

OpenMP

- Introduction
- OpenMP basics
- OpenMP directives, clauses, and library routines
- Readings
 - <http://openmp.org/>
 - <https://computing.llnl.gov/tutorials/openMP/>

Shared memory .vs. distributed memory

Parallel programming

- Parallel programming in general targets two types of systems: shared memory and distributed memory
 - Shared memory parallel programs use a global memory that is shared by all threads
 - Distributed memory parallel programs have multiple processes that do not share memory
 - Python multi-processing is a form of distributed memory parallel program
 - Distributed memory parallel program is inherently much harder to develop than shared memory parallel program
 - OpenMP is probably the simplest way to write shared memory parallel program. See pi.c and piomp.c
 - Python does not have direct OpenMP support. Some library is also not particularly as intuitive as OpenMP for C/C++. We will just talk about OpenMP for C/C++.

What is OpenMP?

- What does OpenMP stands for?
 - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.
- OpenMP is an API that may be used to explicitly direct ***multi-threaded, shared memory parallelism***.
 - API components: compiler directives, runtime library routines, environment variables
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

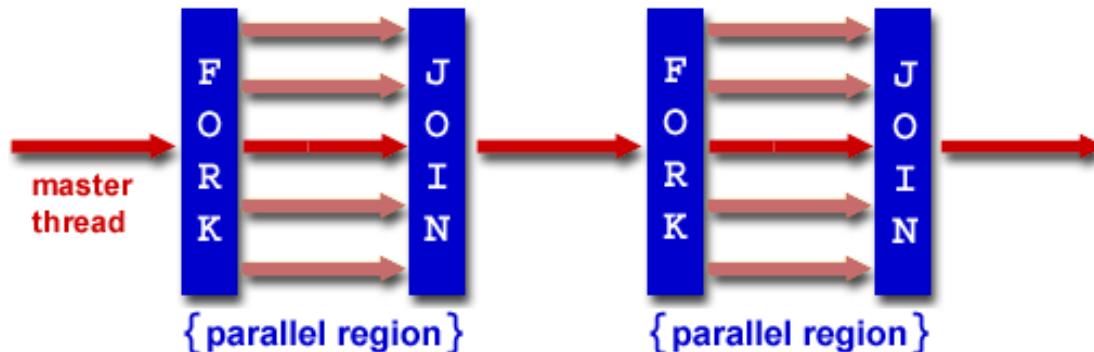
What is OpenMP?

- OpenMP API is specified for C/C++ and Fortran.
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for fortran and pragmas for C/C++.
- OpenMP can be implemented incrementally, one function or even one loop at a time.
 - Very nice way to get a parallel program from a sequential program.
- See mm.c and mm_omp.c
- OpenMP website: <http://www.openmp.org>
 - Materials in this lecture are taken from various OpenMP tutorials in the website and other places.

How to Compile and Run OpenMP Programs?

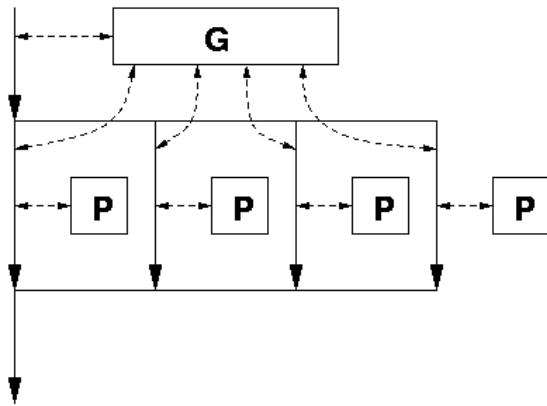
- Gcc 4.2 and above supports OpenMP 3.0
 - gcc –fopenmp a.c or g++ -fopenmp a.cpp
- To run: ‘a.out’
 - To change the number of threads:
 - setenv OMP_NUM_THREADS 4 (tcsh) or
 - export OMP_NUM_THREADS=4 (bash)

OpenMP Programming Model



- OpenMP uses the fork-join model of parallel execution.
 - All OpenMP programs begin with a single master thread.
 - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK). Master is part of the team with thread id 0.
 - When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

OpenMP Data Model



P = private data space
G = global data space

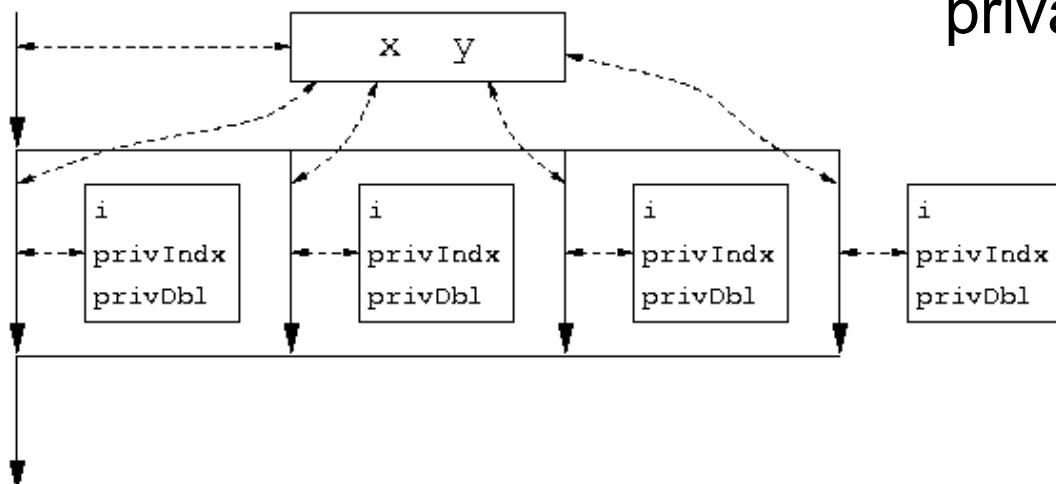
- Private and shared variables
 - Variables in the global data space are accessed by all parallel threads (**shared** variables).
 - Variables in a thread's private space can only be accessed by the thread (**private** variables)
 - several variations, depending on the initial values and whether the results are copied outside the region.

```

#pragma omp parallel for private( privIdx, privDbl )
for ( i = 0; i < arraySize; i++ ) {
    for ( privIdx = 0; privIdx < 16; privIdx++ ) {
        privDbl = ( (double) privIdx ) / 16;
        y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) ) + cos( privDbl );
    }
}

```

Parallel for loop index `i` is private by default.



execution context for "arrayUpdate_II"

OpenMP Terminology

- OpenMP Team := Master + Workers
- A parallel region is a block of code executed by all threads simultaneously
 - "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially
- A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work

Components of OpenMP

Compiler Directives

- Parallel regions
- Work sharing
- Synchronization
- Data scope attributes
 - Private
 - Firstprivate
 - Lastprivate
 - Shared
 - Reduction
- Orphaning

Environment Variables

- Number of thread
- Scheduling type
- Dynamic thread adjustment
- Nest parallelism

Runtime library routines

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Timers
- API for locking

OpenMP General Code Structure

```
#include <omp.h>
main () {
    int var1, var2, var3;
    Serial code
    ...
    /* Beginning of parallel section. Fork a team of threads. Specify variable scoping*/
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        /* Parallel section executed by all threads */
        ...
        /* All threads join master thread and disband*/
    }
    Resume serial code
    ...
}
```

- See example `hello_world_omp.c`

OpenMP Directives

- **Format:**

```
#pragma omp directive-name [clause,..] newline  
(use '\' for multiple lines)
```

- **Example:**

```
#pragma omp parallel default(shared) private(beta,pi)
```

- **Scope of a directive is a block of statements { ... }**

Parallel Region Construct

- A block of code that will be executed by multiple threads.

```
#pragma omp parallel [clause ...]  
{  
    ....  
}
```

(implied barrier)

Example clauses: if (expression), private (list), shared (list), default (shared | none), reduction (operator: list), firstprivate(list), lastprivate(list)

- if (expression): only in parallel if expression evaluates to true
- private(list): list of variables private to each thread
- shared(list): data accessed by all threads
- default (none|shared): default scope for all variables in parallel region

Shared Variables

- A shared variable exists in only one memory location and all threads can read or write to that address.
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections).

Parallel Region Construct (Cont'd)

- Reduction clause

```
sum = 0.0;  
#pragma parallel default(none) shared (n, x) private (l) reduction(+ : sum)  
{  
    for(l=0; l<n; l++) sum = sum + x(l);  
}
```

- Updating sum must avoid racing condition
- With the reduction clause, OpenMP generates code such that the race condition is avoided.
- See add1_omp.c and add2_omp.c
- See example3_omp.c and example3a_omp.c

Work-Sharing Constructs

- `#pragma omp for [clause ...]`
 - `#pragma omp sections [clause ...]`
 - `#pragma omp single [clause ...]`
-
- The work is distributed over the threads
 - Must be enclosed in parallel region
 - No new threads created
 - No implied barrier on entry, implied barrier on exit (unless specified otherwise)

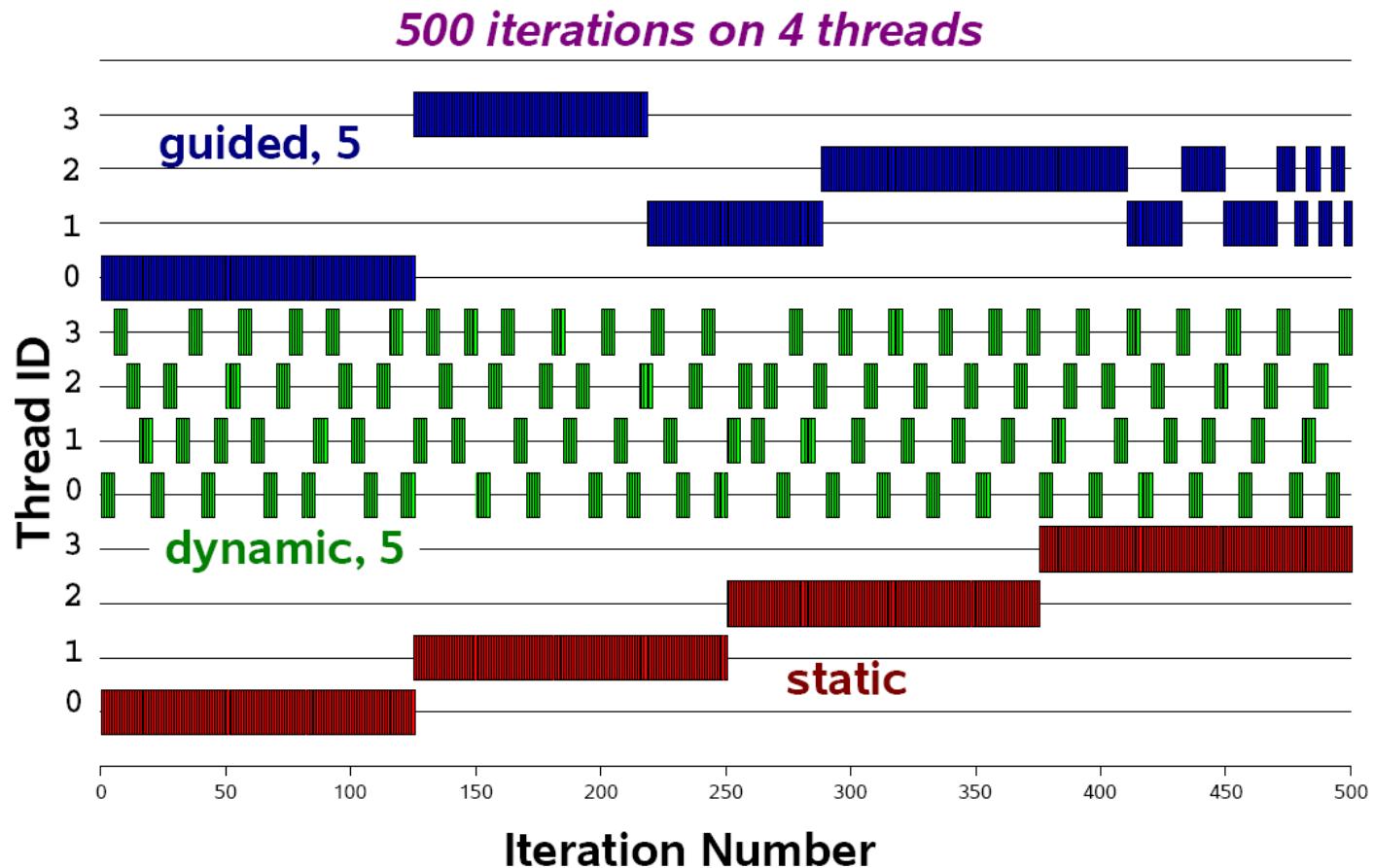
The omp for Directive: Example

```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
}

/*-- End of parallel region --*/
(implied barrier)
```

- Schedule clause (decide how the iterations are executed in parallel):
 - schedule (type [, chunk]). Types: static, dynamic, guided, runtime, auto



The omp sections Directive - Example

```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        {
            for (i=0; i<n-1; i++)
                b[i] = (a[i] + a[i+1])/2;
        }

        #pragma omp section
        {
            for (i=0; i<n; i++)
                d[i] = 1.0/c[i];
        }
    } /*-- End of sections --*/
}

/*-- End of parallel region --*/
```

Combined Parallel Work-Sharing Constructs

- Parallel region construct can be combined with work-sharing constructs
 - See add4_omp.c

```
#pragma omp parallel  
#pragma omp for  
    for (...)
```

Single PARALLEL loop



```
#pragma omp parallel for  
for (....)
```

```
#pragma omp parallel  
#pragma omp sections  
{ ... }
```

Single PARALLEL sections



```
#pragma omp parallel sections  
{ ... }
```

Synchronization: Critical Section

```
For(I=0; I<N; I++) {  
    .....  
    sum += A[I];  
    .....  
}
```

Cannot be parallelized if sum is shared.

Fix:

```
For(I=0; I<N; I++) {  
    .....  
    #pragma omp critical  
    {  
        sum += A[I];  
    }  
    .....  
}
```

- See add3.c

OpenMP Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_THREAD_LIMIT
- etc

OpenMP Runtime Environment

- `omp_get_num_threads()`
- `omp_get_thread_num()`
- `omp_in_parallel()`
- `omp_get_wtime()`
- Routines related to locks
-

Develop an OpenMP program

- Developing an OpenMP program:
 - Start from a sequential program
 - Identify the code segment that takes most of the time.
 - If one loop dominate the execution time, you just need to parallelize this loop. If multiple loops have the same complexity, you will need to parallelize all of the loops in order to get a good speedup
 - Determine whether the important loops can be parallelized
 - Loops with no dependence across iterations can be parallelized.
 - Some loops with dependence can be parallelized by using critical sections, reduction variables, etc
 - Determine the shared, private, reduction variables.
 - Add directives.
 - See for example [pi.c](#) and [piomp.c](#) program.

Examples

```
for (i=0; i<n; i++) {  
    x = b[i] + b[i+1]  
    a[i] = x + 10  
}
```

```
for (i=0; i<n; i++) {  
    x = a[i] + a[i+1]  
    a[i] = x + 10  
    c = c + x  
}
```

```
for (i=0; i<n; i++) {  
    x = a[i] + a[i+1]  
    a[i] = x + 10  
}
```

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        for (k=0; k<n; k++)  
            c[i][j] = c[i][j] + a[i][k]* b[k][j]  
    }
```

Summary

- OpenMP provides a compact, yet powerful programming model for shared memory programming
- OpenMP preserves the sequential version of the program