Lecture 05
Programming Parallel Systems - OpenMP

Dr. Ehtesham Zahoor

Moore’s Law

- **Gordon Moore** – co-founder of Intel
- In 1965 he forecasted that the number of components on an integrated circuit would double every year.
- When it proved correct in 1975, he revised what has become known as Moore’s Law to a doubling of transistors on a chip every two years.

Lecture 05:Programming Parallel Systems - OpenMP

In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox.

We shouldn't be trying for bigger computers, but for more systems of computers.

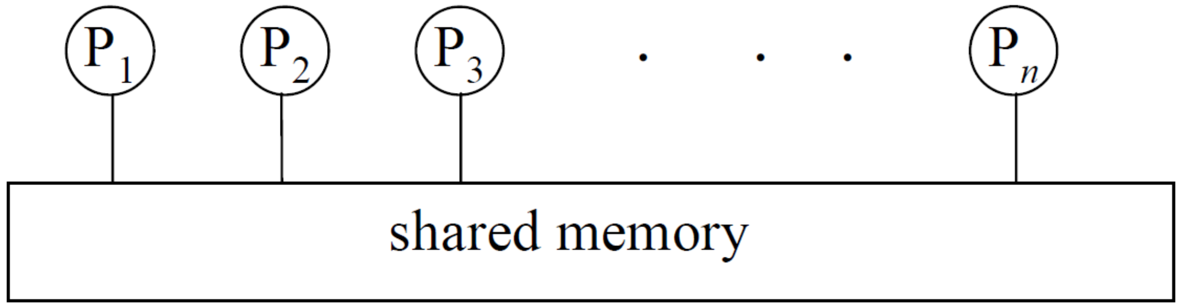
--Grace Hopper

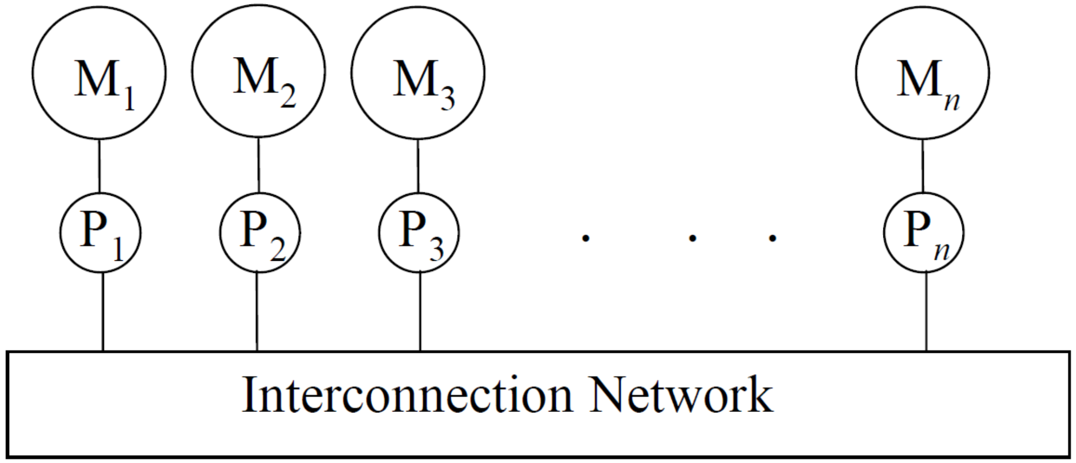
Multi-Processor Systems

- Multiprocessor is any computer with several processors
- Multi-core processor is a special kind of a multiprocessor as all cores share the same memory

Parallel and Distributed Computing

- The difference includes whether the processes communicate using shared or a distributed memory.



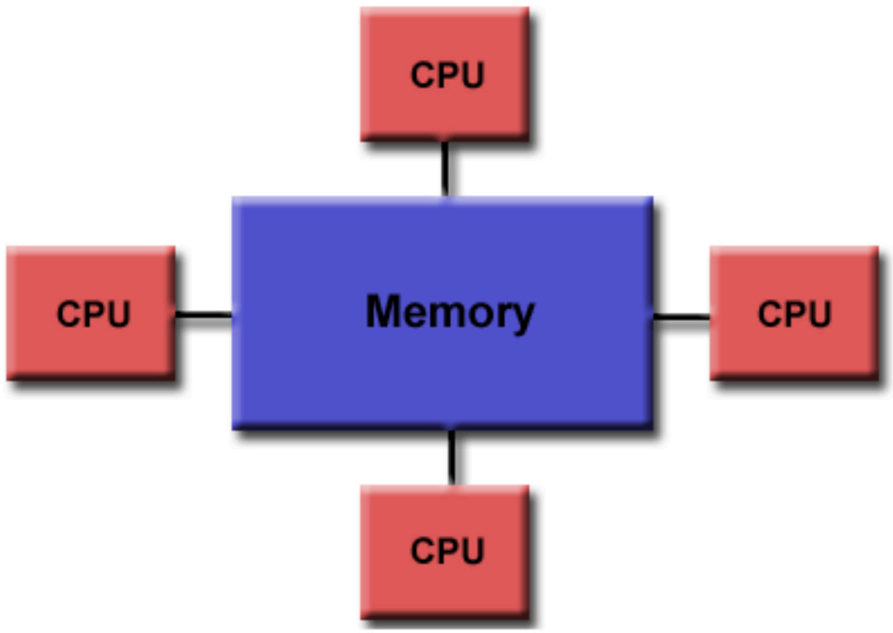


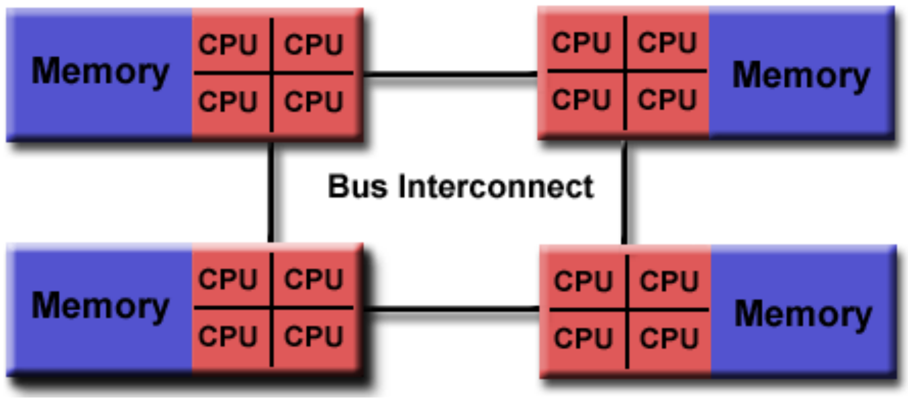
OpenMP

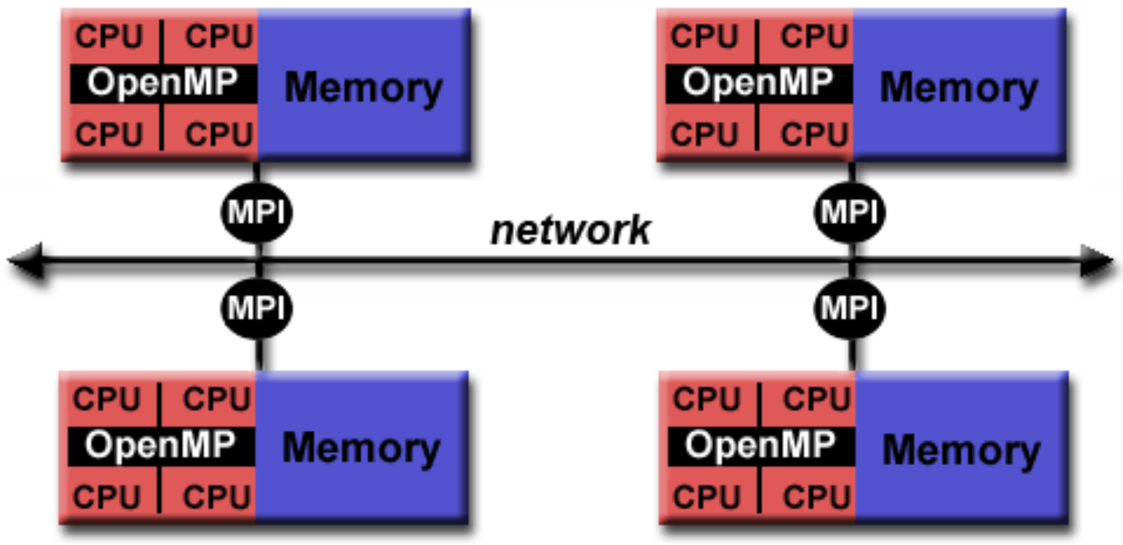
- Programming of shared memory systems
- An API for Fortran and C/C++
  - Directives
  - Runtime routines
  - Environment variables

This Lecture is based on https://computing.llnl.gov/tutorials/openMP/

Lecture 05:Programming Parallel Systems - OpenMP







Goals

- Standardization
  - Provide a standard among a variety of shared memory architectures(platforms)
  - High-level interfaces to thread programming

Not Sold as Yet?

- We know how to program parallel systems and how to create process and threads!
- Lets go back to hello-world, pthread version

Lecture 05:Programming Parallel Systems - OpenMP

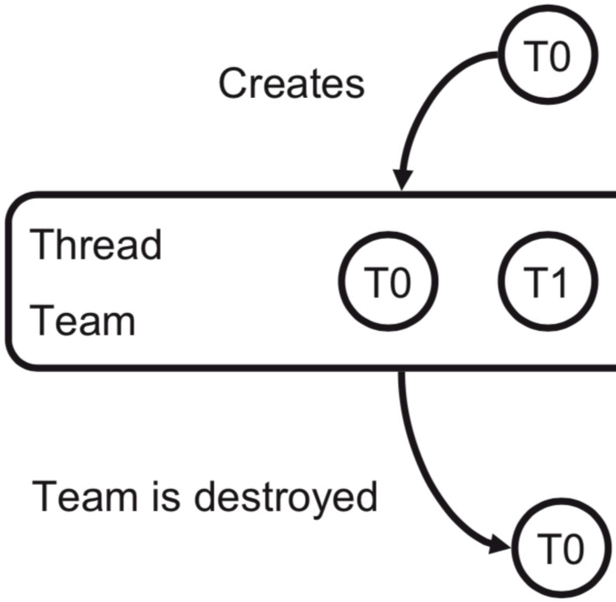
```
#include <pthread.h>
#include <stdio.h>

void* thrfunc(void* arg) {
printf("hello from thread %d\n", *(int*)arg);
}

int main(void) {
pthread_t thread[4];
pthread_attr_t attr;
int arg[4] = {0,1,2,3};
int i;
// setup joinable threads
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
// create N threads
for(i=0; i<4; i++)
pthread_create(&thread[i], &attr, thrfunc, (void*)&arg[i]);


// wait for the N threads to finish
for(i=0; i<4; i++)
pthread_join(thread[i], NULL);
}
```

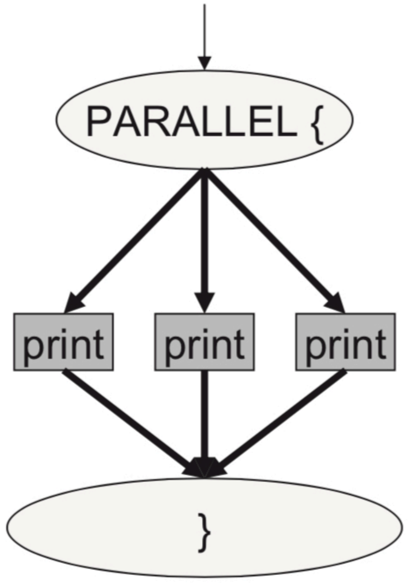
… and the OpenMP version

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

int main(int argc, char* argv[]) {
#pragma omp parallel  {
printf("Hello World... from thread = %d\n",
omp_get_thread_num());
}
}


Compilation: gcc -fopenmp hello_omp.c -o hello_omp
Execution: export OMP_NUM_THREADS=5  ./hello_omp
```

Execution Model

```
#pragma omp parallel
{
 printf("Hello world %d\n", omp_get_thread_num());
}
```



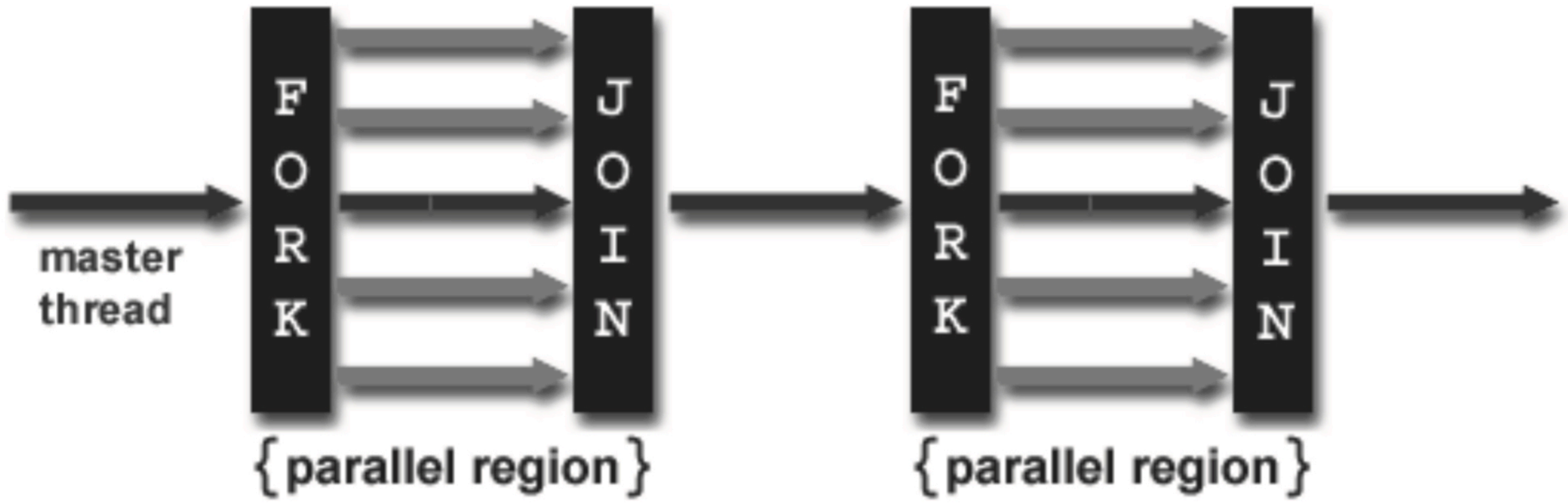


Fork*/Join Execution Model

- An OpenMP-program starts as a single thread (*master thread*).
- Additional threads (*Team*) are created when the master hits a parallel region.
- When all threads finished the parallel region, the new threads are given back to the runtime or operating system.

**Not to be confused with fork() system call*

Fork*/Join Execution Model



**Not to be confused with fork() system call*

OpenMP

- Programming of shared memory systems
- An API for Fortran and C/C++
  - Directives
  - Runtime routines
  - Environment variables

OpenMP

- Programming of shared memory systems
- An API for Fortran and C/C++
  - **Directives**
  - Runtime routines
  - Environment variables

OpenMP - Directives

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - …

sentinel directive-name [clause, …]

#pragma omp parallel

OpenMP - Directives

- **Format:**

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

- **Example:**

```
#pragma omp parallel default(shared) private(beta,pi)
```

The *parallel* directive

- A parallel region is a block of code that will be executed by multiple threads.
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel region. Only the master thread continues execution past this point.

The *parallel* directive

- Some common clauses include:
  - if *(expression)*
  - private *(list)*
  - shared *(list)*
  - num_threads *(integer-expression)*

How Many Threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the **IF** clause
  - Setting of the **NUM_THREADS** clause
  - Use of the **omp_set_num_threads()** library function
  - Setting of the **OMP_NUM_THREADS** environment variable
  - Implementation default - usually the number of CPUs on a node.

- Threads are numbered from 0 (master thread) to N-1

Shared and Private Data

- Shared data are accessible by all threads.
  - A reference a[5] to a shared array accesses the same address in all threads.
- Private data are accessible only by a single thread.
  - Each thread has its own copy.
- The default is shared.

Shared and Private Data

```
int main(int argc, char* argv[])
{
 int threadData = 10;
// Beginning of parallel region
#pragma omp parallel private(threadData)
{
    threadData =200; }
// Ending of parallel region
printf("Value: %d\n", threadData);
}
```

So far so good, but …

- If all the threads are doing the same thing, what is the advantage then?

- Within each “Team” threads are assigned IDs, with master thread assigned ID 0.
- Can we use this to distribute tasks amongst the “team” members?

```
int main(int argc, char* argv[])
{
 int iam, nthreads;
 #pragma omp parallel private(iam,nthreads)
 {
  iam = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf(“ThradID %d, out of %d threads\n”, iam, nthreads);
  if (iam == 0)
   printf(“Here is the Master Thread.\n”);
  else
   printf(“Here is another thread.\n”);
 }
}
```

Work-Sharing Constructs

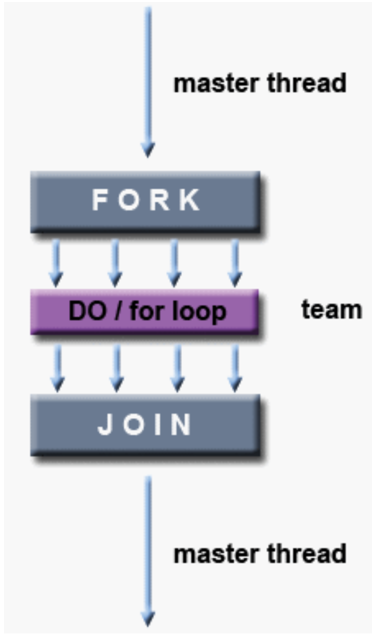
- Work-sharing constructs distribute the specified work to all threads within the current team.

- Work-sharing constructs do not launch new threads

Do/For Work-Sharing Construct

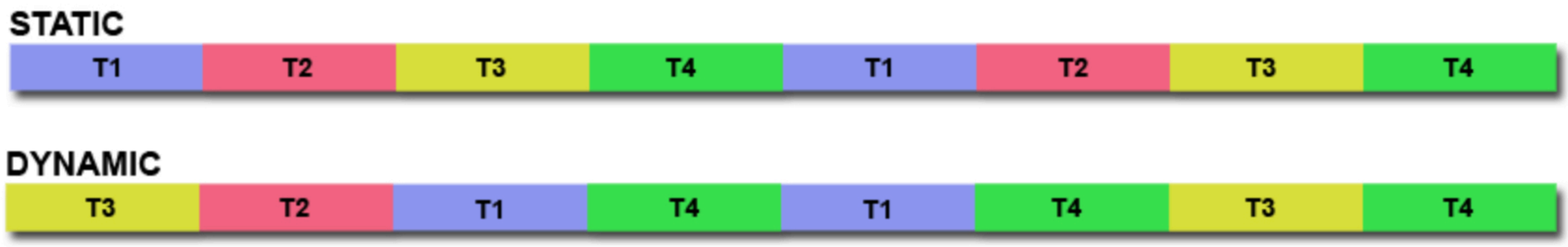
- **DO / for** - shares iterations of a loop across the team.

#pragma omp for [clause ...] newline



Do/For Work-Sharing Construct

- **SCHEDULE** clause describes how iterations of the loop are divided among the threads in the team.



Do/For Work-Sharing Construct

```
int main(int argc, char* argv[])
{
int i, a[10];
#pragma omp parallel num_threads(2)
{
#pragma omp for schedule(static, 2)
   for ( i=0; i<10;i++)
      a[i] = omp_get_thread_num();
}
for ( i=0; i<10;i++)
printf("%d",a[i]);
}
```

Do/For Work-Sharing Construct

```
int main(int argc, char* argv[])
{
int sum, counter, inputList[6] = {11,45,3,5,12,-3};
#pragma omp parallel num_threads(2)
{
#pragma omp for schedule(static, 3)
  for (counter=0; counter<6; counter++) {
printf("%d adding %d to the sum\n",omp_get_thread_num(),
inputList[counter]);
sum+=inputList[counter];
}
}
printf("The summed up Value: %d", sum);
}
```

OpenMP - Synchronization Constructs

- The MASTER directive specifies a region that is to be executed only by the master thread of the team.
- All other threads on the team skip this section of code

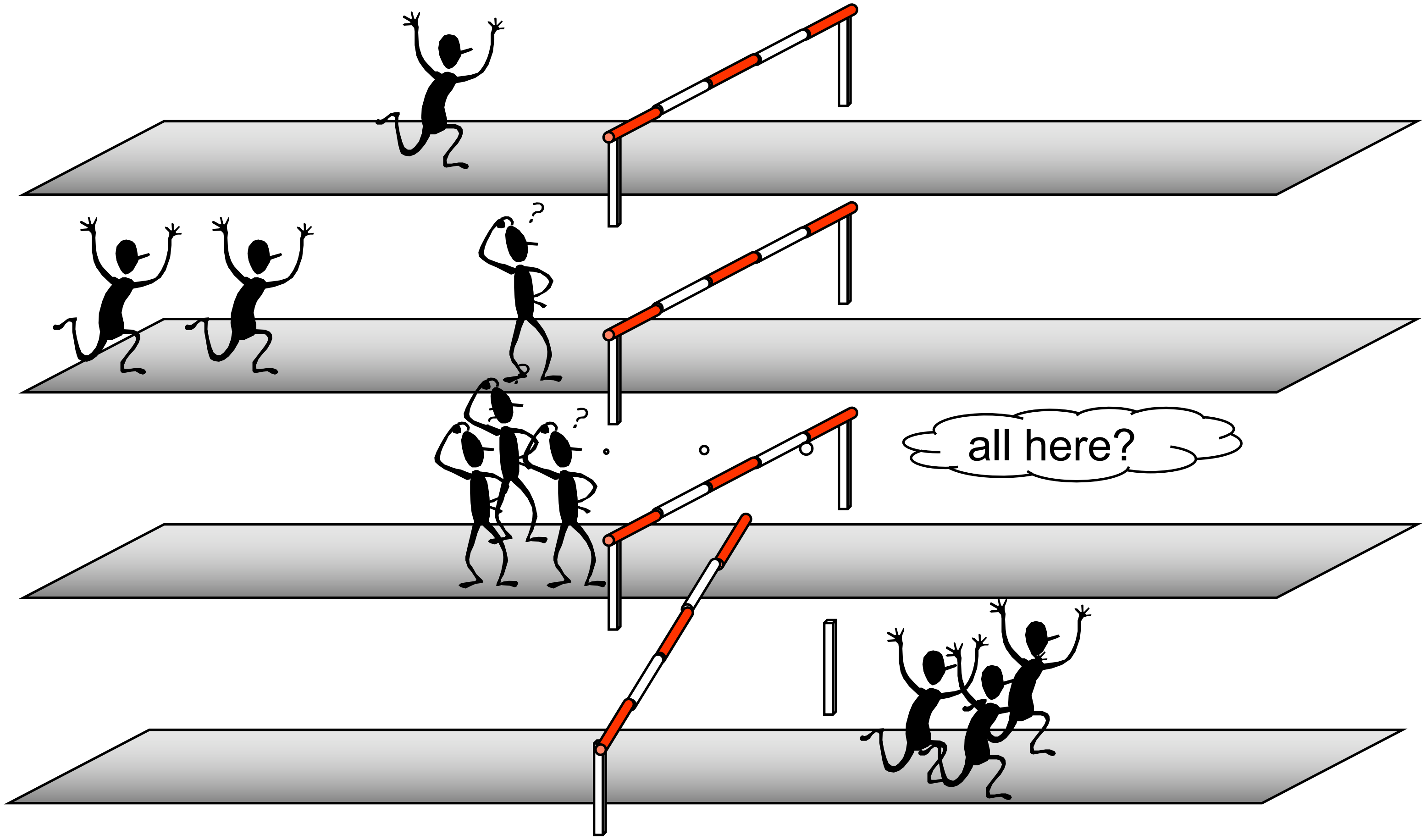
```
#pragma omp master newline
…
```

OpenMP - Synchronization Constructs

- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier.
- All threads then resume executing in parallel the code that follows the barrier.

```
#pragma omp barrier newline
 …
```

Barrier Synchronization



Reduction (Data-sharing Attribute Clause)

- The REDUCTION clause performs a reduction operation on the variables that appear in its list.
- A private copy for each list variable is created and initialized for each thread.
- At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

#pragma omp operator: list newline

```
...
```

operator can be `+,-,*,&&,||,max,min` *…*

Reduction (Data-sharing Attribute Clause)

```
int main(int argc, char* argv[])
{
srand(time(NULL));
int winner;
#pragma omp parallel reduction(max:winner) num_threads(10)
{
winner = (rand() % 1000) + omp_get_thread_num();
printf("Thread: %d has Chosen: %d\n",
omp_get_thread_num(),winner);
  }
printf("Winner: %d\n", winner);
}
```

OpenMP

- Programming of shared memory systems
- An API for Fortran and C/C++
  - Directives
  - **Runtime routines**
  - Environment variables

OpenMP – Runtime routines

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting, initializing and terminating locks
  - …

OpenMP – Basic Runtime routines

- *omp_set_num_threads* - Sets the number of threads in upcoming parallel regions, unless overridden by a num_threads clause.
- *omp_get_num_threads -* Returns the number of threads in the parallel region.
- *omp_get_thread_num* - Returns the thread number of the thread executing within its thread team.
- *omp_in_parallel -* Returns nonzero if called from within a parallel region.

OpenMP

- Programming of shared memory systems
- An API for Fortran and C/C++
  - Directives
  - Runtime routines
  - **Environment variables**

OpenMP – Environment Variables

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - …

Assignment-02

The consensus problem in distributed systems is the problem of getting a set of nodes to agree on something – it might be a value, a course of action or a decision. This problem has been at the core of distributed systems research for over last few decades. A Simple solution, which can work under some constraints, is called two-phase commit, or 2PC.

Think about how you would solve a real-world consensus problem – perhaps trying to arrange a graduate party! You'd call all your friends and suggest a date and a time. If that date/time is good for everybody you have to ring again and confirm, but if someone can't make it you need to call everybody again and tell all your friends that the party is off. More specifically, we can identify two steps in the process:
First, the first proposal phase involves proposing a value to every participant in the system and gathering responses. In the next phase, if everyone agrees, there is need to contact every participant again to let him or her know. Otherwise, contact every participant to abort the consensus.

For this question, you need to ignore the part to contact every participant again, thread share variables and this can be implicit.