

Intro to OpenMP

Parallel Computer Memory Architectures

- ▶ There are three possible parallel computers' memory architectures:

1. Shared memory

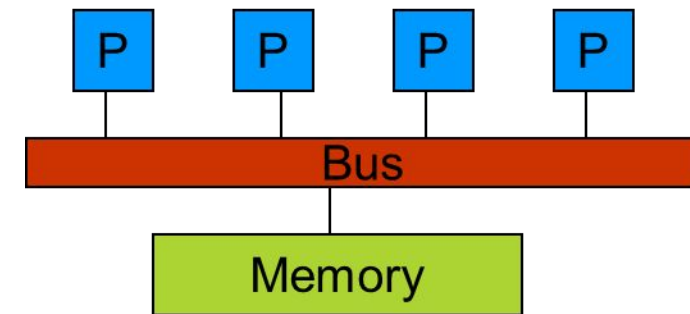
- ▶ *Uniform Memory Access (UMA)*
- ▶ *Non-Uniform Memory Access (NUMA)*

2. Distributed memory

3. Hybrid Distributed Shared memory

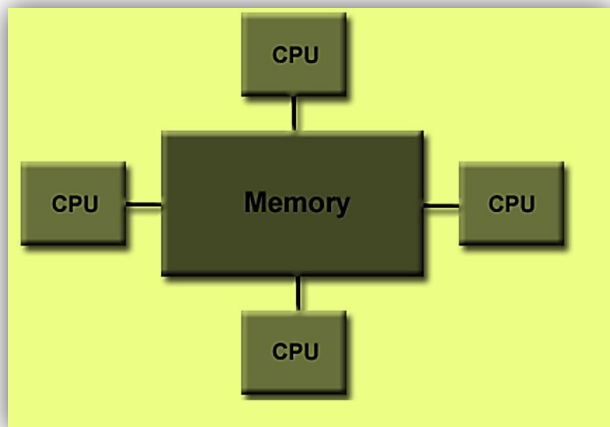
Shared Memory Architecture

- ▶ In shared memory parallelism, threads share a memory space among them.
- ▶ Threads are able to read and write to and from the memory of other threads.
- ▶ One of the standard for shared memory considered is OpenMP, which uses a series of pragmas, or directives for specifying parallel regions of code in C, C++ or Fortran to be executed by threads.
- ▶ **In this architecture, the programmer's task is to specify the activities of a set of processes that communicate by reading and writing shared memory.**

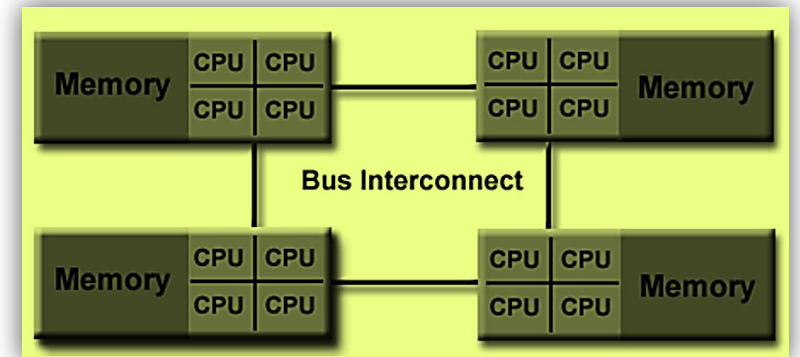


Shared Memory Architectures

- ▶ **Uniform Memory Access (UMA):** in this architecture, the identical processors have equal access times to memory; commonly used in symmetric multiprocessor (SMP) systems.



Non-Uniform Memory Access (NUMA): in this architecture many SMPs are linked together, however all processors do not have equal access times to the memories of all other SMPs. Moreover, the memory access across the link is slower.



Thread v. Process

Process:

- ▶ A process is **created** by the **OS** to execute a **program** with given **resources** (e.g., memory, registers)
- ▶ **Different processes** do **not share** their **memory** with another.

Thread:

- ▶ A thread is a **subset of a process**, and it shares the resources of its parent process,.
- ▶ Exists within a process and uses the process resources
- ▶ Has its own **stack** to keep **track of function calls** and its own **independent** flow of control
- ▶ **Multiple threads** of a **process** will have access to the **same memory** (but can have local variables).
- ▶ Is "**lightweight**" because most of the overhead has already been accomplished through the creation of its process.
- ▶ **Dies** if the **parent process** dies

OpenMP

- ▶ **OpenMP**
- ▶ is an API built for **shared-memory** parallelism.
- ▶ Simplifies writing **multi-threaded (MT) applications**.
- ▶ The **OpenMP API** is comprised of **three distinct components**:
 - ▶ **compiler directives,**
 - ▶ **runtime library routines**
 - ▶ **environment variables**

OpenMP API

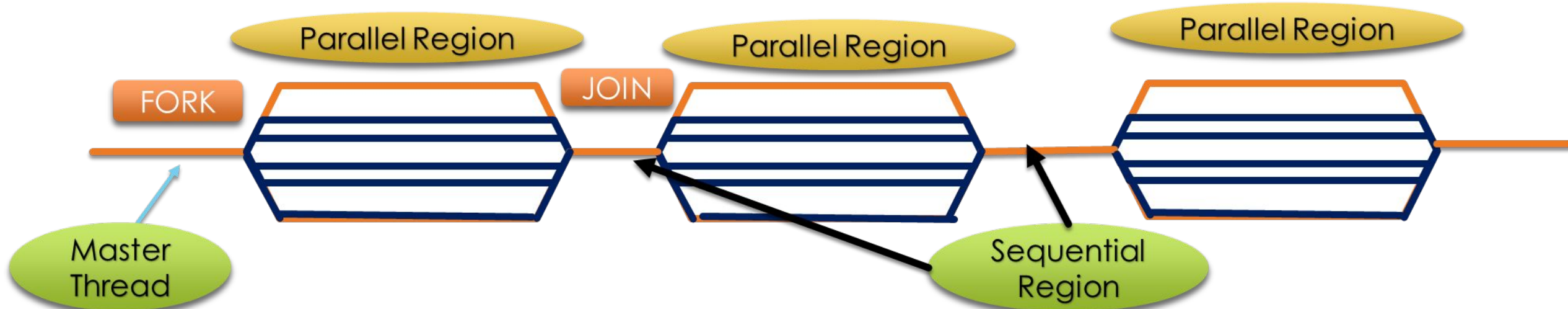
The OpenMP API:

- ▶ **Compiler directives:** typed in your source code, these are instructions for the compiler regarding how to parallelize your code. If you set the right flags at compilation, these directives are read and understood, else, they are ignored. **`#pragma omp construct [clause [clause]...]`**
- ▶ **Runtime library routines:** typed in your source code, these are function calls to functions of the OpenMP library (omp.h in C/C++). **`#include <omp.h>`**
- ▶ **Environment variable:** typed in the terminal, this is a variable used by the system that can be modified or retrieved.

OpenMP Programming Model

Fork-Join Parallelism:

- ▶ Group of threads are spawned by the master thread to permit parallel execution as needed. **[FORK]**
- ▶ The parallel execution merge at a subsequent point to resume sequential execution. **[JOIN]**
- ▶ Parallelism is added incrementally until performance goals are met.



OpenMP Compiler Directives

► parallel compiler directive

USAGE:

```
#pragma omp parallel [options] {  
    Code inside here runs in parallel  
    Among options available: declaring private/shared vars  
}
```

- EXAMPLE: `#pragma omp parallel private(var1, var2) shared(var3) {`
- The parallel pragma starts a parallel block. It creates a team of N threads (where N is determined at runtime), all of which execute the next statement or block (a block of code requires a {...} enclosure). After the statement, the threads join back into one.

OpenMP Compiler Directives

► for compiler directive

USAGE:

```
#pragma omp for [clause]{  
    for loop  
}
```

► EXAMPLE:

```
# pragma omp parallel for  
for (index = 0; index < length; index++) {  
    array[index] = index * index;  
}
```

The iterations of the loop will be computed in parallel (note that they are independent of one another).

OpenMP Environment Variable

- ▶ **OMP_NUM_THREADS** environment variable
- ▶ **USAGE:** OMP_NUM_THREADS=number
- ▶ **EXAMPLE:** export OMP_NUM_THREADS=16 **Sets the value in the shell**
- ▶ This environment variable tells the library how many threads that can be used in running the program. If dynamic adjustment of the number of threads is enabled, this number is the maximum number of threads that can be used, else, it is the exact number of threads that will be used.
- ▶ The default value is the number of online processors on the machine.

OpenMP Functions

- ▶ **omp_get_thread_num** function

- ▶ **EXAMPLE:** `uid = omp_get_thread_num()`

- ▶ This function asks the thread that is executing it to identify itself by returning its unique number. [answers "Who am I?"]

OpenMP Functions



- ▶ **omp_get_num_threads** function

- ▶ **EXAMPLE:** `threadCount = omp_get_num_threads()`

- ▶ This function returns the number of threads in the team currently executing the parallel block from which it is called. [answers "How many of us?"]. If this "get" function exists, might there be a corresponding function?

OpenMP Reduction



- ▶ **The reduction command**

- ▶ **USAGE:** reduction (operator : list)

- ▶ A reduction is a repeated operation that operates over multiple values and yields one value. So, the reduction command performs an operation (using a specified operator) on all of the variables that are in its list. The reduction variable(s) is a shared variable, not a private variable. Operators can include: +, *, -, /, &, ^, |, &&, | |.

- ▶ **EXAMPLE:** #pragma omp parallel for private(privateVar) reduction(+:runningTotal)

OpenMP Scheduling Strategies

OpenMP supports three scheduling strategies:

- ▶ **Static:** The default, as described in the previous slides – good for iterations that are inherently load balanced.
- ▶ **Dynamic:** Each thread gets a chunk of a few iterations, and when it finishes that chunk it goes back for more, and so on until all of the iterations are done – good when iterations aren't load balanced at all.
- ▶ **Guided:** Each thread gets smaller and smaller chunks over time – a compromise.

OpenMP Scheduling

- ▶ The **PARALLEL DO** directive allows a **SCHEDULE** clause to be appended that tell the compiler which variables are shared and which are private:

!\$OMP PARALLEL DO ... SCHEDULE(STATIC)

- ▶ This tells that compiler that the schedule will be static.
- ▶ Likewise, the schedule could be **GUIDED** or **DYNAMIC**.
- ▶ However, the very best schedule to put in the **SCHEDULE** clause is **RUNTIME**.
- ▶ You can then set the environment variable **OMP_SCHEDULE** to **STATIC** or **GUIDED** or **DYNAMIC** at runtime – great for benchmarking!

Acknowledgements

Adapted from:

- ▶ **Shared Memory and OpenMP Basics** at <http://shodor.org/~bplist/bwi/openmp.html>
- ▶ **Supercomputing in Plain English, Distributed Multiprocessing** Slides by Henry Neeman.

Thank You!

