**Justin Ciocoi**

**Oct. 3, 2023**

# CSCI 375 Class Notes

# Operating Systems

## Synchronization

- `#pragma omp {}` denotes parallel portion of code
- The number of threads can be denoted as a parameter of pragma omp
  - If number of threads isnt specified, system will automatically allocate appropriate number
- If a variable is declared inside of the parallel portion, its scope is *private* for each thread
- If a variable is declared outside of the parallel portion, its scope is *shared* among the threads
- The scope of variables can be declared in parameters of the pragma omp directive **Data Races**
- When two things should use a certain portion of data in a certain sequential fashion
  - This will happen when multiple threads are writing or reading the same shared data simultaneously
  - If only data reading is occurring, there is no data dependency involved
- This shouldn't be done within a parallel portion of a code, as this could result in an "incorrect winner" of the data race
  - This could result in programs that will run, but will produce unexpected or incorrect values
- Data dependencies will indicate if a portion of code can be done in parallel or not

#### Dividing for loops in parallel portions

- Start with the loop

  ```
  for(int i=0; i<8; i++) x[i]=0
  ```

- Assume number of threads is 4

- Then, we can assume the associated thread IDs are 0, 1, 2, and 3
- So we can use the following code to divide the parallel sections within a for loop

```
#pragma omp parallel
{
    int id = omp_get_thread_num(); //id 0,1,2,3
    for(int i=id; i<8; i+=numt)
        x[i]=0;
}
```

### Using pragma parallel for

- For the same original for loop as above, we can use the following code to execute the aforementioned for loop in parallel

```
#pragma omp parallel for
{
    for(int i=0;i<8;i++)
        x[i]=0;
}
```

### Atomic Construct

```
sum = 0;
#pragma omp parallel for shared(sum, a, n) private(i)
for(i=0; i<n; i++)
{
    #pragma omp atomic
    sum+=a[i];
}
std::cout<<"Value of sum: "<<sum<<std::endl;
```

- "Atomizes" certain machine language instruction sets (groups them together) such that when they are started, they must first be finished before another thread can operate
- This therefore operates like in a serial fashion, but is even less efficient than a traditional serial code
- The following is a better implementation of the Atomic Construct, where it is only applied for adding local sums in order to obtain the total sum

```
sum = 0;
#pragma omp parallel shared(n, a, sum) private(i)
{
  sumLocal = 0;
  #pragma omp for
  for(i=0; i<n; i++)
      sumLocal+= a[i];
  #pragma omp atomic
      sum+=sumLocal;
}
```

### Critical Construct

- Ensures that multiple threads do not update the same data simultaneously
- Following is also correct, and even faster

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(sumLocal)
{
  sumLocal = 0;
  #pragma omp for
  for(i=0; i<n; i++)
      sumLocal+=a[i];
  #pragma omp critical(update_sum)
  {
      sum+=sumLocal;
  }
}//end of parallel
```

- Critical Constructs help to avoid intermingled outputs when different threads must output sequential values

### Reduction Construct

- Used to automate the processes included in Atomic and Critical Construct optimization
- The following code can be used

```
#pragma omp parallel for default(none) shared(n,a) private(i) reduction(+:sum)
for(i=0; i<n; i++)
  sum+=a[i];
```

- The reduction variable is protected

- This helps to avoid a data race