WRITING PARALLEL PROGRAMS

Dr Noor Mahammad Sk

Center for High Performance Reconfigurable Computing

Hprcse

Need for Parallel Programming

- To solve huge supercomputing problems.
- Every professional software developer must know and master as multi-core processors have emerged
- Parallel programming can be difficult, but its better understood as “just different” not “difficult.”

Parallel Programming

- It includes all the characteristics of more traditional, serial programming
- In parallel programming, there are three additional and well defined steps
- Identify concurrency:
- Analyze a problem to identify tasks that can execute concurrently
- Expose concurrency:
- Restructure a problem so tasks can be effectively exploited.
- This often requires finding the dependencies between tasks and organizing the source code so they can be effectively managed.
- Express concurrency:
- Express the parallel algorithm in source code using a parallel programming notation

Parallel Programming

- We focus on step three: expressing a parallel algorithm in the source code using a parallel programming notation
- This notation can be a parallel programming language, an application programming interface (API) implemented through a library interface, or a language extension added to an existing sequential language

OpenMP

- OpenMP [omp] is an industry standard API for writing parallel application programs for shared memory computers
- The primary goal of OpenMP is to make the loop oriented programs common in high performance computing easier to write
- Constructs were included in OpenMP to support SPMD (single program, multiple data), Master worker, pipeline and most other types of parallel algorithms as well
- OpenMP has been a very successful parallel language

OpenMP

- It is available on every shared memory computer on the market
- Recently Intel© has created a variation on OpenMP to support clusters as well
- OpenMP supports a style of programming where parallelism is added incrementally so an existing sequential program evolves into a parallel program
- This advantage, however, is also OpenMP's greatest weakness
- By using incremental parallelism, a programmer might miss the large scale restructuring of a program often required to get the most performance

OpenMP

- OpenMP is a continuously evolving standard.
- An industry group called “the OpenMP Architecture Review Board” meets regularly to develop new extensions to the language.
- Version 3.0 has been released May 2008
- This will allow OpenMP to handle a wider range of control structures as well as more general recursive algorithms.

OpenMP Overview

- OpenMP is based on the fork-join programming model.
- A running OpenMP program starts as a single thread.
- When the programmer wishes to exploit concurrency in the program, additional threads are forked to create a team of threads.
- These threads execute in parallel across a region of code called a parallel region.
- At the end of the parallel region, the threads wait until all of the threads have finished their work, and then they join back together.
- At that point, the original or “master” thread continues until the next parallel region is encountered (or the end of the program)

When to consider OpenMP?

- Using an automatically parallelizing compiler:
  - It can not find the parallelism
    - The data dependence analysis is not able to determine whether it is safe to parallelize or not
  - The granularity is not high enough
    - The compiler lacks information to parallelize at the highest possible level
- Not using an automatically parallelizing compiler:
  - No choice than doing it yourself

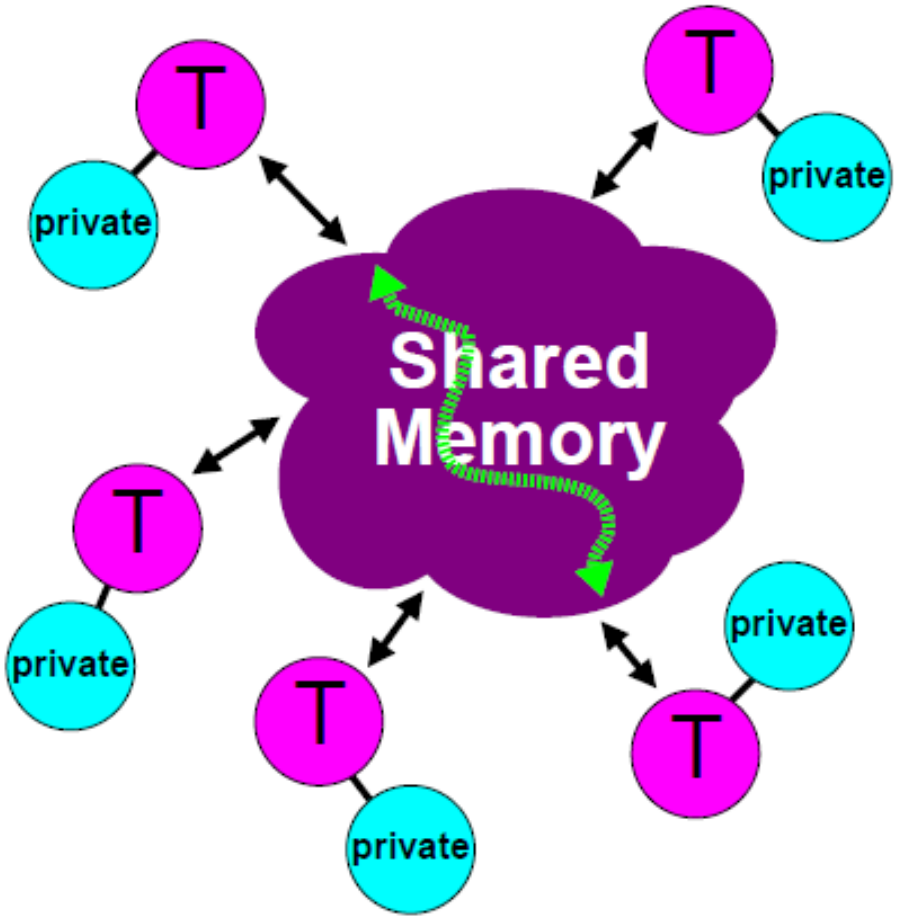
Advantages of OpenMP

- Good performance and scalability
  - If you do it right ....
- De-facto and mature standard
- An OpenMP program is portable
  - Supported by a large number of compilers
- Requires little programming effort
- Allows the program to be parallelized incrementally

OpenMP and Multicore

- OpenMP is ideally suited for multicore architectures
- Memory and threading model map naturally

The OpenMP Memory Model



- All threads have access to the same, globally shared, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit

TUTORIAL ON OpenMP

Dr Noor Mahammad Sk

Center for High Performance Reconfigurable Computing

Hprcse

What is OpenMP?

- An Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory*** parallelism.
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- **OpenMP** is An abbreviation for: **Open Multi-Processing**

OpenMP Is Not:

- Meant for distributed memory parallel systems (by itself)
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, deadlocks, or code sequences that cause a program to be classified as non-conforming
- Designed to handle parallel I/O.
- The programmer is responsible for synchronizing input and output.

Goals of OpenMP

- **Standardization:**
- Provide a standard among a variety of shared memory architectures/platforms
- Jointly defined and endorsed by a group of major computer hardware and software vendors
- **Lean and Mean:**
- Establish a simple and limited set of directives for programming shared memory machines.
- Significant parallelism can be implemented by using just 3 or 4 directives.

Goals of OpenMP:

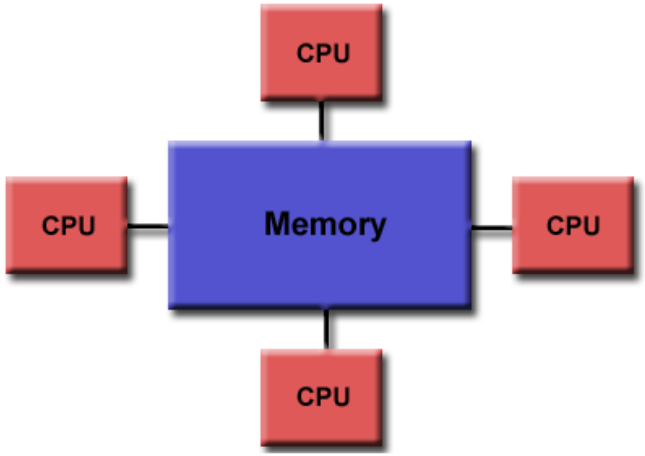
- **Ease of Use:**
- Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
- Provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:**
- The API is specified for C/C++ and Fortran
- Public forum for API and membership
- Most major platforms have been implemented including Unix/Linux platforms and Windows

History

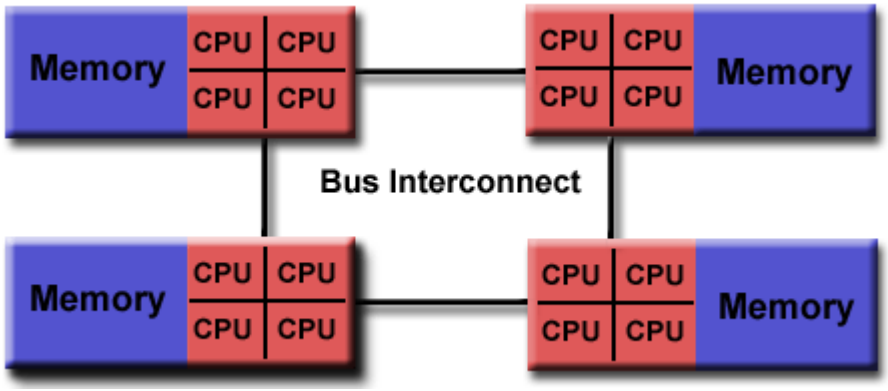
- In the early 90's, vendors of shared-memory machines supplied similar, directive-based, Fortran programming extensions:
- The user would augment a serial Fortran program with directives specifying which loops were to be parallelized
- The compiler would be responsible for automatically parallelizing such loops across the SMP processors
- Implementations were all functionally similar, but were diverging (as usual)
- First attempt at a standard was the draft for ANSI X3H5 in 1994.
- It was never adopted, largely due to waning interest as distributed memory machines became popular.

OpenMP Programming Model

- **Shared Memory Model:**
- OpenMP is designed for multi-processor/core, shared memory machines.
- The underlying architecture can be shared memory UMA or NUMA.



Uniform Memory Access



Non-Uniform Memory Access

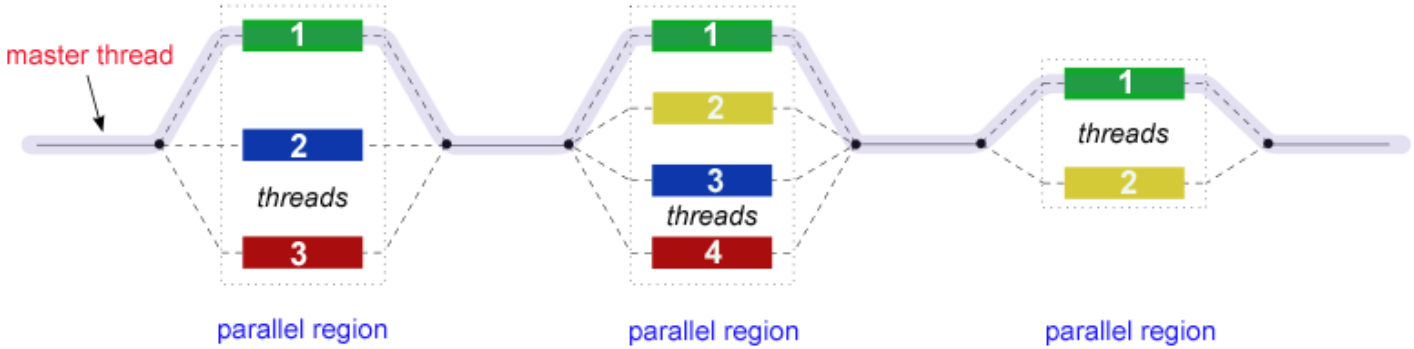
Thread Based Parallelism

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
- The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a single process.
- Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores.
- The actual use of threads is up to the application.

Explicit Parallelism

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives....
- Or
- As complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

Fork - Join Model



- OpenMP uses fork-join model of parallel execution
- All OpenMP programs begin as a single process: the **master thread**.
- The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a team of parallel ***threads***.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

Data Scoping

- Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default.
- All threads in a parallel region can access this shared data simultaneously.
- OpenMP provides a way for the programmer to explicitly specify how data is "scoped" if the default shared scoping is not desired.

Nested Parallelism

- The API provides for the placement of parallel regions inside other parallel regions.
- Implementations may or may not support this feature.

Dynamic Threads

- The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions.
- Intended to promote more efficient use of resources, if possible.
- Implementations may or may not support this feature.

I/O

- OpenMP specifies nothing about parallel I/O.
- This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Three Components of OpenMP

- The OpenMP 3.1 API is comprised of three distinct components
- Compiler Directives
- Runtime Library Routines
- Environment Variables

Compiler Directives

- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag
- **OpenMP compiler directives are used for various purposes**:
- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

C/C++ **#pragma omp parallel default(shared) private(beta,pi)**

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 and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

C/C++

```
#include <omp.h>
int omp_get_num_threads(void)
```

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 interations are divided
- Binding threads to processors
- Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
- Enabling/disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy

sh/bash **export OMP_NUM_THREADS=8**

HelloWorld in C

```
#include 'omp.h'
void main()
{
        #pragma omp parallel
        {
                int ID = omp_get_thread_num()
                printf('Hello(%d) ',ID);
                printf('World(%d) \n',ID);
        }
}
```

> gcc -fopenmp HelloWorld.c -o Hello

> export OMP_NUM_THREADS=4 for 4 threads in a bash shell

A sample output from this program is shown below

```
hprcse@hprcse:~/Desktop$ gcc -fopenmp HelloWorld.c
hprcse@hprcse:~/Desktop$ export OMP_NUM_THREADS=4
hprcse@hprcse:~/Desktop$ ./a.out
Hello(0) World(0)
Hello(1) World(1)
Hello(3) World(3)
Hello(2) World(2)
hprcse@hprcse:~/Desktop$
```

How do I decide if a loop is parallel or not?

- Loops are the most common parts of a code to parallelise.
- When run in parallel, a loop will divide its iterations between multiple threads.
- Not all loops can be parallelised though so it is important to think about what the dependencies are in the loop.
- One useful test is to consider if the loop will give the same answers if it is run backwards.
- If this works then the loop is almost certainly parallelisable.

Example 1

```
for (i=1; i<n; i++)
{
        a[i] = 2 * a[i-1];
}
```

- In this example, a[i] depends on a[i-1], so it can not be parallelised.

Example 2

```
for (i=1; i<n; i++)
{
        b[i] = (a[i] - a[i-1]) * 0.5;
}
```

- In this case, all of the iterations are independent so it is possible to make this loop parallel

What are Private and Shared variables?

```
shared(list)
private(list)
default(shared|none)
```

- Inside a parallel region of a program, variables can either be shared or private or default
- In this model, all threads can see the same copy of shared variables and all threads can read or write shared variables.
- Each thread has its own copy of a private variable that is invisible to all other threads.
- A private variable can only be written by its own thread.
- We define the scope of the variables because it is up to the programmer to make sure that memory is accessed correctly (otherwise the code may give rubbish outputs).

The Default Clause

- The Default clause allows the user to specify the default scope for all of the variables in the parallel region.
- If this is not specified at the beginning of the parallel region, the default is automatically shared.
- Specific variables can be exempted from the default by using one of the other clauses.

```
#pragma omp parallel default(private), shared(a,b)
```

The above declarations state that all the variables in the parallel region are private apart from variables a and b that are shared.

The Shared Clause

- Most variables are shared and these variables exist only in one memory location and all threads can read or write to that address.
- When trying to decide whether a variable is shared, consider what is happening to it inside the parallel region.
- Generally read only variables are shared (they can not accidently be over written) and main arrays are shared.

```
#pragma omp parallel default(none), shared(a,b)
```

In this example, the variables a and b are the only variables in the parallel region and they are both shared

The Private Clause

- A variable that is private can only be read or written by its own thread.
- When a variable is declared as private,
- A new object of the same type is declared once for each thread and all references to the original object are replaced with references to the new object.
- By default, loop indices are private and loop temporaries are private.

```
#pragma omp parallel default(none), shared(a,b), private(i,j)
```

In this example, the only variables in the parallel region are a, b, i and j. The former pair are shared and the latter two are private.

Example

```
#include <omp.h>
#define N 10
int main(void)
{
   float a[N], b[N]; int i;
   #pragma omp parallel default(none), private(i), shared(a,b)
   {
      #pragma omp for
      for (i = 0; i < N; i++)
      {
            a[i] = (i+1) * 1.0;
            b[i] = (i+1) * 2.0;
            printf("%d, %f, %f \n",i+1,a[i],b[i]);
      }
    }
}
```

Example

Serial:

```
void main ()
{
     double x(256);
     for(int i=0; i<256; i++)
     {
        some_work(x[i]);
     }
}
```

OpenMP:

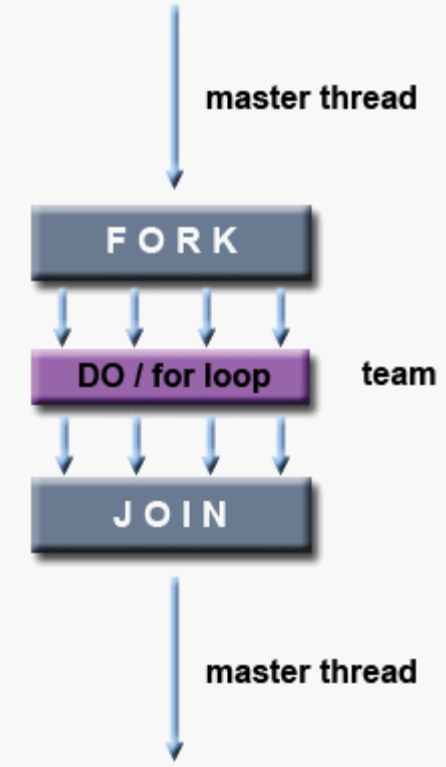
```
#include "omp.h"
Void main ()
{
      double x(256);
      #pragma omp parallel for
      for (int i=0; i<256; i++)
      {
          some_work(x[i]);
      }
 }
```

Work-Sharing Constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

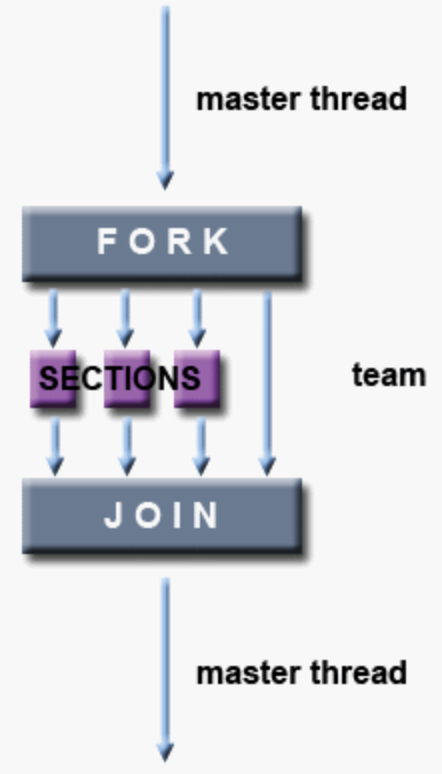
Types of Work-Sharing Constructs

- **DO / for -** shares iterations of a loop across the team. Represents a type of "data parallelism".



Sections

- **SECTIONS** - breaks work into separate, discrete sections.
- Each section is executed by a thread.
- Can be used to implement a type of "functional parallelism".



Single

- **SINGLE** - serializes a section of code