Agenda

- Programming Models:
  - Shared Address space
  - Message Passing

- Programming Shared Address Space:
  - Thread basics
  - OpenMP
    - Programming model
    - Threads creation & termination.
    - Work sharing constructs
    - Reduction clause
    - Synchronization & barriers
    - Mutual execution
Overview of Programming Models

- Programming models provide support for expressing concurrency and synchronization:
  - Shared Address space
  - Message passing

Comparison!!

<table>
<thead>
<tr>
<th>Shared Address Space</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td></td>
</tr>
<tr>
<td>Programming</td>
<td></td>
</tr>
<tr>
<td>Pros/Cons</td>
<td></td>
</tr>
</tbody>
</table>
Programming Shared Address Space

Overview

- assume that all memory is global
  - relaxing the protection domain
  - support much faster manipulation
- Directive based programming models extend the threaded model by facilitating creation and synchronization of threads.
The logical machine model of a thread-based programming paradigm.

Performance??

the cost to access a physically local memory may be an order of magnitude less than that of accessing remote memory

---

Thread Basics

- one may think of the thread as an instance of a function that returns before the function has finished executing.
- The stack corresponding to the function call is local to the thread.
- logical machine model:
  - global memory (default)
  - local memory (stacks).
- threads are scheduled at runtime (no a priori schedule of their execution can be safely assumed)
A thread is a single stream of control in the flow of a program:
- provide software portability.
- Inherent support for latency hiding.
- Implicit scheduling and load balancing.
- Ease of programming and widespread use.

Matrix–matrix multiplication example

```c
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product( get_row(a, row),
                        get_col(b, col));
```

```c
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(  
                dot_product(get_row(a, row),
                            get_col(b, col)));
```
Multithreading APIs

- A number of vendors provide vendor-specific thread APIs:
  - POSIX API; Also referred to as Pthreads
  - NT threads,
  - Solaris threads,
  - Java threads,
  - OpenMP, etc

What is OpenMP?

- Application program interface (API) for shared memory parallel programming
- A specification for a set of compiler directives, library routines, and environment variables
- Make it easy to create multi-threaded (MT) programs in Fortran, C and C++
- Portable / multi-platform, including Unix platforms and Windows NT platforms
- Jointly defined and endorsed by a group of major computer hardware and software vendors
OpenMP is not....

- Not Automatic parallelization
  - User explicitly specifies parallel execution
  - Compiler does not ignore user directives even if wrong
- Not just loop level parallelism
  - Functionality to enable coarse grained parallelism
- Not meant for distributed memory parallel systems
- Not necessarily implemented identically by all vendors
- Not Guaranteed to make the most efficient use of shared memory

OpenMP Programming Model

- OpenMP directives provide support for:
  - concurrency,
  - synchronization,
  - data handling,
  - Mutual exclusion
  - data scope and initialization
OpenMP Directives Format

- OpenMP directives in C and C++ are based on the #pragma compiler directives.
- A directive consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

OpenMP Programming Model

OpenMP programs execute serially until they encounter the parallel directive, which creates a group of threads.
Clauses

- **Conditional Parallelization**: The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads.
- **Degree of Concurrency**: The clause `num_threads(integer expression)` specifies the number of threads that are created.
- **Data Handling**:
  - `private (variable list)`: variables local to each thread.
  - `firstprivate (variable list)`: local variables that are initialized to corresponding values before the parallel directive.
  - `shared (variable list)`: variables are shared across all the threads.

Example

```c
//serial Segement

#pragma omp parallel
if (is_parallel== 1) num_threads(8) private (a) shared (b) firstprivate(c)
{
    // parallel Segement
}

// rest serial Segement
```
PARALLEL Region Construct

- omp_get_num_threads()
- omp_get_thread_num() \rightarrow \text{thread id 0 } \rightarrow \text{the master of the group}

Specify Concurrent Tasks

- Divide the execution of the enclosed code region among the members of the team that encounter it
- Do not launch new threads
- Must be enclosed within a parallel region
- Directives to specify concurrency:
  - sections - concurrent tasks (functional parallelism).
  - for - concurrent iterations (data parallelism)
OpenMP supports non-iterative parallel task assignment using the `sections` directive.

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        { taskA(); }
        #pragma omp section
        { taskB(); }
        #pragma omp section
        { taskC(); }
    }
}
```

The `sections` directive allows a sequence of `for` directives within a parallel construct, with no need to execute an implicit barrier at the end of each `for` directive.

Parallel `for` clauses are:
- `private`
- `firstprivate`
- `lastprivate`
- `schedule`
- `reduction`
- `nowait`
- `ordered`
More on OpenMP

After the break!!

**Parallel for**

a sequence of for-directives within a parallel construct
→ no need to execute an implicit barrier at the end of each for directive → nowait

#pragma omp for [clause list]
/* for loop */

**Clauses are:**
private, firstprivate, lastprivate, schedule, reduction, nowait, and ordered

- master thread
- team
- master thread
Assigning Iterations to Threads

- The schedule clause of the for directive deals with the assignment of iterations to threads.

/* static scheduling of matrix multiplication loops */
#pragma omp parallel
default(private)
shared (a, b, c, dim)  //dim = 128
num_threads(4)

#pragma omp for schedule(static)
for (i = 0; i < dim; i++) {
    for (j = 0; j < dim; j++) {
        c(i, j) = 0;
        for (k = 0; k < dim; k++) {
            c(i, j) += a(i, k) * b(k, j);
        }
    }
}
Static mapping of matrix multiplication example

Three different schedules using the static scheduling class of OpenMP.

Nesting parallel Directives: Example

- generates a logical team of threads on encountering a nested parallel directive → split each of the for loops across various threads
- Nested parallelism must be enabled using the OMP_NESTED environment variable.
The reduction clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.

reduction (operator: variable list)

- The variables in the list are implicitly private to threads.
- The operator can be one of +, *, -, & |, ^, &&, and ||.
- examples??

Example: Dot Product
variable name needs to be looked up in two lists.
If the name exists in a list, it must be processed accordingly.
The name might exist in both lists.

The ORDERED directive

- The ORDERED directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.
- Threads will need to wait before executing their chunk of iterations if previous iterations haven’t completed yet.
- Used within a DO / for loop with an ORDERED clause
Computing the cumulative sum of a list

- The ordered directive represents an ordered serialization point in the program.
- Only a single thread can enter an ordered block when all prior threads (as determined by loop indices) have exited.
- A single for directive is constrained to have only one ordered block.

Synchronization Constructs

- OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.

cumul_sum[0] = list[0];
#pragma omp parallel for private (i)
shared (cumul_sum, list, n) ordered
for (i = 1; i < n; i++)
{
    /* other processing on list[i]...*/
    #pragma omp ordered
    {
        cumul_sum[i] = cumul_sum[i-1] + list[i];
    }
}
Consider a simple example where two threads on two different processors (assume x is initially 0)

THREAD 1:
increment(x) {
    x = x + 1;
}

THREAD 2:
increment(x) {
    x = x + 1;
}

10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)

Why Synchronization ??

both processor is trying to increment a variable x at the same time

Synchronization Constructs in OpenMP

OpenMP provides a variety of synchronization constructs:

- #pragma omp barrier
- #pragma omp single [clause list]
- structured block
- #pragma omp master
- structured block
- #pragma omp critical [(name)]
- structured block
- #pragma omp ordered
- structured block
a barrier holds a thread until all threads participating in the barrier have reached it.

Barriers

- Barriers implemented:
  - A single integer is used to keep track of the number of threads that have reached the barrier.
  - If the count is less than the total number of threads, the threads execute a condition wait.
  - The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.
The Master Directive

- specifies a region that is to be executed only by the master thread.
- All other threads on the team skip this section of code.
- There is no implied barrier associated with this directive.

The single Directive

- the enclosed code is to be executed by only one (arbitrary) thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O).
Mutual Exclusion for Shared Variables:
- All threads in the team will attempt to execute in parallel,
- the CRITICAL construct surrounding the increment of x
- only one thread will be able to read/increment/write x at any time.

```c
int x = 0;
#pragma omp parallel shared(x)
{
    #pragma omp critical
    x = x + 1;
}
```

- provides a mini-CRITICAL section.
  - specifies that a specific memory location must be updated atomically.
  - expressions:
    - x binary_operation = expr
    - x++
    - ++x
    - x
    - --x
- All atomic directives can be replaced by critical directives

```c
#pragma omp atomic
statement_expression
```
Environment Variables in OpenMP

- **OMP_NUM_THREADS**: This environment variable specifies the default number of threads created upon entering a parallel region.
- **OMP_SET_DYNAMIC**: Determines if the number of threads can be dynamically changed.
- **OMP_NESTED**: Turns on nested parallelism.
- **OMP_SCHEDULE**: Scheduling of for-loops if the clause specifies runtime.

That’s all for today

Thanks !!