Goals of OpenMP

Simple support for shared memory

Hide much of thread syntax

Preserve most of thread functionality

- Create/exit, sync

Support standard high level functions (reduction)

Support automatic parallelization

Machine/run independence

- Static and dynamic thread control

Mechanism:

- Pragmas – compiler directives (#pragma omp parallel)
- Functions – OpenMP library (Reduce, get_thread_num())
- Environment variables (setenv OMP_NUM_THREADS 8)

Functions that must be supported

Thread operation (with minimal thread syntax)

- Create threads
- Initialize threads
- Exit threads
- Kill threads
- Local variables
- Thread identity

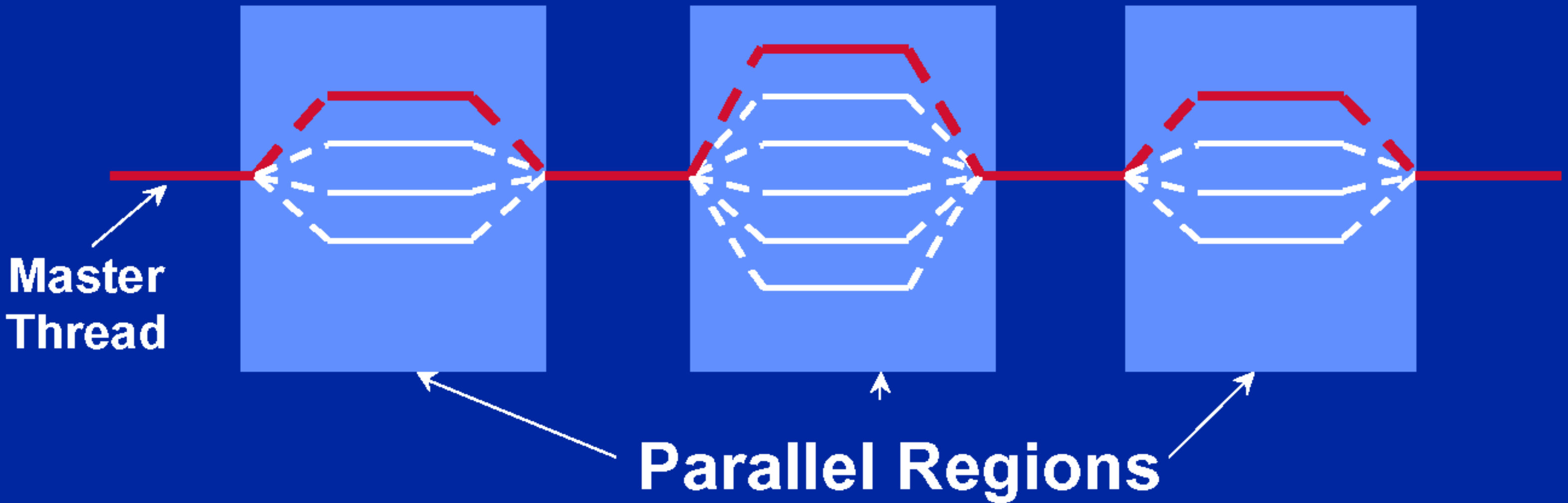
Parallel Operation

- Thread constructs
- Parallel sections
- Synchronization
- Critical sections
- Reductions

OpenMP: Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.



OpenMP:
How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop between multiple threads

```
void main()
{
    double Res[1000];

    for(int i=0;i<1000;i++) {
            do_huge_comp(Res[i]);
    }
}
```

Sequential Program

```
void main()
{
    double Res[1000];
#pragma omp parallel for
    for(int i=0;i<1000;i++) {
            do_huge_comp(Res[i]);
    }
}
```

Parallel Program

OpenMP Rules and Conventions

Rules of OpenMP directives for f77, f90 and C/C++:

Fortran 77:
c$omp parallel default(none)
c$omp&private(i,j) shared(a)

Fortran 90:
!$omp parallel default(none) &
!$omp private(I,j) shared(a,b)

C/C++:
#pragma omp parallel(none) \
   private(i,j) shared(a)

construct: parallel
clauses: default, private, shared

```
#include <stdio.h>
#define n 10
#define m 20

void main()
{
  int  i, j, a[n][m];
#pragma omp parallel default(none) \
    private(i,j) shared(a)
#pragma omp for
          for (j=0; j<m; j++) {
             for (i=0; i<n; i++) {
               a[i][j] = i+(j-1)*n;
             }
          }
}
```

OpenMP:
Structured blocks

- Most OpenMP constructs apply to structured blocks.
  - Structured block: a block of code with one point of entry at the top and one point of exit at the bottom. The only other branches allowed are STOP statements in Fortran and exit() in C/C++.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
      print *,id
```

A structured block

```
C$OMP  PARALLEL
10    wrk(id) = garbage(id)
30    res(id)=wrk(id)**2
      if(conv(res(id))goto 20
      go to 10
C$OMP END PARALLEL
      if(not_DONE) goto 30
20    print *, id
```

Not A structured block

Outline

1. Ways to do multithreading:
    1. Fork off n threads, ending in a barrier
    2. Parallelize FOR loops
    3. Parallel Sections   (parbegin … parend)
    4. Reduction
2. Data – shared and global
3. Synchronization
    1. Critical
    2. Atomic
    3. Barrier
    4. Ordered, Master, Single
    5. Flush
4. Runtime Functions, Environment Variables
    1. Locks
    2. Thread management

OpenMP: Parallel Regions

- You create threads in OpenMP with the "omp parallel" pragma.
- For example, To create a 4 thread Parallel region:

Each thread redundantly executes the code within the structured block

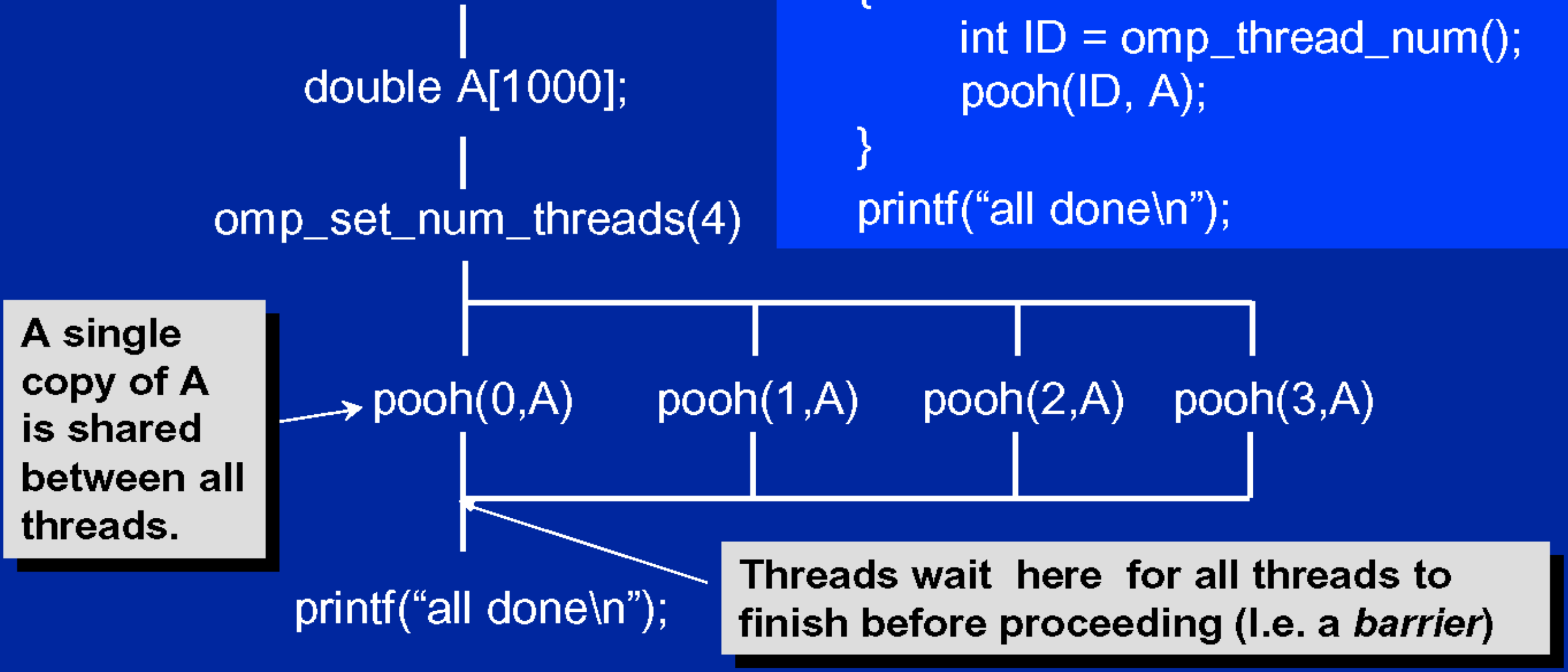
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_thread_num();
        pooh(ID,A);
}
```

- Each thread calls pooh(ID) for `ID` = `0` to `3`

OpenMP: Parallel Regions

- **Each thread executes the same code redundantly.**

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_thread_num();
        pooh(ID, A);
}
printf("all done\n");
```


double A[1000];
omp_set_num_threads(4)
pooh(0,A)
pooh(1,A)
pooh(2,A)
pooh(3,A)
printf("all done\n");
A single copy of A is shared between all threads.
Threads wait here for all threads to finish before proceeding (I.e. a *barrier*)

Exercise 1:
A multi-threaded "Hello world" program

- Write a multithreaded program where each thread prints a simple message (such as "hello world").
- Use two separate printf statements and include the thread ID:

```
int ID = omp_get_thread_num();

printf(" hello(%d) ", ID);

printf(" world(%d) ", ID);
```

- What do the results tell you about I/O with multiple threads?

OpenMP: Some subtle details (don't worry about these at first)

- Dynamic mode (the default mode):
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.
- Static mode:
  - The number of threads is fixed and controlled by the programmer.
- OpenMP lets you nest parallel regions, but…
  - A compiler can choose to *serialize* the nested parallel region (i.e. use a team with only one thread).

OpenMP: Work-Sharing Constructs

- The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.

Work Sharing Constructs

A motivating example

Sequential code

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
            int id, i, Nthrds, istart, iend;
            id = omp_get_thread_num();
            Nthrds = omp_get_num_threads();
            istart = id * N / Nthrds;
            iend = (id+1) * N / Nthrds;
            for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

```
#pragma omp parallel
#pragma omp for schedule(static)
            for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP For constuct:
The schedule clause

- **The schedule clause effects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

OpenMP: Work-Sharing Constructs

- The Sections work-sharing construct gives a different structured block to each thread.

```
#pragma omp parallel
#pragma omp sections
{
        X_calculation();
#pragma omp section
        y_calculation();
#pragma omp section
        z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

OpenMP: Combined Parallel Work-Sharing Constructs

- A short hand notation that combines the Parallel and work-sharing construct.

```
#pragma omp parallel for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

- There's also a "parallel sections" construct.

Data Environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

Private Clause Example

```
#pragma  omp parallel for private(j)
  for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
    for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j], a[i][j] + tmp[j]);


// private copies of j are only accessible inside for loop
```

OpenMP: Reduction

- Another clause that effects the way variables are shared:
  - reduction (op : list)
- The variables in "list" must be shared in the enclosing parallel region.
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+")
  - pair wise "op" is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.

OpenMP:
Reduction example

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
        int i;
        double ZZ, func(), res=0.0;
        omp_set_num_threads(NUM_THREADS)
#pragma omp parallel for reduction(+:res) private(ZZ)
        for (i=0; i< 1000; i++){
              ZZ = func(I);
              res = res + ZZ;
        }
}
```

OpenMP: Synchronization

- OpenMP has the following constructs to support synchronization:
  - atomic
  - critical section
  - barrier
  - flush
  - ordered
  - single
  - master

We discuss this here, but it really isn't a synchronization construct. It's a work-sharing construct that includes synchronization.

We discus this here, but it really isn't a synchronization construct.

OpenMP: Synchronization

- Only one thread at a time can enter a **critical** section.

```
C$OMP PARALLEL DO PRIVATE(B)
C$OMP& SHARED(RES)
      DO 100 I=1,NITERS
            B =  DOIT(I)
C$OMP CRITICAL
            CALL CONSUME (B, RES)
C$OMP END CRITICAL
100   CONTINUE
```

OpenMP: Synchronization

- Atomic is a special case of a critical section that can be used for certain simple statements.
- It applies only to the update of a memory location (the update of X in the following example)

```
C$OMP PARALLEL PRIVATE(B)
      B =  DOIT(I)
C$OMP ATOMIC
      X = X + B

C$OMP END PARALLEL
```

OpenMP: Synchronization

- Barrier: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
        id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc3(id);
}
```

implicit barrier at the end of a for work-sharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

OpenMP: Synchronization

- The **ordered** construct enforces the sequential order for a block.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res = consum(tmp);
        }
```

OpenMP: Synchronization

- The master construct denotes a structured block that is only executed by the master thread. The other threads just skip it (no implied barriers or flushes).

```
#pragma omp parallel private (tmp)
{
          do_many_things();
#pragma omp master
          {     exchange_boundaries();   }
#pragma barrier
          do_many_other_things();
}
```

OpenMP: Synchronization

- The single construct denotes a block of code that is executed by only one thread.
- A barrier and a flush are implied at the end of the single block.

```
#pragma omp parallel private (tmp)
{
          do_many_things();
#pragma omp single
          {     exchange_boundaries();   }
          do_many_other_things();
}
```

OpenMP: Synchronization

- The **flush** construct denotes a sequence point where a thread tries to create a consistent view of memory.
  - All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - Variables in registers or write buffers must be updated in memory.
- Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

This is a confusing construct and we won't say much about it. To learn more, consult the OpenMP specifications.

OpenMP: Library routines

- Lock routines
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()
- Runtime environment routines:
    - Modify/Check the number of threads
        - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
    - Turn on/off nesting and dynamic mode
        - omp_set_nested(), omp_set_dynamic(), omp_get_nested(), omp_get_dynamic()
    - Are we in a parallel region?
        - omp_in_parallel()
    - How many processors in the system?
        - omp_num_procs()

OpenMP: Library Routines

- Protect resources with locks.

```
   omp_lock_t lck;
   omp_init_lock(&lck);
#pragma omp parallel private (tmp)
{
   id = omp_get_thread_num();
   tmp = do_lots_of_work(id);
   omp_set_lock(&lck);
   printf("%d %d", id, tmp);
   omp_unset_lock(&lck);
}
```

OpenMP: Library Routines

- To fix the number of threads used in a program, first turn off dynamic mode and then set the number of threads.

```
#include <omp.h>
void main()
{   omp_set_dynamic(0);
    omp_set_num_threads(4);
#pragma omp parallel
    {   int id=omp_get_thread_num();
        do_lots_of_stuff(id);   }
}
```

OpenMP: Environment Variables

- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
  - OMP_SCHEDULE “schedule[, chunk_size]”
- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*
- Can the program use a different number of threads in each parallel region?
  - OMP_DYNAMIC TRUE || FALSE
- Will nested parallel regions create new teams of threads, or will they be serialized?
  - OMP_NESTED TRUE || FALSE