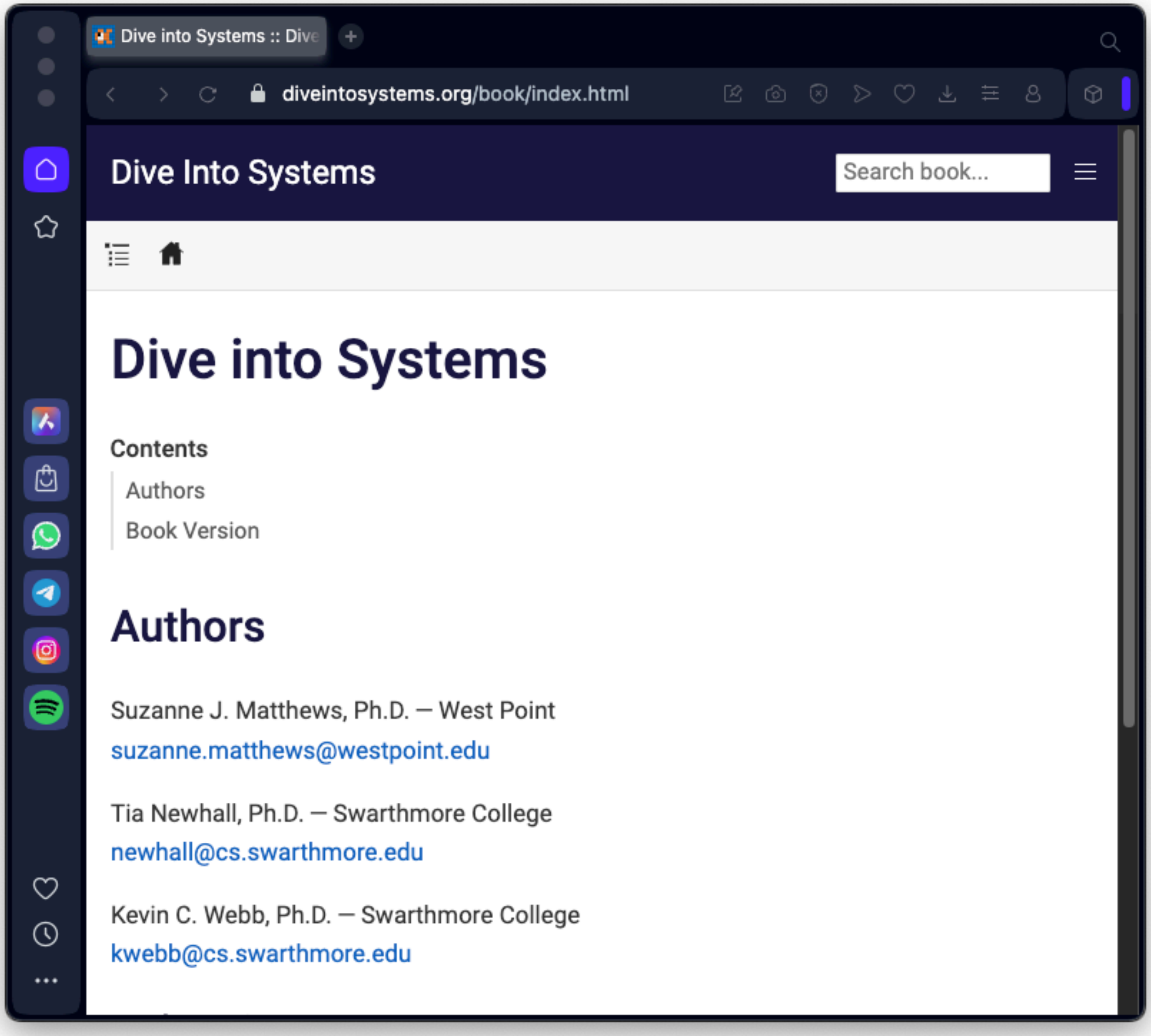
14. Leveraging Shared Memory in the Multicore Era

For COMSC 142

Sam Bowne

Updated Apr 30, 2025

Free online textbook



- https://diveintosystems.org/book/index.html

Topics

Ch 14a:

- 14.1. Programming Multicore Systems
- 14.2. POSIX Threads
- 14.3. Synchronizing Threads

Ch 14b:

- 14.4. Measuring Parallel Performance
- 14.5. Cache Coherence
- 14.6. Thread Safety
- 14.7. Implicit Threading with OpenMP

14.4. Measuring Parallel Performance

Topics

- Speedup
- Efficiency
- Amdahl’s Law
- Gustafson-Barsis Law
- Scalability

Speedup

- Compare the time a program takes to execute on one core to the time on ***c*** cores

$$Speedup_c = \frac{T_1}{T_c}$$

- If a serial program takes 60 seconds to execute,
- while its parallel version takes 30 seconds on 2 cores,
- the corresponding speedup is 2.

Efficiency

$$Efficiency_c = \frac{T_1}{T_c \times c} = \frac{Speedup_c}{c}$$

- If a serial program takes 60 seconds, but a parallel program takes 30 seconds on two cores
  - Efficiency is 1
- If a serial program takes 60 seconds, but a parallel program takes 30 seconds on four cores
  - Efficiency is 0.5

Parallel Performance in the Real World

- Most programs contain a necessarily serial component that exists due to inherent dependencies in the code.
- The longest set of dependencies in a program is referred to as its **critical path**.
- Not all programs are good candidates for parallelism!
    - The length of the critical path can make some programs downright hard to parallelize.
    - As an example, consider the problem of generating the _n_th Fibonacci number.

Parallelization of the countElems function

```
$ ./countElems_p_v3 100000000 0 1
Time for Step 1 is 0.331831 s

$ ./countElems_p_v3 100000000 0 2
Time for Step 1 is 0.197245 s

$ ./countElems_p_v3 100000000 0 4
Time for Step 1 is 0.140642 s

$ ./countElems_p_v3 100000000 0 8
Time for Step 1 is 0.107649 s
```

Table 1. Performance Benchmarks

Number of threads	2	4	8
Speedup	1.68	2.36	3.08
Efficiency	0.84	0.59	0.39

Amdahl’s Law

- ***S*** is the fraction of a program that is inherently serial
- ***P*** is the fraction of a program that can be parallelized
- The maximum improvement is:

$$T_c = S \times T_1 + \frac{P}{c} \times T_1$$

Example

- Program is 90% parallelizable and executes in 10 seconds on 1 core

Number of cores	Serial time (s)	Parallel time (s)	Total Time (T_c s)	Speedup (over one core)
1	1	9	10	1
10	1	0.9	1.9	5.26
100	1	0.09	1.09	9.17
1000	1	0.009	1.009	9.91

Gustafson-Barsis Law

- Amdahl used a fixed problem size and added cores
- Gustafson-Barsis assume that the problem grows as cores are added
  - With time being constant
- So you can always get more work done with more processors
- Even for the serial portion of the work

Scalability

- A program is **scalable**
  - If adding cores improves performance
- **Strongly scalable**
  - If adding cores improves performance at a fixed problem size
- **Weakly scalable**
  - If adding cores and also increasing the problem size in proportion improves performance

General Advice Regarding Measuring Performance

- Run a program multiple times when benchmarking.
- Be careful where you measure timing.
- Be aware of the impact of hyperthreaded cores
  - They may have more resource contention than physical cores
- Beware of resource contention
  - Other processes may slow the one you are testing

14.5. Cache Coherence

Cache Design

- Data/instructions are not transported individually to the cache.
  - Instead, data is transferred in **blocks**, and block sizes tend to get larger at lower levels of the memory hierarchy.
- Each cache is organized into a series of **sets**, with each **set** having a number of **lines**.
  - Each **line** holds a single block of data.
- A **cache hit** occurs when the desired data block exists in the cache.
- Otherwise, a **cache miss** occurs, and a lookup is performed on the next lower level of the memory hierarchy (which can be cache or main memory).

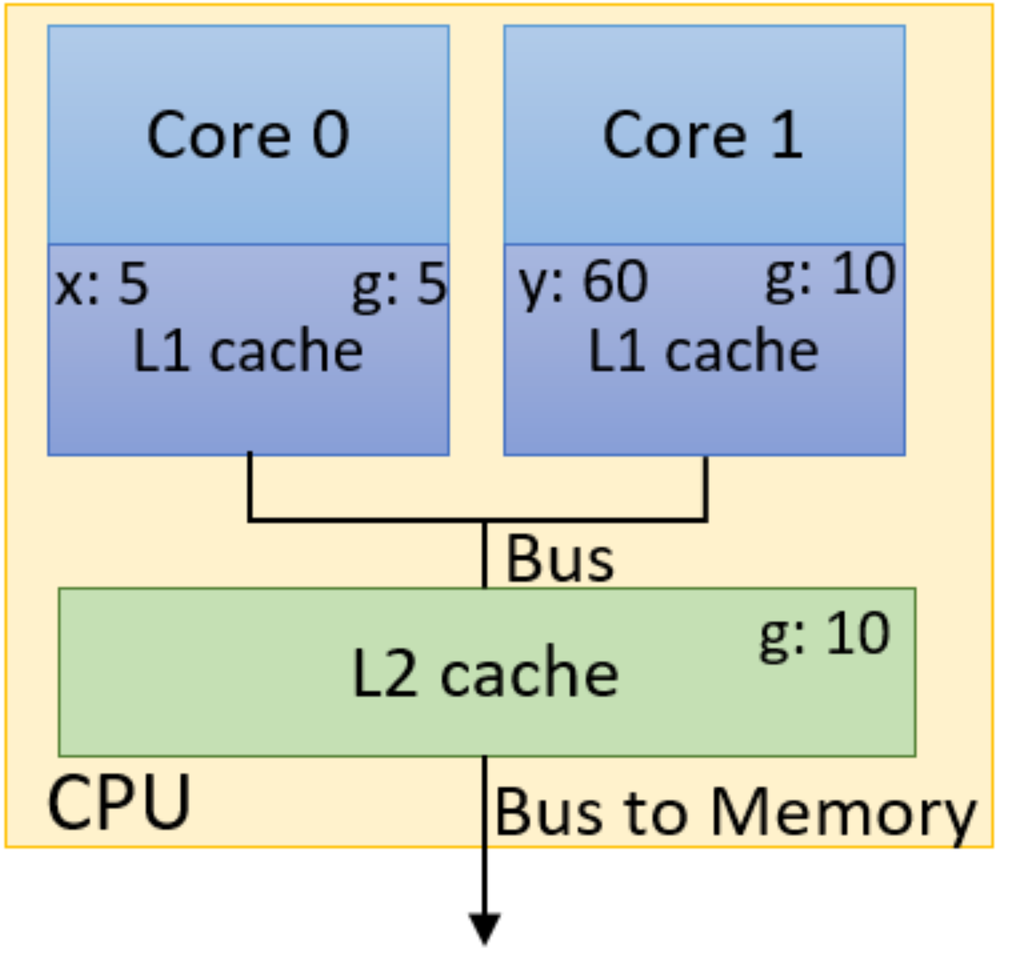
Cache Design

- The **valid bit** indicates if a block at a particular line in the cache is safe to use.
- Information is written to cache/memory based on two main strategies.
  - In the **write-through** strategy, the data is written to cache and main memory simultaneously.
  - In the **write-back** strategy, data is written only to cache and gets written to lower levels in the hierarchy after the block is evicted from the cache.

14.5.1. Caches on Multicore Systems

- Without a **cache coherence strategy** to ensure that each cache maintains a consistent view of shared memory, it is possible for shared variables to be updated inconsistently.

Time	Core 0	Core 1
0	g = 5	(other work)
1	(other work)	y = g*4
2	x += g	y += g*2



MSI protocol

- **Modified Shared Invalid (MSI)** protocol
  - an invalidating cache coherency protocol
- A common technique for implementing MSI is **snooping**.
  - “snoops" on the memory bus for possible write signals
  - If the snoopy cache detects a write to a shared cache block, it invalidates its line containing that cache block.

14.5.2. False Sharing

- This attempt to parallelize the countElems function is inaccurate
  - Because of data races affecting the **counts** array
- But it also gets slower when more cores are added

```
$ ./countElems_p 100000000 0 1
Time for Step 1 is 0.336239 s

$ ./countElems_p 100000000 0 2
Time for Step 1 is 0.799464 s

$ ./countElems_p 100000000 0 4
Time for Step 1 is 0.767003 s
```

```
void *countElems(void *args){
    //extract arguments
    //ommitted for brevity
    int *array = myargs->ap;
    long *counts = myargs->countp;

    //assign work to the thread
    //compute chunk, start, and end
    //ommited for brevity

    long i;
    //heart of the program
    for (i = start; i < end; i++){
        val = array[i];
        counts[val] = counts[val] + 1;
    }

    return NULL;
}
```

Li cache size

```
$ cat /sys/devices/system/cpu/cpu0/cache/coherency_line_size
64
```

Table 2. A Possible Execution Sequence of Two Threads Running `countElems`

Time	Thread 0	Thread 1
i	Reads array[x] (1)	...
i+1	Increments counts[1] (**invalidates cache line**)	Reads array[x] (4)
i+2	Reads array[x] (6)	Increments counts[4] (**invalidates cache line**)
i+3	Increments counts[6] (**invalidates cache line**)	Reads array[x] (2)
i+4	Reads array[x] (3)	Increments counts[2] (**invalidates cache line**)
i+5	Increments counts[3] (**invalidates cache line**)	...

False sharing

- The cache is invalidated each time any process writes to the **count** array
- repeated invalidation and overwriting of lines from the L1 cache is an example of **thrashing**
- The code gives the illusion of sharing the elements among the cores: **false sharing**

14.5.3. Fixing False Sharing

- One way is to pad the array (in our case counts) with additional elements so that it doesn't fit in a single cache line
- A better solution is to have threads write to local storage whenever possible.

```
//heart of the program
for (i = start; i < end; i++){
    val = array[i];
    local_counts[val] = local_counts[val] + 1; //update local counts
}

//update to global counts array
pthread_mutex_lock(&mutex); //acquire the mutex lock
for (i = 0; i < MAX; i++){
    counts[i] += local_counts[i];
}
pthread_mutex_unlock(&mutex); //release the mutex lock
```

14.6. Thread Safety

Safety and re-entrancy

- **Thread safe** functions
  - capable of being run by multiple threads while guaranteeing a correct result without unintended side effects
  - Not all C library functions are thread safe
- A function is **re-entrant** if it can be re-executed/partially executed by a function without causing issue
  - ensures that accesses to the global state of a program always result in that global state remaining consistent

Thread-unsafe functions

asctime()	ecvt()	gethostent()	getutxline()	putc_unlocked()
basename()	encrypt()	getlogin()	gmtime()	putchar_unlocked()
catgets()	endgrent()	getnetbyaddr()	hcreate()	putenv()
crypt()	endpwent()	getnetbyname()	hdestroy()	pututxline()
ctime()	endutxent()	getnetent()	hsearch()	rand()
dbm_clearerr()	fcvt()	getopt()	inet_ntoa()	readdir()
dbm_close()	ftw()	getprotobyname()	l64a()	setenv()
dbm_delete()	gcvt()	getprotobynumber()	lgamma()	setgrent()
dbm_error()	getc_unlocked()	getprotoent()	lgammaf()	setkey()
dbm_fetch()	getchar_unlocked()	getpwent()	lgammal()	setpwent()
dbm_firstkey()	getdate()	getpwnam()	localeconv()	setutxent()
dbm_nextkey()	getenv()	getpwuid()	localtime()	strerror()
dbm_open()	getgrent()	getservbyname()	lrand48()	strtok()
dbm_store()	getgrgid()	getservbyport()	mrand48()	ttyname()
dirname()	getgrnam()	getservent()	nftw()	unsetenv()
dlerror()	gethostbyaddr()	getutxent()	nl_langinfo()	wcstombs()
drand48()	gethostbyname()	getutxid()	ptsname()	wctomb()

- https://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_09.html

14.6.1. Fixing Issues of Thread Safety

- **countElemsStr** parses a string using **strtok()**

```
void countElemsStr(int *counts, char *input_str) {
    int val, i;
    char *token;
    token = strtok(input_str, " ");
    while (token != NULL) {
        val = atoi(token);
        counts[val] = counts[val] + 1;
        token = strtok(NULL, " ");
    }
}
```

```
$ ./countElemsStr 100000 1
contents of counts array:
9963 9975 9953 10121 10058 10017 10053 9905 9915 10040
```

Multithreaded version

- **strtok()** is not thread-safe
- Replace with **strtok_r()**

```
token = strtok(input_str + start, " ");
while (token != NULL) {
    val = atoi(token); //convert to an int
    local_counts[val] = local_counts[val] + 1;
    token = strtok(NULL, " ");
}

pthread_mutex_lock(&mutex);
for (i = 0; i < MAX; i++) {
    counts[i] += local_counts[i];
}
pthread_mutex_unlock(&mutex);
```

```
$ ./countElemsStr_p 100000 1 1
contents of counts array:
9963 9975 9953 10121 10058 10017 10053 9905 9915 10040

$ ./countElemsStr_p 100000 1 2
contents of counts array:
498 459 456 450 456 471 446 462 450 463

$ ./countElemsStr_p 100000 1 4
contents of counts array:
5038 4988 4985 5042 5056 5013 5025 5035 4968 5065
```

14.7. Implicit Threading with OpenMP

14.7. Implicit Threading with OpenMP

- Pthreads are great for simple applications
  - they become increasingly difficult to use as programs themselves become more complex
- POSIX threads are an example of **explicit parallel programming** of threads, requiring a programmer to specify exactly what each thread is required to do and when each thread should start and stop.
- The **Open Multiprocessing (OpenMP)** library implements an implicit alternative to Pthreads.
- Programmers parallelize components of existing, sequential C code by adding **pragmas** (special compiler directives) to parts of the code
  - Starting with **#pragma omp**

14.7.1. Common Pragmas

- **#pragma omp parallel**
  - creates a team of threads, with these clauses
    - **num_threads**
    - **private** variables that are local to each thread
    - **shared** lists variables that should be shared
    - **default** indicates whether the determination of which variables should be shared is left up to the compiler.
      - In most cases, we want to use **default(none)**

14.7.1. Common Pragmas

- **#pragma omp for**
  - each thread executes a subset of iterations of a for loop
- **#pragma omp parallel for**
  - creates a team of threads, then executes a for loop
- **#pragma omp critical**
  - This code is a critical section—only one thread should execute this code at a time

Functions

- There are also several functions that a thread can access that are often useful for execution. For example:
  - **omp_get_num_threads**
    - returns the number of threads in the current team that is being executed.
  - **omp_set_num_threads**
    - sets the number of threads that a team should have.
  - **omp_get_thread_num**
    - returns the identifier of the calling thread.

14.7.2. Hello Threading: OpenMP flavored

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void HelloWorld( void ) {
    long myid = omp_get_thread_num();
    printf( "Hello world! I am thread %ld\n", myid );
}
```

```c
    nthreads = strtol( argv[1], NULL, 10 );

    #pragma omp parallel num_threads(nthreads)
        HelloWorld();
```

```
$ gcc -o hello_mp hello_mp.c -fopenmp

$ ./hello_mp 4
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 0
Hello world! I am thread 1
```

14.7.3. A More Complex Example: CountSort in OpenMP

- The important code in main()

```
//allocate counts array and initializes all elements to zero.
int counts[MAX] = {0};

countElems(counts, array, length); //calls step 1
writeArray(counts, array); //calls step2
```

Parallelizing CountElems Using OpenMP

```
void countElems(int *counts, int *array, long length) {

    #pragma omp parallel default(none) shared(counts, array, length)
    {
        int val, i, local[MAX] = {0};
        #pragma omp for
        for (i = 0; i < length; i++) {
            val = array[i];
            local[val]++;
        }

       #pragma omp critical
       {
           for (i = 0; i < MAX; i++) {
               counts[i] += local[i];
           }
       }
    }
}
```

Results

- Almost linear speedup!

```
$ ./countElems_mp 100000000 1
Run Time for Phase 1 is 0.249893

$ ./countElems_mp 100000000 2
Run Time for Phase 1 is 0.124462

$ ./countElems_mp 100000000 4
Run Time for Phase 1 is 0.068749
```

Kahoot!

Ch14b