Outline

This presentation covers the following topics

- Cilk and the Lucata API
  - *Basic programming*
  - *Data distribution*

Slides originally developed by Janice McMahon, Lucata Corporation

Basic Programming

Parallclism, spawning/syncing thrcads with Cilk

Spawning Threads with Cilk

- Specifies function may run in parallel with caller
  - Child thread spawned to execute function and parent continues in parallel w/child
  - Otherwise, parent executes a standard function call
- Spawn location determines location of
  - Synchronization structure
  - Stack frame (if needed)
- Spawn destination
  - Special functions denote spawn location
  - If no direction is given, then spawn is local

```
long f = cilk_spawn fib(a);
```

Thread Synchronization with Cilk

- Current function cannot continue past the cilk_sync until all children have completed
- Last thread to reach the cilk_sync continues execution – no waiting
- Implicit sync at termination of a function

```
void cilk_sync;
```

Example: Fibonacci

```
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>

long result;
long fib(long n)
{
  if (n < 2) return n;
  long a = cilk_spawn fib(n-1);
  long b = cilk_spawn fib(n-2);
  cilk_sync;
  return a + b;
}

int main(int argc, char **argv)
{
  long input = atol(argv[1]);
  result = fib(input);
}
```

Cilk include file

Spawn a thread for each of the `fib()` calls

Wait for threads to complete to ensure `a` and `b` are valid

- Recursive function with dynamic spawning
- Number of threads depends on data size
  - Second spawn could be function call

Sample Program Execution: fib.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc -o fib.mwx fib.c
>>>>>>>> /usr/local/emu/bin/emusim.x -- fib.mwx 10

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 13:02:34 2021

End untimed simulation with local date and time= Thu Mar 25 13:02:34 2021

>>>>>>>> more fib.cdc
**********************************************
Program Name/Arguments:
fib.mwx
10
**********************************************
Simulator Version: 21.3.24
**********************************************
Configuration Details:
Ring Model = Stratix: 3 GC Clusters, 4 MSPs, 1 SRIO
Number of Nodes=1
Total Memory (in GiB)=64
Logical MSPs per Node=1
Log2 Memory Size per MSP=36
GC Clusters per Node=3
GCs per Cluster=8
Uplinks used in rings=false
**********************************************
**********************************************
Simulator wall clock time (seconds): 0
>>>>>>>> more fib.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 0, 88, 89, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 551, 1858, 320

>>>>>>>>
```

Compile and run in simulator; default configuration is one node

Configuration file with execution summary

Verbose file with thread and memory operation counts

No migrations; all data on Node 0

Extra Credit Question:
Why is #died one more than #spawned?
(Answer at end of section)

Example: SAXPY with spawn loop

```
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>
void saxpy(long n, float a, float *x, float *y)
{
  for (long i = 0; i < n; i++)
    y[i] += a * x[i];
}
int main(int argc, char **argv)
{
  long nth = atol(argv[1]); // number threads
  long size = atol(argv[2]); // array size
  float aval = atof(argv[3]); // constant
  float *x = malloc(size * sizeof(*x));
  float *y = malloc(size * sizeof(*y));
  for (long i = 0; i < size; i++) {
    x[i] = i; y[i] = 0;
  }
  long grain = size / nth; // elts per thread
  for (long i = 0, j = 0; i < nth; i++, j += grain)
    cilk_spawn saxpy(grain, aval, &x[j], &y[j]);
  cilk_sync;
}
```

Allocated on stack at current node

Grain size determined by number of threads

Spawn a thread for each section of array

- Iterative loop with static spawning
- Number of threads fixed in code
- Work per thread depends on data size

Sample Program Execution: saxpy.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc -o saxpy.mwx saxpy.c
>>>>>>>> /usr/local/emu/bin/emusim.x -- saxpy.mwx 4 32 5.0

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 14:02:29 2021

End untimed simulation with local date and time= Thu Mar 25 14:02:29 2021

>>>>>>>> more saxpy.cdc
*********************************************
Program Name/Arguments:
saxpy.mwx
4
32
5.0
*********************************************
Simulator Version: 21.3.24
*********************************************
Configuration Details:
Ring Model = Stratix: 3 GC Clusters, 4 MSPs, 1 SRIO
Number of Nodes=1
Total Memory (in GiB)=64
Logical MSPs per Node=1
Log2 Memory Size per MSP=36
GC Clusters per Node=3
GCs per Cluster=8
Uplinks used in rings=false
*********************************************
*********************************************
Simulator wall clock time (seconds): 0
>>>>>>>> more saxpy.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 0, 4, 5, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 72, 47, 21
>>>>>>>>
```

Number of threads fixed in code

Parallel Loops in Cilk

- Divides loop among parallel threads, each containing one or more contiguous loop iterations
- Max number of iterations in each chunk is grainsize
- Best for situations where:
  - Threads are spawned locally
  - Work per element is fairly uniform

```
#pragma cilk grainsize = 4
cilk_for;(long i=0; i<SIZE; i++)
  {...}
```

Example: SAXPY with parallel loop

```
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>
int main(int argc, char **argv)
{
  long size = atol(argv[1]); // array size
  float aval = atof(argv[2]); // constant
  float *x = malloc(T * sizeof(*x));
  float *y = malloc(T * sizeof(*y));
  for (long i = 0; i < T; i++) {
    x[i] = i; y[i] = 0;
  }
#pragma cilk grainsize = 8
  cilk_for (long i = 0; i < size; i++) {
    y[i] += aval * x[i];
  }
}
```

Grain size determined by pragma

Use parallel construct for spawn loop

- Single thread per loop iteration unless grainsize is specified
- Threads are implicitly synchronized at end of loop

Sample Program Execution: saxpy_for.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc saxpy_for.c -o saxpy_for.mwx
>>>>>>>> /usr/local/emu/bin/emusim.x -- saxpy_for.mwx 32 5.0

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 14:56:29 2021

End untimed simulation with local date and time= Thu Mar 25 14:56:29 2021

>>>>>>>> more saxpy_for.cdc
**********************************************
Program Name/Arguments:
saxpy_for.mwx
32
5.0
**********************************************
Simulator Version: 21.3.24
**********************************************
Configuration Details:
Ring Model = Stratix: 3 GC Clusters, 4 MSPs, 1 SRIO
Number of Nodes=1
Total Memory (in GiB)=64
Logical MSPs per Node=1
Log2 Memory Size per MSP=36
GC Clusters per Node=3
GCs per Cluster=8
Uplinks used in rings=false
**********************************************
**********************************************
Simulator wall clock time (seconds): 0
>>>>>>>> more saxpy_for.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 0, 3, 4, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 99, 73, 44
>>>>>>>>
```

Fewer memory operations; more efficient

Extra Credit Question:
Why is there one fewer thread than spawn loop?
(Answer at end of section)

Unit Summary: Basic Parallelism

- Dynamic vs. static spawning
- Using Cilk_spawn and Cilk_sync for multi-threading
- Using Cilk_for with grain size to parallelize loop iterations
- Using compiler to build executables
- Using simulator to gather basic statistics and thread counts
- Using basic simulator statistics to verify that correct numbers of threads are spawned

Exercises:
Try fib with different values, measure and verify thread counts
Insert printf to verify results
Try different grain sizes, thread counts

Extra Credit Answers:
Total thread count includes initial thread spawned from the SC Parallel loop re-uses initial thread for computation

Data Distribution: Part I

Distributed allocation using memorywcb library, targeted thread spawn

Cyclic Array Distribution

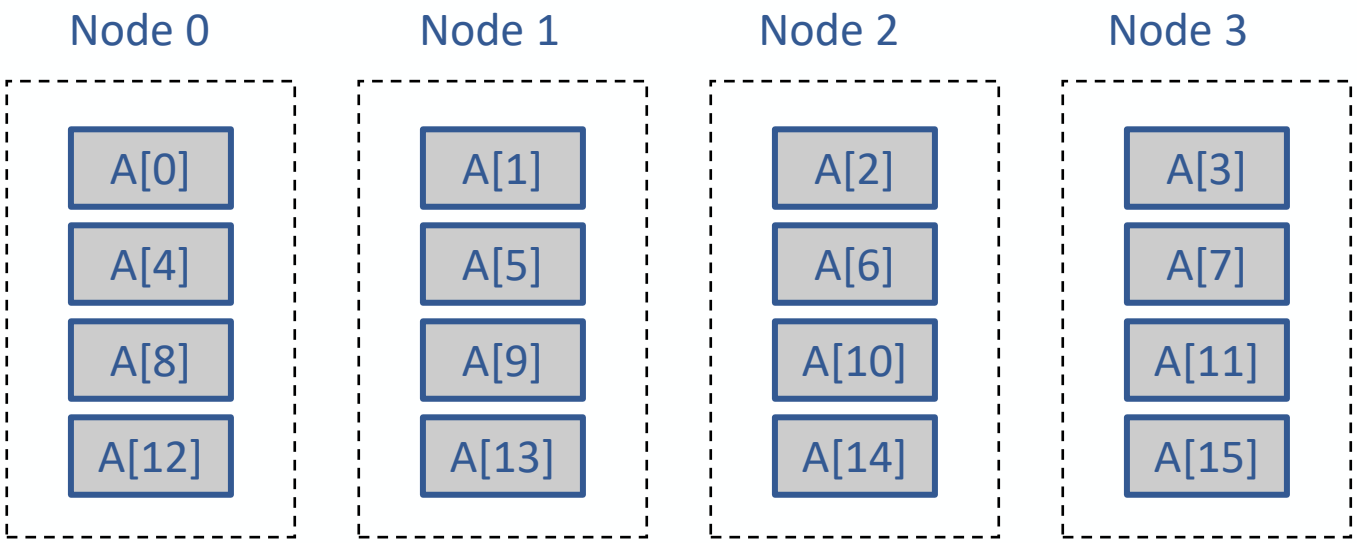
- Array of N long integers striped across nodes round robin
- Accessed using 1D array notation (e.g. A[i])
- ONLY works for 64-bit types (e.g. long)

```
long *A = mw_malloc1dlong(unsigned long N);
```

Example: 1D Array

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

Array of N elements, striped across nodes round-robin



Example: Distributed Vector AX+Y

```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb.h>
void saxpy(long n, long a, long *x, long *y)
{
  for (long i = 0; i < n; i++)
    y[i] += a * x[i];
}
int main(int argc, char **argv)
{
  long nth = atol(argv[1]); // number threads
  long size = atol(argv[2]); // array size
  long aval = atol(argv[3]); // constant
  long *x = mw_malloc1dlong(size);
  long *y = mw_malloc1dlong(size);
  for (long i = 0; i < size; i++) {
    x[i] = i; y[i] = 0;
  }
  long grain = size / nth; // elts per thread
  for (long i = 0, j = 0; i < nth; i++, j += grain)
    cilk_spawn saxpy(grain, aval, &x[j], &y[j]);
  cilk_sync;
}
```

Memoryweb include file

Cyclic distribution of input arrays

- Code is the same except for initial data allocation
- Array access in program automatically finds correct element in correct node

Sample Program Execution: saxpy_1d.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc saxpy_1d.c -o saxpy_1d.mwx
>>>>>>>> /usr/local/emu/bin/emusim.x --total_nodes 4 - saxpy_1d.mwx 4 32 5

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

End untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

>>>>>>>> more saxpy_1d.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 8, 4, 5, 0
1: 8, 0, 0, 0
2: 8, 0, 0, 0
3: 8, 0, 0, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 28, 25, 11
1.0: 70, 67, 27
2.0: 44, 26, 74
3.0: 9, 31, 1
```

Compile and run in simulator with 4 nodes

All threads spawned in node 0, migrate to node with data

- Data distributed across nodes causes thread migrations
- All threads spawned in node 0
- Access is parallel, but thread spawning is sequential

Lucata Cilk Extensions for Thread Placement

- Migrate hint specifies node for next cilk_spawn operation
  - Argument p is a pointer into destination node's memory

```
cilk_migrate_hint(p);
```

- Directed spawn function combines migrate hint and Cilk spawn into a macro for single-command spawn
  - Implemented as C macro; may require braces for correct operation

```
cilk_spawn_at(p) fib(a);
```

Example: Distributed Vector and Spawn

```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb.h>
void saxpy(long n, long a, long *x, long *y)
{
  for (long i = 0; i < n; i++)
    y[i] += a * x[i];
}
int main(int argc, char **argv)
{
  long nth = atol(argv[1]); // number threads
  long size = atol(argv[2]); // array size
  long aval = atol(argv[3]); // constant
  long *x = mw_malloc1dlong(size);
  long *y = mw_malloc1dlong(size);
  for (long i = 0; i < size; i++) {
    x[i] = i; y[i] = 0;
  }
  long grain = size / nth; // elts per thread
  for (long i = 0, j = 0; i < nth; i++, j += grain) {
    cilk_migrate_hint(&y[j]);
    cilk_spawn saxpy(grain, aval, &x[j], &y[j]);
  } cilk_sync;
}
```

Directed spawn

- Code is the same except for additional migrate hint
- Equivalent alternative replaces two function calls:

```
cilk_spawn_at (&y[j])
  saxpy(grain, aval,
        &x[j], &y[j]);
```

Sample Program Execution: saxpy_1d_hint.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc  saxpy_1d_hint.c -o saxpy_1d_hint.mwx
>>>>>>>> /usr/local/emu/bin/emusim.x --total_nodes 4 - saxpy_1d_hint.mwx 4 32 5

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

End untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

>>>>>>>> more saxpy_1d_hint.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 8, 4, 5, 0
1: 8, 0, 0, 0
2: 8, 0, 0, 0
3: 8, 0, 0, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 32, 25, 11
1.0: 70, 67, 27
2.0: 36, 26, 82
3.0: 9, 31, 1
```

All threads spawned in node 0, migrate to node with data

Extra Credit Question:
Why are all threads still spawned in Node 0?
(Answer at end of section)

Block Array Distribution

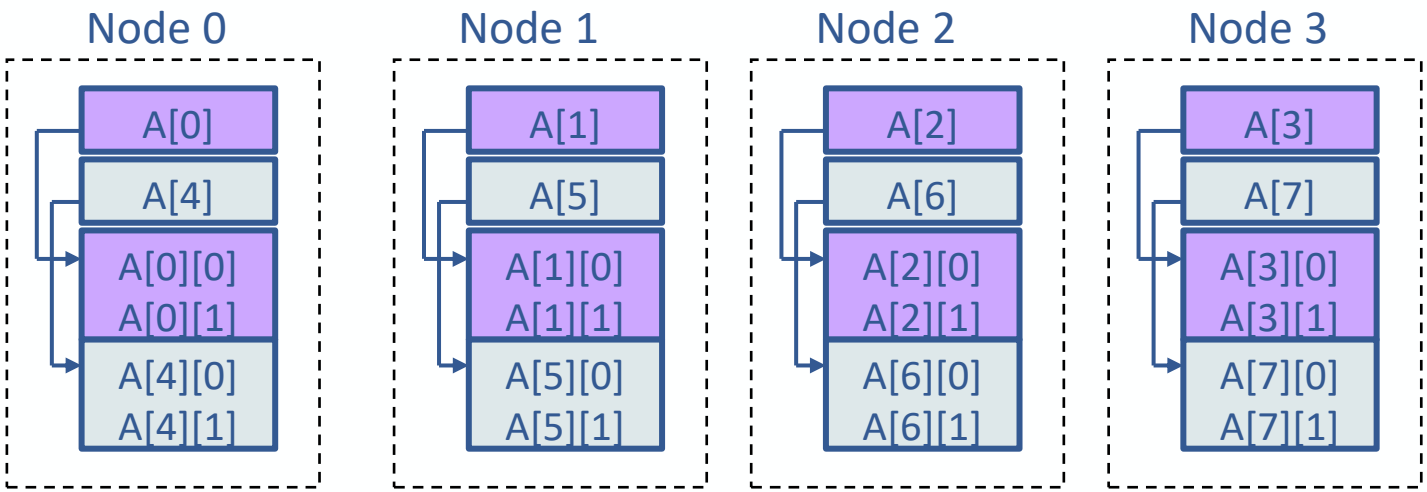
- Array of N pointers striped across nodes round robin
- Each points to co-located memory block of size S
- May be an array of pointers to any type

```
void ** mw_malloc2d(unsigned long N,
                    size_t S);
```

Example: Wrapped Block Distribution

```
#define N 16
long B = 8;
long E = N/B;
long ** A = (long **)
    mw_malloc2d(B, E * sizeof(long));
for (long i=0; i<N; i++)
    A[i/E][i%E] = i;
```

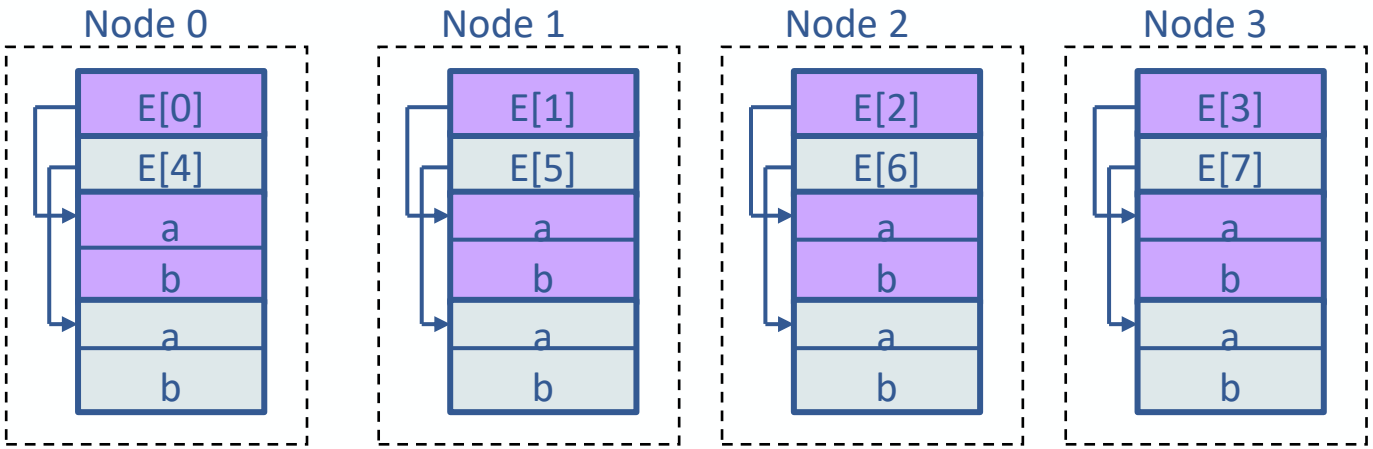
Array of B blocks each with E elements, striped across nodes round-robin



Example: Distributed Array of Structures

```
#define N 8
struct element { long a; long b; };
struct element ** E = (struct element **)
        mw_malloc2d(N, sizeof(struct element));
for (long i=0; i<N; i++) {
        E[i]->a = i;
        E[i]->b = 0;
}
```

Array of N blocks each with containing struct E, striped across nodes round-robin



Accessing Distributed Data

- Inputs:
  - Array allocated with mw_malloc2d
  - Index for first dimension
  - Number of blocks used in mw_malloc2d
  - Blocksize used in mw_malloc2d
- Returns address of A[i][0]

```
long * =
  mw_arrayindex(void *A, unsigned long i,
                unsigned long N, size_t S);
```

Example: SAXPY with block distribution

```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb.h>
void saxpy(long n, float a, float *x, float *y)
{
  for (long i = 0; i < n; i++)
    y[i] += a * x[i];
}
int main(int argc, char **argv)
{
  long num = atol(argv[1]); // number blocks
  long size = atol(argv[2]); // block size
  float aval = atof(argv[3]); // constant
  float **x = mw_malloc2d(num, size * sizeof(*x));
  float **y = mw_malloc2d(num, size * sizeof(*y));
  for (long j = 0; j < num; j++)
    for (long i = 0; i < size; i++) {
      x[j][i] = j * size + i; y[j][i] = 0;
  }
  for (long i = 0; i < num; i++) {
    cilk_spawn_at (y[i]) saxpy(size, aval, x[i], y[i]);
  }
  cilk_sync;
}
```

- 2D Block allocation
- Number of threads is number of blocks
- Work per thread is block size

Spawn a thread for each block

Braces required for macro in loop

Sample Program Execution: saxpy_2d_at.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc saxpy_2d_at.c -o saxpy_2d_at.mwx
>>>>>>>> /usr/local/emu/bin/emusim.x --total_nodes 4 - saxpy_2d_at.mwx 4 8 5.0

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

End untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

>>>>>>>> more saxpy_2d_at.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 12, 1, 5, 0
1: 5, 1, 0, 0
2: 5, 1, 0, 0
3: 5, 1, 0, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 130, 124, 22
1.0: 90, 104, 31
2.0: 73, 53, 30
3.0: 27, 93, 4
```

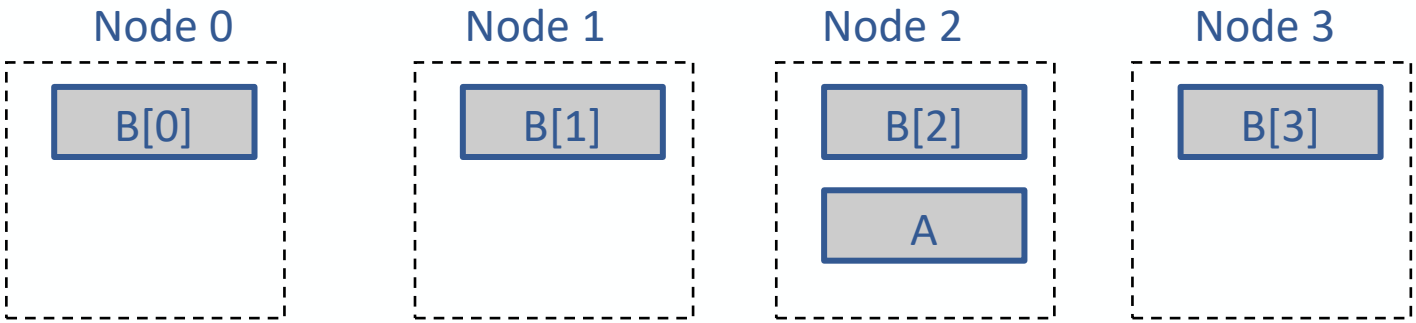
Threads spawned on correct node

Allocate Co-located Data

- Block of memory of size eltsize co-located with address ptr
- Accessed using traditional C semantics
- Leaves you at the new location

```
void * = mw_localmalloc(size_t S, void *p);
```

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



Example: SAXPY with local allocation

```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb.h>
void saxpy4(long n, float a, float *x, float *y)
{
  for (long i = 0; i < n; i++)
    y[i] += a * x[i];
}
int main(int argc, char **argv)
{
  long num = atol(argv[1]); // number blocks
  long size = atol(argv[2]); // block size
  float aval = atof(argv[3]); // constant
  float **x = mw_malloc2d(num, size * sizeof(*x));
  float *y = mw_localmalloc(num * size * sizeof(*y), x[0]);
  for (long j = 0; j < num; j++)
    for (long i = 0; i < size; i++) {
      x[j][i] = j * size + i; y[j * size + i] = 0;
  }
  for (long i = 0; i < num; i++) {
    cilk_spawn_at (x[i]) saxpy4(size, aval, x[i], &y[i * size]);
  }
  cilk_sync;
}
```

- 2D Block allocation for input
- Output uses local allocation on particular node

Place output on Node 0

Sample Program Execution: saxpy_loc_at.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc  saxpy_loc_at.c -o saxpy_loc_at.mwx
>>>>>>>> /usr/local/emu/bin/emusim.x --total_nodes 4 - saxpy_loc_at.mwx 4 8 5.0

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

End untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

>>>>>>>> more saxpy_loc_at.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 108, 1, 5, 0
1: 37, 1, 0, 0
2: 37, 1, 0, 0
3: 37, 1, 0, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 131, 168, 5
1.0: 126, 206, 10
2.0: 97, 72, 154
3.0: 103, 156, 53
```

Spawns in correct locations

More migrations due to local allocation

Distributed Free

- Free data allocated by mw_malloc2d or mw_malloc1dlong

```
void mw_free(void *p);
```

- Free data allocated by mw_localmalloc

```
void mw_localfree(void *p);
```

Unit Summary: Data Distribution

- Cyclic data distribution
- Thread placement
- Block data distribution
- Varying parallelism just by changing data allocation, not algorithm code

Exercises:
Try different grain and block sizes, check migrations and thread counts
Try different pointers to direct spawn location
Try block allocation of structures
Use printf and see how migration counts change

Extra Credit Answers:
Pointer index is a multiple of number of nodes, always on node 0

Data Distribution: Part II

Replicated allocation using memorywcb library

LUCATA

Global Replicated Data

- Instructs compiler to place an instance on each node
- Uses a "View 0" address that always gives local instance
- Must be a global variable
- Example Uses:
  - Constants
    - Copy on each node
    - All initialized to the same unchanging value
    - EX: PI, pointer to shared data structure
  - Local data
    - Copy on each node
    - May have different values
    - Use only when it does not matter which instance you access!
    - EX: random number table, pointer to local work queue

```
replicated long c = 3927883;
```

Dynamic Replicated Pointers

- Allocates a block on each node, returns replicated pointer
- Similar to using the replicated keyword
- Used when the size of the data structure is not known at compile time

```
long * mw_mallocrepl(size_t N);
```

Accessing Replicated Data Structures

- Return a pointer to the instance of a replicated data structure co-located with the destination pointer

```
void * mw_get_localto(void *src, void *dst);
```

- Return a pointer to the nth instance of a replicated data structure

```
void * mw_get_nth(void *src, unsigned n);
```

Example: Replicated Constant AX+Y

```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb.h>
long aval;
replicated long a;
void saxpy(long n, long a, long *x, long *y) {
  for (long i = 0; i < n; i++) y[i] += a * x[i];
}
int main(int argc, char **argv)
{
  long nth = atol(argv[1]); // number threads
  long size = atol(argv[2]); // array size
  aval = atol(argv[3]); // constant
  mw_replicated_init(&a, aval);
  long *x = mw_malloc1dlong(size);
  long *y = mw_malloc1dlong(size));
  for (long i = 0; i < size; i++) {
    x[i] = i; y[i] = 0;
  }
  long grain = size / nth; // elts per thread
  for (long i = 0, j = 0; i < nth; i++, j += grain)
    cilk_spawn saxpy(grain, &x[j], &y[j]);
  cilk_sync;
}
```

- All threads access global variable instance on current node

Constant is in global replicated variable

Sample Program Execution: saxpy_1d_repl.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc saxpy_1d_repl.c -o saxpy_1d_repl.mwx
>>>>>>>> /usr/local/emu/bin/emusim.x --total_nodes 4 -- saxpy_1d_repl.mwx 4 32 5

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

End untimed simulation with local date and time= Thu Mar 25 22:40:38 2021

>>>>>>>> more saxpy_1d_repl.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 8, 4, 5, 0
1: 8, 0, 0, 0
2: 8, 0, 0, 0
3: 8, 0, 0, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 28, 25, 11
1.0: 75, 68, 27
2.0: 43, 26, 74
3.0: 9, 31, 1
```

- Same migration and thread pattern as spawn loop
- Memory accesses are slightly different

Extra Credit Question:
What would happen if global non-replicated variable were used instead?
(Answer at end of section)

Initializing Replicated Data with Value

- Initializes each instance of replicated data structure to the same value

```
void = mw_replicated_init(long *p, long v);
```

```
replicated long N;
int main()
{       …
        long n_elements = compute_n_elements();
        mw_replicated_init(&N, n_elements);
        …
}
```

Initializing Replicated Data using Node ID

- Initializes each instance of the replicated data structure using the result of the user-defined function init_func(n) where n is the node number

```
void mw_replicated_init_multiple
        (long *p, long (*f)(long));
```

```
replicated long B;
long init_func(long nid) {
        return nid * 5;
}
int main()
{       …
        mw_replicated_init_multiple(&B, init_func);
        …
}
```

Initializing Replicated Data using Pointer

- Initializes each instance of replicated data structure using the user-defined function f(&obj, n) where &obj is the address of the replicated data structure and n is the node number

```
void mw_replicated_init_generic
        (long *p, void (*f)(void *, long));
```

```
replicated struct info { long x; long y; } info;
void init_info(void *obj; long nid) {
        struct info *i = (struct info*) obj;
        i->x = node;
        i->y = 5*node + 4;
}
int main()
{       …
        mw_replicated_init_multiple(&info, init_info);
        …
}
```

Unit Summary: Data Replication

- Replicated global variables
- Allocation, initialization and use within threads
- Affects on migrations and memory accesses

Exercises:
Try other replicated data structures
Try more complex initializations
Measure effects of replicated vs. non-replicated data

Extra Credit Answer:
Non-replicated variable will cause more migrations in code