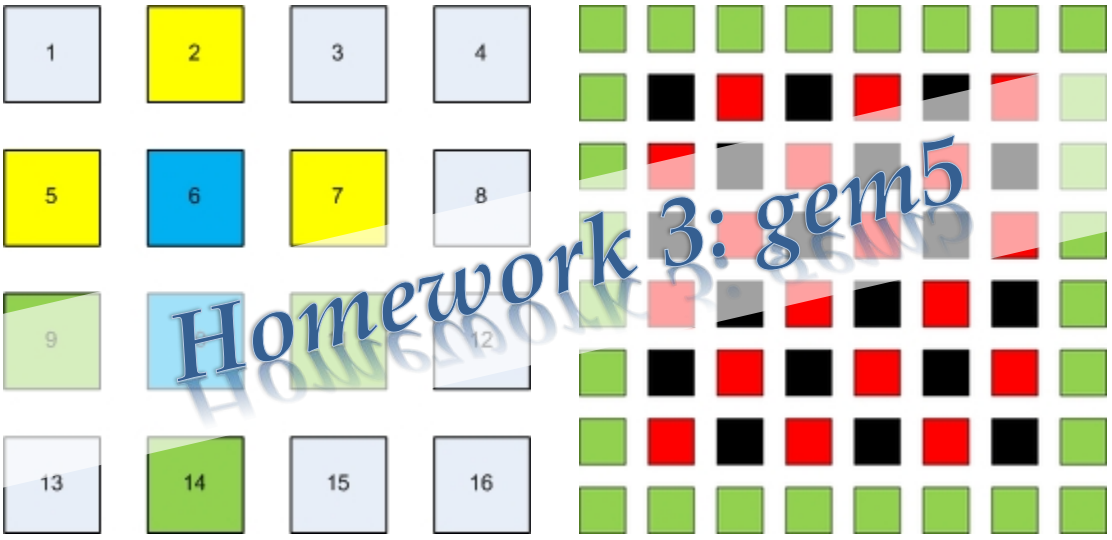
CS 757 - Parallel Computer Architecture

Spring 2014



THE UNIVERSITY of WISCONSIN MADISON

Aman Rakesh Chadha
(906 835 4597)

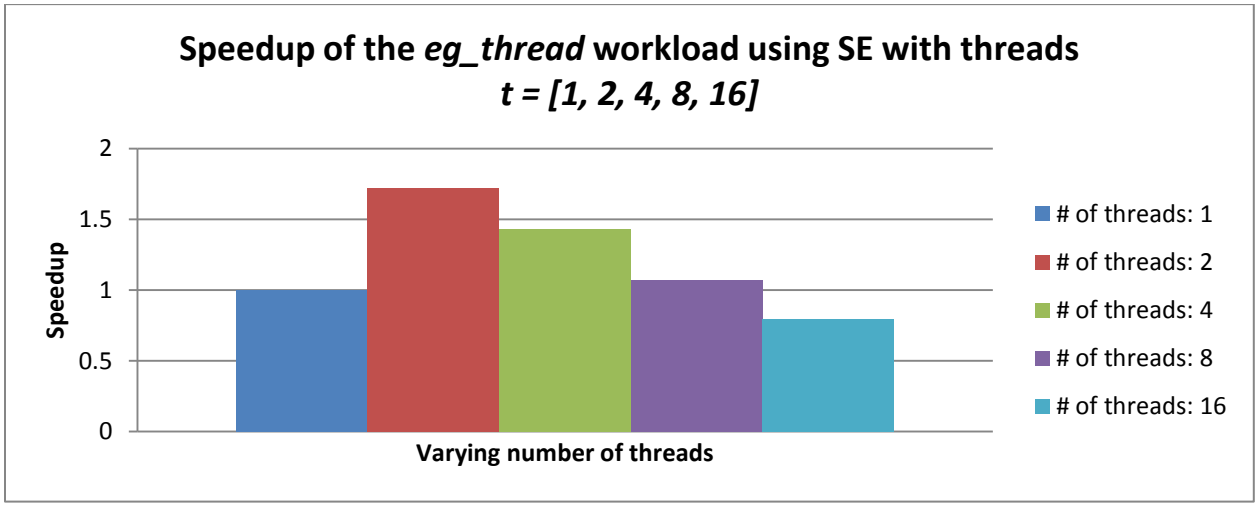
Ranjini Mysore Nagaraju
(906 924 4441)

1 Overview

This assignment consisted of setting up gem5, performing Syscall Emulation (SE), Full System (FS) Simulation, and simulating programs to observe speedup trends using SE and FS modes.

2 Problem 1: SE Mode - *eg_pthread* workload

a Plot of speedup with varying number of threads normalized to the t = 1 case



We observe the decreasing trend in speedup with increasing number of threads. The performance for *eg_pthread* increased initially for two threads. This behavior is due to two threads sharing work concurrently. However, since all the threads in this program operate on single counter variable, this variable is highly contended resulting in performance degrade past two threads. This behavior is due to synchronization cost between multiple threads contending for a single shared variable.

Thus, the downward trend in speedup can be attributed to the fact that *eg_pthreads* lacks scalability and hence, opportunities of concurrency due to coarse-grained locking and the use of a single counter variable, which is being incremented serially by all threads.

3 Problem 2: SE Mode - *Ocean* workload

a Source code

```
/* ************************************************************** *
 * Example pthread code                                           *
 * Each thread increments a shared counter                        *
 * ************************************************************** */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>
#include <sys/errno.h>
#include "m5op.h"

int num_threads;

pthread_mutex_t   SyncLock; /* mutex */
```

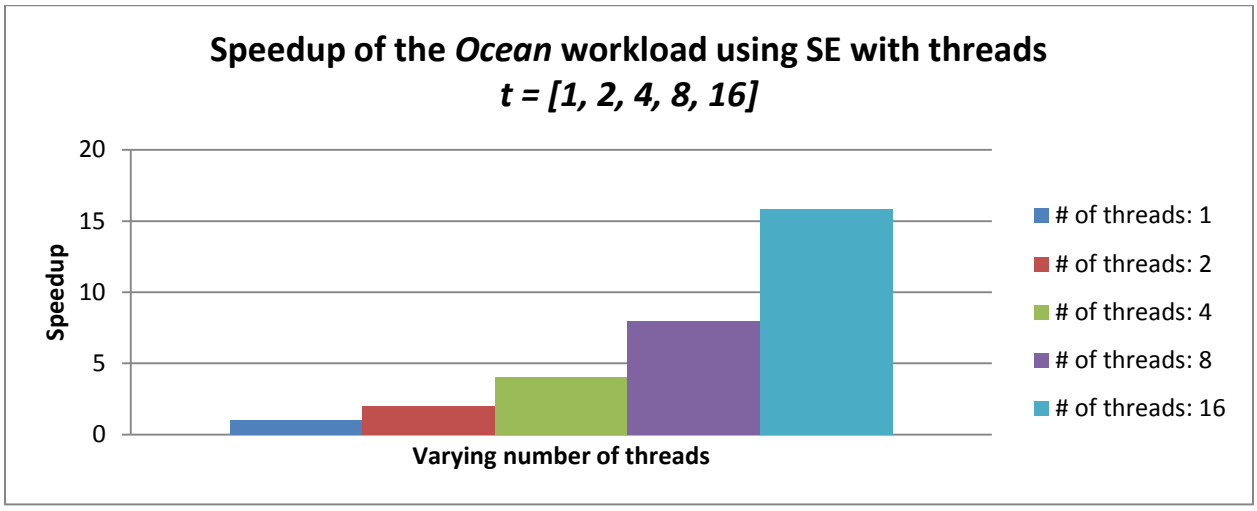
```
pthread_cond_t    SyncCV; /* condition variable */
int               SyncCount; /* number of processors at the barrier so far */

int xdim,ydim,timesteps;
int** grid;

/*  Ad-hoc  Barrier  Code.  This  function  will  cause  all  the  threads  to
synchronize
 * with each other.  What happens is that the mutex lock is used to control
 * access to the condition variable & SyncCount variable.  SyncCount is
 * initialized to 0 in the beginning, and when barrier is called by each
 * thread, SyncCount is incremented.  When it reaches NumProcs, the
 * number of threads/processors, a conditional broadcast is sent out which
 * wakes up all the threads.  The bad part is that each thread will then
 * contend over the mutex lock, SyncLock, and will be released sequentially.
 *
 * see man for further descriptions about cond_broadcast, cond_wait, etc.
 *
 * Barrier locks could also be implemented in many other ways, using
 * semaphores, and other sync. functions
 */

void printGrid(int** grid, int xdim, int ydim){
  int i, j;
  for(i=0;i<ydim;i++){
    for(j=0;j<xdim;j++)
      printf("%-6d  ",grid[i][j]);
    printf("\n");
  }
}

void Barrier()
{
  int ret;

  pthread_mutex_lock(&SyncLock); /* Get the thread lock */
  SyncCount++;
  if(SyncCount == num_threads) {
    ret = pthread_cond_broadcast(&SyncCV);
    SyncCount = 0;
    assert(ret == 0);
  } else {
    ret = pthread_cond_wait(&SyncCV, &SyncLock);
    assert(ret == 0);
```

```
  }
  pthread_mutex_unlock(&SyncLock);
}

/* The function which is called once the thread is allocated */
void* ocean(void* tmp)
{
  /* each thread has a private version of local variables */
  int tid = *((int*) tmp);
  int t,offset,i,j;
  int ret;

  if(tid==0)
    m5_reset_stats(0,0);
  Barrier();

  /* ********************* Execute Job ******************************** */
  for (t = 0; t < timesteps; t++) {
    m5_work_begin(0,tid);
    for(i=(((xdim-2)/num_threads)*tid)+1;i<=(((xdim-2)/num_threads)*(tid+1)); i++)
      {
      int offset = (i+t)%2;
      for(j=1+offset; j<ydim-1; j+=2) {
      //printf("i=%d, j=%d, tid=%d\n",i,j,tid);
      grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
                  + grid[i][j-1] + grid[i][j+1])/5;
      }
    }
    Barrier();
    m5_work_end(0,tid);
  }
}

int main(int argc, char** argv)
{
  pthread_t*     threads;
  pthread_attr_t attr;
  int            ret;
  int            dx;

  int i,j,t;

  if (argc!=5) {
    printf("The Arguments you entered are wrong.\n");
```

```
  printf("./serial_ocean <x-dim> <y-dim> <timesteps> <num_threads>\n");
  return EXIT_FAILURE;
} else {
  xdim = atoi(argv[1]);
  ydim = atoi(argv[2]);
  timesteps = atoi(argv[3]);
  num_threads = atoi(argv[4]);
}

/* Initialize array of thread structures */
threads = (pthread_t *) malloc(sizeof(pthread_t) * num_threads);
assert(threads != NULL);

/* Initialize thread attribute */
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); //sys manages contention

/* Initialize mutexs */
ret = pthread_mutex_init(&SyncLock, NULL);
assert(ret == 0);
//  ret = pthread_mutex_init(&ThreadLock, NULL);
//  assert(ret == 0);

/* Init condition variable */
ret = pthread_cond_init(&SyncCV, NULL);
assert(ret == 0);
SyncCount = 0;

/* declare and initialize grid*/
  grid = (int**) malloc(ydim*sizeof(int*));
  int *temp = (int*) malloc(xdim*ydim*sizeof(int));
  for (i=0; i<ydim; i++) {
    grid[i] = &temp[i*xdim];
  }
  for (i=0; i<ydim; i++) {
      for (j=0; j<xdim; j++) {
      grid[i][j] = rand();
      }
  }

int* id = (int*)malloc(sizeof(int)*num_threads);

for(dx=0; dx < num_threads; dx++) {
  /* ************************************************************
```

```
     * pthread_create takes 4 parameters
     *  p1: threads(output)
     *  p2: thread attribute
     *  p3: start routine, where new thread begins
     *  p4: arguments to the thread
     * ********************************************************** */
    id[dx]=dx;
    ret = pthread_create(&threads[dx], &attr, ocean, (void*) &id[dx]);
    assert(ret == 0);
  }

  /* Wait for each of the threads to terminate */
  for(dx=0; dx < num_threads; dx++) {
    ret = pthread_join(threads[dx], NULL);
    assert(ret == 0);
  }

  pthread_mutex_destroy(&SyncLock);
  pthread_cond_destroy(&SyncCV);
  pthread_attr_destroy(&attr);

  free(temp);
  free(grid);

  return 0;
}
```

b Plot of speedup with varying number of threads for grid size 258x258 for 50 iterations normalized to the t = 1 case

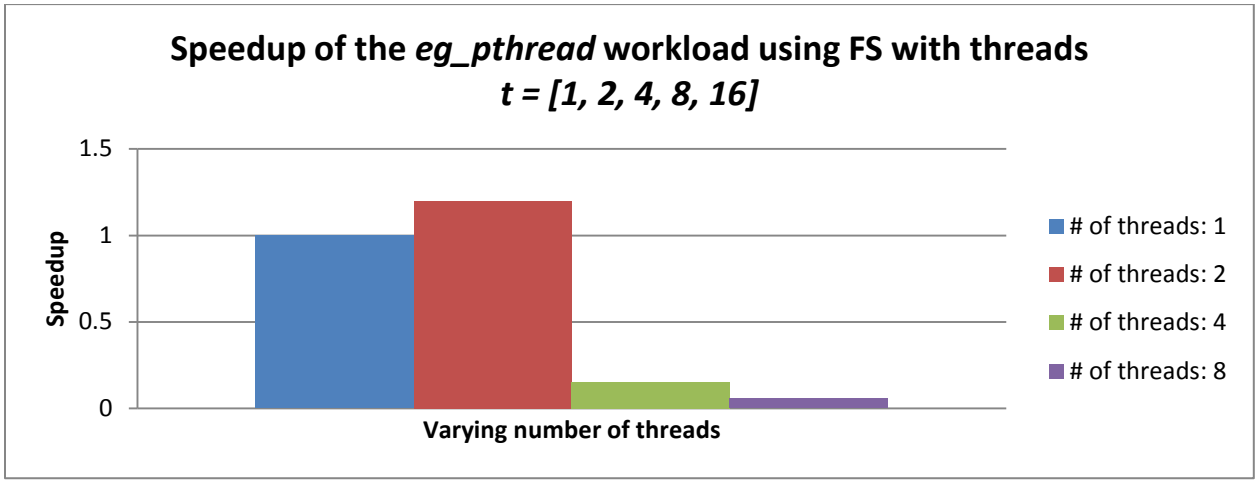


Ocean workload shows nearly linear increase in speedup with increasing number of threads. The difference from perfect linear speedup is due to the fact the threads are synchronized using barrier at each timestep. This synchronization incurs some overhead and in case of a load

imbalance, all threads wait for the delayed thread. Apart from this observation, our implementation of ocean with *pthreads* has linear scalability with number of threads.

4 Problem 3: FS Mode - *eg_pthread* workload

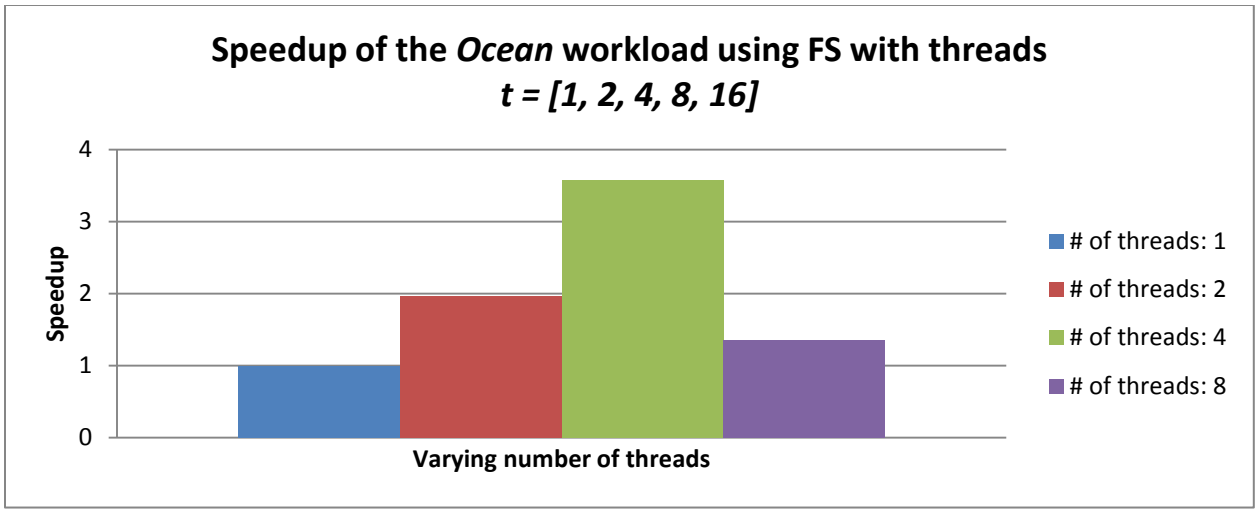
a Plot of speedup with varying number of threads relative to the t=1 case

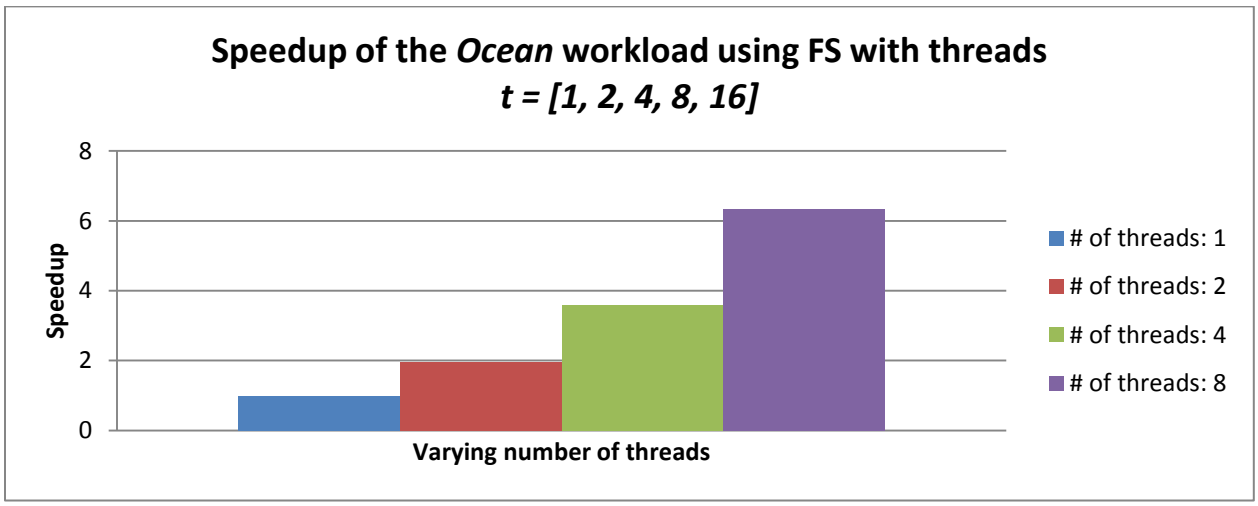


As in the case of problem 1, the downward trend in speedup with increasing number of threads can be attributed to the high contention between threads for a single shared variable. However performance values slightly differ from values obtained in SE mode. This can be attributed to complete system modeling by FS mode.

4 Problem 4: FS Mode - *Ocean* workload

a Plot of speedup with varying number of threads for grid size 258x258 for 50 iterations relative to the t = 1 case





We observe similar trend for ocean in full system mode as we observed in system emulation mode. However values obtained from FS mode reflect values obtained while executing ocean in homework 2. This is due to the fact that Full System mode models complete system including OS and devices.

5 Explanations and Analysis of Trends

a Conclusions on whether speedup numbers for ocean obtained with the simulator match the speedup numbers obtained in homework 2

Speedup numbers obtained by execution in homework 2 matches with speedup numbers obtained from FS mode. However in SE mode *pthread ocean* shows nearly linear speedup. This is because SE mode does not model OS kernel instructions while FS mode does. This shows that FS mode provides more realistic simulation than SE mode.

b Conclusions on whether speedup trends observed with the simulator match the speedup trend from homework 2

Speedup numbers and trend obtained by execution in homework 2 matches with that obtained from FS mode.

c Difference between trace-driven simulation and execution-driven simulation

Trace-driven Simulation:
In case of trace-driven simulation, a real machine is used to execute a benchmark program/software in the native ISA binary. This binary is usually instrumented, i.e., modified, so that as each instruction is executed, information such as the instruction op-code, data address, and branch information is written out in a trace file, which is general very large in size for real benchmark programs. These trace records usually represent memory references, branch outcomes, or specific machine instructions, among others.

Once generated, these traces are read into a simulator which can run on any machine (different ISA) and analyzed for performance study. Thus, the simulator itself does need to execute the ISA but only needs to analyze the 'execution history' for a particular architecture being studied, hence named 'trace-driven'.

Execution-driven Simulation:
In case of execution-driven simulation, the benchmark is directly executed, i.e., the simulator reads a program and simulates the execution of machine instructions on the fly.

As the program is executed, the performance study is performed at the same time.

The selection of input types for simulation is a trade-off between space and time. In particular, a very detailed trace for a highly accurate simulation requires a very large storage space, whereas a very accurate execution-driven simulation takes a very long time to execute all instructions in the program.

d Advantages and disadvantages of trace-driven simulation

Advantages:
1. A trace-driven simulation is known to be relatively fast in comparison to its execution-driven counterpart.
2. Results are highly reproducible.
3. Can run on any machine as long as trace format is known.
4. Different simulators can be used to study performance unlike in the case of execution-driven simulation.

Disadvantages:
1. Two step process: (i) execution of trace (ii) performance study.
2. Requires a large storage space to store traces.
3. Hard to study architectural details such as branch prediction algorithm since you know the branch behavior ahead of time.
4. Reading traces from disk can be slow.

e Advantages and disadvantages of execution-driven simulation

Advantages:
1. No need to store traces.
2. One step process.
3. Can study architectural details.
4. A program file is typically several magnitudes smaller than a trace file.

Disadvantages:
1. Much slower than the trace-driven simulation because it has to process each instruction one-by-one and update all statuses of the microarchitecture components involved.
2. Rebuilding a new simulator each time a new architecture is studied may be hard and tedious.
3. May be slower when a complex architecture is studied (since with execution-driven simulation, speed of the simulator is a very important issue and a complex one).

f Simulation type used when simulating eg_phtreads and ocean

Since we ran the benchmark (program) directly instead of loading in a trace of the benchmark, we used execution-driven simulation in this assignment.

g Observations and reasons for any differences in speedups between SE and FS mode

In FS mode gem5 simulated a full system along with OS and devices. FS mode simulated both user level and kernel level instructions. Hence it provides simulation that is close to actual execution. However in SE mode system level services are emulated. SE mode is like an idealized model which ignores many practical factors and constraints. Hence there is a considerable gap between results of the two modes of simulation.

6 Appendix: Data obtained from Experiments

Problem 1: SE Mode - eg_pthread workload	
Threads	**Time (ticks)**
1	3187233000
2	1847940000
4	2219637000
8	2978778000
16	4005615000
Speedup Table wrt t=1	
Threads	**Speedup**
1	1
2	1.72474918016819
4	1.435925333737
8	1.06998003879443
16	0.795691298339955
Problem 2: SE Mode - Ocean workload	
Grid size 258x258 for 50 iterations	
Threads	**Time (ticks)**
1	956645867000
2	478405274000
4	238601960000
8	119522270000
16	60384554000
Speedup Table wrt t=1	
Threads	**Speedup**
1	1
2	1.99965577093533
4	4.00937975111353
8	8.00391313685726
16	15.8425591253021
Problem 3: FS Mode - eg_pthread workload	
Threads	**Time (s)**
1	0.003609
2	0.003007
4	0.023325
8	0.063116
16	N/A
Speedup Table wrt t=1	
Threads	**Speedup**
1	1
2	1.20019953441969
4	0.154726688102894
8	0.057180429685024
16	N/A

Problem 4: FS Mode - Ocean workload	
Grid size 258x258 for 50 iterations	
Threads	**Time (s)**
1	0.984307
2	0.499591
4	0.275312
8	0.155159
16	N/A
Speedup Table wrt t=1	
Threads	**Speedup**
1	1
2	1.97022564457726
4	3.57524190736328
8	6.34386016924574
16	N/A