EECE7352 - COMPUTER ARCHITECTURE: HOMEWORK 5

ANNA DEVRIES[1]

CONTENTS

LIST OF FIGURES

LIST OF TABLES

[1] *Department of Electrical and Computer Engineering, Northeastern University, Boston, United States*

1 PART A

1.1 Cache Coherency

Description: In this problem you will review cache coherency protocols.

1.1.1 *Discuss the tradeoffs between using a snoopy cache coherency protocol and a directory based cache coherency protocol.*

1.1.2 *Discuss the role of the Exclusive state introduced in the MESI protocol.*

1.1.3 *Provide an example of false sharing.*

1.1.4 *Discuss how the ski rental algorithm could be used to decide when to broadcast versus invalidate?*

1.2 Solution

A) Snoopy and directory bases cache coherency protocols provide methods to safely share memory between multiprocessors by ensuring consistent data across caches. Multiprocessors share address space for communication to one another; however, this creates problems during writes. When one process writes to a location, all caches must be updated to remain consistent.

Snoopy cache coherency protocol works as follows. Processors broadcast to each other via address lines of a shared bus every time memory accesses occur. Caches monitor this broadcasting method, "snooping" the bus. Then, they determine whether they must write-update or write-invalidate. The snoopy protocol is shown in Figure 1 [4].

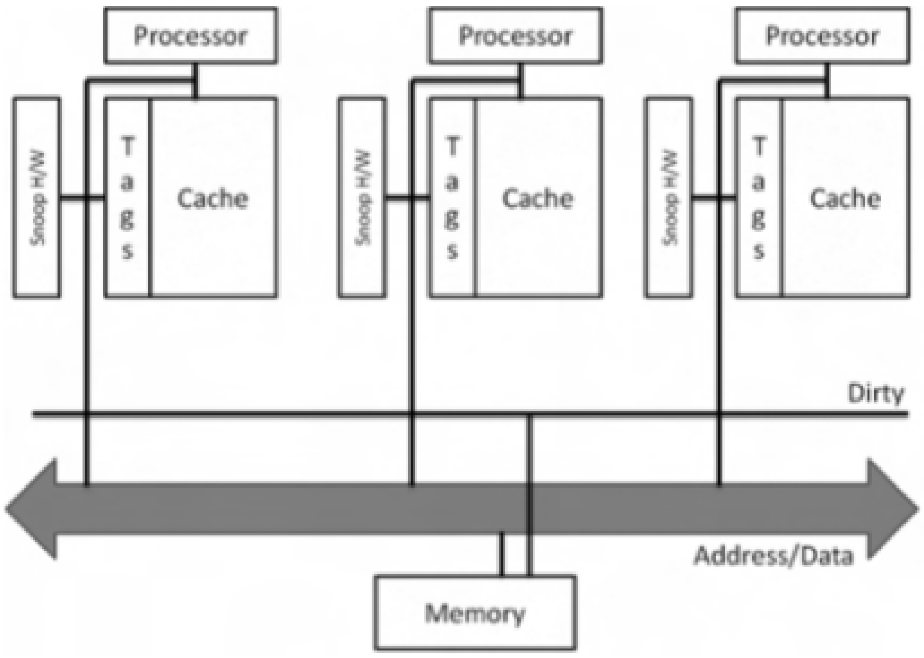


Figure 1: Snoopy cache coherency protocol diagram.

Most notably, this protocol requires a broadcast medium, thus, it is applied to small-scale bus-based multiprocessors. A major advantage to snoopy protocols is the low average miss latency for cache to cache misses [4]. This advantage is due to every memory write being broadcasted to every cache each time. However, snoopy protocols create large overhead and the speed of the shared broadcast method limits the bandwidth required for sending messages [4]. Furthermore, these protocols inefficiency dissipate power.

Directory based cache coherency protocol works as follows. A main directory is shared across processor caches. This directory acts similar to a look-up table, allowing each processor to identify coherency and consistency on data. The directory protocol is shown in Figure 2 [4].

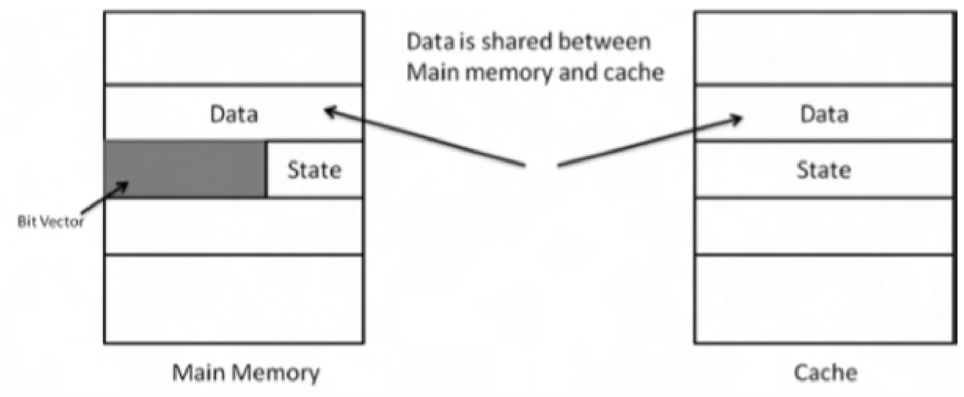


Figure 2: Directory cache coherency protocol diagram.

Most notably, directory-based protocols scale well to a large number of processors in a shared-memory multi-processors unit. A major advantage to directory protocols are there ability to exploit arbitrary point-to-point interconnects [4]. This advantage is due to its use of a shared directory that can be accessed without an expensive broadcasting method. However, directory protocols depend on directory accesses and the extra interconnect traversal to prevent cache to cache misses (increasing likelihood of cache misses), and it requires additional storage units and methods to manipulate that storage [4].

B) MESI protocol places each cache line in one of four states: invalid, shared, exclusive or modified. These states are used to promote memory consistency between caches. The exclusive state specifically states that this content is not in any other caches. Thus, the cache may overwrite without consulting other caches.

Figure 3 provides a visualization for how the cache line state is determined [8]. Exclusive state represents a valid and clean line. This means that the cache line is valid, no other caches have invalidated this line by updating it in memory. Finally, if the line is not shared among other caches, then it is exclusive. If the cache must write the line to memory, the exclusive states requires no further bus transactions to occur to inform other caches of this update.

C) False sharing occurs when threads on different processors modify different parts of the same cache line. As these threads update memory, they invalidate the cache line and may cause thrashing. An example of false sharing is provided in the code below, provided by Intel [3].

```
double sum=0.0, sum_local[NUM_THREADS];
//pragma omp parallel num_threads(NUM_THREADS){
```

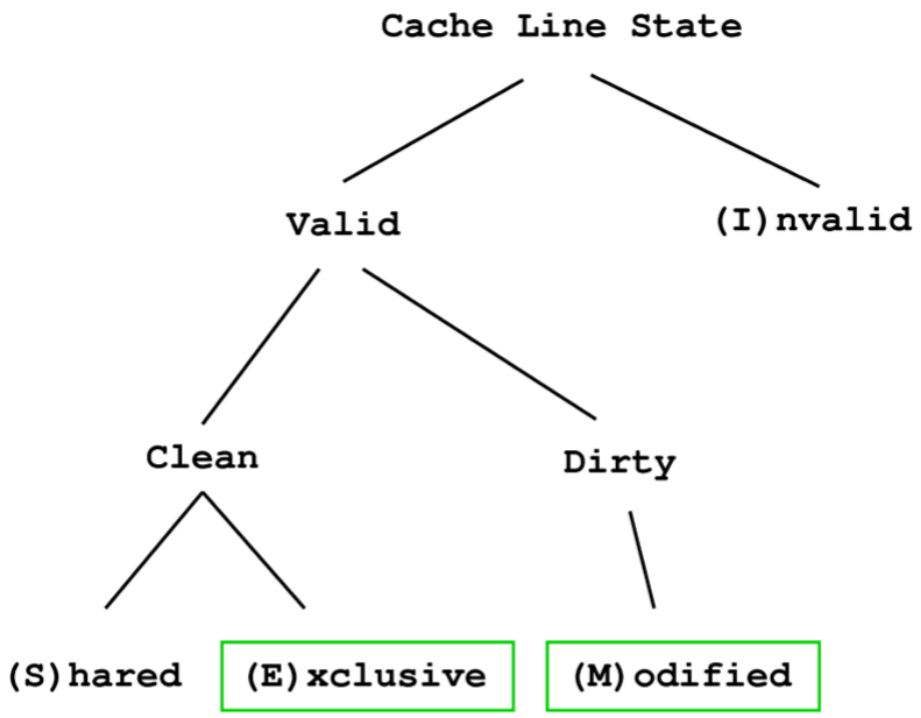


Figure 3: MESI diagram.

```
int me = omp_get_thread_num();
sum_local[me] = 0.0;
//pragma omp for
for (i = 0; i < N; i++)
sum_local[me] += x[i] * y[i];
//pragma omp atomic
sum += sum+local[me];
}
```

Algorithm 1: False Sharing Example

In this code, multiple threads are accesses/modifying variables on the same cache line. Since array sum_local is small enough to fit on a single cache line and dimensioned by number of threads, false sharing may occur [3]. Figure 4 illustrates this problem [3]. As one thread updates the first element of the array, another thread may try to update the second element. This will invalidate the cache line for all processors.

D) Broadcasting is an expensive decision; this requires the cache to send messages to every cache in the system. Broadcasting creates traffic and requires a large amount of power dissipation. Invalidating is a smaller expense but incurs a penalty if a cache needs access to it shortly after. The ski rental problem is similar – it looks at the rent/buy problem. Skis cost a lot; however, eventually, it is no longer worth the cost of renting either. The broadcasting decision is related to buying skis and the invalidation decision is related to renting skis. These can be compared to the break-even algorithm or randomized algorithm presented in the ski-rental problem.

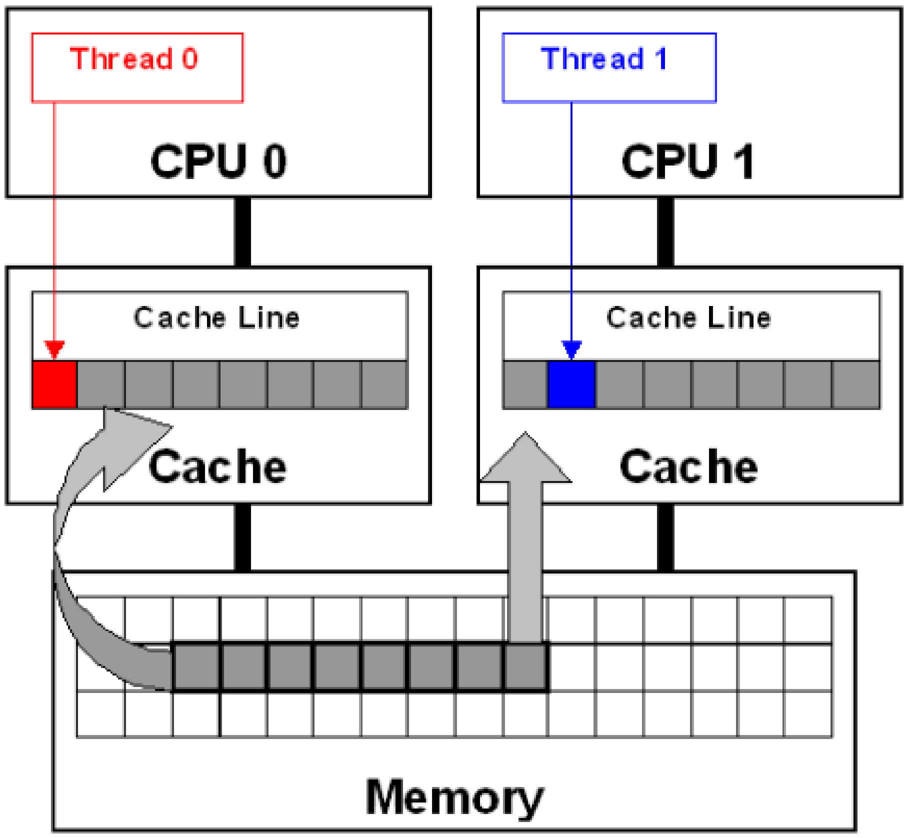


Figure 4: False sharing example illustration.

2 PART B

2.1 TLBs

Description:In this problem, find the organization for the TLBs present on two different processors, one developed by IBM and the other developed by ARM. Make sure to cite your sources. Provide the details of the organization the whole TLB hierarchy, and the format of a single entry in an L1 TLB (either I or D).

2.2 Solution

For this problem, I researched the Power9 processor from IBM and Cortex-A15 MPCore processor from ARM. The Power9 processor has two translation architectures: radix translation and hashing page table translation [7]. The TLB in radix mode contains 512 entries (128 entries x 4-way set-associative) [7]. It utilizes the following hash algorithms: [LPIDR(29:31) XOR EA(45:47)] | | [PIDR(28:31) XOR EA(48:51)] for a 4KB page size, [LPIDR(29:31) XOR EA(41:43)] | | [PIDR(28:31) XOR EA(44:47)] for a 64 KB page size, [LPIDR(29:31) XOR EA(36:38)] | | [PIDR(28:31) XOR EA(39:42)] for 2 MB page size, and [LPIDR(29:31) XOR EA(27:29)] | | [PIDR(28:31) XOR EA(30:33)] for 1 GB page size [7].

The remainder 512 entries are utilized as a page-walk cache to cache page directory entries and each cache level has its own TLB [7]. Furthermore, all entries are tagged with LPID and PIDR values when translating [7]. The TLB in HPT mode contains 1024 entries (256 entries x 4-way set associative) [7]. Each entry can support 256 MB or 1 TB segment sizes and the miss penalty is approximately 30 – 36 cycles [7]. The format of a single entry in an L1 data cache TLB in radix mode is as follows:

$\|\underline{\qquad\qquad\qquad\qquad\qquad}\|\underline{\qquad\qquad\qquad\qquad\qquad}\|\underline{\qquad\qquad\qquad\qquad}\|$
$LPID/PIDRHashValue(25:31) \quad Tag:level1EA(12:24) \quad ByteOffset(0:11)$

The Cortex-A15 MPCore processor utilizes a 2-level TLB architecture [5]. The L1 and L2 level TLBs do not need to be flushed nor do they support locking entries [5]. The L1 data and instruction caches both support 32 entries, are full-associative and only support cache entries at the 4 KB granularity of VA to PA mappings [5]. The L1 instruction and data TLBs provide a single clock cycle access to translation if it receives a hit, return the physical address for comparison [5]. The L1 data TLB, however, provides two separate TLBs with 32 entries each, rather than just one. These are used for separate data loads/stores [5]. The L2 TLB is a unified data and instruction cache. It supports 512 entries and is 4-way set-associative [5]. The L2 TLB supports 4 K, 64 K, 1 MB and 16 MB page sizes [5]. The format of a single entry in an L1 instruction cache TLB is as follows:

$\|\underline{\qquad\qquad\quad}\|\underline{\qquad\qquad\quad}\|\underline{\qquad\qquad}\|\underline{\qquad\qquad}\|$
$Granulesize \quad Tablelevel \quad Blocksize \quad Pagesize$

3 PART C

3.1 Multi-Core

Description: In this problem you will utilize parallel threads to explore the use of multiple cores. Use the pi.c parallel program that utilizes pthreads to compute the value of PI using integration. The program computes the following equation:

PI = 4 * arctan(1)
Where the arctan(x) = integral from 0 to x of (1/(1+x2))

You can adjust the number of intervals chosen for the integration and the number threads used of on the command line. Note that, the more intervals used, the better the approximation.
You will use pthreads to run the pi.c program in parallel. You will need to compile your program with the –lpthread switch and the –lm switch with gcc. Run this on the 64-bit COE Linux systems. Make sure to discuss the CPU you are running on and how many cores/threads are available on the system (look in /proc/cpuinfo).

3.1.1 *Select a value for the number of intervals to get reasonable timing results. Plot the speedup you get by increasing the number of threads. Discuss your results. Include results for 1, 2, 4, 8, 16, 32 and 64 threads. Make sure to use a large enough number of intervals to obtain meaningful runtimes and accurate values for pi.*

3.1.2 *Discuss the trends you are seeing in the graphs in part a.*

3.2 Solution

The CPU has 23 processors. Each processor is an Intel Xeon CPU X5650 at 2.67 GHz. The cache size is 12288 KB with 64-bit cache alignment. Additionally, the address sizes are 40-bits physical, 48-bits virtual and the CPU has 6 cores/threads per processor.

For this problem, I utilized 1,000,000,000 intervals for the pi.c program. I calculated speedup as Speedup = Time at 1 thread / Time at x threads. Table 1 and figure 5 illustrate the results.

Table 1: Pi.c Speedup Results.

Number of Threads	Total Time (sec)	Speedup
1	9.21	1
2	9.21	1
4	9.40	0.9798
8	9.66	0.9534
16	12.38	0.7439
32	17.81	0.5171
64	18.18	0.5066

Threads work best when you have a large amount of independent data (data that does not interact with each other). In this example, the data completely builds upon itself to create an estimated pi value. The local sum is added together but most be locked with a mutex to control memory synchronization between threads. This type of programming slows down with increased threads both due to the overhead to create threads and the locks to allow accurate memory synchronization.

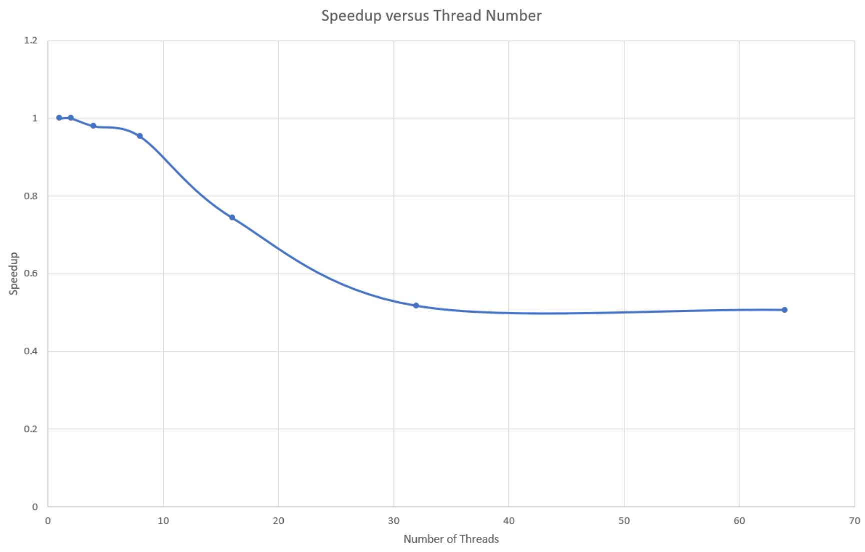


Figure 5: Speedup versus Thread Number.

4 PART D

4.1 Power Challenges

Description: Since the turn of the century, power has become a first-rate architectural design constraint in the design of microprocessors. Trevor Mudge describes these challenges in his 2001 paper that in provided.

4.1.1 *Read Mudge's paper. Then for one of the approaches that he describes, find a current microprocessor available on the market that has adopted that technique. Try to provide as much detail as possible about the implementation.*

4.1.2 *Find performance and power rating information for one current embedded CPU and one current high-performance multi-core CPU. Discuss if it would make sense to combine many embedded CPUs to replace one high-performance CPU, while reducing the power budget. Back up your conclusions with quantitative arguments. Make sure to cite your sources.*

4.2 Solution

A) The Intel i7 processors utilizes clock gating to reduce power consumption. Clock gating is commonly utilized in processors; it turns off components currently not being used. Specifically, i7 has a power saving mode which affects C-states. C-states refers to states the processor or individual cores may be in. The first four states are C0, C1, C2, and C3. C0 is the default state, C1 is an idle state that may immediately becoming working again, C2 is an idle state that will take a delay to return to working, and C3 is the sleeping state [6]. C3 is the most power efficient, it scales down all frequencies to their minimum operating values and flushes L1-L3 caches. These states allow the processor to shut down different cores so that only cores actively working are consuming normal power.

B) For this question, I researched the Intel Atom C3958 embedded processor and the Intel Xeon E-2186G high-performance multi-core CPU. I utilized Anandtech Benchmark Comparison to create table

2 [1]. This table illustrates the performance ratings of each processor at various benchmarks. Higher numbers represent better performance unless otherwise stated.

Table 2: Processor Performance Comparison.

Benchmark	Intel Atm C3958	Intel Xeon E-2186G
Mozilla Kraken 1.1 (lower is better)	3,210 ms	913 ms
Google Octane 2.0	11,544	401,941
Speedometer 2	24	89
7-Zip 1805 Combined	32,667	50,683
PCMark10-1 Essential Set	5,349	10,209
Geekbench 4 – ST Crypto	972	5,419
Geekbench 4 – ST Integer Set	1,608	5,742
Geekbench 4 – ST Floating Point Set	1,096	5,599

The Intel Atom produces a thermal design power of 31 Watts (average power consumption) and operates at a frequency of 2000 MHz; whereas, the Intel Xeon produces a thermal design power of 95 Watts and operates at a frequency of 3800 MHz [2].

The Xeon outperforms the Atom in all benchmark tests; however, with a much larger power draw (three times the power draw). Suppose we added two additional embedded processors (three in total) of the Intel Atom, we would expect benchmark performance above to triple. For some applications, this performance would still underperform the Xeon (i.e. Google Octane 2.0 and Geekbench benchmarks); however, performance for all other benchmarks above would outperform the Xeon with a thermal design power of about 93 Watts. For example, the 7-Zip 1805 benchmark would rise to approximately 98,000 and the PCMark10-1 Essential Set would become 16,047, both performances far exceeding the Xeon. Therefore, replacing the embedded processor could make sense depending on the workload.

5 EXTRA CREDIT 1

5.1 GPU Architecture Comparison

Description: Graphics Processing Units are highly parallel single-instruction multi-threaded architectures that achieve impressive speedups for a range of applications. Using the NVIDIA V100 architecture, describe the memory hierarchy present on this device, and compare how it differs from the memory hierarchy of the AMD Ryzen CPU.

5.2 Solution

N/A

6 EXTRA CREDIT 2

6.1 Parallel Programming (Extra credit)

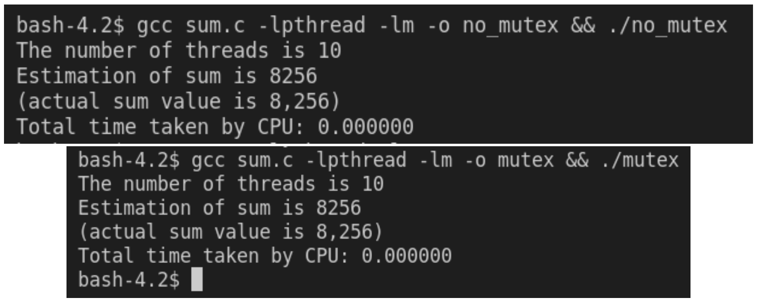
Description:Using the darts.c program as a model, write a pthreads parallel program that computes the sum of 128 numbers. The numbers are: 1, 2, 3, 4. . . . 128. The sum should be: 129 * 64 = 8,256. Declare sum as a global variable. Vary the number of threads. Discuss whether you need to use a mutex to protect sum. Discuss what happens when you don't use a mutex, and show the output of your program when you do. Include your source code in your homework submission.

6.2 Solution

Table 3 shows the results of my summation program and Figure 6 illustrates an example run.

Table 3: Parallel Programming Summation Results.

Number of Threads	With Mutex Lock	Without Mutex Lock
1	8,256	8,256
2	8,256	8,256
3	8,256	8,256
4	8,256	8,256
5	8,256	8,256
6	8,256	8,256
7	8,256	8,256
8	8,256	8,256
9	8,256	8,256
10	8,256	8,256
11	8,256	8,256
12	8,256	8,256
13	8,256	8,256
14	8,256	8,256
15	8,256	8,256
16	8,256	8,256
32	8,256	8,256
64	8,256	8,256



```
bash-4.2$ gcc sum.c -lpthread -lm -o no_mutex && ./no_mutex
The number of threads is 10
Estimation of sum is 8256
(actual sum value is 8,256)
Total time taken by CPU: 0.000000
```

```
bash-4.2$ gcc sum.c -lpthread -lm -o mutex && ./mutex
The number of threads is 10
Estimation of sum is 8256
(actual sum value is 8,256)
Total time taken by CPU: 0.000000
bash-4.2$
```

Figure 6: Summation code with no lock mutex [A], summation code with lock mutex [B].

Both the mutex locked and non-mutex locked programs produces identical and correct results (8,256). The mutex lock is utilized to prevent simultaneous accesses of multiple threads to the same memory location. However, the summation program manages to avoid this issue by assigning separate sections of the array to each thread for summing. Therefore, for this program, the mutex is not needed; however, other parallel programs (such as pi.c) need a lock to prevent a thread from accessing a resource currently in use.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

/* global variables */
volatile int sum = 0.0;
pthread_mutex_t sumLock;        /* how we synchronize writes to 'sum' */
int numThreads;                 /* how many threads we use */

#define THREAD_NUM 10
#define SIZE 128

void *computeSUM(void *id)
{
    int localSum = 0;
    int threadID = *((int*)id);
    int i;

    int array[SIZE];
    for(i = 0; i < SIZE; i++){
      array[i] = i + 1;
    }
    int partition = SIZE / THREAD_NUM;
    int max = THREAD_NUM - 1;

    for(i = (threadID * partition); i < SIZE; i++){
      if(THREAD_NUM != 0 && (THREAD_NUM & (THREAD_NUM - 1)) != 0){
        if(threadID != max && i < ((threadID * partition) + partition)){
          localSum += array[i];
        }
        if(threadID == max){
          localSum += array[i];
        }
      }
      else{
        if(i < ((threadID * partition) + partition)){
          localSum += array[i];
        }
      }
    }

    pthread_mutex_lock(&sumLock);
    sum += localSum;
    pthread_mutex_unlock(&sumLock);

    return NULL;
}

int main(int argc, char *argv[]){
  clock_t start_t, end_t;
  double total_t;
  pthread_t *threads;          /* dynarray of threads */
  void *retval;                /* unused; required for join() */
  int *threadID;               /* dynarray of thread id #s */
  int i;                       /* loop control variable */

  printf("The number of threads is %i \n", THREAD_NUM);

  start_t = clock();

  threads = malloc(THREAD_NUM*sizeof(pthread_t));
  threadID = malloc(THREAD_NUM*sizeof(int));

  pthread_mutex_init(&sumLock, NULL);

  for (i = 0; i < THREAD_NUM; i++) {
    threadID[i] = i;
    pthread_create(&threads[i], NULL, computeSUM,threadID+i);
  }

  for (i = 0; i < THREAD_NUM; i++) {
     pthread_join(threads[i], &retval);
  }

  end_t = clock() - start_t;

  printf("Estimation of sum is %d \n", sum);
  printf("(actual sum value is 8,256)\n");

  total_t = ((double)end_t) / CLOCKS_PER_SEC;
  printf("Total time taken by CPU: %f\n", total_t  );

  return 0;
}
```

Algorithm 2: Summation Parallel Programming Code

REFERENCES

[1] Cpu 2019 benchmarks. URL https://www.anandtech.com/bench/product/2351?vs=2286. [Online; accessed 22-November-2019].

[2] Intel processor specifications. URL http://www.cpu-world.com/. [Online; accessed 23-November-2019].

[3] Avoiding and identifying false sharing among threads, 2011. URL https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads. [Online; accessed 19-November-2019].

[4] S. Al-Hothal, S. Soomro, K. Tanvir, and R. Tuli. *Snoopy and Directory Based Cache Coherence Protocols: A Critical Analysis*. 2010.

[5] ARM. *ARM Cortex-A15 MPCore Processor Technical Reference Manual*.

[6] Dungeoner. Overclocking and intel's power management settings (eist, c-states, turbo boost). URL http://www.dungeoner.com/en/overclocking-and-intels-power-management-settings-eist-c-states-turbo-boost/. [Online; accessed 20-November-2019].

[7] IBM. *POWER9 Processor User's Manual*. 2018.

[8] R. Mullins. Chip multiprocessors. URL https://www.cl.cam.ac.uk/~rdm34/acs-slides/lec4.pdf. [Online; accessed 19-November-2019].