Introduction to OpenMP

Dr. Axel Kohlmeyer

Research Professor, Department of Chemistry
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

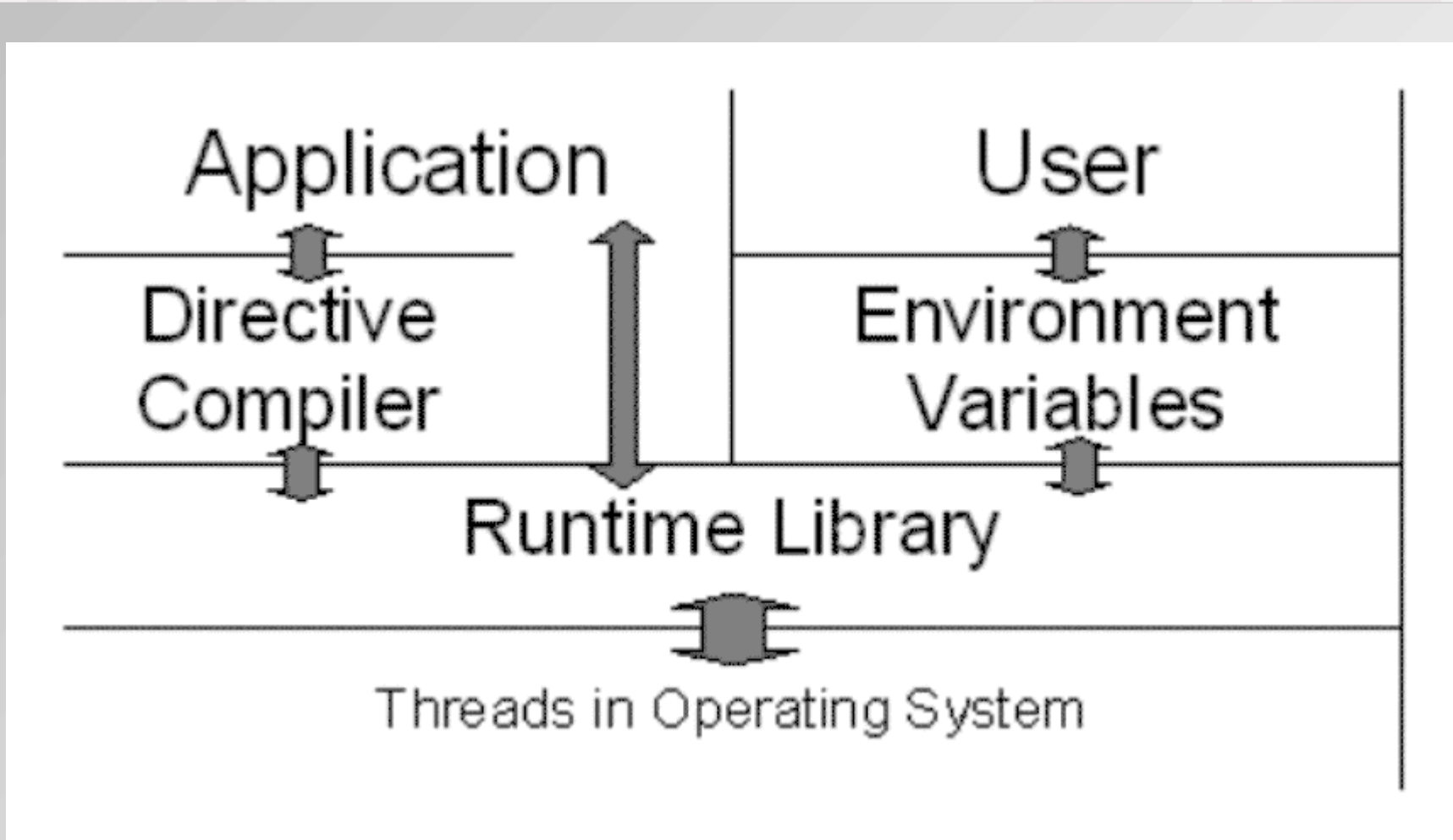
axel.kohlmeyer@temple.edu

TEMPLE UNIVERSITY®

OpenMP Overview

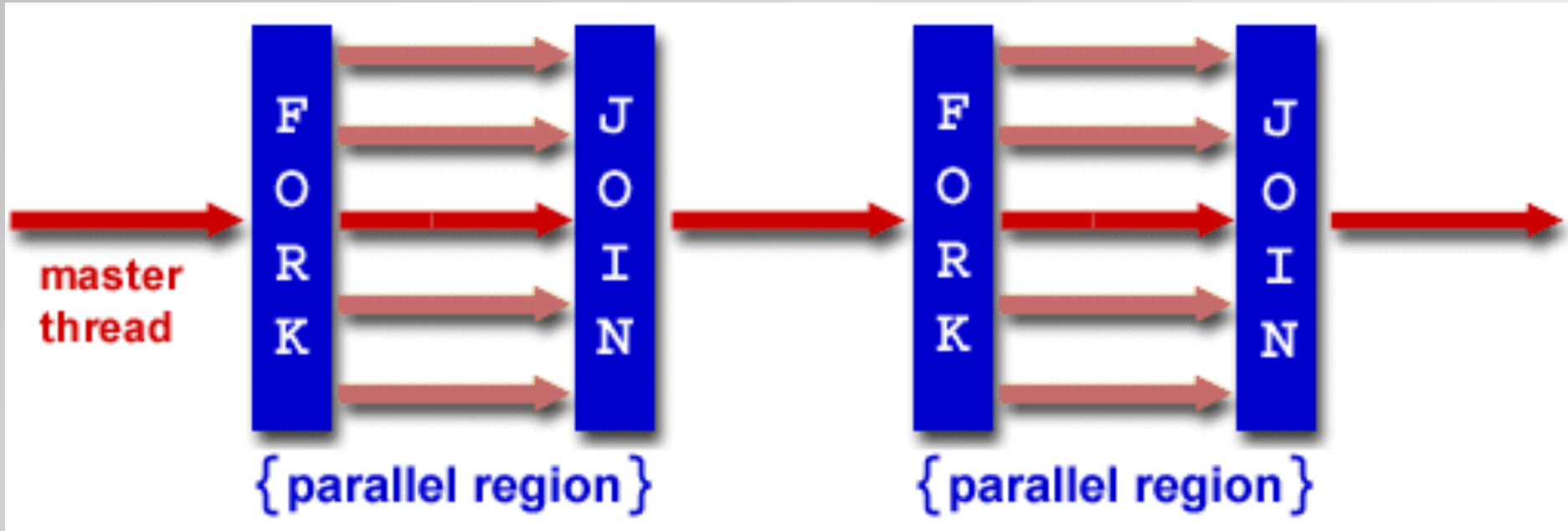
- Fine grained (loop) parallelism
- For shared memory SMP machines
- Directive based parallelization:
  Code should compile unaltered in serial mode
- Fortran 77/95 and C/C++ interface
- Incrementally parallelize a serial program
- Independent from and orthogonal to MPI
- http://www.openmp.org

OpenMP Architecture



OpenMP Execution Model

Fork-Join model on thread based machines



Most OpenMP implementations now use a thread-pool architecture to reduce overhead

Directives Example: Fortran

```
      PROGRAM HELLO
      INTEGER VAR1, VAR2, VAR3
  Serial code
      ...
!$OMP PARALLEL PRIVATE(VAR1,VAR2) SHARED(VAR3)
  Section executed in parallel by multiple threads
      ...
!$OMP END PARALLEL
  Resume serial code
      END
```

Directives Example: C/C++

```
#include <omp.h> /* for calling API functions
*/
int main(int argc, char **argv) {
    int var1, var2, var3;
Serial code
   ...
#pragma omp parallel private(var1) shared(var2)
  {  /* note: var3 is shared by default, too */
 Section executed in parallel by multiple threads
   ...
  }
```

Parallel Region

```
      PROGRAM HELLO
      INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
     +   OMP_GET_THREAD_NUM
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID
      IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF
!$OMP END PARALLEL
      END
```

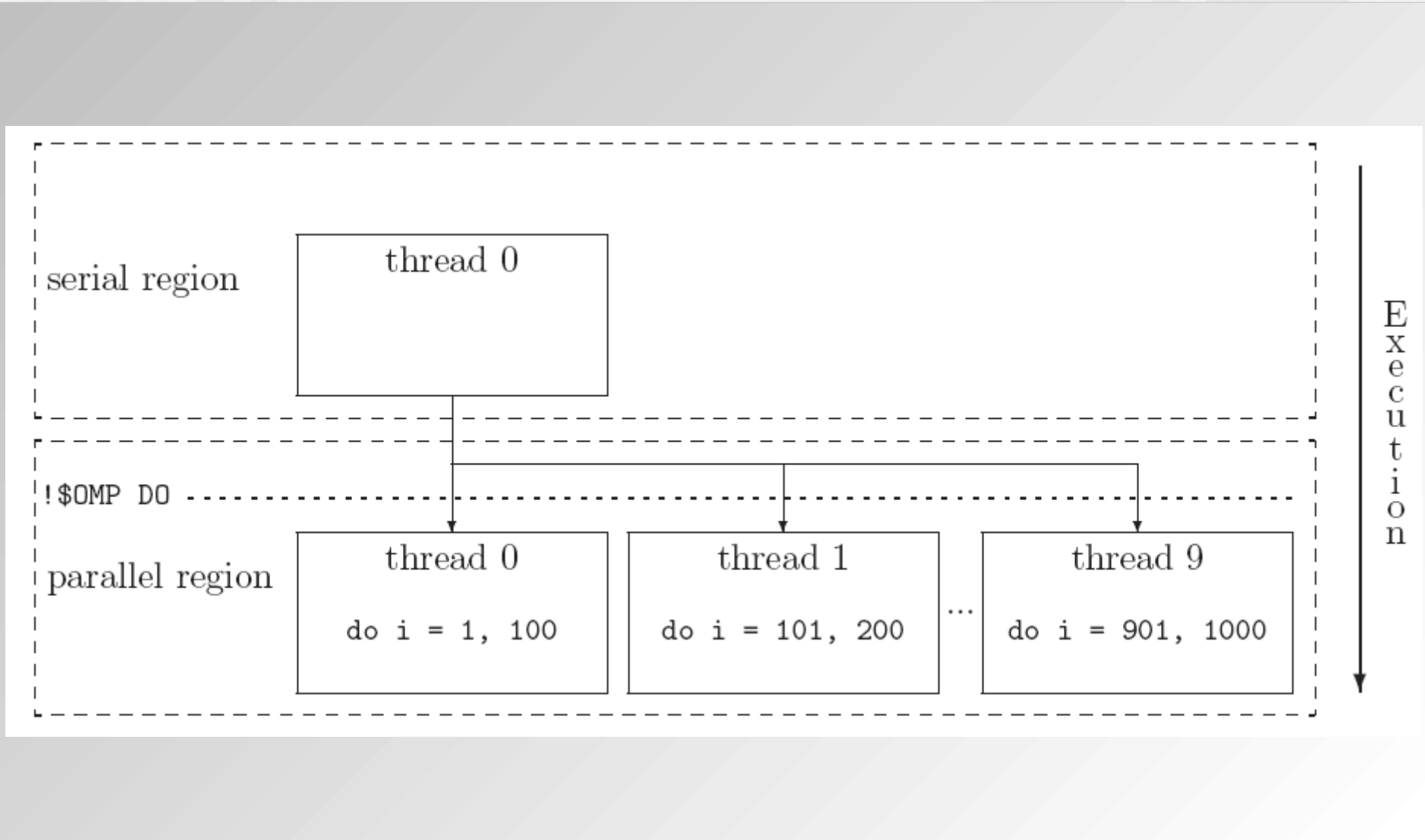
Loop Parallelization

```
      PROGRAM VEC_ADD_DO
      INTEGER I
      REAL*8 A(1000), B(1000), C(1000)
      DO I = 1, 1000
        A(I) = I * 1.0d0
        B(I) = A(I)*2
      ENDDO
!$OMP PARALLEL DO SHARED(A,B,C) PRIVATE(I)
      DO I = 1, 1000
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END PARALLEL
      END
```

Remove data dependency between threads.
Each thread will have its own copy of “I”.

Outside of the parallel region
the value of “I” is undefined.
“I” is 'thread-local'.

Loop Parallelization, cont'd



Reduction Operation

```
      PROGRAM VEC_ADD_DO
      INTEGER I
      REAL*8 A(1000), B
      DO I = 1, 1000
        A(I) = I * 1.0d0
      ENDDO
!$OMP PARALLEL DO SHARED(A) PRIVATE(I) REDUCTION(+:B)
      DO I = 1, 1000
         B = B + A(I)
      ENDDO
!$OMP END PARALLEL
      END
```

Each thread will do part of the sum and the result from the threads will be combined into one final sum.

Due to changing the order of the summation of floating point numbers, the total sum can vary when changing the number or scheduling of threads.

Non-Parallelizable Operation

```
      PROGRAM VEC_ADD_DO
      INTEGER I
      REAL*8 A(1000),B(1000),C(1000)
      ...
!$OMP PARALLEL DO SHARED(A,B) PRIVATE(I)
      DO I = 2, 999
         C(I) = 0.25d0*(A(I-1)+A(I+1))-0.5d0*A(I)
         B(I) = 0.25d0*(C(I-1)+C(I))+0.5d0*A(I)
      ENDDO
!$OMP END PARALLEL
      END
```

A step of the iteration depends of the result of a previous step, but with threading, we cannot know if that result is already available.

Race Condition

```
#if defined(_OPENMP)
#pragma omp parallel for default(shared) schedule(static)\
            private(i,j) reduction(+:epot)
#endif
    for(i=0; i < natoms-1; ++i) {
        for(j=i+1; j < natoms; ++j) {
            d=r[j] - r[j];
            d=d*d;
            if (d < rcutsq) {
                r2 = 1.0/d;
                r6=r2*r2*r2;
                ffac = (12.0*c12*r6 - 6.0*c6)*r6*r2;
                epot += r6*(c12*r6 - c6);
                f[i] += ffac;
                f[j] -= ffac;
            }
        }
    }
```

The “i” loop index will be distributed across multiple threads, so the “j” on some thread may be the same number as “j” or “i” on some other thread.

Avoiding Race Conditions (1)

```
#pragma omp parallel for default(shared) schedule(static)\
            private(i,j) reduction(+:epot)
    for(i=0; i < natoms-1; ++i) {
        for(j=i+1; j < natoms; ++j) {
            d=r[j] - r[j];
            d=d*d;
            if (d < rcutsq) {
                r2 = 1.0/d;
                r6=r2*r2*r2;
                ffac = (12.0*c12*r6 - 6.0*c6)*r6*r2;
                epot += r6*(c12*r6 - c6);
#pragma omp critical
                {
                  f[i] += ffac;
                  f[j] -= ffac;
                }
            }
        }
    }
```

The critical directive will guarantee, that only one thread at a time, will execute this part of the code.
Problems: not parallel and overhead to aqcuire and release lock
=> slow

Avoiding Race Conditions (2)

```
#pragma omp parallel for default(shared) schedule(static)\
            private(i,j) reduction(+:epot)
    for(i=0; i < natoms-1; ++i) {
        for(j=i+1; j < natoms; ++j) {
            d=r[j] - r[j];
            d=d*d;
            if (d < rcutsq) {
                r2 = 1.0/d;
                r6=r2*r2*r2;
                ffac = (12.0*c12*r6 - 6.0*c6)*r6*r2;
                epot += r6*(c12*r6 - c6);
#pragma omp atomic
                f[i] += ffac;
#pragma omp atomic
                f[j] -= ffac;
            }
        }
    }
```

The “atomic” directive will protect a single memory location. Much less Overhead than “critical”, but still Slower than unprotected execution

Avoiding Race Conditions (3)

```
#pragma omp parallel for default(shared) schedule(static)\
            private(i,j) reduction(+:epot)
    for(i=0; i < natoms; ++i) {
        for(j=0; j < natoms; ++j) {
            if (i == j) continue
            d=r[j] - r[j];
            d=d*d;
            if (d < rcutsq) {
                r2 = 1.0/d;
                r6=r2*r2*r2;
                ffac = (12.0*c12*r6 - 6.0*c6)*r6*r2;
                Epot += 0.5*r6*(c12*r6 - c6);
                f[i] += ffac;
            }
        }
    }
```

By looping over all pairs of atoms twice we avoid the the race condition by only adding the computed force to one atom indexed by “i”. But now we have to compute the force twice.

False Sharing

- Not strictly a bug, code will execute correctly
- **But...** can have a large performance impact on cache-coherent memory architecture because:
  - Data is cached in “lines” (aligned blocks, 64 bytes)
  - When one CPU core modifies cached data and the same cache line is cached elsewhere, both need to be flushed (committed to RAM and read back)
  - This scenario can happen, e.g. when collecting per-thread data in an array index by thread id or when threads operate on different elements of a struct

How To Activate OpenMP

- Compile with special flags:
  - GNU: -fopenmp
  - Intel: -qopenmp

    => implies -D_OPENMP to be set as well
- Set number of threads:
  - Implementation default (use all CPU cores)
  - Environment: $OMP_NUM_THREADS
  - Function: omp_set_num_threads()
- For optimal performance, use with threaded, and re-entrant BLAS/LAPACK library (MKL, GotoBLAS)

OpenMP vs. MPI

- OpenMP does not require code layout change in principle, ... but it may help a lot
- OpenMP requires shared memory
- Fine grained parallelism inefficient in MPI
- There is overhead associated with creating and deleting or waking up threads
- MPI + OpenMP = 2-level parallelization most efficient on cluster of SMP nodes
- No MPI calls within OpenMP block