The OpenMP Crash Course

(How to Parallelize Your Code with Ease and Inefficiency)

Tom Logan

Course Overview

I. Intro to OpenMP

II. OpenMP Constructs

III. Data Scoping

IV. Synchronization

V. Practical Concerns

VI. Conclusions

Section I: Intro to OpenMP

- **What's OpenMP?**
  - Fork-Join Execution Model
  - How it works
  - OpenMP versus Threads
  - OpenMP versus MPI
- **Components of OpenMP**
  - Compiler Directives
  - Runtime Library
  - Environment Variables

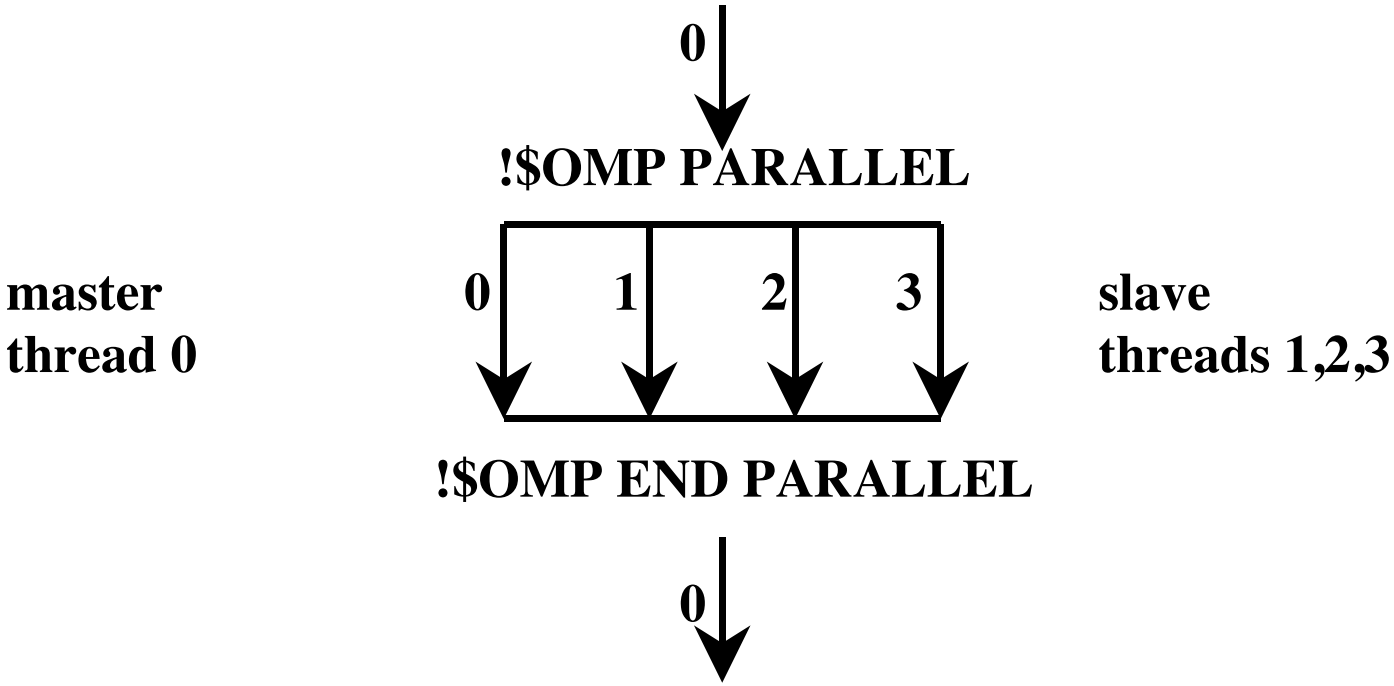
What's OpenMP?

- **OpenMP is a standardized shared memory parallel programming model**
  - Standard provides portability across platforms
  - Only useful for shared memory systems
  - Allows incremental parallelism
  - Uses directives, runtime library, and environment variables

Fork-Join Execution Model

- **Execution begins in a single master thread**
- **Master spawns threads for parallel regions**
- **Parallel regions executed by multiple threads**
  - master and slave threads participate in region
  - slaves only around for duration of parallel region
- **Execution returns to the single master thread after a parallel region**

How It Works


0
!$OMP PARALLEL
master
thread 0
0
1
2
3
slave
threads 1,2,3
!$OMP END PARALLEL
0

OpenMP versus Threads

- **Both Threads and OpenMP use the same Fork-Join parallelism**
- **Threads**
  - Explicitly create processes
  - More programmer burden
- **OpenMP**
  - Implicitly create processes
  - Relatively easy to program

OpenMP versus MPI

- **OpenMP**
  - 1 process, many threads
  - Shared architecture
  - Implicit messaging
  - Explicit synchronization
  - Incremental parallelism
  - Fine-grain parallelism
  - Relatively easy to program
- **MPI**
  - Many processes
  - Non-shared architecture
  - Explicit messaging
  - Implicit synchronization
  - All-or-nothing parallelism
  - Coarse-grain parallelism
  - Relatively difficult to program

Components of OpenMP

Compiler Directives

- **Compiler directive based model**
  - Compiler sees directives as comments unless OpenMP enabled
  - Same code can be compiled as either serial or multi-tasked executable
  - Directives allow for
    - Work Sharing
    - Synchronization
    - Data Scoping

Runtime Library

- **Informational routines**

  omp_get_num_procs() - number of processors on system
  omp_get_max_threads() - max number of threads allowed
  omp_get_num_threads() - get number of active threads
  omp_get_thread_num() - get thread rank

- **Set number of threads**

  omp_set_num_threads(integer)
  - set number of threads
  - see OMP_NUM_THREADS

- **Data access & synchronization**
  - omp_<*>_lock() routines - control OMP locks

Environment Variables

- **Control runtime environment**
  - OMP_NUM_THREADS - number of threads to use
  - OMP_DYNAMIC - enable/disable dynamic thread adjustment
  - OMP_NESTED - enable/disable nested parallelism
- **Control work-sharing scheduling**
  - OMP_SCHEDULE
    - specify schedule type for parallel loops that have the RUNTIME schedule
    - *static* - each thread given one statically defined chunk of iterations
    - *dynamic* - chunks are assigned dynamically at run time
    - *guided* - starts with large chunks, then size decreases exponentially
    - Example would be: *setenv OMP_SCHEDULE "dynamic,4"*

Section II: OpenMP Constructs

- **Directives**
- **Constructs**
  - Parallel Region
  - Work-Sharing
    - DO/FOR Loop
    - Sections
    - Single
  - Combined Parallel Work-Sharing
    - DO/FOR Loop
    - Sections

Directives: Format

sentinel directive_name [clause[[,] clause] …]

- Directives are case insensitive in FORTRAN and case-sensitive in C/C++
- Clauses can appear in any order separated by commas or white space

Directives: Sentinels

- **Fortran Fixed Form**

```
123456789
!$omp
c$omp
*$omp
```

- **Fortran Free Form**

```
!$omp
```

- **C/C++**

```
#pragma omp
        { … }
```

Directives: Continuations

- **Fortran Fixed Form - character in 6th column**

```
123456789
c$omp parallel do shared(alpha,beta)
c$omp+            private(gamma,delta)
```

- **Fortran Free Form - trailing "&"**

```
!$omp parallel do shared(alpha,beta) &
!$omp             private(gamma,delta)
```

- **C/C++ - trailing "\"**

```
#pragma omp parallel do \
        shared(alpha) private(gamma,delta)
           { … }
```

Directives: Conditional Compilation

- **Fortran Fixed Form**

```
123456789
!$
c$
*$
```

- **Fortran Free Form**

```
!$
```

- **C/C++**

```
#ifdef _OPENMP
  …
#endif
```

Example: Conditional Compilation

- **conditional.F (note the .F invokes cpp)**

```
PROGRAM conditional
      print *,' Program begins'
!$    print *,' Used !$ sentinel'
#ifdef _OPENMP
      print *,' Used _OPENMP environment variable'
#endif
#ifdef _OPENMP
!$    print *,' Used both !$ and _OPENMP'
#endif
      print *,' Program ends'
END
```

Example: Conditional Compilation

```
% f90 -o condf cond.F90
% ./condf
 Program begins
 Program ends

% f90 -mp -o condf cond.F90
% ./condf
 Program begins
 Used !$ sentinel
 Used _OPENMP environment variable
 Used both !$ and _OPENMP
 Program ends
```

OpenMP Constructs

Constructs: Parallel Region

- **FORTRAN**

```
!$ OMP parallel [clause] …
    structured-block
!$ OMP end parallel
```

- **C/C++**

```
#pragma omp parallel [clause]…
    structured-block
```

- **All code between directives is repeated on all threads**
- **Each thread has access to all data defined in program**
- **Implicit barrier at the end of the parallel region**

Example: Parallel Region

```
!$omp parallel private(myid, nthreads)
      myid = omp_get_thread_num()
      nthreads = omp_get_num_threads()
      print*,'Thread',myid,'thinks there are',nthreads,&
            'threads'
      do i=myid+1,n,nthreads
            a(i)=a(i)*a(i)
      end do
!$omp end parallel
```

Constructs: Work-Sharing

- **FORTRAN**

!$ OMP do

!$ OMP sections

!$ OMP single

- **C/C++**

#pragma omp for

#pragma omp sections

#pragma omp single

- **Each construct must occur within a parallel region**
- **All threads have access to data defined earlier**
- **Implicit barrier at the end of each construct**
- **Compiler copes with how to distribute work**
- **Programmer provides guidance using clauses**

Work-Sharing: Do/For Loop

- **FORTRAN**

```
!$ omp do [clause] …
   do-loop
[!$ omp end do [nowait]]
```

- **C/C++**

```
#pragma omp for
   [clause] ...
   for-loop
```

- **Iterations are distributed among threads**
- **Distribution controlled by clauses & env. vars.**
- **Data scoping controlled by defaults & clauses**
- **Implicit barrier can be removed by nowait clause**

Example: Do/For Loop

```
!$omp parallel
!$omp do
    do i = 1, n
        a(i) = a(i) * a(i)
    end do
!$omp end do
!$omp end parallel
```

```
#pragma omp parallel
 {
#pragma omp for
    for (i=0; i<n; i+)
      {
        a[i] = a[i]*a[i];
      }
 }
```

Work-Sharing: Sections

- **FORTRAN**

```
!$ omp sections [clause] ...
[!$ omp section
    code for this section] ...
!$ omp end sections [nowait]
```

- **C/C++**

```
#pragma omp sections [clause]...
{
  [#pragma omp section
        code for this section] ...
}
```

- **Defines concurrent sections of code**
- **Distributes sections among threads**

Example: Sections

```
do i = 1, nlines
    call read_line(filea,ina)
    call read_line(fileb,inb)
    do j = 1, linelen
      out(j,i)=ina(j)*conj(inb(j))
    end do
end do
```

```
!$omp parallel
do i = 1, nlines
!$omp sections
    call read_line(filea,ina)
!$omp section
    call read_line(fileb,inb)
!$omp end sections
!$omp do
    do j = 1, linelen
      out(j,i) = ina(j)*conj(inb(j))
    end do
end do
!$omp end parallel
```

Work-Sharing: Single

- **FORTRAN**

```
!$ omp single [clause] ...
    structured-block
!$ omp end single [nowait]
```

- **C/C++**

```
#pragma omp single
    [clause] ...
    structured-block
```

- **Defines serial code in a parallel region**
- **An unspecified single thread executes the code**
- **No implicit barrier at the start of the construct**
- **Common use is for performing I/O operations**

Example: Single

```
integer len
real in(MAXLEN), out(MAXLEN)

do i = 1,nlines
   call read_line(in,len)
   do j=1,len
     call compute(out(j), in, len)
   end do
   call wrt_line(out,len)
end do
```

```
integer len
real in(MAXLEN), out(MAXLEN)
!$omp parallel
      do i = 1,nlines
      !$omp single
          if (i.GT.1) call wrt_line(out,len)
          call read_line(in,len)
      !$omp end single
      !$omp do
          do j=1,len
              call compute(out(j), in, len)
          end do
      end do
!$omp end parallel
call wrt_line(out,len)
```

Constructs: Parallel Work-Sharing

- **FORTRAN**

!$ OMP parallel do

!$ OMP parallel sections

- **C/C++**

#pragma omp parallel for

#pragma omp parallel sections

- **Combines parallel construct with work-sharing**
- **Basically it's a shorthand convenience**
- **Most common OpenMP construct is parallel do**

Parallel Work-Sharing: Do/For

- **FORTRAN**

!$ omp parallel do[clause] …
    do-loop
[!$ omp end parallel do]

- **C/C++**

#pragma omp parallel for [clause]…
    for-loop

!$omp parallel do

```
    do i = 1, n
          a(i) = a(i) * a(i)
    end do
```

!$omp end parallel do

#pragma omp parallel for

```
    for (i=0; i<n; i+)
          a[i] = a[i]*a[i];
```

Parallel Work-Sharing: Sections

- **FORTRAN**

```
!$ omp parallel sections [clause] …
[!$omp section
    code for this section]…
!$omp end parallel sections
```

- **C/C++**

```
#pragma omp parallel sections [clause]…
{
  [#pragma omp section
        code for this section] …
}
```

Section III: Data Scoping

- Defaults
- Rules
- Data Scoping Clauses
- Guide to Choosing Scope Attributes

What is Data Scoping?

- **Most common sources of error in shared memory programs are**
  - Unintended sharing of variables
  - Privatization of variables that need to be shared
- **Determining whether variables should be shared or private is a key to parallel program correctness and performance**
- **Most difficult part of OpenMP programming**

OpenMP Data Scoping

- **Every variable has scope *shared* or *private***
- **Scoping clauses consist of**
  - *shared* and *private* explicitly scope specific variables
  - *firstprivate* and *lastprivate* perform initialization and finalization of privatized variables
  - *default* changes the default rules used when variables are implicitly scoped
  - *reduction* explicitly identifies a reduction variable

Default Data Scopes

- **If a variable in a parallel region is not explicitly scoped, then it is shared**
  - Correct behavior if variable is read but not modified
  - If the variable is assigned in the parallel region, then may need to
    - Explicitly scope
    - Add synchronization

Scoping Example 1

- **BAD**

```
#pragma omp parallel for
for (i=0; i<nl; i++)
 {
     int x;
     x = func1(i);
     y =func2(i);
     a[i] = x*y;
 }
```

- **GOOD**

```
#pragma omp parallel for private(y)
for (i=0; i<nl; i++)
  {
     int x;
     x = func1(i);
     y =func2(i);
     a[i] = x*y;
  }
```

Scope Terminology

- ***lexical scope*** **refers to the scope of accessibility of variables within a program (e.g. global or local)**
- ***lexical extent*** **of a parallel region consists of code contained directly within the region**
- ***dynamic extent*** **of a parallel region includes lexical extent and code contained within subroutines invoked within parallel region**

Default Data Scopes

- **Exceptions to default shared**
  - Parallel loop indices (in Fortran, index variables of sequential loops within the lexical extent of parallel regions are also scoped private)
  - Nested declarations occurring within lexical extent of parallel regions (C/C++)
  - Variables in subroutines called from parallel regions
  - Exception to the exception
    - variables in subroutines called from parallel regions marked as save (FORTRAN) or static (C/C++) have a *shared* scope

Example 2 - Fortran

```
subroutine caller(a,n)
integer n, a(n), i, j, m
m=3

!$omp parallel do
      do i=1,n
         do j=1,5
              call callee(a(i), m, j)
         end do
      end do
end
```

(Answers at end of slides)

```
subroutine callee(x,y,z)
common /com/ c
integer x,y,z,c,ii,cnt
save cnt

cnt = cnt+1
do ii=1,z
     x= y+c
enddo
end
```

Example 2 - C

```
void caller(int a[],int n)
{
  int i, j, m=3;
  #pragma omp parallel for
  for (i=0; i<n; i++)  {
        int k=m;
        for (j=1;j<=5;j++) callee(&a[i], &k, j);
  }
}
```

```
extern int c;
void callee(int*x,int*y,int z)
{
      int ii;
      static int cnt;
      cnt++;
      for (ii=0; ii<z; ii++)
            *x = *y + c;
}
```

(Answers at end of slides)

Scoping Rules

- **Any variable can be scoped, including**
  - globals (C/C++)
  - common blocks and module variables (Fortran)
- **Scope clauses must be in lexical extent of each variable named**
- **Scoping clauses apply only to accesses to named variables occurring in lexical extent of directive containing scoping clause**

Scoping Rules

- **An individual variable can appear in at most a single scoping clause**
- **Variable to be scoped must refer to an entire object, not a portion of an object**
  - Can not scope array element or field of a structure
  - Can scope struct or class (C/C++) and entire common blocks (Fortran)

Data Scoping Clauses

- **SHARED: shared(varlist)**
  - Declares variables that have a single common memory location for each thread
  - All threads access the same memory location, thus access must be carefully controlled

Data Scoping Clauses

- **PRIVATE: private(varlist)**
  - Declares variables for which each thread has its own copy (separate memory locations)
  - Private variables only exist in parallel region
  - Value is undefined at start of parallel region
  - Value is undefined after end of parallel region

Private Clause Notes

- **Exceptions to undefined private variables**
  - Parallel loop control variables are defined
  - C++ class objects
    - each thread invokes constructor upon entering region
    - each thread invokes destructor upon completion of region
  - Allocatable arrays in Fortran 90
    - Serial copy before parallel region must be unallocated
    - Each thread gets a private unallocated copy of the array
    - Copy must be allocated before use
    - Copy must be explicitly deallocated before end of loop
    - Original serial copy is unallocated after loop

Private Clause Notes

- **Must be able to determine size of the variable declared private**
  - Fortran formal array parameter of adjustable size must fully specify the bounds
  - C/C++ can not use variables with incomplete types
- **Can not declare C++ reference type variables as private**

Data Scoping Clauses

- **FIRSTPRIVATE: firstprivate(varlist)**
  - Declares variables for which each thread has its own copy (separate memory locations)
  - Each thread's copy of the variable is initialized to the value the master held before entering the loop
  - This may be more efficient than declaring some variables to be shared

Data Scoping Clauses

- **LASTPRIVATE: lastprivate(varlist)**
  - Loads the master's variable with the last value, as set by the last iteration of the loop
  - If you do not do the last iteration, behavior is undefined

Data Scoping Clauses

- **DEFAULT:**
  - Fortran Syntax: **default(private | shared | none)**
  - C/C++ Syntax: **default(share | none)**
    - Can't have default(private) since standard libraries often reference global variables
  - changes the default scoping attributes for any variables not specifically declared otherwise
  - default(none) helps catch scoping errors by forcing explicit scoping of all variables in lexical extent

Data Scoping Clauses

- **REDUCTION:reduction(redn_oper:varlist)**
  - Declares that a scalar variable will be involved in a reduction operation
  - In Fortran, reduction operators can be +, *, -, .AND., .OR., .EQV., .NEQV., MAX, MIN, IAND, IOR, IEOR
  - In C/C++, reduction operators can be +, *, -, &, |, ^ , &&, ||
  - Floating point arithmetic may not be commutative!

Example: Reduction

```
!$omp parallel do reduction(+:sum)
   do i=1,ndata
         a(i) = a(i) * a(i)
         sum = sum + a(i)
   end do
```

```
#pragma omp parallel for reduction(+:sum)
   for (i=0; i<ndata; i++){ a[i]=a[i]*a[i]; sum+=a[i];}
```

Data Scoping Clauses

- **SCHEDULE: schedule(type[,chunk])**
  - Control distribution of iterations in do/for directives
  - Type can be
    - *Static* chunks assigned in round-robin order
    - *Dynamic* chunks given out dynamically
    - *Guided* first large and then smaller chunks are allocated to each processor
    - *Runtime* schedule determined at runtime using environment variable OMP_SCHEDULE
  - Chunk is a positive integer

Scheduling Types

- **Static**
  - Use if each iteration takes about the same time
  - Has the least overhead
  - Chunk size iterations assigned in round-robin
  - Default chunk size is #(iterations)/#(threads)
- **Dynamic**
  - Use if each iteration may take different amount of time
  - First come first served policy of chunk size iterations
  - Default chunk size is 1; make sure to specify larger

Scheduling Types

- **Guided**
  - Similar to dynamic - first come first served
  - Different from dynamic
    - Initial assignment of work is large
    - Size of work assignments decreases exponentially
    - Minimum work assignment size specified by chunk
    - Default chunk is 1

Data Scoping Clauses

- **COPYIN: copyin(list)**
  - initializes threadprivate variables to the master's value

Data Scoping Clauses

- **IF: if (*expr*)**
  - Controls loop execution for parallel or serial
  - Used with parallel constructs (parallel, parallel do, parallel sections)
  - If *expr* evaluates to true, loop is executed in parallel
  - If *expr* evaluates to false, loop is executed in serial

Data Scoping Clauses

- **ORDERED: ordered**
  - Specifies that an ordered construct will appear within a loop
  - Used with a do directive
  - Must be included if an ordered directive is used in the loop

Data Scoping Clauses

- **NOWAIT: nowait**
  - specifies that no implicit barrier exists at the end of a construct

Threadprivate

- **FORTRAN**

!$omp threadprivate (list)

- **C/C++**

#pragma omp threadprivate (list)

- **Makes global variable(s) private**
- **This over rides the default of shared**

Scope Attribute Choosing Guide

- **Consider a PARALLEL DO construct, assume the loop is actually parallel**

```
      sum = 0.0
!$omp PARALLEL DO  SHARED (??)  PRIVATE (??)  ??
      DO I = 1, N
        A(I) = x
        B(I) = Y(K)
        temp = A(I) * B(I)
        sum = sum + temp
      END DO
      print *, sum
```

Scope Attribute Choosing Guide

0) Make a list of all the variables in the loop

- sum, I,n, A, x, B, Y, k, temp

1) Variables only simply subscripted by the parallel loop index are SHARED

- See A and B
- Subscript must be only the loop index. If A(I) = x were A(I-1) = x, then there is a dependence between iterations that may prevent parallelization
- Variables can appear on either side of the assignment statements
- Each OpenMP thread accesses different elements of these arrays based on the iterations they are assigned. Two threads cannot access the same element because no two threads are assigned the same iteration

Scope Attribute Choosing Guide

2) Variables not subscripted by the parallel loop index

a) Variables only on right hand side of assignment statements are SHARED

1. See N, x, Y and K
2. Variables assigned values before parallel loop; values are not changed
3. Each thread accesses the same variable. Read access does not change the value of the variable, and no conflicts or races can occur.

b) Variables used in reduction operations are SHARED

1. See sum.
2. Reduction variable initialized before the parallel loop & used after
3. For each REDUCTION variable, an invisible PRIVATE variable is created for each thread to compute partial results. PRIVATE variables are accumulated to visible SHARED variable before loop is completed.

Scope Attribute Choosing Guide

c) Variables that are defined before being used are PRIVATE

1. See temp.
2. The variable is not referenced before or after the parallel loop.
3. Compiler creates variable with same name but with different storage location for each thread. Threads cannot access variables of the same name in another thread.

3) The parallel loop index is handled by the compiler, and should not appear in a PRIVATE or SHARED clause. The compiler will ignore whatever specification is given, and do the right thing.

- See i

Scope Attribute Choosing Guide

- Correct scoping specifications for this parallel loop are

  !$omp PARALLEL DO SHARED (A, B, N, x, Y, K, sum) &

  !$omp PRIVATE (temp) REDUCTION (+:sum)

- More simply, using the fact that default scope is SHARED

  !$omp PARALLEL DO PRIVATE (temp) REDUCTION (+:sum)

- If all variables in a loop cannot be described as in the guidelines above, then the loop needs deeper analysis or it is not parallel. If it can, then the loop is parallel and you got the scoping right!

Section IV: Synchronization

Constructs - Synchronization

- **FORTRAN**

!$ OMP [end] master

!$ OMP [end] critical [(name)]

!$ OMP barrier

!$ OMP atomic

!$ OMP flush [(list)]

!$ OMP [end] ordered

- **C/C++**

#pragma omp master

#pragma omp critical [(name)]

#pragma omp barrier

#pragma omp atomic

#pragma omp flush [(list)]

#pragma omp ordered

Master

- **FORTRAN**

```
!$omp master
    structured-block
!$omp end master
```

- **C/C++**

```
#pragma omp master
    structured-block
```

- **Region is executed only by the master thread**
- **There is no barrier either before or after a MASTER region**

Critical

- **FORTRAN**

```
!$omp critical [(name)]
    structured-block
!$omp end critical [(name)]
```

- **C/C++**

```
#pragma omp critical [(name)]
    structured-block
```

- **Allows one thread exclusive access to code block**
- **Order of threads in block is non-deterministic**

Barrier

- **FORTRAN**

```
!$omp barrier
```

- **C/C++**

```
#pragma omp barrier
```

- **Synchronizes all threads in a team**
- **Halts program execution until all threads arrive**

Atomic

- **FORTRAN**

```
!$omp atomic
    expression-statement
```

- **C/C++**

```
#pragma omp atomic
    expression-statement
```

- **Specialized case of the CRITICAL directive**
- **Protects from simultaneous updates on a scalar variable by multiple threads**

Flush

- **FORTRAN**

!$omp flush [(list)]

- **C/C++**

#pragma omp flush [(list)]

- **Sets a memory fence (synchronization point)**

Ordered

- **FORTRAN**

```
!$omp ordered
    structured-block
!$omp end ordered
```

- **C/C++**

```
#pragma omp ordered
    structured-block
```

- **Used to order loop iterations for a code block**
- **Forces section of a loop to be ordered as if it were executing serially**
- **Must be combined with the ordered clause**

Input Serialization (fico)

```
 /*At each grid point, read in a chunk of each image...*/
#pragma omp critical
        readMatrix(szImg1,s,FLOAT_COMPLEX,srcSize,srcSize,
           x1- srcSize/2+1, y1-srcSize/2+1, wid, len,0,0);
#pragma omp critical
        readMatrix(szImg2,t,FLOAT_COMPLEX,trgSize,trgSize,
           x2 - trgSize/2 + 1, y2 - trgSize/2 + 1, wid, len,0,0);
```

Output Serialization (fico)

```
#pragma omp critical
  {
   fprintf(fp_output,"%6d %6d %8.5f %8.5f %4.2f\n",
                     x1,y1,x2+dx,y2+dy,snr);
   fflush(fp_output);
   if (!quietflag&&(goodPoints<=10||!(goodPoints%100)))
       printf("\t%6d %6d %8.5f %8.5f %4.2f/%4.2f\n",
                     x1,y1,dx,dy,snrFW,snrBW);
  }
```

Section V: Practical Concerns

- OpenMP Usage Suggestions
- Common OpenMP Problems & Gotchas
- Restrictions on Parallel Loop Indices
- Conclusions

OpenMP Usage Suggestions

- **Don't use clauses if at all possible - instead**
  - Declare private variables in parallel regions (C/C++)
  - Convert parallel regions into subroutines (FORTRAN)
- **Don't use synchronization if at all possible - particularly 'flush'**
- **Don't use locks if at all possible - if have to use them, make sure that you unset them**

OpenMP Usage Suggestions

- **Only use parallel do/for constructs**
  - Everything else in the language will slow you down
  - Unfortunately, they are often necessary evils
- **Scheduling**
  - Use static unless load balancing is a real issue
  - If load balancing is a problem, use static chunks
  - Using Dynamic or Guided requires too much overhead

Typical Overheads

- **Some example overheads (in clock cycles)**
  - Thread Id 10-50
  - Static Do/For 100-200
  - barrier 200-500
  - Dynamic Do/For 1000-2000
  - Ordered statement 5000-10000

OpenMP Usage Suggestions

- Know your application
- Use a profiler whenever possible, this will point out bottle necks and other problems

Common OpenMP Problems

- Too fine grain parallelism
- Overly synchronized
- Load Imbalance
- True Sharing (ping-ponging in cache)
- False Sharing (another cache problem)
- Data Race Conditions
- Deadlocks

Gotchas!

- Declaring a shared pointer makes the pointer shared, not memory it points to
- Declaring a private pointer makes the pointer private, not memory it points to
- Remember that reduction clauses need a barrier - make sure not to use 'nowait' when a reduction is being computed

Gotchas!

- **Don't assume order of execution for loops**
  - Access to shared variables can occur anytime and, thus, in any order
- **If you suspect a race condition, run the loops in reverse order and see if you get the same result**

To use OpenMP

- **C/C++ use #include <omp.h>**
- **Setting number of threads**
  - Environment variable OMP_NUM_THREADS
  - Run-time library omp_set_num_threads()

Restrictions on Parallel Loops

- **Must be possible to compute the trip-count without having to execute the loop**
- **Must complete all iterations of the loop**
  - cannot use constructs that exit the loop early
  - single top entry point and single bottom exit point
  - *exception* - execution can be terminated within the loop using usual mechanisms (*stop* in Fortran, *exit* in C/C++)

Loop Restrictions: Fortran

- ***parallel do* directive must be immediately followed by do loop statement of the form:**

  DO index = lowerbound, upperbound [,stride]

  - do-while loops are not allowed
  - do loops that lack iteration control are not allowed
  - array assignment statements are not allowed
  - can not use *exit* or *goto* to branch out of loop
  - can use *cycle* to complete current iteration
  - can use *goto* to jump from one statement in loop to another

Loop Restrictions: C

- ***parallel for* pragma must be immediately followed by a for loop of the form:**

  for (index = start; index < end; incr_exp)

  - *index* must be an integer
  - comparison operator may be <,<=, >, or >=
  - *start* and *end* are any numeric expression whose value does not change during execution of loop
  - *incr_exp* must change value of *index* by same amount each iteration

Loop Restrictions: C

- *incr_exp* may use only following operators and forms:

Operator	Form
++	*index*++ or ++*index*
--	*index*-- or --*index*
+=	*index* += *incr*
-=	*index* –= *incr*
=	*index* = *index* + *incr* or *index* = *incr* + *index* or *index* = *index* – *incr*

(where *incr* is a numeric expression that does not change during the loop)

Loop Restrictions: C/C++

- can not use a *break* or *goto* to leave the loop
- can use *continue* to complete current iteration
- can use *goto* to jump from one statement inside the loop to another
- **C++**
  - can not *throw* an exception caught outside loop
  - can use *throw* if it is caught by *try* block in loop body

Simplified ‘for’ Loop (c2p)

```
>  for (j=0,cP=cpx,aP=amp,pP=phase; j<i ; j++,cP++,aP++,pP++)
---
< #pragma omp parallel for
<       for (j=0; j < i; j++)
<       {
<                       FComplex *cP = &cpx[j];
<                       float    *aP = &amp[j];
<                       float    *pP = &phase[j];
137,140d127
<                       cP++;
<                       aP++;
<                       pP++;
<       }
```

Convert ‘while’ into ‘for’ (fico)

```
>       while (getNextPoint(&x1,&y1,&x2,&y2))
...
>       } /* end while(getNextPoint) */
---
< #pragma omp parallel for reduction(+:attemptedPoints,goodPoints)
<  for (pointNo=0;pointNo<gridResolution*gridResolution;pointNo++)
...
<       } /* end for pointNo */
```

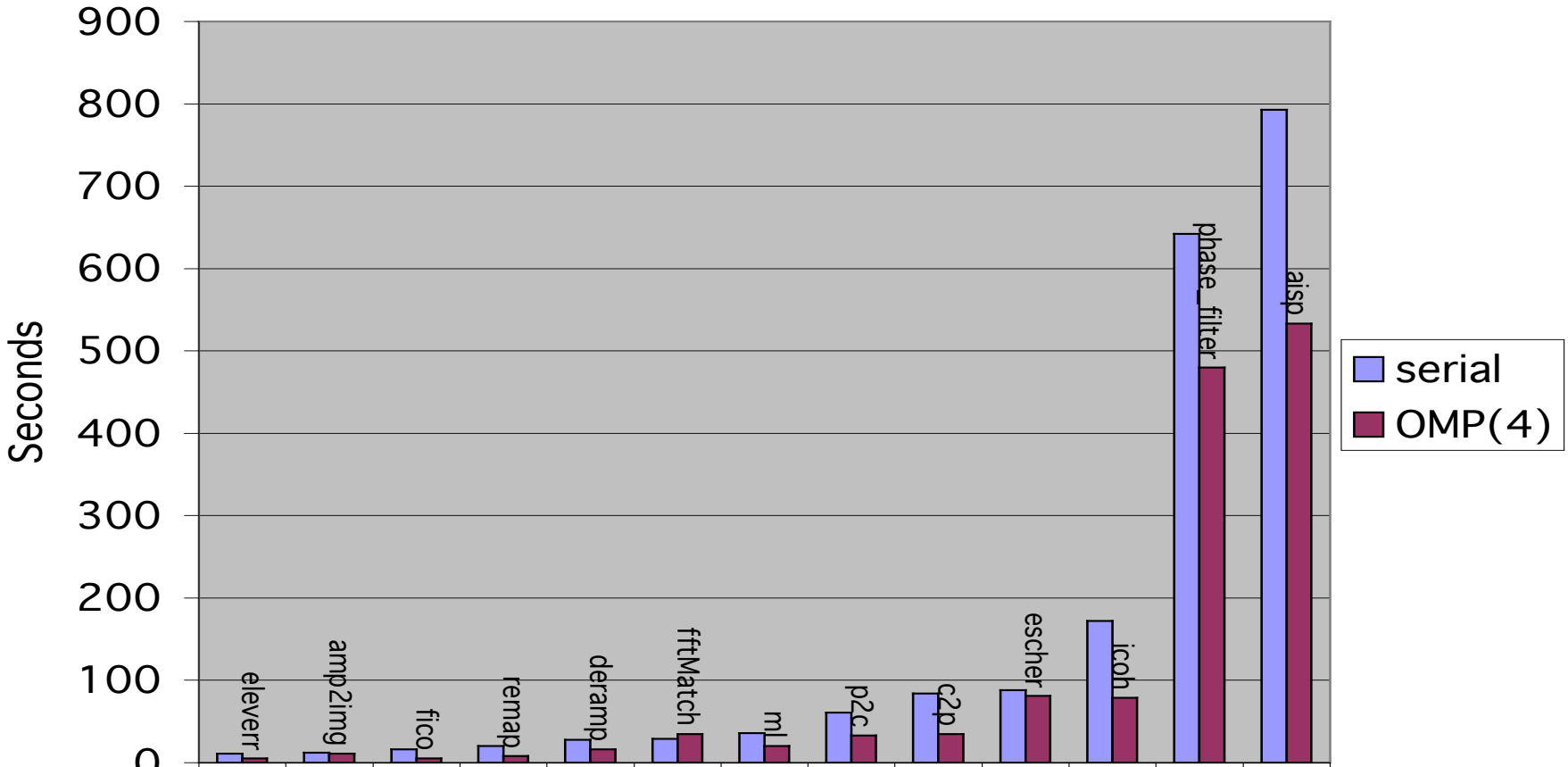
Section VI: Conclusions

- **OpenMP is DEFINITELY the simplest parallel programming language I have ever seen**
- **Effective for small-scale parallelism**
  - For 4 to 16 processors, gets decent performance
  - Can not be beat for ease of programming
- **Not very useful for large-scale parallelism**
  - Difficult to get good parallel efficiency for any significant number of processors

Conclusions

- OpenMP is designed for "flat address space" that doesn't exist in modern computer systems
- This is probably the biggest impediment to obtaining scalable parallel performance

Example OpenMP Results



	eleverr	amp2img	fico	remap	deramp	fftMatch	ml	p2c	c2p	escher	icoh	phase_filter	aisp
serial	11	12	16	20	28	29	36	61	84	88	172	642	793
OMP(4)	5	11	5	8	16	35	20	33	35	81	79	480	533

Example 2 - Fortran Answers

Variable	*Scope*	*Safe?*	*Reason for Scope*
a	Shared	Yes	Declared outside construct
n	Shared	Yes	Declared outside construct
i	Private	Yes	Parallel loop index
j	Private	Yes	Fortran sequential loop index in parallel region
m	Shared	Yes	Declared outside construct
x	Shared	Yes	Actual parameter is a, which is shared
y	Shared	Yes	Actual parameter is m, which is shared
z	Private	Yes	Actual parameter is j, which is private
c	Shared	Yes	In a common block
ii	Private	Yes	Local stack-allocated variable of subroutine
cnt	Shared	No	Local variable of subroutine with save attribute

Example 2 - C Answers

Variable	*Scope*	*Safe?*	*Reason for Scope*
a	Shared	Yes	Declared outside parallel construct
n	Shared	Yes	Declared outside parallel construct
i	Private	Yes	Parallel loop index
j	Shared	No	Declared outside parallel construct
m	Shared	Yes	Declared outside parallel construct
k	Private	Yes	Declared inside parallel construct
x	Private	Yes	Value parameter
*x	Shared	Yes	Actual parameter is a, which is shared
y	Private	Yes	Value parameter
*y	Private	Yes	Actual parameter is k, which is private
z	Shared	Yes	Value parameter
c	Shared	Yes	Declared as extern
ii	Private	Yes	Local stack-allocated variable
cnt	Shared	No	Declared as static