3.2.3 emu_local_for_copy_long

```
void
emu_local_for_copy_long(long * dst, long * src, long n);
```

Arguments:

- dst: Pointer to destination array
- src: Pointer to source array
- n: Array length

Functionality: Copies `src` to `dst` in parallel.

Notes:

- If `src` and `dst` are on different nodes, then it is most efficient to call this function on the source node, so that remote writes can be used to send the data.
- Copying striped arrays with this function will be inefficient, use `emu_1d_array_apply` for that instead.

Example:

```
long n = 1024;
long * a = malloc(n * sizeof(long));
long * b = malloc(n * sizeof(long));
// Copy a to b
emu_local_for_copy_long(b, a, n);
```

5.3.2.4 emu_memcpy

```
void *
emu_memcpy(void * dst, void * src, size_t n)
```

Arguments:

- dst: Pointer to destination array
- src: Pointer to source array
- n: Number of bytes to copy

Functionality: The parallel equivalent of `memcpy`.

Notes:

Example:

```
size_t n = 10000;

// Allocate two buffers
long * array1 = malloc(sizeof(long) * n); assert(array1);
// Give the second array a weird offset
long * array2 = malloc(sizeof(long) * n); assert(array2);

// Initialize to known values
for (long i = 0; i < n; ++i) {
    array1[i] = i;
    array2[i] = 0;
}

// Do the copy in parallel
emu_memcpy(array2, array1, n * sizeof(long));

// Check
for (long i = 0; i < n; ++i) {
    TEST_ASSERT_EQUAL(i, array2[i]);
}

// Clean up
free(array1);
free(array2);
```

5.3.2.5 LOCAL_GRAIN_MIN

```
static inline long
LOCAL_GRAIN_MIN(long n, long min_grain);
```

Return Value: Returns a grain size suitable for a loop processing `n` items on a single node.

Arguments:

- n: Number of elements in the array.
- min_grain: Minimum number of elements per thread.

Functionality: This function is designed to help prevent cases of too many threads executing on a single node as may happen with a large arrray and a small grain size. By controlling the grain size, the number of threads can be controlled.

Notes:

Example:

5.3.3 Working with distributed (striped) arrays

As discussed in Section 3.3.2, an array allocated with `mw_malloc1dlong` will be striped across nodes in a round-robin fashion. The most efficient way to access this type of array is with a stride of `NUM_NODES()`, such that each thread remains on a single node. These functions implement a two-level spawn tree and split the array into striped slices for each worker function.

5.3.3.1 emu_1d_array_apply

```
void
emu_1d_array_apply(
    long * array,
    long size,
    long grain,
    void (*worker)(long * array, long begin, long end, va_list args),
    ...
);
```

Return Value:

Arguments:

- array: Pointer to striped array allocated with `malloc1dlong()`.
- size: Length of the array (number of elements)
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function should go from `begin` to `end` and have a stride of `NUM_NODES()`. Each worker function will be assigned elements on a single node.
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel for over a `malloc1dlong()` array.

Notes:

- Each worker function will be assigned an iteration space on a single node. These should use striped indexing (see example)!
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
long * x = malloc1dlong(n);

void worker(long * array, long begin, long end, va_list args)
{
    long * x = array;
    long b = va_arg(args, long);
    for (long i = begin; i < end; i += NUM_NODES()) {
        x[i] += b;
    }
}
emu_1d_array_apply(x, n, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.3.3.2 emu_1d_array_reduce_sum

```
long
emu_1d_array_reduce_sum(
    long * array,
    long size,
    long grain,
    void (*worker)(long * array, long begin, long end, long * sum, va_list args),
    ...
);
```

Return Value: Returns the sum of all the elements in the array.

Arguments:

- array: Pointer to striped array allocated with `malloc1dlong()`.
- size: Length of the array (number of elements)
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function should go from `begin` to `end` and have a stride of `NUM_NODES()`. Each worker function will be assigned elements on a single node.
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel reduce over a `malloc1dlong()` array.

Notes:

- Each worker function will be assigned an iteration space on a single node. Use striped indexing (see example)!
- The `sum` argument to the worker function points to a local accumulation variable that is shared by all the threads on a given node. An atomic function such as `REMOTE_ADD` should be used to update this sum. Additionally, it may be more efficient for each thread to calculate a local sum before adding to the node-local sum. See the example for more details.
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
```

```
long * x = malloc1dlong(n);

void worker(long * array, long begin, long end, long * sum, va_list args)
{
    long * x = array;
    long b = va_arg(args, long);
    long local_sum = 0;
    for (long i = begin; i < end; i += NUM_NODES()) {
        local_sum += x[i] * b;
    }
    REMOTE_ADD(sum, local_sum);
}
long sum = emu_1d_array_reduce_sum(x, n, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.3.3.3 GLOBAL_GRAIN_MIN

```
static inline long
GLOBAL_GRAIN_MIN(long n, long min_grain);
```

Return Value: Returns a grain size suitable for a loop processing `n` items on the entire system.

Arguments:

- n: Number of elements in the array.
- min_grain: Minimum number of elements per thread.

Functionality: This function is designed to help prevent cases of too many threads executing on the system as may happen with a large arrray and a small grain size. By controlling the grain size, the number of threads can be controlled.

Notes:

Example:

5.3.4 Working with distributed (chunked) arrays

The term "chunked" (or "blocked") refers to a particular strategy for using `mw_malloc2d()`. The first dimension of the 2D array now holds pointers to a chunk of memory on each node instead of a pointers to each element. Unlike a striped array, elements with consecutive indices have spatial locality.

```
typedef struct point { long x; long y;} point;
long n = 1024;
point * last_point;
point ** striped_array = mw_malloc2d(n, sizeof(point));
last_point = striped_array[n - 1];
point ** chunked_array = mw_malloc2d(NUM_NODES(), n/NUM_NODES() * sizeof(point));
last_point = &chunked_array[NUM_NODES() - 1][n/NUM_NODES() - 1];
```

Working with chunked arrays introduces several complications:

- Converting logical array indexes into 2D index operations (i.e. `array[i / N][i % N]`)
- Keeping track of the block size `N` and per-element size to use in each indexing operation.
- Using `mw_arrayindex` for indexing operations to avoid migrating where possible.
- Initializing and replicating the array metadata to all nodes.

The struct `emu_chunked_array` encapsulates this logic within an abstract data type. It holds the array metadata in replicated storage, and functions are provided to calculate indices and apply functions to the array in parallel.

```
typedef struct point { long x; long y; } point;
long n = 1024;
emu_chunked_array *point_array = emu_chunked_array_replicated_new(n, sizeof(point));
point * last_point = emu_chunked_array_index(point_array, n-1);
```

5.3.4.1 emu_chunked_array_replicated_new

```
emu_chunked_array *
emu_chunked_array_replicated_new(
    long num_elements,
    long element_size
);
```

Return Value: A relative pointer to a replicated `emu_chunked_array`.

Arguments:

- num_elements: Number of elements in the array
- element_size `sizeof()` each array element

Functionality: Allocates replicated storage for an `emu_chunked_array`, initializes the storage on each node, and returns a relative pointer to the data structure. There will be `num_elements` elements with `element_size` bytes each.

Notes:

- Use `emu_chunked_array_replicated_free()` to free the array.

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
emu_chunked_array_replicated_free(x);
```

5.3.4.2 emu_chunked_array_replicated_free

```
void emu_chunked_array_replicated_free(emu_chunked_array * self);
```

Return Value:

Arguments:

- self: Pointer initialized with `emu_chunked_array_replicated_new`

Functionality: Frees a pointer allocated with `emu_chunked_array_replicated_new`.

Notes:

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
emu_chunked_array_replicated_free(x);
```

5.3.4.3 emu_chunked_array_replicated_init

```
void
emu_chunked_array_replicated_init(
    emu_chunked_array * self,
    long num_elements,
    long element_size
);
```

Return Value:

Arguments:

- self: Pointer to uninitialized struct, which MUST be located in replicated storage.
- num_elements: Number of elements in the array
- element_size: `sizeof()` each array element

Functionality: Initializes an `emu_chunked_array`. There will be `num_elements` elements with `element_size` bytes each. Array metadata will be replicated onto each node.

Notes:

- The first argument MUST point to replicated storage, i.e. a `replicated` global variable.
- Use `emu_chunked_array_replicated_deinit()` to free the array.
- `emu_chunked_array_replicated_new` is the dynamic-allocation equivalent of this function, and is generally more convenient since it handles the replicated allocation for you.

Example:

```
long n = 1024;
replicated emu_chunked_array x;
emu_chunked_array_replicated_init(&x, n, sizeof(long));
emu_chunked_array_replicated_deinit(&x);
```

5.3.4.4 emu_chunked_array_replicated_deinit

```
void emu_chunked_array_replicated_deinit(emu_chunked_array * self);
```

Return Value:

Arguments:

- self: Pointer to struct initialized with `emu_chunked_array_replicated_init`

Functionality: Deallocates the array associated with a `emu_chunked_array` struct.

Notes:

- Use only in conjunction with `emu_chunked_array_replicated_init`. If you created the array with `emu_chunked_array_replicated_new`, use `emu_chunked_array_replicated_free` instead.

Example:

```
long n = 1024;
replicated emu_chunked_array x;
emu_chunked_array_replicated_init(&x, n, sizeof(long));
emu_chunked_array_replicated_deinit(&x);
```

5.3.4.5 emu_chunked_array_index

```
static inline void *
emu_chunked_array_index(emu_chunked_array * self, long i);
```

Return Value:

Arguments:

- self: Pointer to `emu_chunked_array`
- i: Index of requested element

Functionality: Returns a pointer to an element within the array.

Notes:

- Remember to cast the returned value to the appropriate type before dereferencing. It may be helpful to define a helper macro to make the syntax less verbose (see example).

Example:

```
long n = 1024;
emu_chunked_array * array = emu_chunked_array_replicated_new(n, sizeof(long));
#define X(I) *(long*)emu_chunked_array_index(array, I)
X(10) = X(20) + X(30);
```

5.3.4.6 emu_chunked_array_size

```
void
emu_chunked_array_size(emu_chunked_array * array);
```

Return Value: Returns the number of elements in the array (same as `num_elements` argument that was passed during initialization).

Arguments:

- array: Pointer to initialized `emu_chunked_array`

Functionality: Returns the number of elements in the array (same as `num_elements` argument that was passed during initialization).

Notes:

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
assert(emu_chunked_array_size(x) == n);
```

5.3.4.7 emu_chunked_array_apply

```
void
emu_chunked_array_apply(
    emu_chunked_array * array,
    long grain,
    void (*worker)(emu_chunked_array * array, long begin, long end, va_list args),
    ...
);
```

Return Value:

Arguments:

- array: Pointer to `emu_chunked_array`
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function is responsible for array elements from `begin` to `end` with a stride of 1. Because each worker function will be assigned elements on a single node, it is more efficient to call `emu_chunked_array_index` once before the loop, and do linear indexing from that pointer (see example).
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel for over an `emu_chunked_array`.

Notes:

- Each worker function will be assigned an iteration space on a single node. After indexing, you may treat the local slice like a contiguous array (see example).
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));

void
worker(emu_chunked_array * array, long begin, long end, va_list args)
{
    long b = va_arg(args, long);
```

```
    long * x = emu_chunked_array_index(array, begin);

    for (long i = 0; i < end-begin; ++i) {
        x[i] += b;
    }
}
emu_chunked_array_apply(x, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.3.4.8 emu_chunked_array_set_long

```
void
emu_chunked_array_set_long(emu_chunked_array * array, long value);
```

Return Value:

Arguments:

- array: `emu_chunked_array` to initialize
- value: Set each element to this value

Functionality: Initializes each element of the array to a single value, in parallel.

Notes:

- The array must have been initialized to store a long datatype.

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
emu_chunked_array_set_long(x, 0);
emu_chunked_array_replicated_free(x);
```

5.3.4.9 emu_chunked_array_reduce_sum

```
long
emu_chunked_array_reduce_sum(
    emu_chunked_array * array,
    long grain,
    void (*worker)(emu_chunked_array * array, long begin, long end,
          long * partial_sum, va_list args),
    ...
);
```

Return Value: Returns the sum of all the elements in the array.

Arguments:

- array: Pointer to `emu_chunked_array`
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function is responsible for array elements from `begin` to `end` with a stride of 1. Because each worker function will be assigned elements on a single node, it is more efficient to call `emu_chunked_array_index` once before the loop, and do linear indexing from that pointer (see example).
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel reduction over an `emu_chunked_array` of `long` type.

Notes:

- This function only makes sense if the `emu_chunked_array` was initialized to store elements of type `long`.
- Each worker function will be assigned an iteration space on a single node. After indexing, you may treat the local slice like a contiguous array (see example).
- The `partial_sum` argument to the worker function points to a local accumulation variable that is shared by all the threads on a given node. An atomic function such as `REMOTE_ADD` should be used to update this sum. Additionally, it may be more efficient for each thread to calculate a local sum before adding to the node-local sum. See the example for more details.
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));

void
worker(emu_chunked_array * array, long begin, long end,
       long * partial_sum, va_list args)
{
    long b = va_arg(args, long);
    long * x = emu_chunked_array_index(array, begin);
    long local_sum;
    for (long i = 0; i < end-begin; ++i) {
        local_sum += x[i] * b;
    }
    REMOTE_ADD(partial_sum, local_sum);
}
long sum = emu_chunked_array_reduce_sum(x, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.3.4.10 emu_chunked_array_from_local

```
void
emu_chunked_array_from_local(emu_chunked_array * self, void * local_array);
```

Return Value:

Arguments:

- self: Pointer to `emu_chunked_array`
- local_array: Pointer to local array

Functionality: Copies elements from a local array to an emu_chunked_array in parallel.

Notes:

- The chunked array must have already been initialized with the same number of elements as the local array, and the same element size.

Example:

```
long n = 1024;

// Allocate a local and a chunked array
long * x = malloc(n * sizeof(long));
emu_chunked_array * y = emu_chunked_array_replicated_new(n, sizeof(long));

// Assign index to each value of local array
for (long i = 0; i < n; ++i) {
    x[i] = i;
}
// Set chunked array to invalid values
emu_chunked_array_set_long(y, -1);

// Do the scatter
emu_chunked_array_from_local(y, x);

// Check
for (long i = 0; i < n; ++i) {
    long val = *(long*)emu_chunked_array_index(y, i);
    TEST_ASSERT_EQUAL(i, val);
}

// Clean up
free(x);
emu_chunked_array_replicated_free(y);
```

5.3.4.11 emu_chunked_array_to_local

```
void
emu_chunked_array_to_local(emu_chunked_array * self, void * local_array);
```

Return Value:

Arguments:

- self: Pointer to `emu_chunked_array`
- local_array: Pointer to local array

Functionality: Copies elements from an emu_chunked_array to a local array in parallel.

Notes:

- The local array must have already been allocated with the same number of elements as the chunked array, and the same element size.

Example:

```
long n = 1024;

// Allocate a local and a chunked array
long * x = malloc(n * sizeof(long));
emu_chunked_array * y = emu_chunked_array_replicated_new(n, sizeof(long));

// Assign index to each value of chunked array
emu_chunked_array_apply(y, GLOBAL_GRAIN_MIN(n, 64), assign_index_worker);

// Set local array to invalid
for (long i = 0; i < n; ++i) {
    x[i] = -1;
}

// Do the gather
emu_chunked_array_to_local(y, x);

// Check
for (long i = 0; i < n; ++i) {
    TEST_ASSERT_EQUAL(i, x[i]);
}

// Clean up
free(x);
emu_chunked_array_replicated_free(y);
```

5.3.5 Timing Hooks

These functions implement a timer subsystem suitable for performance analysis, along with tools to annotate the output and control the level of detail in the simulator.

5.3.5.1 hooks_region_begin; hooks_region_end

```
void
hooks_region_begin(const char* name);

double
hooks_region_end(const char* name);
```

These functions work together to provide timing information for a program section of interest.

Return Value: For `hooks_region_end`: the time elapsed since the call to `hooks_region_begin`, in milliseconds.

Arguments:

- name: Name of the region

Functionality: `hooks_region_begin` marks the beginning of a region of interest, and takes the following actions:

- Calls `starttiming()` to trigger detailed timing mode when running within the simulator.
- Starts a timer by calling the `CLOCK()` macro on node 0.

`hooks_region_end` marks the end of a region of interest, and takes the following actions:

- Stops the timer by calling the `CLOCK()` macro on the same node it was called on initially.
- Outputs information about this region in JSON format, including the region name, the time elapsed in milliseconds, the number of ticks elapsed, and any attributes that were set within the region with `hooks_set_attr` functions.

Notes:

- Each call to `hooks_region_begin()` should be matched with a call to `hooks_region_end()`. Regions may not be nested, however there may be many regions within the same program, and the same region name can be reused multiple times.
- Since software cannot currently detect the clock frequency, timing accuracy depends on the environment variable `CORE_CLK_MHZ` being set correctly. The number of ticks elapsed, however, will be correct regardless.
- The timing output can be directed to an arbitrary file by setting the environment variable `HOOKS_FILENAME`.

Example:

```
// Initialization section, will not be timed
long n = 1024;
long * x = malloc1dlong(n);
for (long i = 0; i < n; ++i) { x[i] = i; }
// Region of interest, will be timed
hooks_region_begin("sum");
long sum = 0;
for (long i = 0; i < n; ++i) { sum += x[i]; }
double time_ms = hooks_region_end();
// Output: {"region_name": "sum", "time_ms": 3.14, "ticks": 1234567}
```

5.3.5.2 hooks_set_active_region

```
void
hooks_set_active_region(const char* name);
```

Return Value:

Arguments:

- name: Name of the region

Functionality: Sets which region name will trigger `starttiming()`. All "inactive" regions will not call starttiming. If this function is never called, all regions will be active. Only the simulator pays attention to which regions are active/inactive, the hardware executes all regions normally.

Notes:

- Until `stoptiming()` is implemented in the simulator, it is impossible to switch back to functional simulation for regions after the active region.
- You may wish to set the active region directly from an environment variable at the start of your program like this:

  ```
  // Set active region from environment variable
  hooks_set_active_region(getenv("HOOKS_ACTIVE_REGION"));
  ```

Example:

```
// Set active region
hooks_set_active_region("sum");

// Initialization section, will be simulated in functional mode
hooks_region_begin("init");
long n = 1024;
long * x = malloc1dlong(n);
for (long i = 0; i < n; ++i) { x[i] = i; }
hooks_region_end();

// Region of interest, will be simulated in timing mode
hooks_region_begin("sum");
long sum = 0;
for (long i = 0; i < n; ++i) { sum += x[i]; }
hooks_region_end();
```

5.3.5.3 hooks_set_attr

```
void hooks_set_attr_u64(const char * key, uint64_t value);
void hooks_set_attr_i64(const char * key, int64_t value);
void hooks_set_attr_f64(const char * key, double value);
void hooks_set_attr_str(const char * key, const char* value);
```

Return Value:

Arguments:

- key: Name to be associated with the value in the output
- value: Value to print after the end of this region

Functionality: Adds a custom key/value pair to the JSON output printed after a region ends. This can be used to annotate each piece of timing data with additional information, such as the array size, version number, or other configuration parameters.

Notes: Attributes must be set before a region or within a region; `hooks_region_end()` resets all attributes.

Example:

```
long n = 1024;
long * x = malloc1dlong(n);
for (long i = 0; i < n; ++i) { x[i] = i; }
hooks_set_attr_i64("N", n);
hooks_set_attr_str("version", "2.0rc3");
hooks_region_begin("sum");
long sum = 0;
for (long i = 0; i < n; ++i) { sum += x[i]; }
hooks_region_end();
// Output: {"region_name": "sum", "time_ms": 3.14, "ticks": 1234567, "N": 1024,
// "version": "2.0rc3"}
```

5.3.6 Building

The following examples assume that the toolchain is installed in the default location, `/usr/local/emu`.

5.3.6.1 Targeting Pathfinder

The emu_c_utils library is incorporated into the toolchain but is not included by default. The following item will need to be passed to `emu-cc` to be able to utilize the library:

```
-lemu_c_utils
```

If using CMake, the following line can be added to CMakeLists.txt:

```
link_libraries(emu_c_utils)
```

5.3.6.2 x86 toolchain

The toolchain also provides x86 based libraries to allow for native x86 compilation; this allows developers to quickly build and test codes using familiar IDE's and debuggers prior to running on `emusim` and/or Pathfinderhardware. When running an application targeting the x86 architecture, the application treats the system as having a single node and Gossamer Core.

To compile directly for the x86 architecture a compiler with Cilk support is required. For GCC, this is generally the GCC5, GCC6, and GCC7 versions as support for Cilk in GCC was removed starting with GCC 8.

It will be necessary to pass the following flags to GCC: `-lcilkrts` and `-fcilkplus`.

The include and library paths for the x86 implementation are as follows:

- include: `/usr/local/emu/x86/include/`
- library: `/usr/local/emu/x86/lib/`

Add the following lines to your Makefile to link with the x86 libraries for memoryweb and emu_c_utils:

```
CFLAGS+=-I/usr/local/emu/x86/include
LDFLAGS+=-lemu_c_utils -L/usr/local/emu/x86/lib
```

Or, with CMake:

```
include_directories(/usr/local/emu/x86/include)
link_libraries(/usr/local/emu/x86/lib/libemu_c_utils.a)
```

5.4 Emu Cilk Reducers

A Cilk Reducer is an object that allows several threads to efficiently combine partial results into a single value while avoiding data races and preserving serial semantics.

THESE ARE NOT FUNCTIONAL IN THE CURRENT RELEASE. It is expected that these Reducers will be available in a future release.

The toolchain currently only provides a subset of all possible reducers, implemented using remote atomics for 64-bit integers. One view is allocated on each node in replicated memory, so that threads never need to migrate in order to reduce with a view.

A `PerNodeReducer` is instantiated with a single template argument: the Monoid which encapsulates the the view type with corresponding identity and reduce operators. The following Monoid implementations are provided:

```
cilk::op_add<long> // Signed addition
cilk::op_and<long> // Bitwise AND
cilk::op_or<long>  // Bitwise OR
cilk::op_xor<long> // Bitwise XOR
cilk::op_min<long> // Minimum
cilk::op_max<long> // Maximum
```

Example:

```
#include <cilk/cilk.h>
#include <cilk/reducer.h>
#include <vector>

int main(int argc, char* argv[])
{
    // Allocate local array, initializing each element to 1
    long n = 64;
    std::vector<long> data(n, 1);
    // Create a reducer
    // Using dynamic allocation here to force replicated storage
    auto result = new cilk::PerNodeReducer<cilk::op_add<long>>(0);
    // Convert pointer to reference for convenience
    auto&r = *result;
    cilk_for (long i = 0; i < n; ++i) {
        (*r) += data[i];
    }
    // Expected result is n
    return r.get_value();
}
```

Notes:

1. Reducers are only available when compiling a C++ file.
2. A PerNodeReducer MUST be allocated using `new`, other forms of allocation will lead to incorrect behavior. Example:

```
// GOOD
auto good_reducer = new cilk::PerNodeReducer<cilk::op_add<long>>(0);
// BAD
```

```
cilk::PerNodeReducer<cilk::op_add<long>> bad_reducer(0);
```

3. Reducers are compatible with any parallel construct, not only `cilk_for`. This includes custom uses of `cilk_spawn` and the parallel functions provided by `emu_c_utils`.

5.5 Ongoing Efforts

Work continues to extend and optimize the compilation toolchain and libraries. Key efforts include:

- Optimize the performance of thread spawning for cilk_spawn and cilk_for.
- Extend the set of available reducers.
- Extend C++ support and provide a set of distributed C++ containers to improve programmer productivity.

Chapter 6

Examples

There has been a significant amount of function renaming recently and the examples have been removed until they have been updated and verified.

Chapter 7

Simulation Execution

7.1 Simulation Overview

The simulation environment (`emusim`)[1] executes Pathfinder machine instructions in a blend of an untimed functional model and a SystemC timed architectural performance model. Various command line arguments can be passed to `emusim` to configure the simulation and collect various statistics. It is generally expected that `emusim` will be running simulations in the SystemC based architectural model. Section 7.3 describes how to control which model is used.

The functional model models the system as a set of nodes that are executing threads and their associated request and response packets. In this model, the node consists of a simple thread execution engine and the individual memory channels; there is no attempt made to model timing or limit resources. The main purpose for this model is determining program correctness and generating initial statistics related to where spawns and memory references occur.

The SystemC architectural performance model is designed to provide a cycle-approximate model of how an application will run on the actual hardware. Within this model, the packets[2] are treated as tokens and move amongst the various modules within the system to perform their operations. While `emusim` attempts to model the actual hardware implementation, there are some abstractions to simplify the model and improve simulator performance. Updating the simulator and validating against hardware results is always an ongoing process.

For the Pathfinder, we do not fully model the SRIO interconnect network, but instead use a simple switch to connect the nodes to each other. When a packet is ready to be released to the network, an estimated delay length is computed and the packet must wait for that length of time before it is released to the simple switch. The main limitation of this model is that effects such as collisions are not modeled.

While nodes in the full Pathfinderwill have stationary cores attached to them, these are not explicitly modeled in `emusim` at this time. The stationary core samples the service queue at a slower rate to model the delay a typical thread would see while awaiting service.

NOTE: While there are multiple physical MSPs in the node to service memory operations, the application is unaware of their existence. The simulation environment and, more notably, the profiling

[1] The simulator executable is `emusim.x`.

[2] A packet can be a thread, request packet, or response packet.

statistics and tools are continuing to be refined to present data in a user-friendly manner.

7.2 Application Development

It is highly encouraged to start development with a small, easy to debug program before moving on to larger data sets and a large number of threads since there is not a fully developed debugger as of yet.

Application failures, such as segmentation faults, are typically observed as exceptions in the simulator and on the hardware. Frequently, the first step in debugging these failures is to identify the offending instruction (TPC = Thread Program Counter) in both the exception message and the disassembled executable. The `gossamer64-objdump` tool has been ported and this tool can be used to generate a program's symbol table using the `-t` flag or disassembled using the `-d` flag. Documentation on using this tool can be found in the Linux manual (`man objdump`). The disassembly format is the only divergence from the standard tool. The format is as follows:

```
Instruction Byte-address: Instruction Nibble-address: Disassembled Instruction
```

Other files can be generated with `emu-cc` to further aid in debugging. In particular, users can output `.ll` (LLVM IR) and `.s` (assembly) files which can aid in correlating a line of C code to assembly code for debugging.

NOTE: The current toolchain releases have a "shared" bit set. This is bit 31 in program counters, e.g. 0x8000_2000, and bit 55 in addresses, e.g. 0x0180_0000_0000_8128. When running on the simulator, this bit can be safely ignored. When running on the hardware, this bit is only an issue if it is no longer set to 1. This bit will be set when viewing a disassembled executable program; thus, `Instruction Byte-address` values will start with a 4 and `Instruction Nibble-address` values will start with an 8.

7.3 Simulation Control Functions

Table 7.1 shows the functions used to control simulation execution and statistics in `emusim`. These functions are defined in the `timing.h` file which is included when using the `memoryweb.h` file.

See Section 4.5 for information on using the `CLOCK()` intrinsic for timing comparisons in the simulator and on the hardware. Note also that the `time.h` functions, such as `clock_gettime()`, use the SC clock. The time involved in doing a system call to the SC to get the time may be significant for small programs, so `CLOCK()` is the preferred mechanism for timing.

The following subsections define each function and describe their operation and usage.

7.3.1 starttiming

```
void starttiming()
```

Return Value: None

Table 7.1: Statistics Control Functions

Function	Description
void starttiming()	• Ends the functional portion of the simulation and moves all threads to the architectural model. Simulation restarts in the architectural performance model. • No-op in the hardware.
void stoptiming()	**Not yet implemented**

Arguments:

- None

Functionality: Within `emusim` programs are initially started in the untimed model and after a call to `starttiming()` in the source code, the untimed model will complete all migrations, remote operations, and acknowledgments. All threads that exist will be moved from the untimed node that models them to the equivalent architectural node. Once all threads have been moved to their proper architectural node, the simulation begins executing in the architectural model. If no call to `starttiming()` in the source code is made, the simulation runs completely in the functional model.

If the `--ignore_starttiming` flag is given to `emusim`, the statistics being gathered by the untimed functional model are reset when the call to `starttiming()` is made. This allows the programmer to view statistics for the same portion of the program.

Notes: While it is safe to make multiple calls to this function, it is recommended to call it only once.

Example:

7.4 Using the Simulator

The simulator is executed as follows:

```
emusim.x [<emusim options>] -- cilk_example.mwx <mwx options>
```

It is recommended that applications be verified with the functional model before using the timing model!

Table 7.2 lists the command line arguments that are currently recognized.

Table 7.2: Command Line Arguments

Command Line Argument	Action
Short/Long Options	

-h, --help	Prints the command line argument options and short descriptions.
-m, --log2_memory_per_node	Set log2 memory size per node (default 36=64GB, range 20-38).
-o, --base_ofile	Set the base file name for all output files (default is program name without the `.mwx` extension).
Long Options	
--total_nodes	Set the number of nodes to model; must be a power-of-2.
--output_monitor_period	Set the period, in ms, for the output monitor (default=1); setting to 0 will maximize this value (essentially no montior)
--output_instruction_count	Collect instruction count data for each function in the application. Data is written to a `.uis` file.
--capture_timing_queues	When doing a timed simulation, collect statistics for various queues/resources in the system to use with the visualization tool. Data is written to a `.tqd` file.
--timing_sample_interval	Set timing model queue depth sampling frequency in ns (default 10000; equivalent to a 100 KHz clock frequency). Requires --capture_timing_queues also.
--return_value	Output the return value from your program as an integer.
--return_value_hex	Output the return value from your program as a hex value.
--forward_return_value	If the simulation finishes correctly, the exit code of the simulator will be that of the simulated program rather than that of the simulator.
--verbose_isa	Print each ISA instruction as executed.
--short_trace	Print spawn and quit instructions; also print instructions causing migrations.
--untimed_short_trace	For the untimed (functional) simulation, print spawn and quit instructions; also print instructions causing migrations.
--verbose_tid	Print each ISA instruction as executed for a specific thread number. This is typically best done only after a simulation has failed and you know which thread you would like to make verbose.
--max_sim_time	Set the max simulation time in milliseconds (floating point input is accepted).
--ignore_starttiming	Run simulation purely in untimed mode; stats will be reset if a call to `starttiming()` is made in the source program.
--initialize_memory	This will initialize all memory to garbage values which may be useful for catching programs with bad pointers. However, this is resource intensive and the simulator will try to prevent you from using this if the simulated memory total is larger than the physical memory on the system running the simulator.

7.4.1 Simulator Configuration and Parameters

Please refer back to Fig. 1.1 for the organization of a single node. Within each node, the following components are modeled:

- Gossamer Core Clusters (GcClusters, 3 per node): These units contain the eight Gossamer Cores (GCs) and provide shared access to the communication rings. The Gossamer Cores are the lightweight processing cores for the threads; each core can support 64 active threads (512 active threads per cluster).
- Memory Side Processors (MSPs, 8 per node): These units serve all memory transactions within the node.
- SRIO Ports (6 in/out per node): These provide access to the system SRIO network. These ports share a common interface to the rings.
- Communication Rings: These are concentric rings that provide paths for request/response packets to move through the node. There are two request and three response rings per node. Each ring has a set of stations used to move packets around the ring.

The simulator defaults to modeling only a single node; use the `--total_nodes` flag to model additional nodes (must be a power-of-2).

The network model is "perfect" in that there are no collisions, dropped packets, broken links, etc.

7.5 `emusim` Screen Output

`emusim` does generate a modest amount of output to the screen with each simulation in default mode. Additionally, the application program itself may write data to the screen.

The main output generated by `emusim` is a progress marker that is printed every 1ms of simulated Pathfinder machine time with the time and date of the system. This allows a user to ensure that the simulation is progressing. *Note: adding nodes to the simulated machine will increase the length of time necessary to complete each 1ms of simulated time; this may be offset by a quicker program completion.*

Other screen output may be observed if there is some form of error in the simulation. The most common case is an exception for a bad address; this is essentialy a segmentation fault. Assertions have also been used in the simulator code to identify situations that should not happen; if one of these assertions is hit, please file a bug report.

The `--short_trace` and all of the `--verbose` options print data to the screen. It is strongly recommended that screen output be redirected to a file if using any of these options due to the amount of data they generate.

7.6 `emusim` File Output

Each run of `emusim` uses either the executable file name (without the `.mwx` extension) or the name specified with the `-o`, `--base_ofile` command line argument to generate three output files. The first output file has a `.cdc` extension and contains the configuration details and input command line. The second output files has a `.vsf` extension and contains more verbose statistics information. The contents of these files, especially the `.vsf` file, changes considerably when running in untimed functional rather than timed mode. A third output file, this one with a `.mps` extension, contains a set of memory maps identifying the source node and destination MSP of memory operations. This is formatted as a JSON file and is intended to be interpreted by the profiler described in Chapter 8.

Other output files can be generated by using the proper command line arguments as defined in Table 7.1.

7.6.1 Configuration Data Output (.cdc)

The following list describes the data written to the `.cdc` file when doing a timed architectural simulation:

- System Configuration details including clock rates and bandwidths.
- Pathfinder system run time and number of cycles for the core, interconnect, and memory.
- Number of threads that are active, created, and died. The active count will be 1 to account for the thread that executed. Died should be 1 less than the number created.
- Simulation wall clock time

The interconnect and memory (DDR) clock rates are separate from the GC Cluster and core clock frequency and are set to match the expected bandwidths of those components. **These rates are subject to change in future iterations of the simulator as it is validated against the actual hardware.**

7.6.2 Verbose Statistics Information (.vsf)

When running an untimed simulation, this file provides basic information on where threads were created and died along with counters for each MSP as to what memory transaction requests were made.

In timed mode, this file outputs a wide range of counters and ratios that may be used to identify potential bottlenecks. *This data is intended to be used by advanced users and should not be necessary in most cases.* The contents of this file are subject to change at any time and are not utilized by the profiler.

The most useful counters currently presented in this file are the Gossamer Core stats (headed with `GC STATS:`) and the MSP Data Cache stats (headed with `MSP DataCache STATS:`). The first `GC STATS:` section displays the IPC and various spawning/thread loading stats. The second `GC STATS:` section provides data on how often the GC was stalled (unable to issue an instruction). The first

`MSP DataCache STATS:` is the more useful and displays typical cache stats such as the number of access and the hit rate.

7.6.3 Memory Map Output (.mps)

NOTE: Currently, these maps generally show counters based on the application perspective rather than the physical perspective. Work is ongoing on how best to provide both views.

The memory reference maps are the same in both timed and untimed mode. If a program is executed without a call to `starttiming()`, the program will execute completely in untimed mode and the map statistics will be captured for the entire program. When a call is made to `starttiming()`, the maps are reset, even if the `--ignore_starttiming` flag is used. Using `starttiming()` in the source code with the `--ignore_starttiming` simulator flag allows a developer to run fully in untimed mode but capture only the maps of interest rather than any setup or initialization components.

As mentioned above, this file is formatted as a JSON file to allow for processing by the profiling tool. There is an abundance of whitespace, thus it can be human-readable as well.

These maps are formatted as tables where the rows are the source nodes and the columns are the destination node (for the enqueue map) or destination MSP (for the remaining maps).

The enqueue map captures when a thread is being moved from a GC into either the run queue or the service queue. Threads that are moving into the run queue on a different node are in the process of being migrated. Moves into the local run queue happen on reschedule instructions and on some spawns.

The remaining maps identify the key memory operation types (read, write, atomic, and remote) and the source/destination pairs for each of these. Note that reads and atomics to a different node will lead to migrations which are shown in the enqueue map.

7.6.4 Instruction Execution Statistics (.uis)

The `.uis` file consists of a set of instruction counts for each function in the program. Within each function, the number of migrations and half of the number of registers in the thread for each migration (migrations_with_reg_count[ ]) are also tracked (migrations use a 128bit bus, hence the half). Additionally, the average number of registers removed at each resize (noted by TPC, Thread Program Counter) instruction is also collected.

7.6.5 Timed Activity Tracing (.tqd)

In prior versions of `emusim`, the simulator captured the depth of various queues in the hardware model. This has been updated to better capture the activity level of various hardware units during the timed portion of the simulation. To maintain backwards compatibility, this activity can be captured by using the
`--capture_timing_queues` option. Using this option will generate a JSON based `.tqd` file which has some system configuration data and then a set of statistics data. This file is intended for use with the profiling tools rather than being viewed as a text file.

The following data is captured at each sample:

- Per Node: Number of live threads
- Thread activity counts: active, waiting in run queues, currently migrating
- Per MSP: number of incoming requests this sample period
- Per SRIO Port: Number of incoming and output requests this sample period

There may be some minor differences between the total number of live threads and the sum of the thread activity count at a given sample. This difference is the number of threads actively being transmitted within the nodes (e.g, moving from a GC to an SRIO port).

By default the sampling interval between timesteps is set to 10000 ns, but this can be changed using the command line argument `--timing_sample_interval`.

7.6.6 Memory Tracing (.mt)

This capability has been removed in the current version of `emusim`.

Chapter 8

Simulation Profiling

NOTE: While there are multiple physical MSPs in the node to service memory operations, the application is unaware of their existence. The profiling statistics and tools are continuing to be refined to present data in a user-friendly manner.

8.1 emusim_profile

The `emusim_profile` script is a wrapper around `emusim`which will set the proper flags for capturing execution statistics and then generate a variety of useful plots after the execution has completed. The results generated by this script can be used as a profiler to identify performance issues.

Before using `emusim_profile`, it is recommended to test the program in timed mode in the simulator to ensure that it runs to completion in a reasonable amount of time, reducing the input size as necessary. Secondarily, it may be worthwhile to adjust the sampling rate to avoid generating an overly-large `.tqd` file.

The program source will need to call `starttiming()` to get the full set of results.

The profiler is invoked in the following manner, passing the profile output directory followed by the benchmark command line:

```
emusim_profile <profile directory> [<emusim options>] -- mybenchmark.mwx
  --param 1 --param 2
```

Note: The profiler uses the following simulator flags, so they should not be passed into the profiler: `-o`, `--capture_timing_queues`, `--output_instruction_count`.

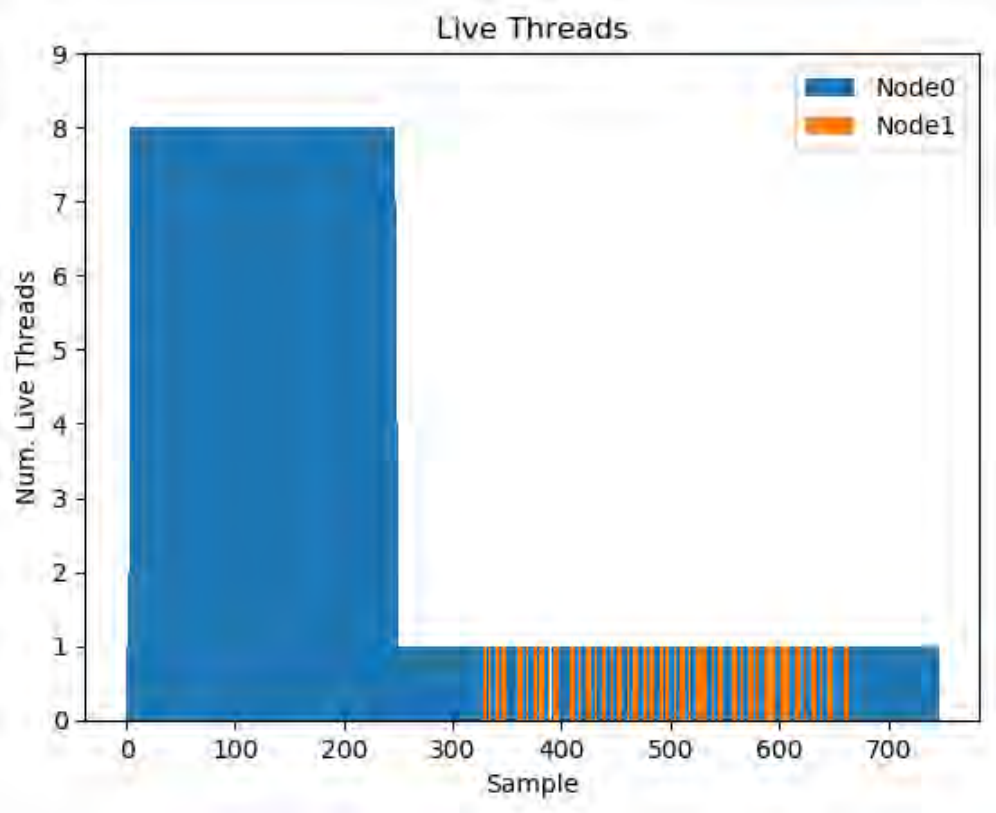
The timing queue samples are collected at a rate of 100 KHz; if running a longer program, it may be worthwhile to use the `--timing_sample_interval` and reduce the sampling rate. This will improve the performance of the profiling tool when generating the resulting graphs.

Several output files will be placed in the `<profile_directory>`. After the execution completes, you can open the HTML document generated in the profile directory to view the report. Alternatively, you can browse through the profile directory and open each plot image individually.

Note on presented graphs: The profiling figures shown in this section are from executing a very small GUPS (random access) program on two nodes. For this version of GUPS, 8 threads are

generated and all run on node 0 during the update phase. A single thread is then used to verify the GUPS result.

8.1.1 Threads per Node over Time

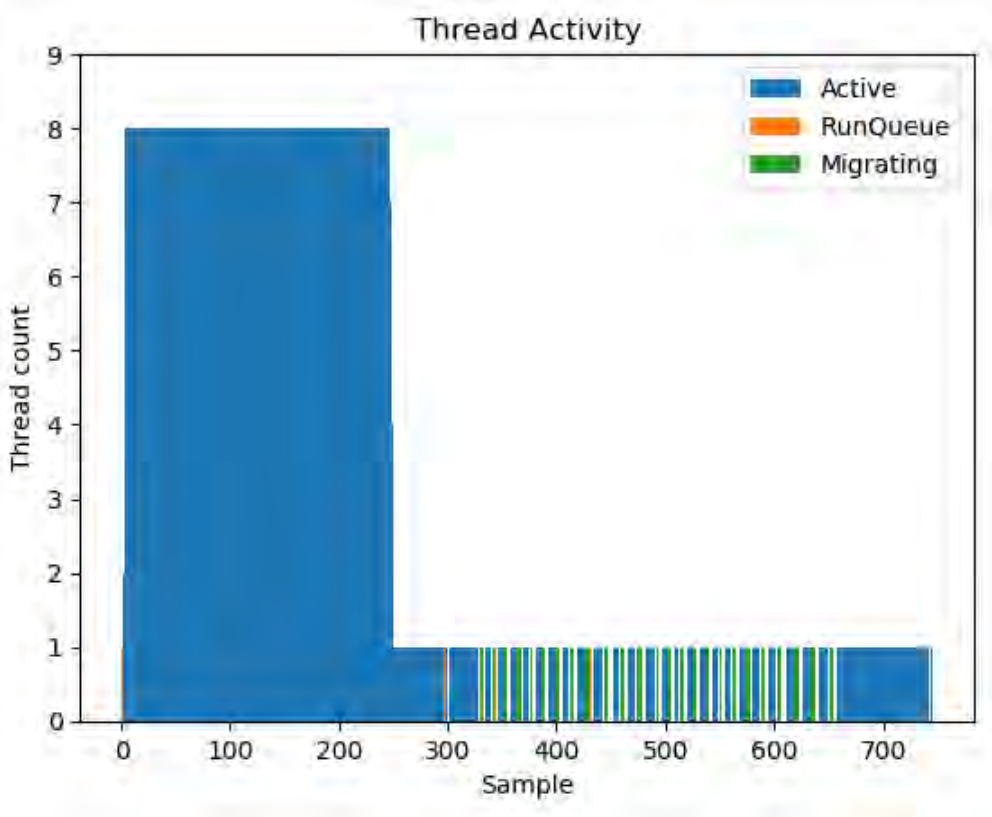


Description: A stacked bar plot with the number of executing threads on the Y axis, and the sample number on the X axis. Each colored stripe represents a different node. The combined height represents the total number of actively running threads on the entire system.

Performance Hints:

- Long flat sections of 1 thread indicate that serial code is dominating the execution. Look for functions that can be parallelized.
- Spans of time with only one color indicate that one node is doing all the work while other nodes sit idle. Consider changing the data layout to distribute more evenly across the system.
- A large filled area in which all color stripes are equally represented indicates excellent parallelism and thread balance.

8.1.2 Thread activity summary



Description: Similar in format to the active threads plot, except that threads are grouped by status instead of by location. The overall height represents the total number of threads in the system, while the colored areas represent how many threads are in each state:

- Active (blue): The thread is occupying an execution slot in a GC, actively running instructions.
- Run Queue (orange): The thread is occupying a slot in the node run queue, waiting for an execution slot in a GC to become available.
- Migrating (green): The thread is "in flight" between its source node and the run queue at its destination node.

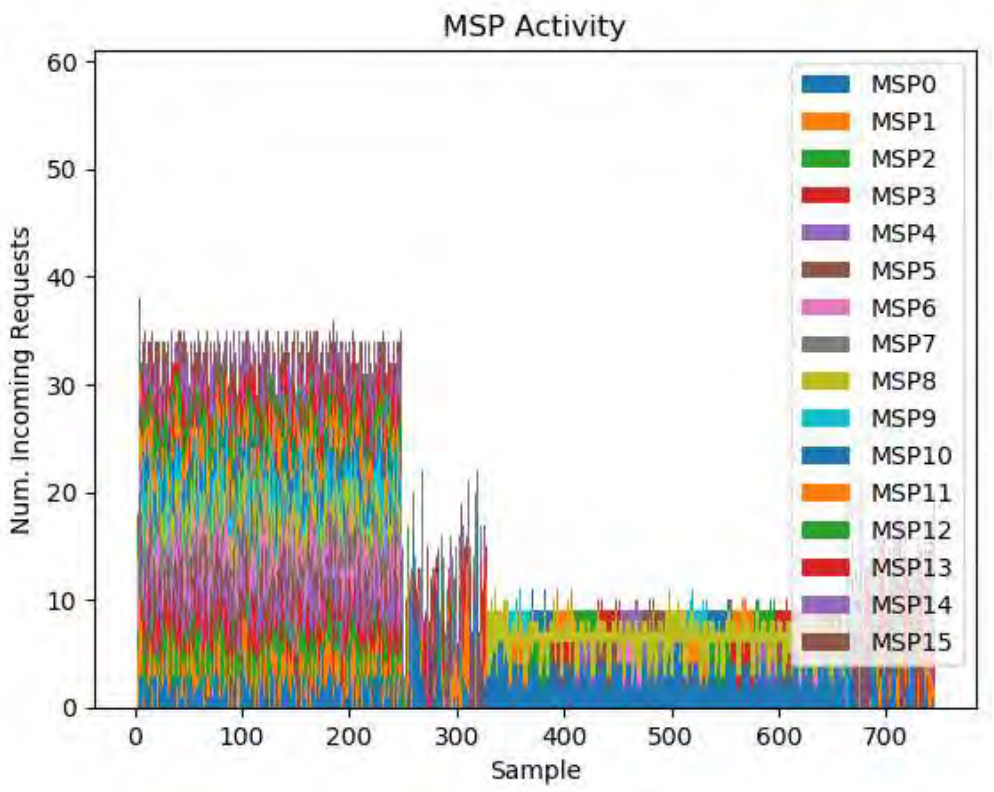
Performance Hints:

- Large number of migrating (green) threads: The system is migration-bound. Try finding ways to reduce the total number of thread migrations, or to do more work per migration.
- Large number of waiting (orange) threads: This may indicate a load imbalance, as many threads migrate to the same node and wait to execute. Try to improve the data distribution. Assuming the workload distribution is even, a large number of waiting threads may indicate that the system is over-subscribed, with more live threads than can execute at once. A

moderate level of over-subscription is good to keep the system busy, but consider reducing the number of thread spawns.

- Large idle time (white): Find ways to parallelize serial code, or spawn more threads by reducing the amount of work each thread does.

8.1.3 Incoming Commands per MSP

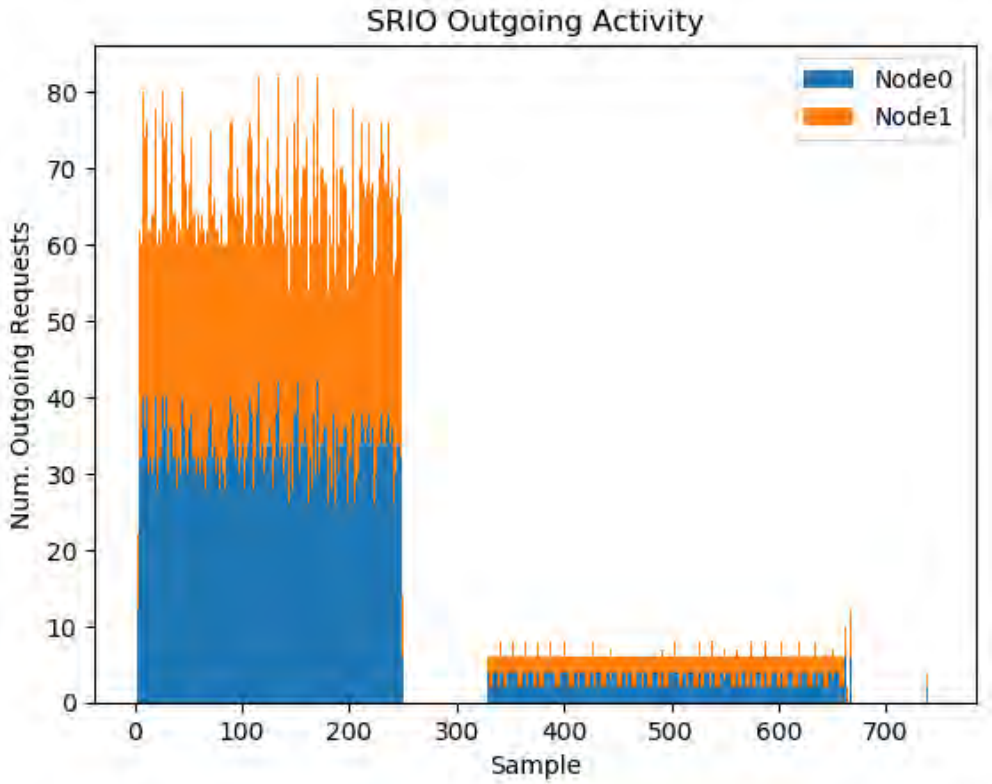


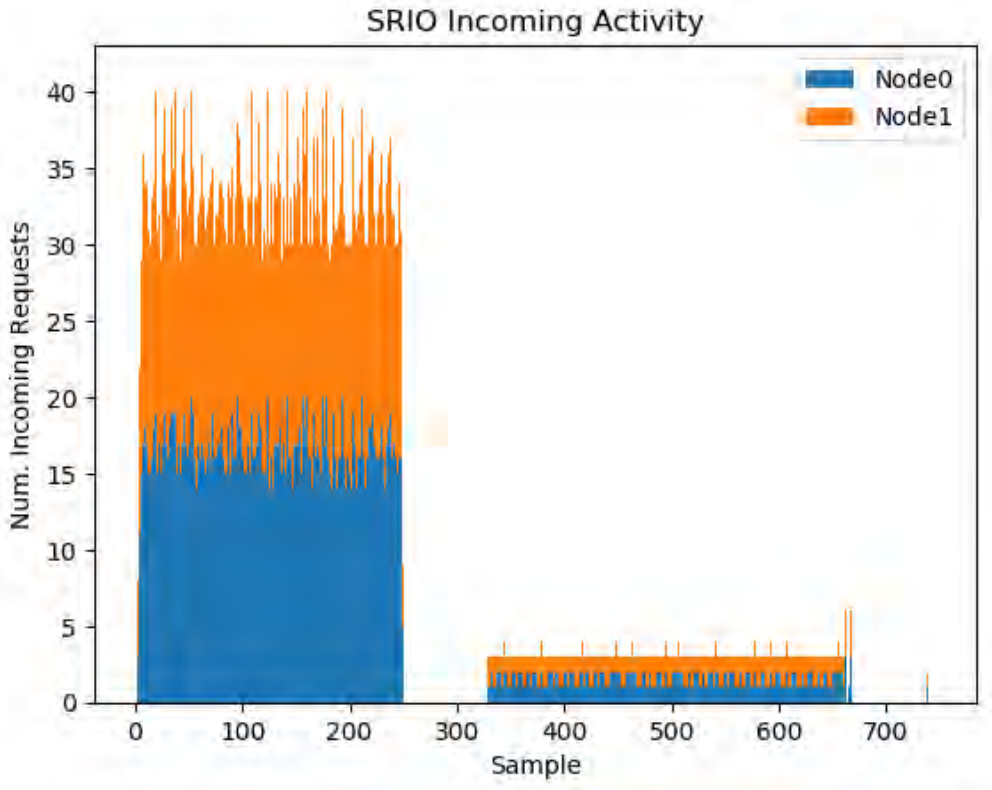
Description: A stacked bar plot with the total number of incoming requests on the Y axis and the samples on the X axis. Each sample captures the number of incoming memory requests seen by a specific MSP during the sample period.

Performance Hints:

- Spans of time with only one color indicate that one MSP is receiving all memory requests while other MSPs sit idle. Consider changing the data layout to distribute more evenly across the system.
- A large filled area in which all color stripes are equally represented indicates excellent data parallelism.

8.1.4 Number of Outgoing/Incoming SRIO Packets

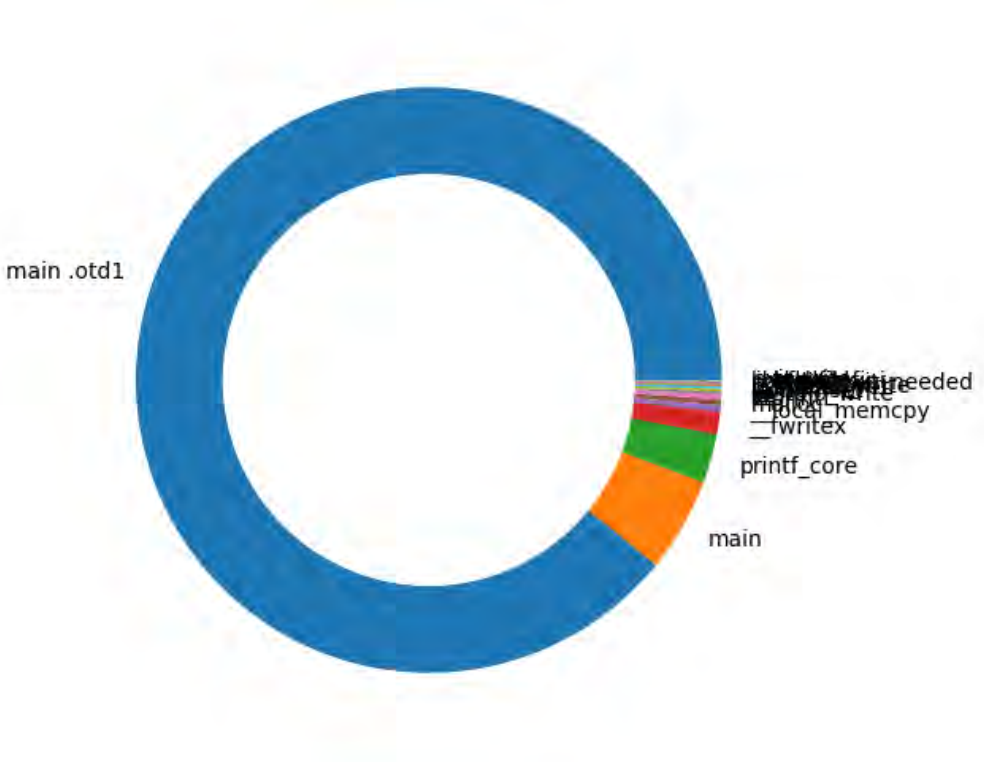




Description: A stacked bar plot with the total number of outgoing(incoming) requests on the Y axis and the samples on the X axis. Each sample captures the number of outgoing(incoming) packets seen by the SRIO ports (summed) during the sample period.

These can be used to identify total SRIO traffic and ensure that it is fairly balanced between nodes over time. Hotspots may be identified if one node has excessive incoming traffic over a period of time.

8.1.5 Total instructions executed by function

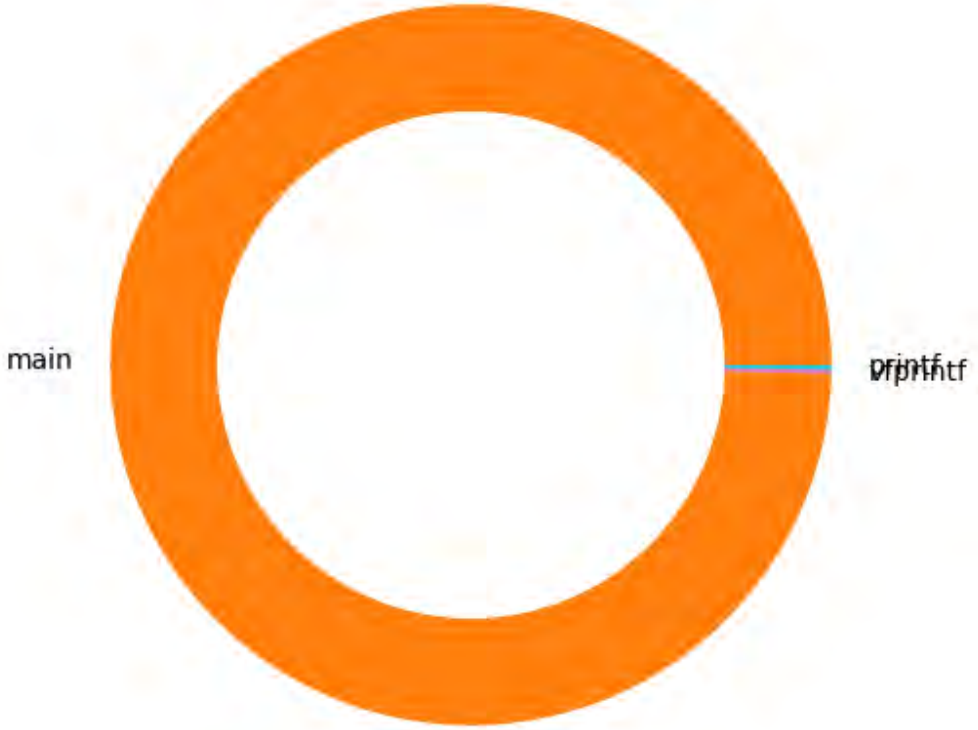


Description: The area of each slice of this donut plot is proportional to the number of instructions executed by each function, summed across all threads throughout the entire program. This plot is generated by aggregating the instruction counts from the UIS file.

Performance Hints:

- The functions with the largest area are the best candidates for optimization.
- Note that this plot counts total instructions executed by all threads, and not execution time per function. Functions that consume a lot of time in serial code may be underrepresented in this plot.

8.1.6 Percent of total migrations grouped by function



Description: The area of each slice of this donut plot is proportional to the number of migrations triggered by each function, summed across all threads throughout the entire program. This plot is generated by aggregating the migration counts from the UIS file.

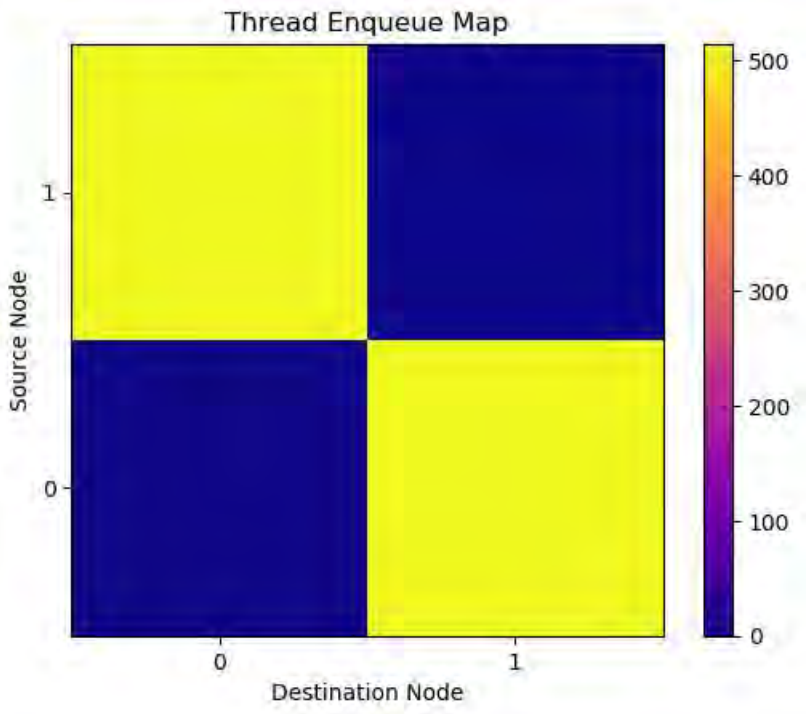
Performance Hints:

- The functions with the largest area are the best candidates for optimization, especially if a function was not expected to migrate.

8.1.7 Memory Maps

All plots shown in this section are 2D heat maps.

8.1.7.1 Thread Enqueue Map

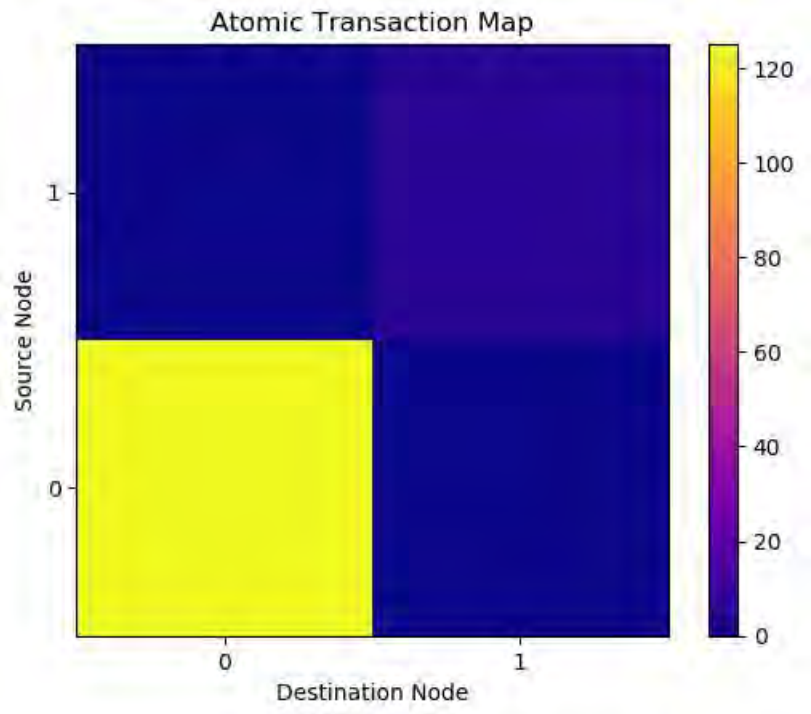


Description: This map shows the number of threads being enqueued on each node. Threads can be enqueued due to migrations (source node and destination differ), reschedules (enqueue on current node), service calls from the SC, and spanws (if the thread cannot be spawned direct into a GC context).

Performance Hints:

- A large number of local enqueues may indicate a large number of service calls, reschedules, or spawns through the run queue. It may be worth investigating ways to reduce these.
- A large number of enqueues to a given destination node may imply a memory hotspot or undesired back migrations to access a stack frame or some other data structure that has not been distributed or replicated.

8.1.7.2 Atomic Transaction Map

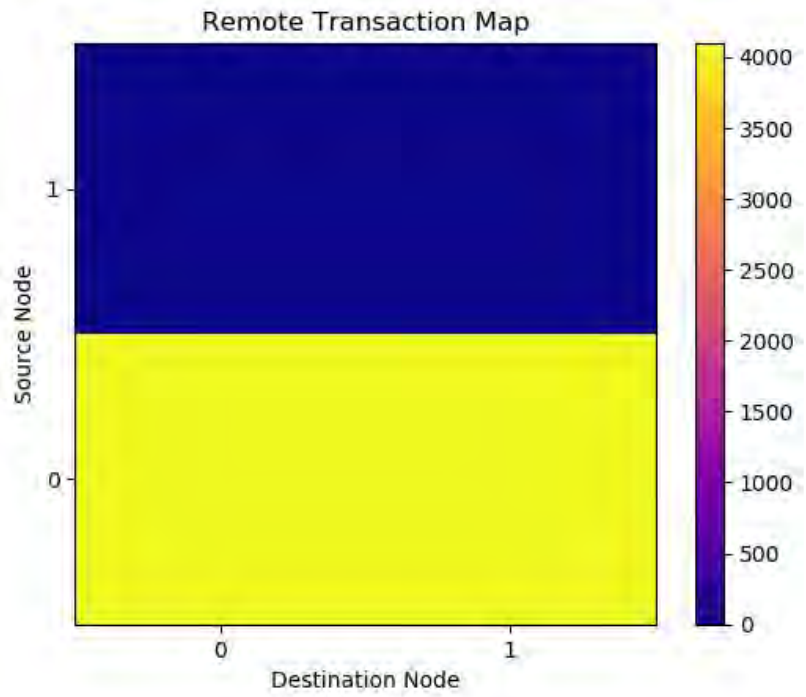


Description: This map shows the number of atomic operations being sent to each MSP from a given source node. If the atomic operation is to an address on a different node, the thread will be migrated to the proper node before the operation is redone.

Performance Hints:

- A highly unbalanced map, especially if atomics are not explicitly in the source code, suggests that thread stacks may not be well distributed. In multinode tests, it may be worth reviewing the spawn pattern(s) to see if a better balance is possible. Note that the program source code cannot control which MSP within a node will contain the thread stack (if one is necessary) for a spawned thread. The MSP to target will be a factor of the current clock cycle at the time of the spawn.
- An excess of atomics on MSP[0] may be a result of many small memoryweb library malloc calls.

8.1.7.3 Remote Transaction Map

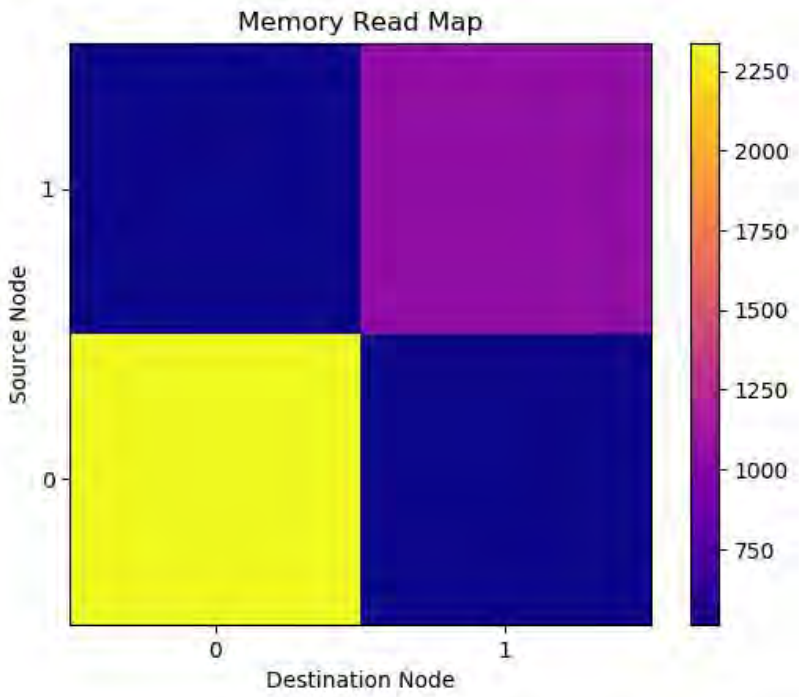


Description: This map shows the number of remote atomic operations being sent to each MSP from a given source node.

Performance Hints:

- The remote atomic operations are invoked by the program source code. If the map is unbalanced, then the data distribution and/or thread locations should be reconsidered.

8.1.7.4 Memory Read Map

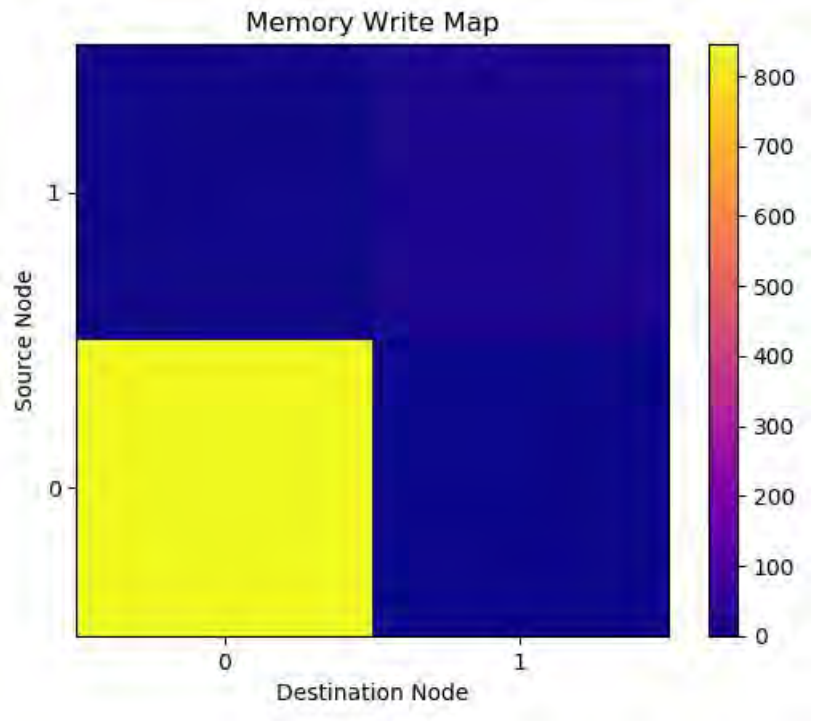


Description: This map shows the number of read operations made to each MSP from every node. If the read operation is to a different node, the thread will be migrated to the proper node before the operation is redone.

Performance Hints:

- If there are a large number of unexpected migrations, then the program is likely suffering from spurious back-migrations. Threads are frequently migrating back to their origination node to access a stack frame or some other data structure that has not been distributed or replicated.
- Take note of the numerical labels on the legend: if the overall number of migrations is low, the migration pattern is less important.
- Finally, remember that eliminating all migrations is not the only way to improve performance. Thread migrations are an intended feature of the system. Use this chart to compare the expected migration pattern with the actual migration pattern and diagnose performance issues.

8.1.7.5 Memory Write Map



Description: This map shows the number of write operations made to each MSP from every node. All write operations are done without migrations; all writes (and remotes) must be ACK'ed before the thread can leave its execution slot.

Performance Hints:

- Ensure the map is not excessively unbalanced.
- Take note of the numerical labels on the legend: if the overall number of writes is low, this data is likely to be of minimal importance.