Lecture Notes: February 29, 2023

What is OpenMP?

Open Multi-Processing (OpenMP - OMP) is an API for parallel processing in C, C++, and Fortran.

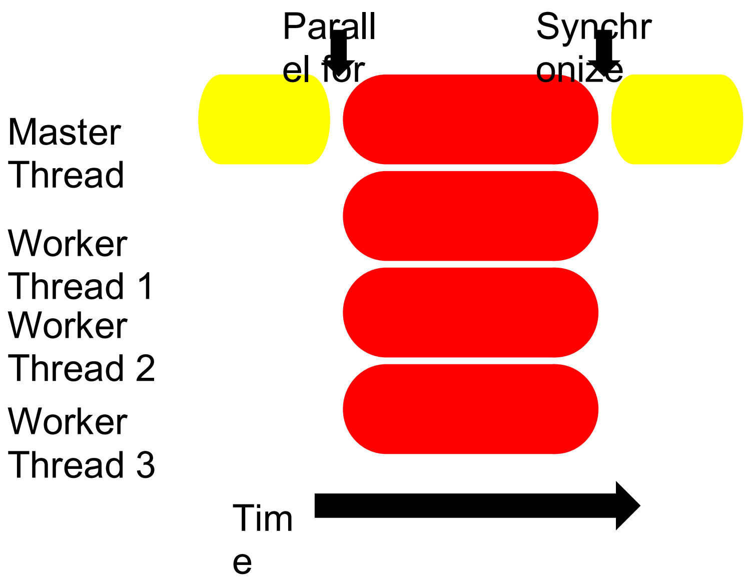
It simplifies parallelization, allowing different directives to indicate which code sections can execute concurrently.

OMP has Fork/Join, a model of parallelism, where the program begins as a single thread and then forks into multiple parallel threads and rejoins afterward at the end.

Overall, OMP helps make parallel computing easy by using the complexities of multi-threading programming.

Parallel for execution:

In the context of OMP, the term "Master thread" applies to the initial thread of execution before any parallelism occurs. The "Master thread" typically starts executing the program and manages the parallel execution. Working threads arise when the Master thread forks into many threads to carry out a program. The work is divided across the threads, including the Master thread, via loop iterations.



The Master thread performs tasks such as initialization, coordination of parallel tasks, etc.

The number of threads:

There are two ways to set the number of threads.

- You can set the number of threads on the command line before you launch the executable, using the following command

```
export OMP_NUM_THREADS=X
```

This will be performed for the entire execution of the program.

- If you would like to be able to set the number of threads differently for different core regions, then you can use the Library route using the following

```
void omp_set_num_threads(int num_threads)
```

If you would like to know how many cores are available on your physical node, you can use the following

```
int omp_get_num_procs(void);
```

This can be used to decide the number of threads to create.

Data Sharing:
When you run an OMP program most variables are shared by default.
As opposed to MPI, where each process gets its own virtual interface, and one process cannot directly access the memory of another process.
In a threaded program such as OPM and other threading models, all threads can access most things, such as the following:

- Most variables are shared by default - must be careful about data erase conditions
- Global variables are shared - defined at the top of the file

Some of the Exceptions to this are as follows:

- Loop index variables are private by default -

```
Int main(int argc, char **argv) {
     int a[100000];
     #pragma omp parallel for
     for (int i = 0; i < 100000; i++) {
         a[i] = 2* i;
     }

     return 0;
}
```

  If we look at the above loop, *i* is a private variable and every thread has its copy of *i*. The reason why each thread has its copy of *i* is that if we had a shared variable across all the threads, the threads would be writing at the same space and modifying *i* in some random order. This is not what we want, what we want to do with OPM is try to assign different numbers of iterations to different threads.
- Stack variables in function calls from parallel regions are also private to each

thread (thread-private).

Saxpy (Single Precision a*x+y) example:
Now we are going to look at another example called the Saxpy example. Saxpy stands for *Single Precision a*x+y*.
Single Precision refers to a 32-bit computation such as floats. If we were dealing with doubles, it would refer to a 64-bit known as the *Double Precision*.
In the loop, we are doing *a*x+y* where *a* is a scaler and *x/y* are vectors/arrays. Hence the name is Saxpy.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
     z[i] = a* x[i] + y[i];
}
```

If we were to add this OMP parallel, would any issues arise? Do we think this would work?
No issues arise and it will work, this is because x and y are read-only arrays and z is the only array being written into. To add on, across loop iterations, we are not reading z and writing it in a different place (doing both). Therefore we do not expect any rise in conditions from this example.

Overriding defaults using clauses:
Clauses are addition things that you can add after the initial pragma OMP parallel for loop. These can help the user specify if they want to make certain threads private or shared. As a result, the clauses help specify how data is shared between threads executing a parallel region. Listed below are some examples of clauses:

```
private(list)
shared(list)
default(shared | none)
reduction(operator: list)
firstprivate(list)
lastprivate(list)
```

A) **Private Clause:**
With private(list), one can put a list of variables into the private clause, and what that entails is each thread will have a private copy of the variables of that list. By

using a private clause, if some variables are shared by default you can tell the OPM at run time that you want to make the variables private. Private variables are uninitialized when a thread starts therefore, you can not assume that there is already a value that was assigned from a master thread to those variables. You must assume that when the threads start executing the parallel region the initial values of those private variables will be uninitialized, if you want to give it a value it must be done inside the loop. The value of a private variable is unavailable to the master thread after the parallel region has been executed.

Let us see an example:

```
val = 5;
#pragma omp parallel for private(val)
for (int i = 0; i < n; i++) {
      ... = val + 1;
}
```

What is the problem with the following code snippet?
The problem is that even though the Master thread is assigning a value of 5 to *val*, once we enter the parallel region, where val is private, what the OPM run time does is create new instances of the variable *val* where it does not know what *val* was initially initialized to.

```
#pragma omp parallel for private(val)
for (int i = 0; i < n; i++) {
      ... = val + 1;
}

printf("%d\n", val);
```

What is the problem with the following code snippet as we add the *printf* at the end?
In this case, what happens is that the value of val is not available to the master thread outside the loop.

B) Default Clause:
With default(shared | none), if you put some variables in the default clause it

suggests that the OPM run time is going to determine the date-sharing attributes for the variables where they would be implicitly determined otherwise.

C) Firstprivate Clause:

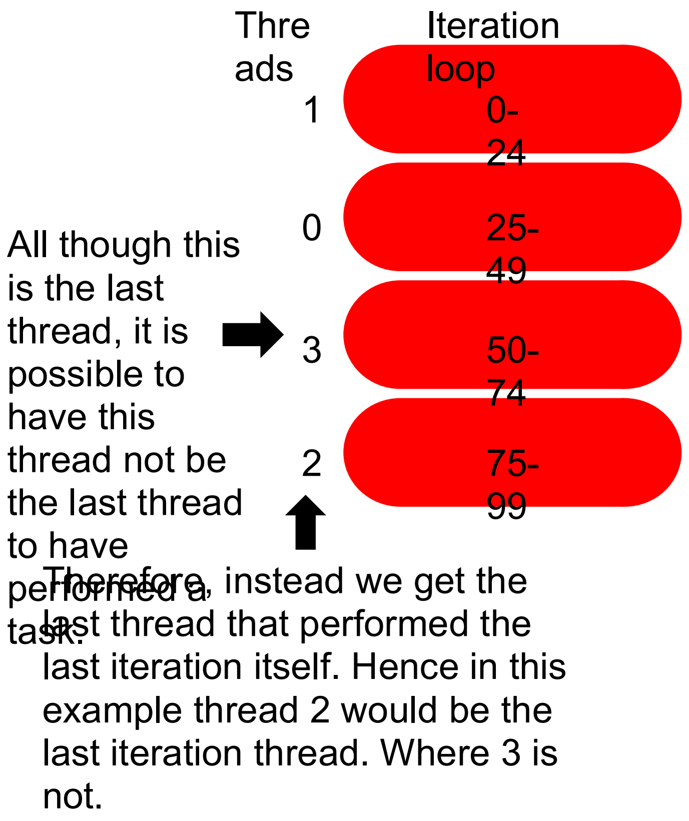
What this clause does is set the OMP run time to initialize each thread's private copy to the value of the Master thread copy.

```
val = 5;
#pragma omp parallel for firstprivate(val)
for (int i = 0; i < n; i++) {
     ... = val + 1;
}
```

Looking at the following example, if you want *val* to be 5 inside the loop then instead of using private you would use firstprivate on *val*. Ensure that the Master thread variable is shared inside the loop.

D) Lastprivate Clause:

This clause writes the value belonging to the thread that executed the last iteration of the loop to the Master thread copy. This iteration is determined by sequential order. For example, consider the following image:


Threads
Iteration loop
1
0-24
0
25-49
3
50-74
2
75-99
All though this is the last thread, it is possible to have this thread not be the last thread to have performed a task.
Therefore, instead we get the last thread that performed the last iteration itself. Hence in this example thread 2 would be the last iteration thread. Where 3 is not.

E) Reduction Clause:

```
#pragma omp parallel for reduction(+: val)
for (int i = 0; i < n; i++) {
     val += 1;
}

printf("%d\n", val);
```

Based on the following example, if we ignore the reduction(+: val), we notice this code snippet has a couple of issues. One of the issues is that the same variable is being read from and written on. However, we can see the code wants to grab the sum of all *n* elements. This is where reduction comes into play, it can reduce values across private copies of a variable.

Loop scheduling:

How are threads assigned to different things in a loop? OMP uses Loop scheduling. Loop scheduling is the assignment of loop iterations to different worker threads. There is a default schedule that decides how to divide the threads. It usually balances the threads equally. However, there are user-specified schedules, where the user can set/divide the threads as they wish. `schedule(type[, chunk])`. There are four types of user-scheduling, the type types are as followers: Static, Dynamic, Guided, and Runtime