A SHORT INTRODUCTION TO OPENMP

AND A FEW MORE STUFF AS WELL :-)

PLAN FOR THE DAY

- The plan is to mix the OpenMP lecture with short exercises
- During the exercises we will cover some base ground as well
- Please follow the exercises as we present them
  - Don't peak into the solutions beforehand (try them out first)

WHY DO WE FOCUS SO MUCH ON MPI AND OPENMP

- They have both gained significant ground
  - Think of `FORTRAN` as an example
- Several applications have been developed for decades (before we were here?)
  - and *will be continued*...
- Other frameworks (i.e. PGAS languages) gain trend slowly but steadily
- Accelerators are also trending in HPC environments

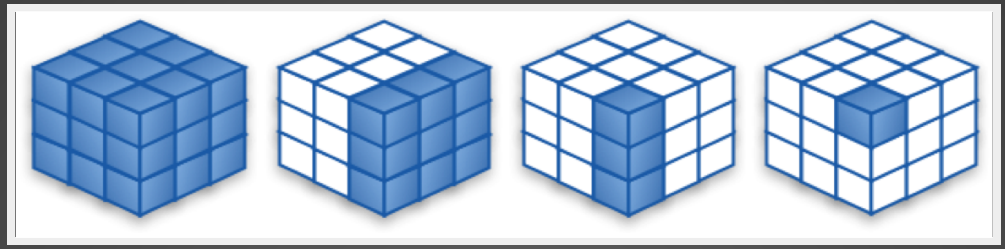
THE QUESTION *WHY DO WE NEED PARALLEL SOFTWARE PROGRAMMING?* HAS BEEN ANSWERED

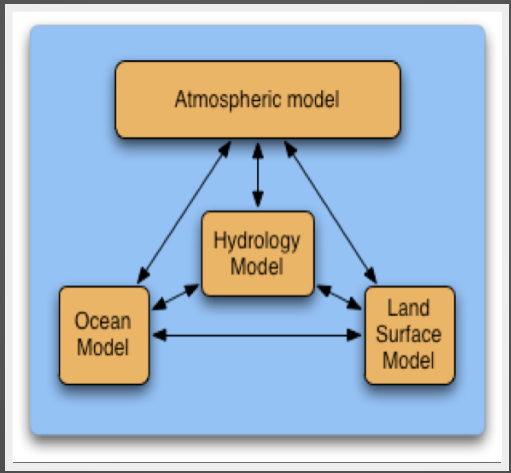
- However it is useful to keep the following in mind
  - before applying parallel technics on a given algorithm make sure you have done your best to *optimize* the serial version
  - think deeply where to divide/split the workload among processes and processors
  - be comfortable to use other people's code in the process

TWO APPROACHES HAVE BEEN AROUND FOR QUITE SOME TIME:

- Domain decomposition
- Functional decomposition

DOMAIN AND FUNCTIONAL DECOMPOSITION APPROACHES





OVERVIEW OF OPENMP

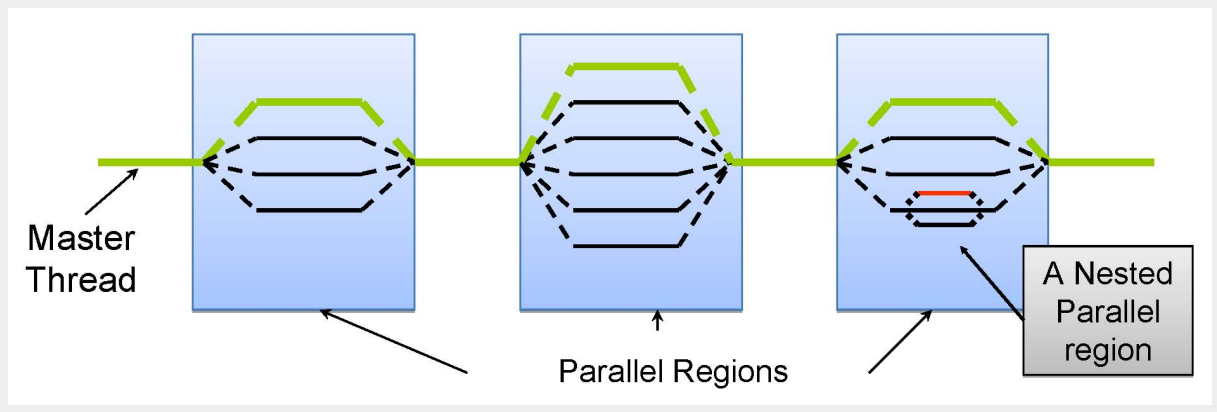
- Shared vs Distributed memory models
- Why OpenMP
- How OpenMP works
- Basic examples
- How to execute the executable

SOME BASICS

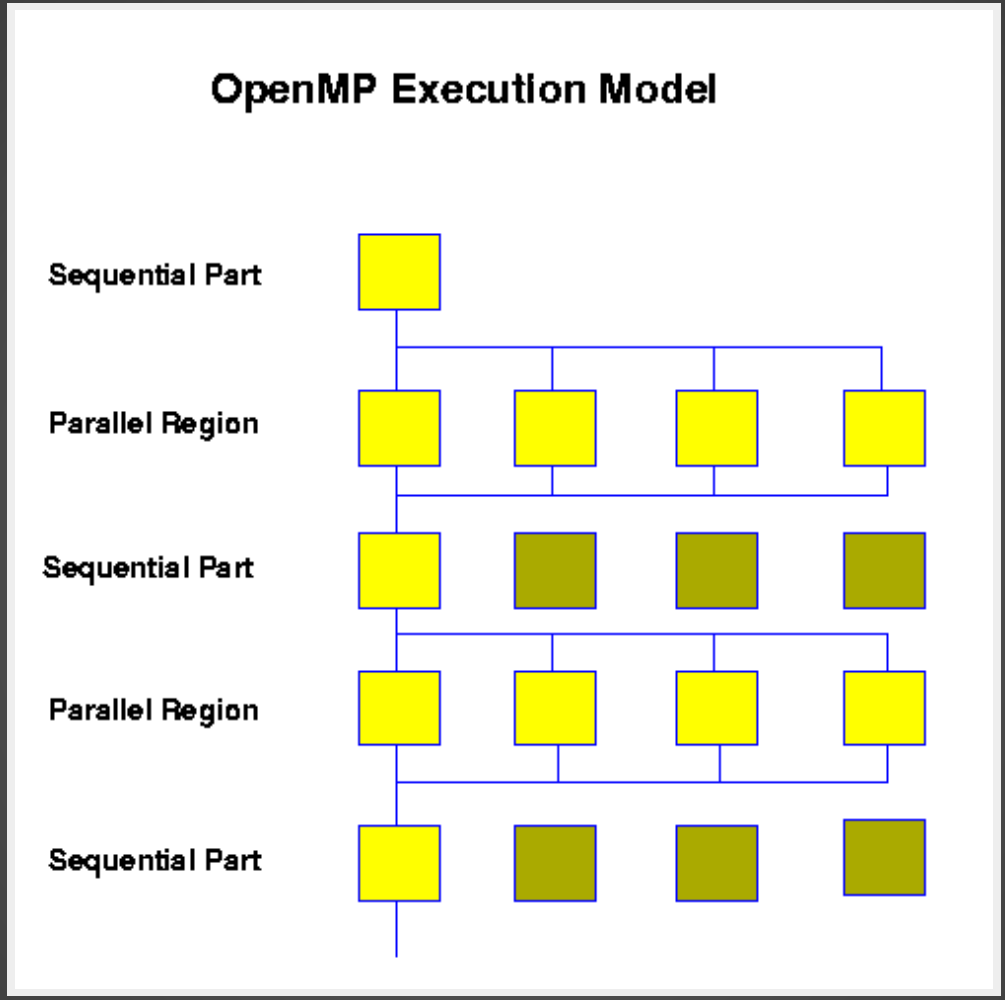
- API extension to C/C++ and Fortran languages
  - Most compilers support OpenMP
    - GNU, IBM, Intel, PGI, EkoPATH, Open64
- Extensively used for writing programs for shared memory architectures over the past decade
- Thread (process) communication is implicit and uses variables pointing to shared memory locations; this is in contrast with MPI which uses explicit messages passed among each process

THREADED PROGRAMMING

1. Fork - join model
2. Master and slave threads



EXECUTION MODEL



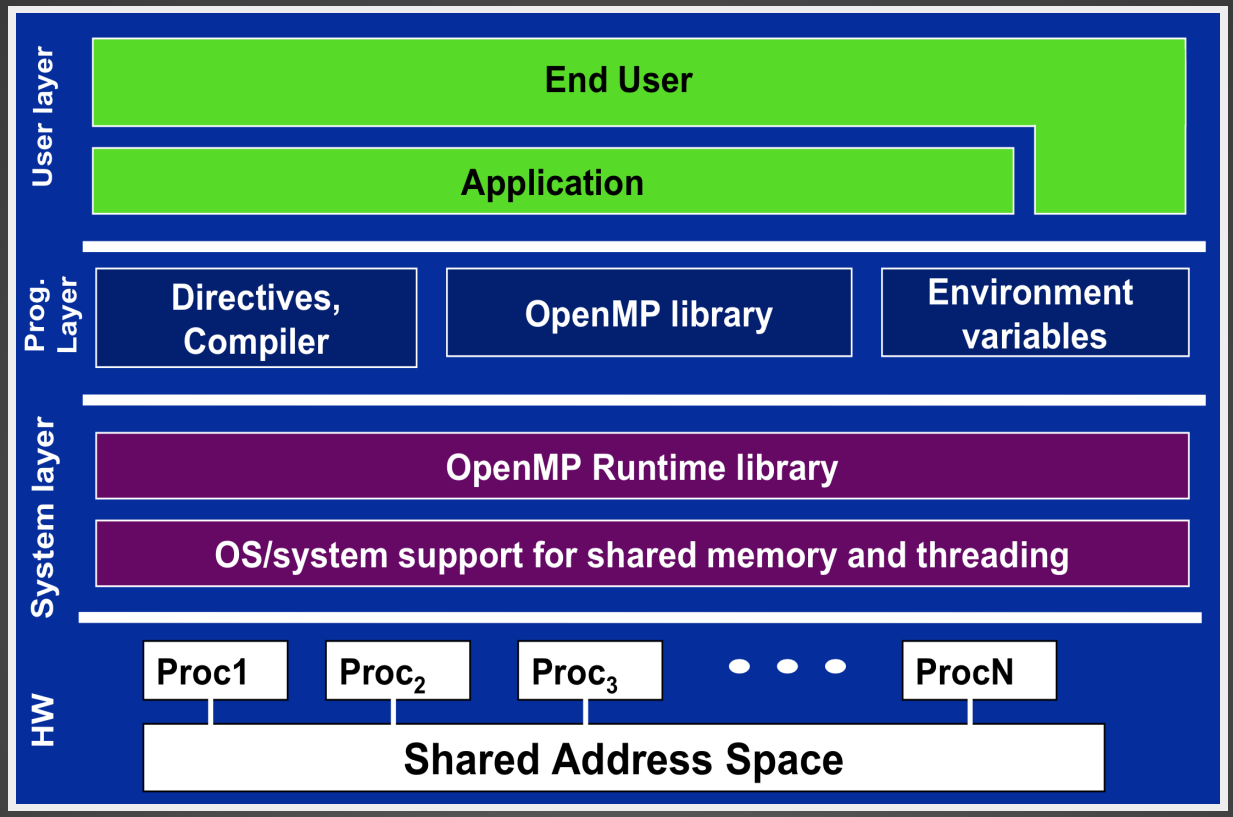
MEMORY ISSUES

- Threads have access to the same address space
- Communication is implicit
- Developer needs to define
  - private data (per thread)
  - shared data (among threads)

CODING BASICS

- Include the library
  - `#include <omp.h>`
- Use the appropriate compiler flag
  - `$ gcc -fopenmp`
  - `$ icc -openmp`

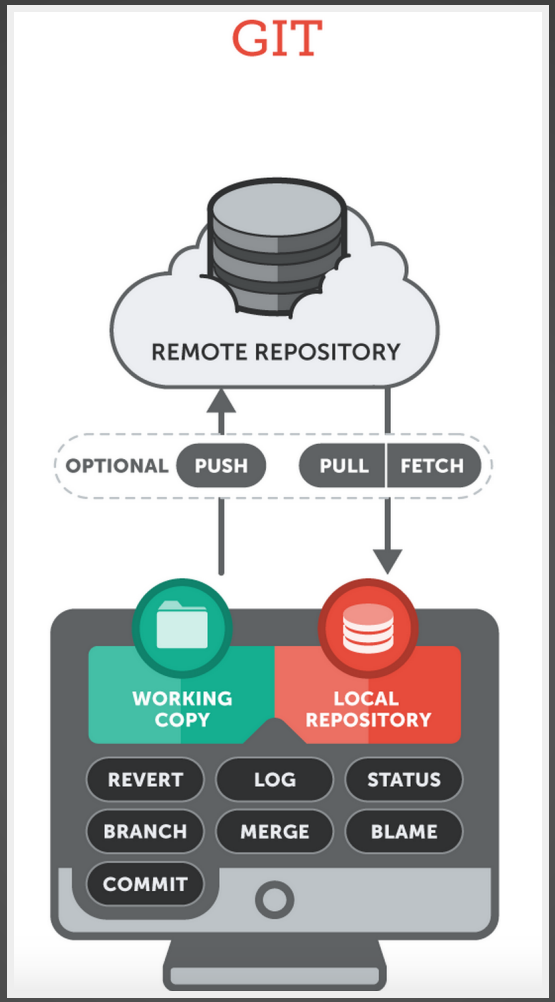
OPENMP SOFTWARE STACK



COMMON API CALLS

Call	Description
`omp_get_num_threads()`	Returns the number of threads in the concurrent team
`omp_get_thread_num()`	Returns the id of the thread inside the team
`omp_get_num_procs()`	Returns the number of processors in the machine
`omp_get_max_threads()`	Returns maximum number of threads that will be used in the next parallel region
`omp_get_wtime()`	Returns the number of seconds since a time in the past
`omp_in_parallel()`	1 if in parallel region, 0 otherwise

TIME FOR SOME HANDS-ON TO BETTER DISCUSS STUFF


GIT
REMOTE REPOSITORY
OPTIONAL
PUSH
PULL
FETCH
WORKING COPY
LOCAL REPOSITORY
REVERT
LOG
STATUS
BRANCH
MERGE
BLAME
COMMIT

EXERCISE

- Add thread number and thread id in print out message
- Make it thread safe
- If you are done,
  - count the time it takes to complete
  - add a couple more lines to print out
    - Number of processors
    - Max number of threads
  - and create a submission script

- Make use of `omp_get_num_threads()` and `omp_get_thread_num()` calls and adjust `printf`
- Will it now compile without the `-fopenmp` flag?
- Have you used and initialized and variables?
  - If yes, are these common (i.e. shared) among threads?
    - If yes, how much meaning does a `thread_id` hold?
      - Make such a variable private (next subject)

- To count the time in seconds use the `omp_get_wtime` call
- To print out number of processors and number of max threads in omp section use `omp_get_num_procs` and `omp_get_max_threads` respectively
- For a sample submission script see slide below

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -c 8

export OMP_NUM_THREADS=8
./hello_omp
```

Use the guidelines from here to submit it. If you are not sure just ask!

DATA SCOPING

- For each parallel region the data environment is constructed through a number of clauses
  - `shared` (variable is common among threads)
  - `private` (variable inside the construct is a new variable)
  - `firstprivate` (variable is new but initialized to its original value via the master thread)
  - `default` (used to set overall defaults for construct)
  - `lastprivate` (variable's last value is copied outside the openmp region)
  - `reduction` (variable's value is reduced at the end)

A FEW EXAMPLES

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
  x++;
  printf("x=%d\n",x);
}
printf("x=%d\n",x);
```

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
  x++;
  printf("x=%d\n",x);
}
printf("x=%d\n",x);
```

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
  x++;
  printf("x=%d\n",x);
}
printf("x=%d\n",x);
```

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

int main()
{
   int x=1;
   #pragma omp parallel shared(x) num_threads(2)
   {
     x++;
     printf("x=%d\n",x);
   }
   printf("x=%d\n",x);
}
```

SYNCHRONIZATION

- OpenMP provides several synchronization mechanisms
  - `barrier` (synchronizes all threads inside the team)
  - `master` (only the master thread will execute the block)
  - `critical` (only one thread at a time will execute)
  - `atomic` (same as critical but for one memory location)

A FEW EXAMPLES

```
int x=1;
#pragma omp parallel num_threads(2)
{
  x++;
  #pragma omp barrier
  printf("x=%d\n",x);
}
```

```
int x=1;
#pragma omp parallel num_threads(2)
{
  #pragma omp master
  {
    x++;
  }
  printf("x=%d\n",x);
}
```

```
int x=1;
#pragma omp parallel num_threads(2)
{
  #pragma omp critical
  {
    x++;
    printf("x=%d\n",x);
  }
}
```

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

int main()
{
  int x=1;
  #pragma omp parallel num_threads(2)
  {
    x++;
    #pragma omp barrier
    printf("x=%d\n",x);
  }
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"barrier.c" 13L, 166C
```

$://showterm slow fast stop

DATA PARALLELISM

- Worksharing constructs
  - Threads cooperate in doing some work
  - Thread identifiers are not used explicitly
  - Most common use case is loop worksharing
  - Worksharing constructs should not be nested
- DO/for directives are used in order to determine a parallel loop region

THE FOR LOOP DIRECTIVE

```
#pragma omp for  [clauses]
for (iexpr ; test ; incr)
```

- Where clauses may be
  - private, firstprivate, lastprivate
  - Reduction
  - Schedule
  - Nowait
- Loop iterations must be independent
- Can be merged with parallel constructs
- Default data sharing attribute is shared (attention!)

A SIMPLE EXAMPLE

```
#pragma omp parallel
#pragma omp for private(j)
for(i=0; i<N; i++){
  for(j=0; j<N; j++ ){
    m[i][j] = f(i,j);
  }
}
```

- `j` must be declared explicitly as private
- `i` is privatized by default

THE SCHEDULE CLAUSE

- Schedule clause may be used to determine the distribution of computational work among threads
  - static, chunk; The loop is equally divided among pieces of size chunk which are evenly distributed among threads in a round robin fashion
  - dynamic, chunk; The loop is equally divided among pieces of size chunk which are distributed for execution dynamically to threads. If no chunk is specified chunk=1
  - guided; similar to dynamic with the variation that chunk size is reduced as threads grab iterations
- Configurable globally via OMP_SCHEDULE
  - i.e. setenv OMP_SCHEDULE "dynamic,4"

EXERCISE

- On the adding two vectors example from this morning
  - add some timing using `omp_get_wtime` call and
  - check if scheduling can affect the timing result
    - (Hint: you will probably need two large vectors to properly distribute the load)

REDUCTION CLAUSE

- Useful in the case one variable's value is accumulated within a loop
- Using the reduction clause
  - A private copy per thread is created and initialized
  - At the end of the region the compiler safely updates the shared variable
  - Operators may be +, *, –, /, &, ^, |, &&, | |

REDUCTION CLAUSE EXAMPLE

Add vector elements:

```
int i, N=atoi(argv[1]);
vector  A(N);
double sum;
// Initialize vector values (now that we know
// we will use omp for the init part as well)
#pragma omp parallel for shared(A,N) private(i)
for(i=0; i<N; i++)
  A[i] = some_arbitrary_function(i);
// Calculate the sum now
#pragma omp parallel for shared(A,N) private(i) reduction(+:s)
for(i=0; i<N; i++)
  sum += A[i];
printf("sum = %d\n", sum);
return 0;
```

EXERCISE

Based on the previous slide can you create a parallel factorial function?

MORE EXERCISES (CHOOSE ONE DEPENDING ON HOW YOU FEEL LIKE)

- Parallel pi calculation (simple)
  - Try to perform some benchmarking (i.e. measure times it takes to calculate Pi from one to using more processors and threads). What do you observe?
- Matrix-matrix multiplication (harder)
  - Try to perform some benchmarking (i.e. measure times it takes to multiply two matrices). Going from the serial to the parallel versions (one with MPI and one with OpenMP) what do you observe? Can you think of a way to mix MPI and OpenMP?

Please discuss and share your observations with us!

THANK YOU!