

An introduction to OpenMP (Open Multi-Processing)

Dr. Abhijit Chatterjee

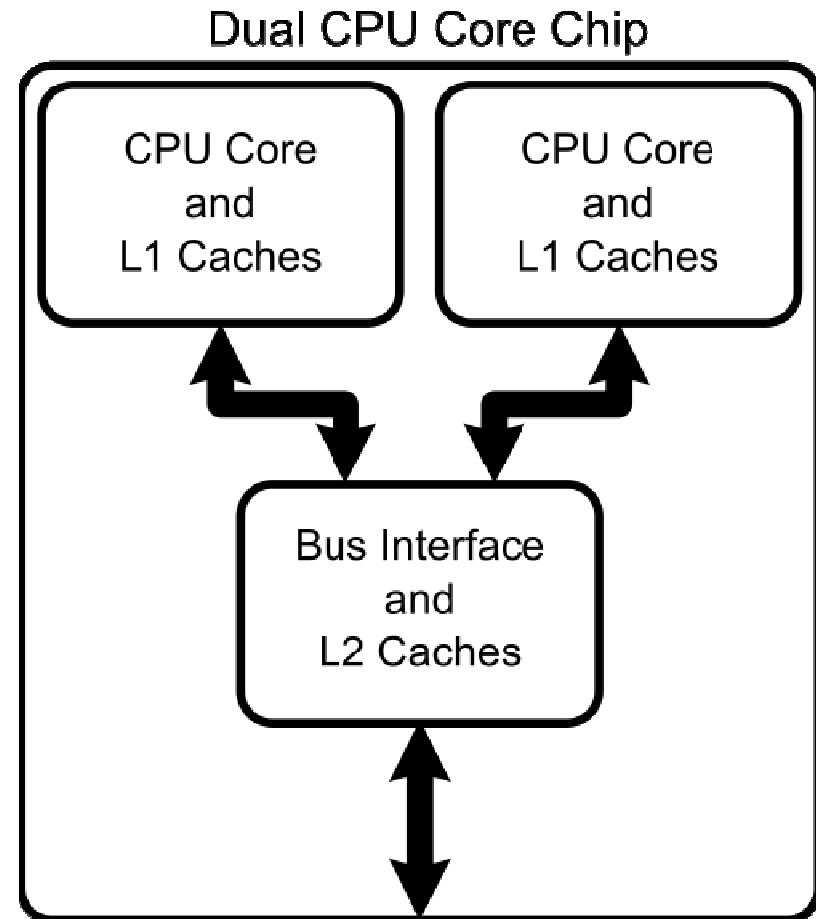
Department of Chemical Engineering

IIT Kanpur

November 12, 2010



Multicore processors



From http://en.wikipedia.org/wiki/Multi-core_processor

Abhijit Chatterjee, Department of Chemical Engineering, IIT Kanpur



What is OpenMP?



<http://openmp.org/wp/>

- Application program interface (API) for shared memory parallel applications in C, C++, Fortran
- OpenMP is not a language
- Compiler directives, runtime library, environment variables are available
- An excellent reference for beginners:

<https://computing.llnl.gov/tutorials/openMP>



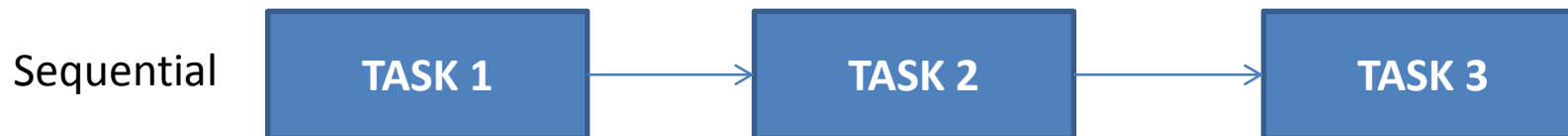
Advantages of Open MP

- Very easy to implement
- Lower communication time required in comparison to MPI
- Unlike MPI, preserves the sequential code
- Makes good use of present day multicore processors



What is a thread?

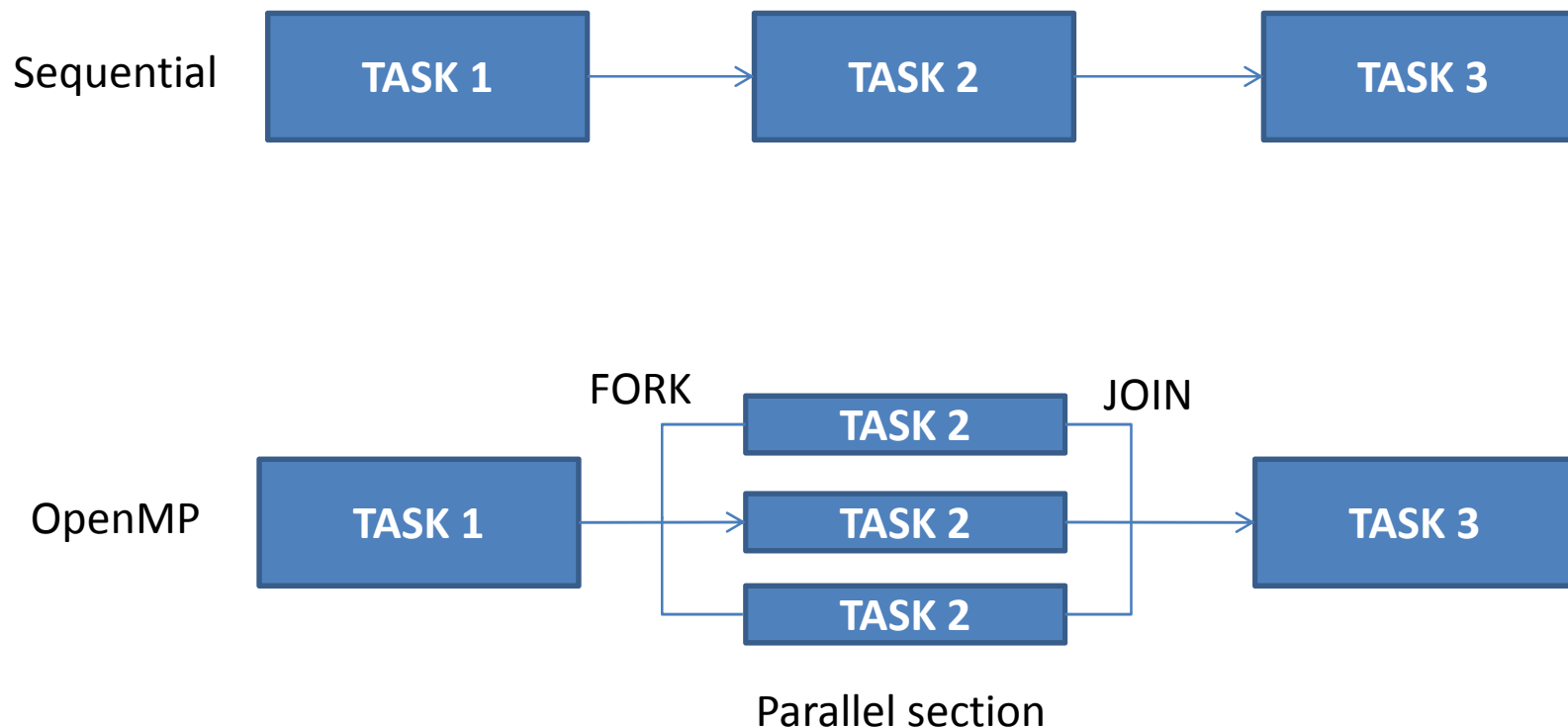
- A thread carries out a series of instructions



- OpenMP program executes threads in parallel
- There is a master thread in OpenMP with id=0
- New threads can be spawned dynamically and later released during the course of a computer calculation



Basic concept of OpenMP



When is parallelization possible?

- Certain degree of independence in the order of the operations

$$x(t + \Delta t) = 2x(t) - x(t - \Delta t) + F(t) / m$$

$$v(t) = \frac{x(t + \Delta t) - x(t - \Delta t)}{2\Delta t}$$

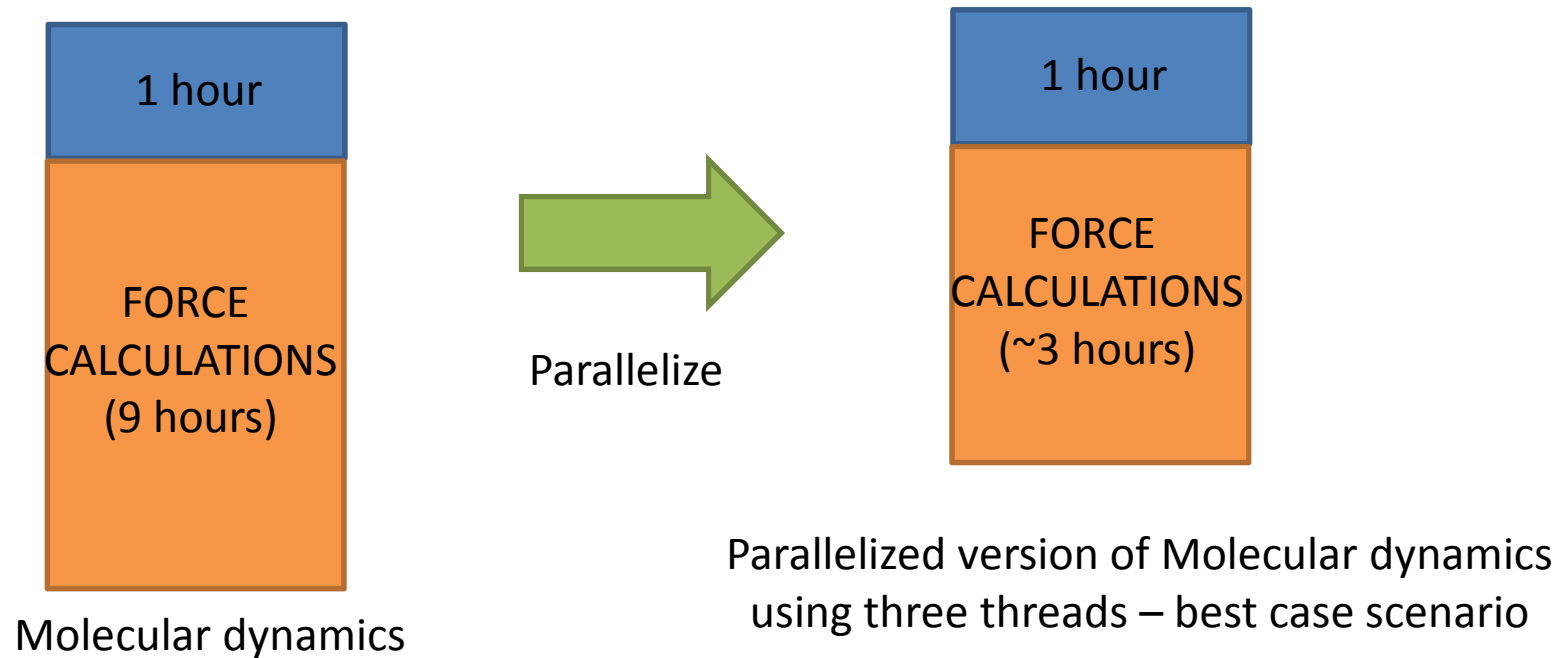
```
DO i=1,n  
  A(i)=B(i)+C(i)  
END DO
```

- It does not matter which order chunks of operations are performed

```
DO i=1,n  
  A(i)=B(i)+C(i)  
END DO  
DO i=1,n  
  D(i)=B(i)-C(i)  
END DO
```



Maximum speed-up



How effective is parallelization?



$$1.1 \text{ hour} + 6.0 \text{ hour} + 0.5 \text{ hour} + 3.6 \text{ hour} = 11.2 \text{ hours}$$

$$\frac{1.1 \text{ hour}}{1} + \frac{6.0 \text{ hour}}{5} + \frac{0.5 \text{ hour}}{20} + \frac{3.6 \text{ hour}}{3} = 3.53 \text{ hours}$$

$$\text{Speed-up} = 3.2 \text{ times}$$



How to fix the number of threads?

- Set the OMP_NUM_THREADS environment variable

```
bash>> OMP_NUM_THREADS = 4
```

- Use the omp_set_num_threads() library function

```
PROGRAM MAIN  
  
CALL OMP_SET_NUM_THREADS(4)  
  
END PROGRAM MAIN
```

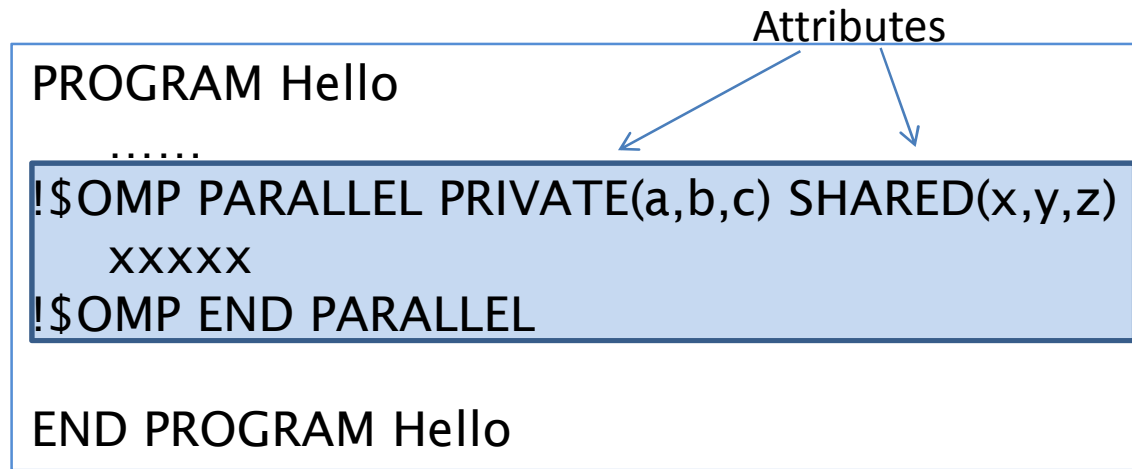


Main directives in OpenMP

- Control structures – serial vs parallel
- Worksharing constructs – who will get to do what
- Synchronization – collecting information from the threads
- Data scoping – variable scope



Typical structure of OpenMP code



```
gfortran -fopenmp -o Hello.x Hello.f90
```

OpenMP

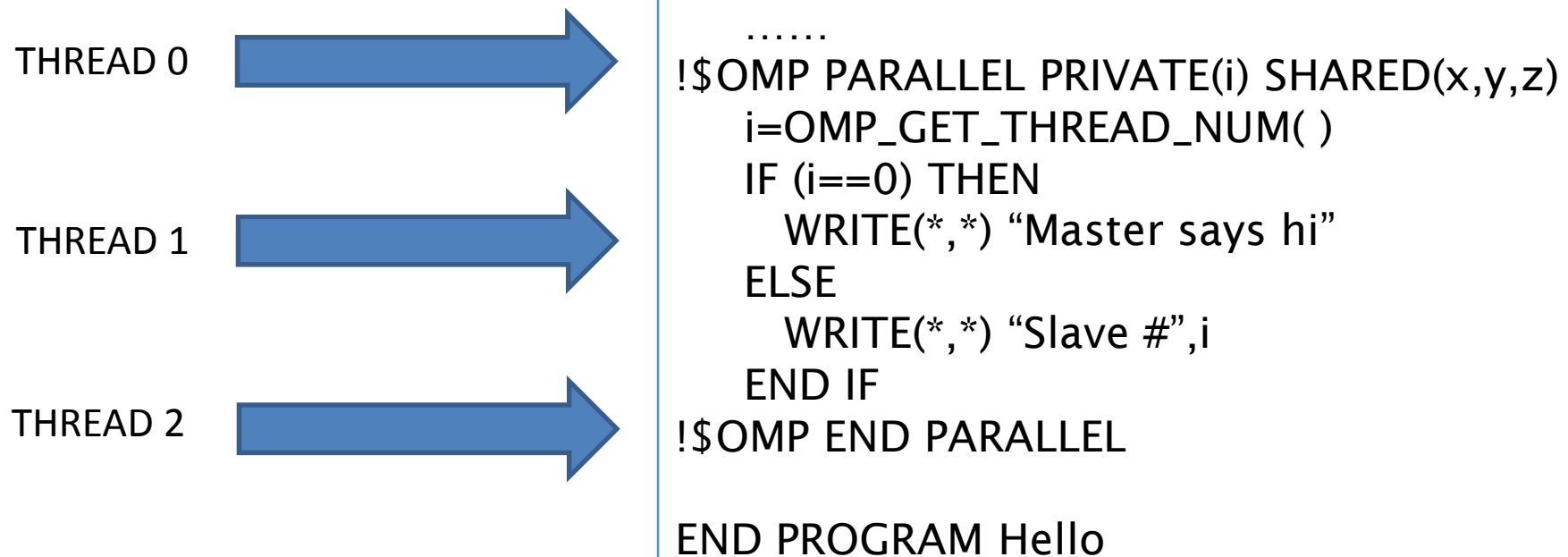
```
gfortran -o Hello.x Hello.f90
```

Sequential



Thread number

- Numbering of thread can be used to identify which thread is in operation



Work sharing constructs

- DO/FOR construct – for loops
- SECTION construct – divide work into separate sections
- SINGLE construct – serialize a section of the code



DO/FOR directive

- DO loop cannot be a DO WHILE loop, or a loop without a control

```
!$OMP DO [clause]
    SCHEDULE (type [,chunk])
    ORDERED
    PRIVATE(list)
    FIRSTPRIVATE (list)
    LASTPRIVATE (list)
    SHARED (list)
    REDUCTION (operator | intrinsic : list)
    COLLAPSE (n)
    do loop
!$OMP END DO [NOWAIT]
```



DO/FOR directive - example

```
PROGRAM VEC_ADD_DO
  INTEGER, PARAMETER :: N=1000,CHUNKSIZE=100
  INTEGER :: CHUNK,I
  REAL :: A(N),B(N),C(N)

  DO I=1,N
    A(I)=REAL(I)
    B(I)=A(I)
  END DO

  !$OMP PARALLEL SHARED (A,B,C,CHUNK) PRIVATE(I)
  !$OMP DO SCHEDULE(DYNAMIC,CHUNK)
  DO I=1,N
    C(I)=A(I)+B(I)
  END DO
  !$OMP END DO
  !$OMP END PARALLEL
END PROGRAM VEC_ADD_DO
```



DO/FOR directive

- Schedule
 - STATIC – loops are divided into pieces of size CHUNK; if CHUNK size is not specified then iterations are evenly divided
 - DYNAMIC – dynamic allocation of size CHUNK
 - RUNTIME – determined at runtime
 - AUTO – Decision is taken by the compiler



SECTIONS directive

- Used whenever iterations are NOT present, but work can be distributed among threads

```
!$OMP SECTIONS [clause]
    PRIVATE(list)
    FIRSTPRIVATE (list)
    LASTPRIVATE (list)
    REDUCTION (operator | intrinsic : list)
!$OMP SECTION
    block
!$OMP SECTION
    block
!$OMP END SECTIONS [NOWAIT]
```



SECTIONS directive - example

```
PROGRAM VEC_ADD_SECTIONS
  INTEGER, PARAMETER :: N=1000
  INTEGER :: I
  REAL :: A(N),B(N),C(N),D(N)

  DO I=1,N
    A(I)=REAL(I)
    B(I)=A(I)
  END DO

  !$OMP PARALLEL SHARED (A,B,C,D) PRIVATE(I)
  !$OMP SECTIONS
  !$OMP SECTION
  DO I=1,N
    C(I)=A(I)+B(I)
  END DO
  DO I=1,N
    D(I)=A(I)*B(I)
  END DO
  !$OMP END SECTIONS
  !$OMP END PARALLEL
END PROGRAM VEC_ADD_SECTIONS
```



WORKSHARE directive

- WORKSHARE divides the execution of a structured block into separate units of work

```
!$OMP WORKSHARE  
  block  
!$OMP END WORKSHARE [NOWAIT]
```

- The structure block must only contain
 - Array assignments, scalar assignments, FORALL statements, WHERE statements



WORKSHARE directive - example

```
PROGRAM VEC_ADD_WORKSHARE
  INTEGER, PARAMETER :: N=1000
  REAL :: A(N),B(N),C(N),D(N)

  DO I=1,N
    A(I)=REAL(I)
    B(I)=A(I)
  END DO

  !$OMP PARALLEL SHARED (A,B,C,D)
  !$OMP WORKSHARE
    C=A+B
    D=A*B
  !$OMP END WORKSHARE NOWAIT
  !$OMP END PARALLEL
END PROGRAM VEC_ADD_WORKSHARE
```



SINGLE directive

- Used while dealing with sections that are not thread safe

```
!$OMP SINGLE [clause]
        PRIVATE(list)
        FIRSTPRIVATE (list)
    block
!$OMP END SINGLE [NOWAIT]
```

- Threads that do not execute the enclosed code, wait unless a NOWAIT clause is specified



Data scope and Data sharing

- Most variables are shared by default, since memory is shared
- Variables can be shared (e.g., variables common to different threads, module variables, common block etc.) or they can be private (e.g., do loop, variables used independently by threads, etc.)



Data scope and Data sharing

- THREADPRIVATE directive – data to be shared between parallel regions
- Other attributes used
 - PRIVATE(x) – new object x for each thread
 - SHARED(x) – same object x is shared between threads
 - DEFAULT – define default scope, e.g., DEFAULT PRIVATE
 - FIRSTPRIVATE(x) – PRIVATE + Initialized
 - LASTPRIVATE(x) – PRIVATE + use last value
 - REDUCTION(x) – perform reduction on x
 - COPYIN



THREADPRIVATE directive

- Each thread gets its copy of the variable, which be used by it during execution in multiple parallel regions

```
COMMON /Block1/ a

!$OMP THREADPRIVATE(/Block1/,x)

!$OMP PARALLEL
!$OMP END PARALLEL

!$OMP PARALLEL
!$OMP END PARALLEL
```



PRIVATE(list)

- List of variables is private to each thread
- VERY IMPORTANT: Each of these variables is uninitialized in the beginning
- Can be initialized using FIRSTPRIVATE
- Does not persist over multiple parallel regions unlike THREADPRIVATE
- Can be used for variables other than those from a common block



SHARED(list)

- List of variables is shared between threads
- VERY IMPORTANT: Shared variable exists on only one memory location, the thread would read from and write to the address
- Use BARRIER, CRITICAL constructs to ensure synchronization



FIRSTPRIVATE and LASTPRIVATE

- FIRSTPRIVATE(list) Combines the PRIVATE clause with initialization for a list of variables
- LASTPRIVATE(list) copies the value of the list of PRIVATE variables to the original variable object
- The value of the private variable from the thread, which performs the last DO loop iteration or the last section, gets copied




REDUCTION

```
PROGRAM DOTPRODUCT
  INTEGER, PARAMETER:: N=1000, CHUNKSIZE=10
  INTEGER :: CHUNK,I
  REAL :: A(N),B(N),RESULT

  DO i=1,N
    A(I)=REAL(I)
    B(I)=SIN(A(I))
  END DO
  RESULT=0.
  CHUNK=CHUNKSIZE
  !$OMP PARALLEL
  !$OMP DO DEFAULT(SHARED) PRIVATE(I) SCHEDULE(STATIC,CHUNK)
  !$OMP& REDUCTION(+:RESULT)
  DO I=1,N
    RESULT=RESULT+A(I)*B(I)
  END DO
  !$OMP END DO
  !$OMP END PARALLEL
END PROGRAM DOTPRODUCT
```

Equal sized blocks



REDUCTION(operator | list)

- Perform a reduction on the variables in the list
- Private copy of the variable in the list are created for each thread
- Finally result is written to global shared variable
- Variables on which reduction is performed should be scalar and shared



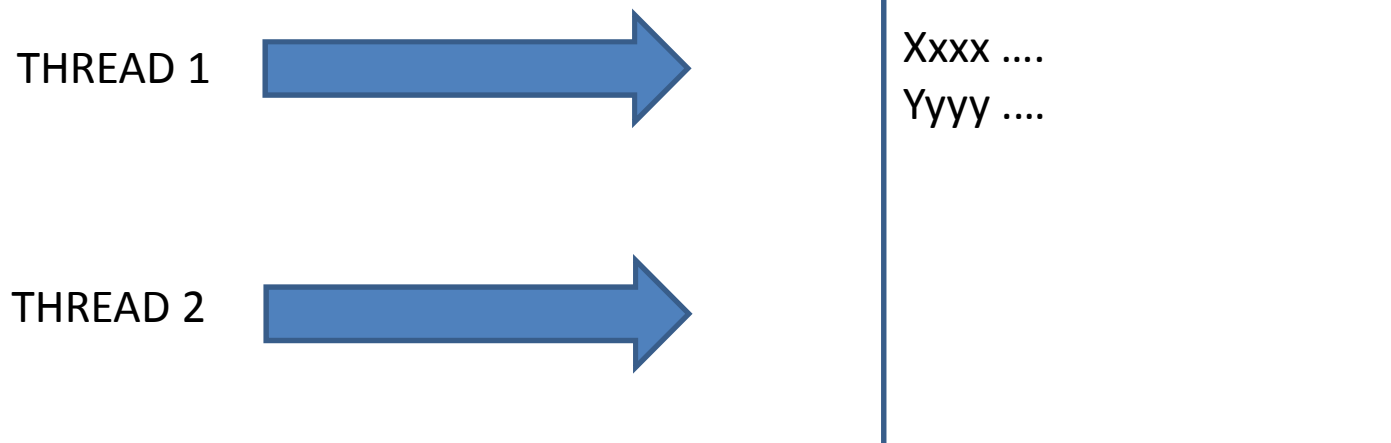
COPYIN

- Obtains the value of a variable x declared under THREADPRIVATE
- The value of x for all threads will be assigned the same value
- Master thread is used as copy source



Race condition

- When two threads compete with each other to influence the output
- Example 1 – writing output to a file



Race condition

- When two threads compete with each other to influence the output
- Example 2 – Data corruption
 - Read x_sum
 - Update according to $X_{sum} = X_{sum} + x(i)$
 - Return value of x_sum



Synchronization constructs

- MASTER directive
- CRITICAL directive
- BARRIER directive
- TASKWAIT directive
- ATOMIC directive
- FLUSH directive
- ORDERED directive



MASTER directive

- Only master thread can execute this region
- All other threads can skip this region

```
!$OMP MASTER  
  block  
!$OMP END MASTER
```



CRITICAL directive

- The region can be executed by only one thread at a time

```
INTEGER :: x_sum
x_sum=0
!$OMP PARALLEL SHARED(x)
!$OMP CRITICAL
  x_sum=x_sum+1
!$OMP END CRITICAL
!$OMP END PARALLEL
```



BARRIER directive

- This can synchronize all threads in the team
- When a BARRIER is reached, the thread will wait till all threads reach the barrier

```
!$OMP BARRIER  
  block  
!$OMP END BARRIER
```



ATOMIC directive

- Specific memory location must be updated atomically, rather than letting multiple threads to write

!\$OMP ATOMIC
statement



FLUSH directive

- The thread-visible variables need to be written back to memory

```
!$OMP FLUSH(list)
```



Runtime routines

- OMP_SET_NUM_THREADS()
- OMP_GET_NUM_THREADS()
- OMP_GET_MAX_THREADS()
- OMP_GET_THREAD_NUM()
- OMP_SET_DYNAMIC()
- OMP_GET_DYNAMIC()
- OMP_SET_NESTED()
- OMP_GET_WTIME()
- Other routines ...



OMP_GET_WTIME()

- Obtain the OpenMP walk clock time
- Returns the number of seconds elapsed (per thread), as a double precision floating point, since some time in the past

```
Time0=OMP_GET_WTIME( )  
$OMP DO SCHEDULE (DYNAMIC,CHUNK  
  xxxxx  
$OMP END DO  
Time1=OMP_GET_WTIME( )
```



Some important issues in OpenMP

- Total CPU time over all threads created can exceed the sequential time
- Some other issues that are important
 - Communication
 - Race conditions and synchronization
 - Load balancing
 - Scalability
 - Portability
 - Problem size (memory related issues)

