<u>Parallel Processing</u>

- Traditional software design was centered around **serial** computation which has the following characteristics:

    - A problem is broken into a discrete sequential series of instructions
    - Instructions are executed sequentially one after another
    - Instructions are executed on a single processor
    - Only one instruction may execute at any moment in time

- **Parallel computing** on the other hand is the simultaneous use of multiple compute resources to solve a **computational problem** as follows:

    - A problem is broken into discrete components that can be solved concurrently
    - Each component is further broken down into a series of instructions
    - Instructions from each component are then executed simultaneously within threads assigned to separate cores/processors
    - An implicit synchronization mechanism is usually used: causes the thread to block until all other threads have arrived at the same point and ready to step to the next processing node

- In order to achieve the advantages of parallel computing, the computational problem must have the following characteristics:

    - Be able to be partitioned into discrete pieces of work that can be solved simultaneously
    - Be able to execute multiple program instructions at any moment in time
    - Be able to be solved in less time with multiple computing resources than with a single resource.

- The computational resources are typically:

    - A single computer with multiple processors/cores
    - An arbitrary number of networked computers operating as a Distributed System

- All modern computer architectures use multi-core CPUs and in addition, architectures also support multiple CPUs. In other words all modern architectures are parallel computers:

    - Multi-core servers, workstations, smartphones, tablets, laptops
    - Multi-node clusters, supercomputers

- In addition to the multiple cores/CPUs, the overall architecture is made up of:

    - Cache: L1 cache, L2 cache
    - Functional Units: branch, prefetch, decode, floating-point, integer
    - Graphics processing (GPU)

- For example a dual CPU motherboard with 12 cores per CPU provides and overall processing power of 24 cores (Intel's Broadwell Xeon server CPUs offer up to 22 cores per socket).

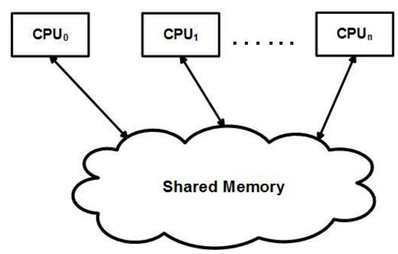
- However, applications cannot automatically take advantage of multiple cores unless they are designed specifically to utilize all available cores in a machine.

- Parallel computing allows applications to utilize all available resources to solve a single computing problem.

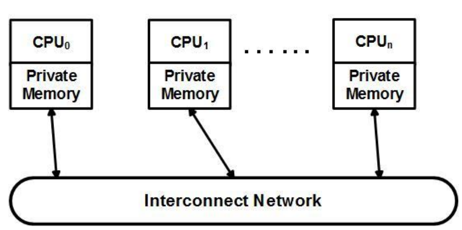
Memory Models

- There are two memory models: **Shared** and **Distributed** Memory models

- **Shared Memory Model**
    - All CPUs have access to the (shared) memory (e.g. Workstations, Servers, smartphones, etc.).
    - The following diagram illustrates this model:



- **Distributed Memory Model**
    - Each CPU has its own private memory which is protected (not accessible to other CPUs).
    - The inter-CPU communications is facilitated by a high-speed interconnect network such as InfiniBand.
    - The following diagram illustrates this model:



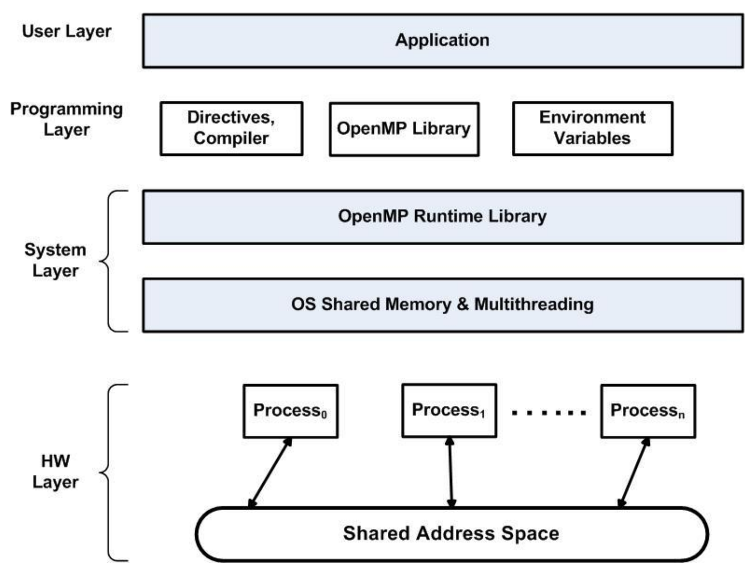
Open Multi-Processing (OpenMP)

- **OpenMP** is an API that implements a multi-threaded, shared memory form of parallelism. Using the API we can write **shared memory** parallel applications in C, C++, and Fortran.

- The OpenMP API is basically comprises a set of:

    - Compiler Directives
    - Runtime library routines/functions
    - Environment variables

- The following diagram illustrates the overall application solution stack:



- The programming constructs are primarily a set of complier directives with the following syntax:

    #pragma omp construct [clause [clause]..]

- Clauses are used to specify the precise behavior of the **parallel region**.

- For example, we can set the total number of threads as follows:

    #pragma omp parallel num_threads(4)

- Note the following:

    - You must include the OpenMP include file in yoru code: **#include <omp.h>**
    - You must link in the OpenMP library: **"-fopenmp**"

- The ***parallel*** construct is one of the most important constructs in OpenMP. The parallel region is specified in C/C++ as follows:

```
//sequential code here (master thread)

#pragma omp parallel [clauses]
{
        //   Execute parallel instructions here
}
// end of parallel block
//   sequential code here (master thread)
```

- OpenMP follows the ***fork/join*** model:

  - OpenMP programs start with a single (master) thread: **Thread$_0$**
  - At start of parallel region the master thread creates a set of “worker” threads (***fork***)
  - Statements in parallel block are executed in parallel by every thread
  - At end of parallel region, all threads synchronize, and ***join*** master thread (***join***)

- The number of openMP threads can be set using the environmental variable “**OMP_NUM_THREADS”.**

- Alternatively we can use the runtime function: **omp_set_num_threads(n)** to set the number of threads required by the application.

- The following are some other useful functions that can be used to get information about threads:

- **omp_get_num_threads()**

  - Returns number of threads in parallel region.
  - By default this is the number of cores.

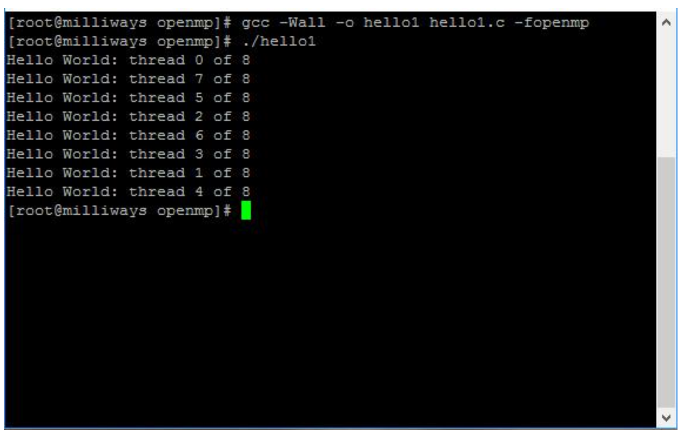
- **omp_get_thread_num()**

  - Returns id of thread between 0 and n-1. Where n = number of threads.
  - The master thread always has an id = 0.

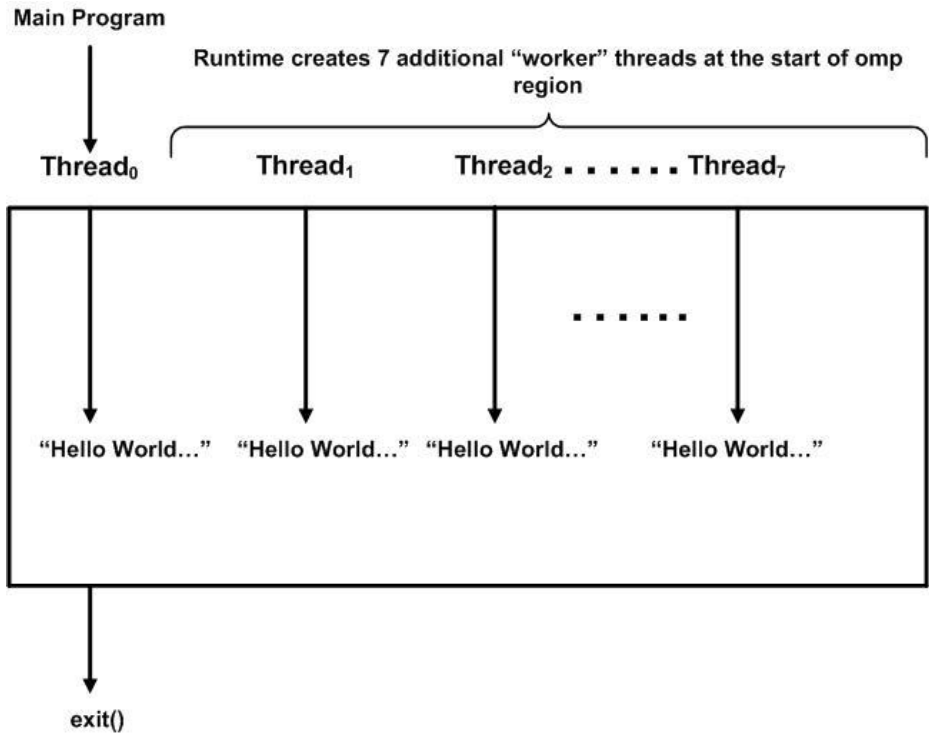
- Consider the following “hello world” program that is “parallelized”:

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

int main(int argc, char *argv[])
{
  int thread_num = 0, num_core = 1;

  // parallel region with default number of threads
  #pragma omp parallel private (thread_num, num_core)
  {
      num_core = omp_get_num_threads();
      thread_num = omp_get_thread_num();
      printf("Hello World: thread %d of %d\n", thread_num, num_core);
  } // end of parallel region
  exit(0);
}
```

- The above will produce the following output (on an 8-core machine):



- The following diagram illustrates the creation of all the threads:



- In the omp region, every thread executes all the instructions in the omp block.

***private* Clause**

- The values of ***private*** data are undefined upon entry to and exit from the specific construct.

- In the above example, variables ***thread_num*** and ***num_core*** are private to each thread. Each thread will maintain its own "private" values for these variables.

- Private variables become undefined after the parallel block.

- In order ensure that the last value is accessible after the construct, we can use the "***lastprivate***" data sharing attribute clause.

- To pre-initialize private variables with values available prior to the parallel region, we can use the "***firstprivate***" data sharing attribute clause.

- Note that any loop iteration variables are private by default

shared **Clause**

- Unless specified as private, all other variables are ***shared*** by default. This means that they are shared among the team of threads executing the parallel region.

- Each thread can read or modify shared variables, which can lead to data corruption is possible when multiple threads attempt to update the same memory location.

- This can also result in data race conditions, where multiple threads attempt to simultaneously read/modify shared variables.

- It is the developer's responsibility to ensure that the above problems for not occur.

Work Sharing

- So far the example we have looked at a parallel region in which all threads executed the same instructions.

- This is not practical in a real-life application where the main objective of parallel computing is to share the workload among multiple threads in order to expedite the solution of problems.

- OpenMP provides the following **Work-sharing constructs**, which can be used to assign independent work to one or all of the threads:

    - ***omp for*** (C/C++) or ***omp do*** (Fortran): used to split up loop iterations among the threads, also called loop constructs.
    - ***sections***: assigning consecutive but independent code blocks to different threads
    - ***single***: specifying a code block that is executed by only one thread, a barrier is implied in the end
    - ***master***: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

- Consider the example provided (**workshare.c**). This is a simple example that illustrates the use of multiple threads to initialize the value of a large array in parallel, using each thread to do part of the work including carrying out a summation.

- The loop counter ***i*** is declared inside the parallel ***for*** loop so that each thread has a unique and private version of the variable.

- Basic guidelines for working with loops:

    - Identify computing intensive loops
    - Make the loop iterations independent in order to remove loop-carried dependencies.

<u>The *critical* Synchronization clause</u>

- Race conditions can become a serious problem when using global variables due to the fact that multiple threads could modify the variable simultaneously.
- In order to avoid that, we can specify a "critical" section of code, which can be executed by all threads but not simultaneously, i.e., only **one thread** can manipulate it at a time.
- This essentially serializes the process but it allows applications to for example, update a global variable with local results from each thread safely without race conditions.
- However, it is important to keep in mind that this method may serialize the whole parallel computation and introduce a scalability bottleneck (Amdahl's law).
- The clause is used as follows:

```
#pragma omp critical
{
        <code-block>
}
```

- The code example provided (**critical.c**) illustrates the use of the critical clause.
- Each thread is assigned a portion of the total iterations carried out in a loop. The number of iterations carried out by each thread is summed within a critical section.
- The individual thread loop iterations are then summed to produce the total number of overall iterations.

The *reduction* clause

- Reduction refers to the process of combining the results of several sub-calculations into a final result. This is a very common paradigm (Google refers to it as a "map-reduce" framework).

- Consider a very common type of computation such as:

```
double avg = 0;
double Data[MAX];
int i;

for (i = 0; I < MAX; i++)
{
    avg += Data[i];
}
avg = avg/MAX;
```

- Here, ***avg += Data[i]*** is what is referred to as a "**reduction operation**". There is dependence between loop operations ("loop carried flow dependence") which cannot be trivially removed and thus the variable "**avg**" prohibits parallelization.

- For these kind of cases OpenMP provides a reduction clause:

    reduction (op : list)

    - The allowable operators for "**op"** are +, * , -
    - ***list*** is the reduction variable: "***avg***" in our example.

- This creates a parallel or work-sharing construct as follows:

    - A local copy of each list variable is created and initialized
    - The compiler determines standard reduction expressions containing "***op***" and uses them to update the local copy.
    - Local copies are then **reduced** into a single value and **combined** with the **original global value**.

- The variables in "**list**" must be shared within the parallel region.

- We can now modify our example as follows:

```
double avg = 0;
double Data[MAX];
int i;

#pragma omp parallel for reduction (+:avg)
for (i = 0; I < MAX; i++)
{
    avg += Data[i];
}
avg = avg/MAX;
```

- There are several different associative operands that can be used with reduction:

Operator	Initial Value
+	0
*	1
-	0
& (AND)	0
| (OR)	0
^ (Complement)	0
&& (conditional AND)	1
|| (conditional OR)	0

- The code example provided (**reduction.c**) illustrates this paradigm. The code is essentially the same as in the **critical.c** example, except there is no need to use the ***critical*** clause anymore.

- Let us look at another example. The code example provided (**critsum.c**) shows a summation task where there is a dependency with the calculation of ***Global_Sum***.

- It is calculated within a critical section may reduce the advantages of parallelization depending on the architecture.

- This is addressed in the next example (**reducsum.c**), which uses reduction operation, which is a much better way to solve this problem.

- Notice how much cleaner (and simpler) the code becomes using ***reduction***.

Loop Scheduling – The *schedule* clause

- Consider the following code fragment:

```
#pragma omp parallel for
{
        for (i = 0; i < 1000; i++)
        {
                foo(i);
        }
}
```

- The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause:

  schedule (type, chunk)

- The three types of scheduling are:

- ***static*** (**default**)

  - Scheduling is done at compile time.
  - Each thread is assigned a fixed-size chunk and all the threads are allocated iterations before they execute the loop iterations.
  - The iterations are divided among threads equally by default.
  - Specifying an integer for the parameter ***chunk*** will allocate a "chunk" number of contiguous iterations to a particular thread.

- ***dynamic***

  - Scheduling is done at run-time.
  - Work is assigned as a thread requests it. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are remaining.
  - The parameter ***chunk*** defines the number of contiguous iterations that are allocated to a thread at a time.

- ***guided***

  - This is a special case of dynamic used to reduce scheduling overhead.
  - A large chunk of contiguous iterations are allocated to each thread dynamically (as above).
  - The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter ***chunk***

Synchronization clauses

- Assume that an application has communication between threads using shared memory; a consumer thread must not start reading the memory before the producer completes writing of the data into the shared memory. This is achieved by **synchronization**.

- In parallel programming, it can be done by barrier synchronization as follows:

- Each thread writes the data to be sent into the memory.
- Do a ***barrier*** synchronization.
- Each thread reads the data to be received from the memory.
- Do a ***barrier*** synchronization.

- The last barrier is required to prevent a next write to the same memory area.

- As more tangible example consider the following two loops running in parallel over variable ***i***:

```
for (i=0; i < N; i++)
        a[i] = b[i] + c[i];

for (i=0; i < N; i++)
        d[i] = a[i] + b[i];
```

- This is almost certain to produce an error because all of a[ ] must be updated first, before using a[ ] in the second loop.

- The solution is to have all threads wait at the ***barrier*** point and only continue when all threads have reached the ***barrier*** point.

- The following are all of the synchronization clauses provided by OpenMP:

- ***critical***

    - The enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.

- ***atomic***

    - The memory update (write, or read-modify-write) in the next instruction will be performed atomically.
    - It does not make the entire statement atomic; only the memory update is atomic.

- ***ordered***

    - The structured block is executed in the order in which iterations would be executed in a sequential loop.

- ***barrier***

    - A work-sharing construct has an implicit barrier synchronization at the end.
    - Each thread waits until all of the other threads of a team have reached this point.

- ***nowait***

    - Specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish.
    - In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

OpenMP Advantages

- Simplicity. In most cases, "the right way" to implement it cleanly and simply.

- Incremental parallelization possible. We can incrementally parallelize a sequential code, one block at a time.

- Very easy to debug and validate.

- Thread management is left to the compiler, thus making it very easy to implement.

- A lot more information can be found at:

  http://www.openmp.org/

  https://computing.llnl.gov/tutorials/openMP/