ECE 420 Assignment 1

Arun Woosaree

February 10, 2021

1 Flynn's Taxonomy

1.1 SISD

SISD stands for Single Instruction Stream, Single Data Stream. A SISD system has a single uniprocessor, which executes a single instruction stream to operate on data stored in a single.

An example that would fit in this category is a single core superscalar processor like the AMD 29050 Reference: `https://en.wikipedia.org/wiki/Superscalar_processor`

1.2 SIMD

SIMD stands for Single Instruction Stream, Multiple Data Stream. A SIMD system has multiple processing units which all simultaneously do the same operation on multiple points in a data pool in a synchronized fashion.

An example that would fit in this category is a modern graphics processing unit, like the AMD Radeon 6900 XT

1.3 MISD

MISD stands for Multiple Instruction Stream, Single Data Stream. A MISD system has multiple processing units performing different operations on the same data simultaneously. It can be used for reliability or fault tolerance, if the same operation is being done multiple times on the same data.

An example of an MISD system is the computer responsible for the Space Shuttle flight control. The Space Shuttle is also known as the low orbit spacecraft that helped launch the Hubble telescope, among other space missions. References: `https://en.wikipedia.org/wiki/MISD#cite_note-1` `https://en.wikipedia.org/wiki/Space_Shuttle`

1.4 MIMD

MIMD stands for Multiple Instruction Stream, Multiple Data Stream. A MIMD system has many processing units which each perform operations independently from each other at the same time. That is, each unit can be doing different operations on different pieces of data at any given time.

An example of a MIMD system is almost any modern multicore processor, like the AMD Ryzen 9 5950X. Another example could be a computer cluster, or a network of workstations.

2 Shared Memory vs Distributed Memory

2.1 Shared Memory

2.1.1 Pros

- user friendly programming perspective to memory
- data sharing between tasks is fast and uniform

2.1.2 Cons

- no scalability between memory and CPUs
- if you want more power,

2.2 Distributed Memory

2.2.1 Pros

- memory is scalable with the number of processors
- this is cost effective, because lots of cheap, commodity hardware can be used as opposed to upgrading a monolithic, more expensive structure

2.2.2 Cons

- data communication between processors is more difficult
- it is difficult to map existing data structures to the memory organization
- memory access times can be very different. For example, local access times vs over a network access times can be much slower compared to a MIMD system
- more overhead for the programmer. The programmer has to think about message passing

3 Amdahl's Law

1. *If y fraction of a serial program cannot be parallelized, $1/y$ is an upper bound on the speedup of its parallel program, no matter how many processing elements are used.*

Proof. If y is the fraction of a serial program that cannot be parallelized, then the fraction x which is the fraction of the program that can be parallelized is found by:

$$x = (1 - y) \tag{1}$$

According to Amdahl's law, the upper limit for speedup of a parallel program is:

$$\lim_{p \to \infty} S(p) \leq \frac{1}{1 - x} \tag{2}$$

where p is the number of processing elements.

Substituting equation 1 into 2, we get:

$$\lim_{p \to \infty} S(p) \leq \frac{1}{1 - (1 - y)} \tag{3}$$

Simplifying 3, we get:

$$\lim_{p \to \infty} S(p) \leq \frac{1}{y} \tag{4}$$

This proves that $1/y$ is an upper bound for the speedup of a program, no matter how many processing elements are used, given that y is the fraction of a serial program that cannot be parallelized □

4 Efficiency

Given:

p is the number of processors, n is the problem size, and

$$T_{serial} = n$$

and

$$T_{parallel} = \frac{n}{p} + \log_2(p)$$

Using Amdahl's law, Efficiency is:

$$E(p) = \frac{S(p)}{p} = \frac{T_{serial}/T_{parallel}}{p}$$

Substituting the above equations, we get

$$E(p) = \frac{n}{p(\frac{n}{p} + \log_2(p))}$$

This simplifies to

$$E(p) = \frac{n}{n + p\log_2 p}$$

If we increase p by a factor of k, how much would we need to increase n by a factor of say, q?

$$\begin{aligned}
\frac{n}{n + p\log_2 p} &= \frac{qn}{qn + pk\log_2 pk} \\
\frac{1}{n + p\log_2 p} &= \frac{q}{qn + pk\log_2 pk} \\
qn + pk\log_2(pk) &= qn + qp\log_2 p \\
pk\log_2(pk) &= qp\log_2 p \\
k\log_2(pk) &= q\log_2 p \\
q &= \frac{k\log_2(pk)}{\log_2 p}
\end{aligned}$$

Thus, we can clearly see that if p is increased by a factor of k, the factor q which n would also have to be increased by to maintain the same efficiency is not equal to k.

There is one interesting result: If you were to somehow have *infinite* processors p:

$$\lim_{p\to\infty} \frac{k\log_2(pk)}{\log_2 p} = k$$

so, if there were infinite processors, n would increase at the same rate as p, but this theoretical scenario does not make much sense haha.

5 Stack vs Heap allocation in pthread

The two programs have different output. The first program always has an exit code of 10, while the second program seems to return a random number.

What's going on here is that in program 1, the memory is allocated on the heap, while in program 2, the memory is allocated on the stack. In the first program, because the memory is allocated on the heap, it is available globally to the entire program, including other threads, until the program ends, or until the memory is manually freed. The memory allocated for pointer b is set to the value of 10, and when the thread exits, the memory is left unchanged because `free()` has not been called on it. This is why the correct value of 10 is printed out by the program. In program 2, the pointer b points to a variable allocated on the stack. This is not good! When the thread exits, that region of memory is freed because the number variable goes out of scope. That leaves b pointing to an unallocated region in memory, and this can cause undefined behaviour. This is why I get random numbers on my machine. When the number variable in program 2 goes out of scope, the pointer to that value is known as a dangling pointer.

6 Condition variables and mutex

```c
#include <pthread.h>
typedef struct {
  pthread_mutex_t count_lock;
  pthread_cond_t ok_to_proceed;
  int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
  b->count = 0;
  pthread_mutex_init(&(b->count_lock), NULL);
  pthread_cond_init(&(b->ok_to_proceed), NULL);
}

void mylib_barrier(mylib_barrier_t *b, int num_threads) {
  pthread_mutex_lock(&(b->count_lock));
  b->count++;
  if (b->count == num_threads) {
    b->count = 0;
    pthread_cond_broadcast(&(b->ok_to_proceed));
  } else
    while (pthread_cond_wait(&(b->ok_to_proceed), &(b->count_lock)) != 0)
      ;
  pthread_mutex_unlock(&(b->count_lock));
}
```

7 Producer/consumer buffer using mutex and condition variables

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

#define BSIZE 3
#define NUMITEMS 6

typedef struct {
  char buf[BSIZE];
  int occupied;
  int nextin, nextout;
  pthread_mutex_t mutex;
  pthread_cond_t more;
  pthread_cond_t less;
} buffer_t;
buffer_t buffer;
```

```
void *producer(void *);
void *consumer(void *);

#define NUM_THREADS 2
pthread_t tid[NUM_THREADS]; /* array of thread IDs */

int main(int argc, char *argv[]) {

  int i;

  pthread_cond_init(&(buffer.more), NULL);
  pthread_cond_init(&(buffer.less), NULL);
  pthread_mutex_init(&buffer.mutex, NULL);
  pthread_create(&tid[1], NULL, consumer, NULL);
  pthread_create(&tid[0], NULL, producer, NULL);
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

  printf("\nmain() reporting that all %d threads have terminated\n", i);
} /* main */

void *producer(void *parm) {

  char item[NUMITEMS] = "HELLO.";
  int i;

  printf("producer started.\n");

  for (i = 0; i < NUMITEMS; i++) {
    /* produce an item, one character from item[] */
    if (item[i] == '\0')
      break; /* Quit if at end of string. */

    pthread_mutex_lock(&(buffer.mutex));

    if (buffer.occupied >= BSIZE)
      printf("producer waiting.\n");
    while (buffer.occupied >= BSIZE)
      pthread_cond_wait(&(buffer.less), &(buffer.mutex));
    printf("producer executing.\n");

    buffer.buf[buffer.nextin] = item[i];
    buffer.nextin++;
    buffer.nextin %= BSIZE;
    buffer.occupied++;
    /* now: either buffer.occupied < BSIZE and buffer.nextin
       is the index of the next empty slot in the
       buffer, or buffer.occupied == BSIZE and
       buffer.nextin is the index of the next
       (occupied) slot that will be emptied by a consumer

       (such as buffer.nextin == buffer.nextout) */
    pthread_cond_signal(&(buffer.more));
    pthread_mutex_unlock(&(buffer.mutex));
  }
  printf("producer exiting.\n");
  pthread_exit(0);
}

void *consumer(void *parm) {
  char item;
  int i;
  printf("consumer started.\n");
  for (i = 0; i < NUMITEMS; i++) {
    /*Insert here*/

    pthread_mutex_lock(&buffer.mutex);
    if (buffer.occupied < 1)
      printf("consumer waiting.\n");
    while (buffer.occupied < 1)
      pthread_cond_wait(&buffer.more, &buffer.mutex);
    printf("consumer executing.\n");

    item = buffer.buf[buffer.nextout];
    printf("%c\n", item);
    buffer.nextout++;
    buffer.nextout %= BSIZE;
    buffer.occupied--;

    pthread_cond_signal(&buffer.less);
    pthread_mutex_unlock(&buffer.mutex);
  }
  printf("consumer exiting.\n");
  pthread_exit(0);
}
```

8 False sharing

The matrix with dimensions 8 x 8,000,000 is most likely to have false sharing. In the Chapter 1 slides, we see that the efficiency with 2 and 4 threads is lower than with the other two matrices with 8,000,000 x 8 and 8000 x 8000 dimensions. This is because when we are doing the matrix multiplication, we are updating only eight locations that are close together in memory. So, when we assign more threads to do the multiplication task, they are continuously updating regions in memory that are close together. False sharing happens when threads with separate caches access different variables that belong to the same cache line. Because there are only 8 location in memory here, it is likely that they are in the same cache line, so when multiple threads are updating those regions in memory, it is very likely that they are going to invalidate the entire cache line, even if the variable another thread is currently reading has not been updated. The other two matrices do not see the same problem, to the same degree, because with more locations to write to, it is less likely that a thread will accidentally invalidate the cache line of another thread since the data that a thread is currently working on is less likely going to be on the same cache line as another thread.

There are a few ways we can avoid false sharing. Here are two of them:

8.1 Padding

We could pad the global data structure with dummy elements to make sure that any update by one thread won't affect the other's cache line. This would involve using the compiler `__declspec (align(n))` statement

8.2 Local variable

We could avoid frequent writing to the global data that is accessed from multiple threads by having each thread use its own private storage during the multiplication loop. When the thread is done, then we can update the shared storage. It should be noted that this method does not completely avoid false sharing, since we update the global object at the end, however, the number of writes to the global object is greatly reduced. This lowers the chance of the cache being invalidated by a lot, compared to not using a local variable.