Open Multi-Processing (OpenMP) Programming

Acknowledgement: Dr. Prasanth Jose, Assosiate Professor, SBS, IIT Mandi

OpenMP

- OpenMP standard defined in 1998 with C/C++
- Parallelizing within a core itself i.e. single core/multicore
- Simplest of all the parallel programming models
- Idea: Parallelize the code, check the time for execution
  - Need not have to massively parallelize the code that may end up in a program that is slower than serial program
- Shared memory multicore architecture

[1] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.

[2] Barbara Chapman, Gabriele Jost and Ruud van der Pas, *Using OpenMP - Portable Shared Memory Parallel Programming*, MIT Press, 2007.

[3] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan and Jeff McDonald, *Parallel programming in OpenMP*, Morgan Kaufmann Publishers, 2001.

Multiprocessors

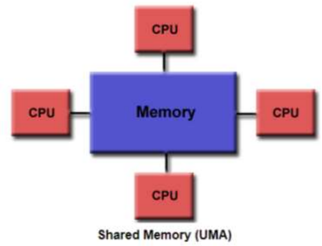
- Multiprocessor:
  - Computers consists of tightly coupled processors
  - Coordination and usage are controlled by a single operating system
  - Share memory through a shared address space
- Each processor fetches its own instructions and operate on its own data
- Exploits thread-level parallelism
- Thread is a light-weight process
  - Threads of a process share the same virtual address space
  - Multiple cooperating threads operate in parallel
- Parallel processing: Tightly coupled set of threads collaborating on a single task

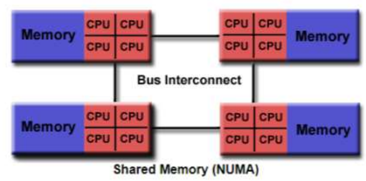
Multiprocessors

- This architecture ranges from single or dual processor to dozens of processors
  - Communicate and coordinate through the sharing memory
- Multiprocessors include in
  - Computers consists of single chip with multiple cores (multicore)
  - Computers consisting of several chips, each may be multicore design
- Shared memory model architecture

Shared Memory Model

- Centralised multiprocessors:
  - Multiple processors attached to a bus
  - all the processors share the same primary memory
    - Uniform Memory Access (UMA) or Symmetric Multiprocessor (SMP)
- Distributed multiprocessors:
  - Processors are distributed and connected by bus interconnect
  - Distributed collection of memories forms one logical address space
    - Same address on different processors refers to the same memory location
    - Nonuniform Memory Access (NUMA)





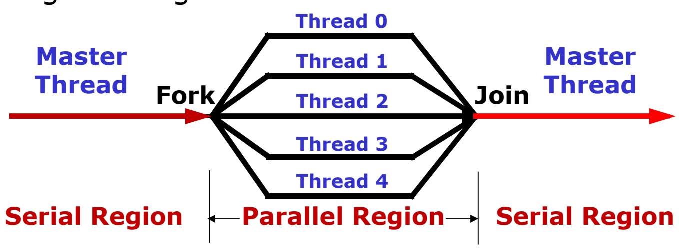
Support of **Cache Coherence**

OpenMP – Shared Memory Programming

- OpenMP has emerged as shared-memory standard
- OpenMP is an application programming interface (API) for parallel programming on multiprocessors
- It utilizes all the cores to perform a particular operation if there is no dependency in iterations i.e. no loop level dependency
- It has set of compiler directives and library of support functions
- OpenMP works in conjunction with standard Fortran, C or C++

Fork/Join Parallelism

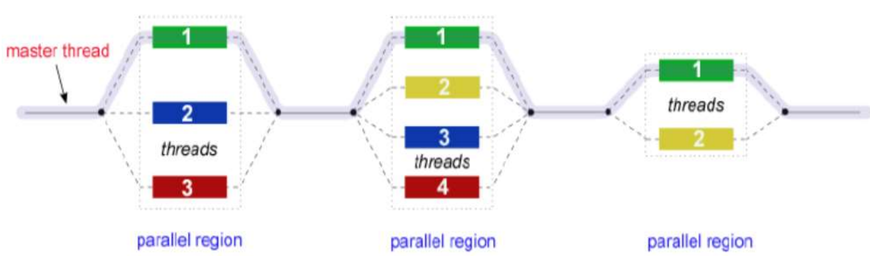
- The standard view of parallelism in shared-memory programming



- At the beginning of execution only a single thread, called master thread is active
  - Master thread executes the sequential portion of the code
- Fork: Mater thread create parallel threads when parallel region is approached
- Join: Once parallel job is completed the threads synchronize and terminate

Incremental Parallelism

- OpenMP (shared memory programming) supports incremental parallelism
- Incremental parallelism is the process of transforming a sequential program into parallel program one block of code at a time



- Sequential program is a special case of shared memory parallel program with no fork/join

Work Sharing Among the Threads

- Programmer has to explicitly specify how work is shared among different threads
- If programmer does not specify the way work is shared, all threads will redundantly execute work specified inside a parallel section
  - This may slow down the execution of the code rather and reducing the time
- Work sharing compiler directive specify how work is shared among threads in the parallel region
- Most common work sharing is to distribute work in iterations in work loop among different threads
- It is possible to control how exactly threads take up jobs in the threads

OpenMP-C Program Compile and Execution

- Compiling (in GCC):
  - `gcc -fopenmp -o example.out example.c`
- Execution:
  - `./example.out`
    - Runs the executable
  - `time ./example.out`
    - Runs the executable and display the running time of complete program

OpenMP Compiler Directives

- Pragma:
  - Pragmatic information
  - Compiler directive to indicate that this code onwards OpenMP starts
  - `#pragma omp <rest of pragma>`
- `parallel` Pragma:
  - OpemMP compiler directive to indicate that block of succeeding codes to be executed by multiple threads
  - It indicate that parallel region starts from this code
  - `#pragma omp parallel`
- `parallel for` Pragma:
  - Compiler directive to run the iteration in `for` loop concurrently
  - This is one of the most used method in programs that involve large recitative operations
  - `#pragma omp parallel for`

Important OpenMP Functions

- **Function** `omp_get_thread_num`
  - It returns the thread identification number
  - `int omp_get_thread_num()`
- **Function** `omp_get_num_procs`
  - It returns the number of physical processors available for the use by the parallel program
  - `int omp_get_num_procs()`
- **Function** `omp_get_num_threads`
  - It returns the number of threads active in the current parallel region
  - `int omp_get_num_threads()`
- **Function** `omp_set_num_threads`
  - It uses parameter value to set the number of threads to be active in parallel section of code
  - `void omp_set_num_threads(int t)`

OpenMP Environment Variable

- OMP_NUM_THREADS
  - It provides default number of threads for parallel section of code
  - This variable can also used to set the number of threads for the programs from command line
    - OMP_NUM_THREADS = *n* (*n indicate the number of threads*)
  - Example:

```
$ export OMP_NUM_THREADS=2
```


Clause - Handling Data Inside Parallel Region

- Clause:
  - Additional component to a pragma
- `private` Clause:
  - `#pragma omp parallel private(<variable list>)`
  - The private clause declares the variables in the list to be private to each thread in the parallel region
- `firstprivate` Clause:
  - `#pragma omp parallel firstprivate(<variable list>)`
  - The `firstprivate` clause directs the compiler to copy the values held by the variable list controlled by master thread (serial region), into the parallel region
  - The `firstprivate` clause provides a superset of the functionality provided by the private clause

Clause - Handling Data Inside Parallel Region

- `lastprivate` Clause:
  - `#pragma omp parallel lastprivate(<variable list>)`
  - The `lastprivate` clause provides a superset of the functionality provided by the private clause
  - It directs the compiler to copy the values held by the variable list at the end of parallel section into the serial section
- `shared` Clause:
  - `#pragma omp parallel shared(<variable list>)`
  - It declare the variable list to be completely shared among all the threads
  - If this clause is not used, the variables are shared by default

Clause - Handling Data Inside Parallel Region

- `reduction` Clause:
  - `#pragma omp parallel reduction(<op>:<variable list>)`
  - It directs the compiler to perform reduction on the scalar variable in the list, `<variable list>`, with a specified operator `<op>`
  - OpenMP reduction operators for C and C++:

Operator	Meaning	Allowable type	Initial value
+	Sum	float, int	0
*	Product	float, int	1
&	Bitwise and	int	all bits 1
|	Bitwise or	Int	0
^	Bitwise exclusive or	int	0
&&	Logical and	int	1
||	Logical or	int	0

  - Reduction of multiple variable and multiple operations can be done
  - `reduction(<op>:<variable>) reduction(<op>:<variable>)`

Clause - Handling Data Inside Parallel Region

- `default` Clause:
  - `default()`
  - The default clause allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct
- OpenMP treats all unspecified variable as shared
- It is equivalent to specifying
  - `#pragma omp parallel default(shared)`
- In C, other supported default variable is
  - `#pragma omp parallel default(none)`
- `default(none)` is a good programming practice as it will avoid accidental sharing of the variable
- Index variables of `for` loop are always private by default

OpenMP Compiler Directives – Contd.

- **Critical Section**
- `critical` Pragma:
  - This directs the compiler to enforce mutual exclusion among the threads trying to execute the block of code
  - Only one thread at a time may execute the section of code
  - `#pragma omp critical { Statements }`
- `atomic` Pragma:
  - This directs the compiler to enforce mutual exclusion among the threads trying to execute the block of code
  - It does not allow the threads to update shared data simultaneously
  - ```
    #pragma omp atomic
        statement
    ```
- atomic section is efficient alternative to reduction
- However, reduction clause is more preferred
    ```

Conditionally Executing Loops

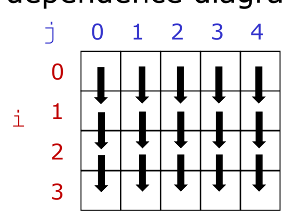
- If loops does not have enough iterations, time spent on forking and joining threads may exceed the time saved by dividing the loop iterations among multiple threads
- `if` Clause:
  - This directs the compiler to insert code that determines at run-time whether the loop should be executed in parallel or sequentially
  - `#pragma omp parallel if (<scalar expression>)`
  - If the scalar expression evaluates to true, the loop will be executed in parallel
  - Otherwise, it will be executed serially

Inverting Loops

- Sometimes transforming a sequential loop into `parallel for` loop can increase execution time
- Consider the code segment:

```
for (i=1; i<m; i++)
   for (j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

- Data dependence diagram: Let $m=4$ and $n=5$



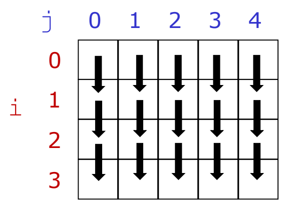
- The loop indexed by `j` maybe executed in parallel, but not loop indexed by `i`

Inverting Loops

- Insert `parallel for` pragma before inner loop

```
for (i=1; i<m; i++)
  #pragma omp parallel for
  for (j=0; j<n; j++)
    a[i][j] = 2 * a[i-1][j];
```

- Execution : Let `m=4`, `n=5` and `Num_Threads` = 5



- It may not exhibit good performance
- There will be `m-1` fork/join steps i.e. one fork/join per iteration of outer loop

Inverting Loops

- Invert the loop

```
for (j=0; j<n; j++)
  for (i=1; i<m; i++)
    a[i][j] = 2 * a[i-1][j];
```

- Insert `parallel for` pragma before outer loop

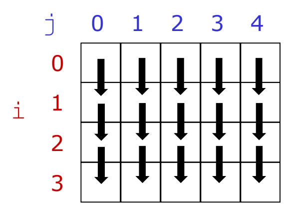
```
#pragma omp parallel for private(i)
for (j=0; j<n; j++)
  for (i=1; i<m; i++)
    a[i][j] = 2 * a[i-1][j];
```

Inverting Loops

- Insert `parallel for` pragma before outer loop

```
#pragma omp parallel for private(i)
for (j=0; j<n; j++)
    for (i=1; i<m; i++)
        a[i][j] = 2 * a[i-1][j];
```

- Execution : Let m=4, n=5 and Num_Threads = 5



- It exhibit good performance
- There will be `single` fork/join step

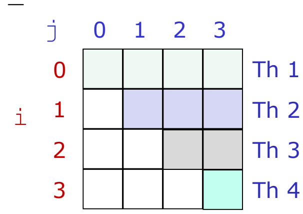
Scheduling Loops

- In some loops, the time needed to execute different loop iterations varies considerable
- Some iterations require more work and some may require less
- Consider the code segment:

```
for (i=1; i<n; i++)
    for (j=i; j<n; j++)
        a[i][j] = some_function(i, j);
```

- No dependences among iterations
- Prefer to execute each rows (outermost loop) in parallel

- Execution : Let n=4 and Num_Threads = 4



Scheduling Loops

- Suppose there are n iterations being executed on t threads
- By default, each thread is assigned with contiguous block of either $\left\lceil \frac{n}{t} \right\rceil$ or $\left\lfloor \frac{n}{t} \right\rfloor$ iterations
- The parallel loop execution will have poor efficiency
  - Some threads will complete their share of work faster than others

Scheduling Loops

- `schedule` Clause:
  - It allows us to specify how iterations of a loop should be scheduled i.e., allocated to threads
- Two types of the schedule:
  - Static schedule:
    - All iterations are allocated to the threads before they execute any loop iterations
  - Dynamic schedule:
    - Only some of the iterations are allocated to threads at the beginning of the loop execution
    - Threads that complete their iterations are eligible to get additional work
    - The allocation process continues until all the iterations have been distributed to threads
  - Static schedule have low overhead but may exhibit high load imbalance
  - Dynamic schedule have higher overhead but reduced load imbalance

Scheduling Loops

- In both static and dynamic schedules, contiguous ranges of iterations called chunks are assigned to threads
- `#pragma omp parallel schedule(<type>[, <chunck>])`
  - Schedule type is compulsory (static or dynamic)
  - The chunk size is optional
- `schedule (static)`: A static allocation of about $\frac{n}{t}$ contiguous iterations to each thread
- `schedule (static, C)`: An interleaved allocation of chunks to tasks. Each chunk contains `C` contiguous iterations
- `schedule (dynamic)`: Iterations are dynamically allocated, one at a time, to threads
- `schedule (dynamic, C)`: Dynamic allocation of `C` iterations at a time to the tasks

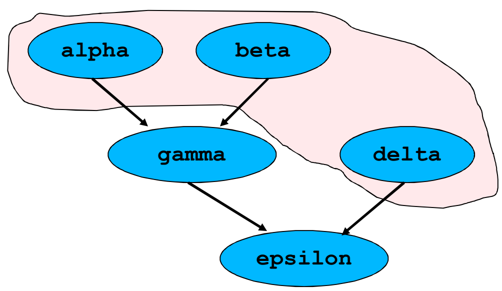
Scheduling Loops

- When the `schedule` clause is not included in `parallel for` pragma, default scheduling is static scheduling
- Increasing the chunk size increases the cache hit rate at the expense of increasing the load imbalance
- Reducing the chunk size can allow finer balancing of work loads
- The best value for the chunk size is system-dependent

Functional Parallelism

- OpenMP allows us to assign different threads to different portions of code

```
v = alpha();
w = beta();
x = gamma(v,w);
y = delta();
printf("%f\n", epsilon(x,y));
```



Functional Parallelism

- `parallel sections` Pragma:
  - It precedes a block of k block of codes that may be executed concurrently by k threads
  - `#pragma omp parallel sections`
  - `section` Pragma:
    - Each of the k blocks of codes are preceded by this pragma
    - The `section` pragma precedes each block of code with in the encompassing block preceded by the `parallel sections` pragma

```
#pragma omp parallel sections
{
 #pragma omp section
     v = alpha();
 #pragma omp section
     w = beta();
 #pragma omp section
     y = delta();
}
x = gamma(v,w);
printf("%f\n", epsilon(x,y));
```

- Suppose we have only 2 processors
  - Is this approach is efficient?

Functional Parallelism

- `parallel sections` Pragma:
  - It precedes a block of k block of codes that may be executed concurrently by k threads
  - `#pragma omp parallel sections`
  - `section` Pragma:

```
#pragma omp parallel sections
{
  #pragma omp section
    v = alpha();
  #pragma omp section
    w = beta();
}
#pragma omp parallel sections
{
  #pragma omp section
    y = delta();
  #pragma omp section
    x = gamma(v,w);
}
printf("%f\n", epsilon(x,y));
```

- Suppose we have only 2 processors
  - Efficient approach

Text Books

- M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.
- Barbara Chapman, Gabriele Jost and Ruud van der Pas, *Using OpenMP - Portable Shared Memory Parallel Programming*, MIT Press, 2007.
- Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan and Jeff McDonald, *Parallel programming in OpenMP*, Morgan Kaufmann Publishers, 2001.