Parallel Programming with OpenMP

Science and Technology Support Group

High Performance Computing

Ohio Supercomputer Center

1224 Kinnear Road

Columbus, OH 43212-1163

OSC

Parallel Programming with OpenMP

OSC

Setting the Stage

- [Overview of parallel computing](#)
- [Introduction to OpenMP](#)
- [The OpenMP programming model](#)

Overview of Parallel Computing

- Parallel computing is when a program uses concurrency to either
    - decrease the runtime needed to solve a problem
    - increase the size of problem that can be solved
- The direction in which high-performance computing is headed!
- Mainly this is a price/performance issue
    - Vector machines very expensive to engineer and run
    - Commodity hardware/software - Clusters!

OSC

Writing a Parallel Application

- Decompose the problem into tasks
  - Ideally, these tasks can be worked on independently of the others
- Map tasks onto "threads of execution" (processors)
- Threads have shared and local data
  - Shared: used by more than one thread
  - Local: Private to each thread
- Write source code using some parallel programming environment
- Choices may depend on (among many things)
  - the hardware platform to be run on
  - the level performance needed
  - the nature of the problem

Parallel Architectures

- Distributed memory
  - Each processor has local memory
  - Cannot directly access the memory of other processors
- Shared memory (OSC Altix 3000, Sun 6800, Cray SV1ex)
  - Processors can directly reference memory attached to other processors
  - Shared memory may be *physically* distributed
    - The cost to access remote memory may be high!
  - Several processors may sit on one memory bus (SMP)
- Combinations are very common (OSC IA32 and IA64 clusters)
  - IA32 cluster
    - 128 compute nodes, each with 2 processors sharing 2GB of memory on one bus
  - IA64 cluster
    - 150 compute nodes, each with 2 processors sharing 4GB of memory on one bus
  - High-speed interconnect between nodes

Parallel Programming with OpenMP

Parallel Programming Models

- Distributed memory systems
  - For processors to share data, the programmer must explicitly arrange for communication - **"Message Passing"**
  - Message passing libraries:
    - MPI ("Message Passing Interface")
    - PVM ("Parallel Virtual Machine")
    - Shmem (Cray only)
- Shared memory systems
  - "Thread" based programming
  - Compiler directives (OpenMP; various proprietary systems)
  - Can also do explicit message passing, of course

OSC

Parallel Computing: Hardware

- In very good shape!

- Processors are cheap and powerful
  - Intel, MIPS, …
  - Theoretical performance approaching 1 GFLOP/sec

- SMP nodes with 1-8 CPUs are common

- Affordable, high-performance interconnect technology is becoming available – Myrinet, Infiniband

- Systems with a few hundreds of processors and good inter-processor communication are not hard to build

Parallel Computing: Software

- Not as mature as the hardware

- The main obstacle to making use of all this power
  - Perceived difficulties with writing parallel codes outweigh the benefits

- Emergence of standards is helping enormously
  - MPI
  - OpenMP

- Programming in a shared memory environment generally easier

- Often better performance using message passing
  - Much like assembly language vs. C/Fortran

Introduction to OpenMP

_____

- OpenMP is an API for writing multithreaded applications in a shared memory environment
- It consists of a set of compiler directives and library routines
- Relatively easy to create multi-threaded applications in Fortran, C and C++
- Standardizes the last 15 or so years of SMP development and practice
- Currently supported by
  - Hardware vendors
    - Intel, HP, SGI, Sun, IBM
  - Software tools vendors
    - Intel, KAI, PGI, PSR, APR, Absoft
  - Applications vendors
    - ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, ...
- Support is common and growing

The OpenMP Programming Model

- A *master* thread spawns *teams* of threads as needed
- Parallelism is added incrementally; the serial program evolves into a parallel program

Master thread
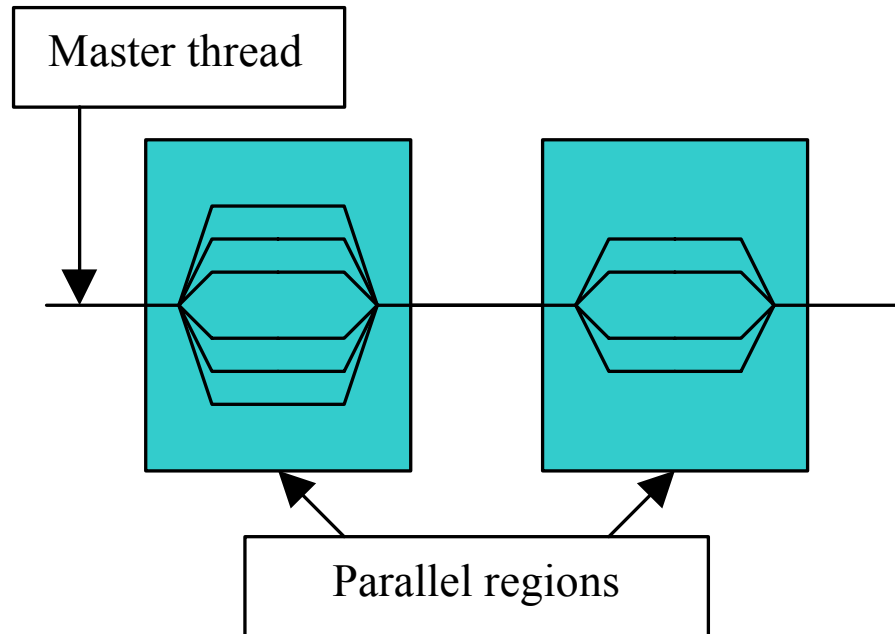
Parallel regions

The OpenMP Programming Model

- Programmer inserts OpenMP directives (Fortran comments, C `#pragmas`) at key locations in the source code.
- Compiler interprets these directives and generates library calls to parallelize code regions.

Serial:

```
void main(){
  double x[1000];
  for (int i=0; i<1000; i++){
    big_calc(x[i]);
  }
}
```

Parallel:

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

Split up loop iterations among a team of threads

The OpenMP Programming Model

- Number of threads can be controlled from within the program, or using the environment variable **OMP_NUM_THREADS.**

- The programmer is responsible for managing synchronization and data dependencies!

- Compiling on OSC systems:

Intel Compiler – IA64 Cluster   Intel Compiler – IA32 Cluster

```
efc -openmp prog.f
efc -openmp prog.f90
ecc -openmp prog.c
```

```
ifc -openmp prog.f
ifc -openmp prog.f90
icc -openmp prog.c
```

How do Threads Interact?

- Shared memory model
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to "race conditions"
  - When the program's outcome changes as the threads are scheduled differently.
- To control race conditions, use synchronization to avoid data conflicts
- Synchronization is expensive!
  - Think about changing how data is organized, to minimize the need for synchronization.

The Basics of OpenMP

- [General syntax rules](#)
- [The parallel region](#)
- [Execution modes](#)
- [OpenMP directive clauses](#)
- [Work-sharing constructs](#)
- [Combined parallel work-sharing constructs](#)
- [Environment variables](#)
- [Runtime environment routines](#)
- [Interlude: data dependencies](#)

General Syntax Rules

- Most OpenMP constructs are compiler directives or C `pragmas`
  - For C and C++, `pragmas` take the form

  ```
  #pragma omp construct [clause [clause]...]
  ```

  - For Fortran, directives take one of the forms:

  ```
  c$omp construct [clause [clause]...]
  !$omp construct [clause [clause]...]
  *$omp construct [clause [clause]...]
  ```

- Since these are directives, compilers that don't support OpenMP can still compile OpenMP programs (serially, of course!)

OSC

General Syntax Rules

- Most OpenMP directives apply to structured blocks
    - A block of code with one entry point at the top, and one exit point at the bottom. The only branches allowed are `STOP` statements in Fortran and `exit()` in C/C++

```
c$omp parallel

10      wrk(id) = junk(id)
        res(id) = wrk(id)**2
        if (conv(res)) goto 10

c$omp end parallel

        print *, id
```

A structured block

```
c$omp parallel

10      wrk(id) = junk(id)
30      res(id) = wrk(id)**2
        if (conv(res)) goto 20
        goto 10

c$omp end parallel

        if (not_done) goto 30
20      print *, id
```

Not a structured block!

OSC

The Parallel Region

- The fundamental construct that initiates parallel execution
- Fortran syntax:

```
c$omp parallel
c$omp& shared(var1, var2, …)
c$omp& private(var1, var2, …)
c$omp& firstprivate(var1, var2, …)
c$omp& reduction(operator|intrinsic:var1, var2, …)
c$omp& if(expression)
c$omp& default(private|shared|none)

   a structured block of code

c$omp end parallel
```

Parallel Programming with OpenMP

The Parallel Region

- C/C++ syntax:

```
#pragma omp parallel                                  \
        private (var1, var2, …)                        \
        shared (var1, var2, …)                         \
        firstprivate(var1, var2, …)                    \
        copyin(var1, var2, …)                          \
        reduction(operator:var1, var2, …)  \
        if(expression)                                 \
        default(shared|none)                           \
{

    …a structured block of code…

}
```

OSC

The Parallel Region

- The number of threads created upon entering the parallel region is controlled by the value of the environment variable **`OMP_NUM_THREADS`**
  - Can also be controlled by a function call from within the program.
- Each thread executes the block of code enclosed in the parallel region
- In general there is no synchronization between threads in the parallel region!
  - Different threads reach particular statements at unpredictable times.
- When all threads reach the end of the parallel region, all but the master thread go out of existence and the master continues on alone.

OSC

The Parallel Region

- Each thread has a thread number, which is an integer from 0 (the master thread) to the number of threads minus one.
  - Can be determined by a call to **omp_get_thread_num()**
- Threads can execute different paths of statements in the parallel region
  - Typically achieved by branching on the thread number:

```
#pragma omp parallel
{
    myid = omp_get_thread_num();
    if (myid == 0)
      do_something();
    else
       do_something_else(myid);
}
```

Parallel Regions: Execution Modes

- "Dynamic mode" (the default)
  - The number of threads used in a parallel region can vary, under control of the operating system, from one parallel region to the next.
  - Setting the number of threads just sets the *maximum* number of threads; you might get fewer!
- "Static mode"
  - The number of threads is fixed by the programmer; you must always get this many (or else fail to run).
- Parallel regions may be nested, but a compiler may choose to "serialize" the inner parallel region, *i.e.*, run it on a single thread.
- Execution mode is controlled by
  - The environment variable `OMP_DYNAMIC`
  - The OMP function `omp_set_dynamic()`

OpenMP Directive Clauses

- **`shared(var1,var2,…)`**
  - Variables to be shared among all threads (threads access same memory locations).
- **`private(var1,var2,…)`**
  - Each thread has its own copy of the variables for the duration of the parallel code.
- **`firstprivate(var1,var2,…)`**
  - Private variables that are initialized when parallel code is entered.
- **`lastprivate(var1,var2,…)`**
  - Private variables that save their values at the last (serial) iteration.
- **`if(expression)`**
  - Only parallelize if `expression` is true.
- **`default(shared|private|none)`**
  - Specifies default scoping for variables in parallel code.
- **`schedule(type [,chunk])`**
  - Controls how loop iterations are distributed among threads.
- **`reduction(operator|intrinsic:var1,var2…)`**
  - Ensures that a reduction operation (e.g., a global sum) is performed safely.

The `private`, `default` and `if` clauses

private & default

```
c$omp  parallel shared(a)
       private(myid,x)
    myid=omp_get_thread_num()
    x = work(myid)
    if (x < 1.0) then
      a(myid) = x
    end if


       is:

     parallel do default(private)
      shared(a)
      …
```

- Don't want to parallelize a loop if the overhead outweighs the speedup.

- Each thread has its own private copy of `x` and `myid`
- Unless `x` is made private, its value is indeterminate during parallel operation
- Values for private variables are undefined at beginning and end of the parallel region!
- `default` clause automatically makes `x` and `myid` private.

if(expression)

```
c$omp parallel do if(n.ge.2000)
     do i = 1, n
       a(i) = b(i)*c + d(i)
     enddo
```

firstprivate

- Variables are private (local to each thread), but are initialized to the value in the preceding serial code.

```fortran
program first
  integer :: myid,c
  integer,external :: omp_get_thread_num
  c=98
!$omp parallel private(myid)
!$omp& firstprivate(c)
  myid=omp_get_thread_num()
  write(6,*) 'T:',myid,' c=',c
!$omp end parallel
end program first
-----------------------------------
T:1 c=98
T:3 c=98
T:2 c=98
T:0 c=98
```

- Each thread has a private copy of c, initialized with the value 98

OpenMP Work-Sharing Constructs

- Parallel `for/DO`
- Parallel `sections`
- `single` directive
- Placed inside parallel regions
- Distribute the execution of associated statements among existing threads
  - No *new* threads are created.
- No implied synchronization between threads at the start of the work sharing construct!

OpenMP work-sharing constructs - `for/DO`

- Distribute iterations of the immediately following loop among threads in a team

```
#pragma omp parallel shared(a,b) private(j)
{
    #pragma omp for
      for (j=0; j<N; j++)
        a[j] = a[j] + b[j];
}
```

- By default there is a barrier at the end of the loop
  - Threads wait until all are finished, then proceed.
  - Use the **nowait** clause to allow threads to continue without waiting.

Detailed syntax - `for`

```
#pragma omp for [clause [clause]…]
    for loop
```

where each `clause` is one of

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(operator: list)`
- `ordered`
- `schedule(kind [, chunk_size])`
- `nowait`

OSC

Detailed syntax - `DO`

```
c$omp do [clause [clause]…]
    do loop
[c$omp end do [nowait]]
```

where each `clause` is one of

  - `private(list)`
  - `firstprivate(list)`
  - `lastprivate(list)`
  - `reduction(operator: list)`
  - `ordered`
  - `schedule(kind [, chunk_size])`

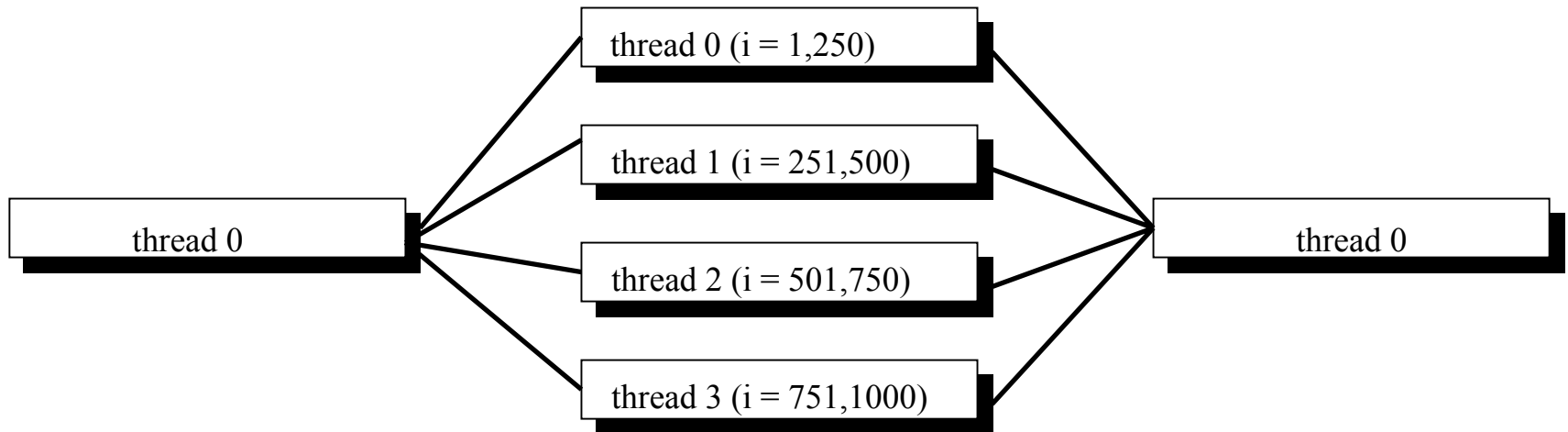- For Fortran 90, use `!$OMP` and F90-style line continuation.

OSC

The `schedule(type,[chunk])` clause

- Controls how work is distributed among threads
- `chunk` is used to specify the size of each work parcel (number of iterations)
- `type` may be one of the following:
  - `static`
  - `dynamic`
  - `guided`
  - `runtime`
- The `chunk` argument is optional. If omitted, implementation-dependent default values are used.

schedule(static)

- Iterations are divided evenly among threads
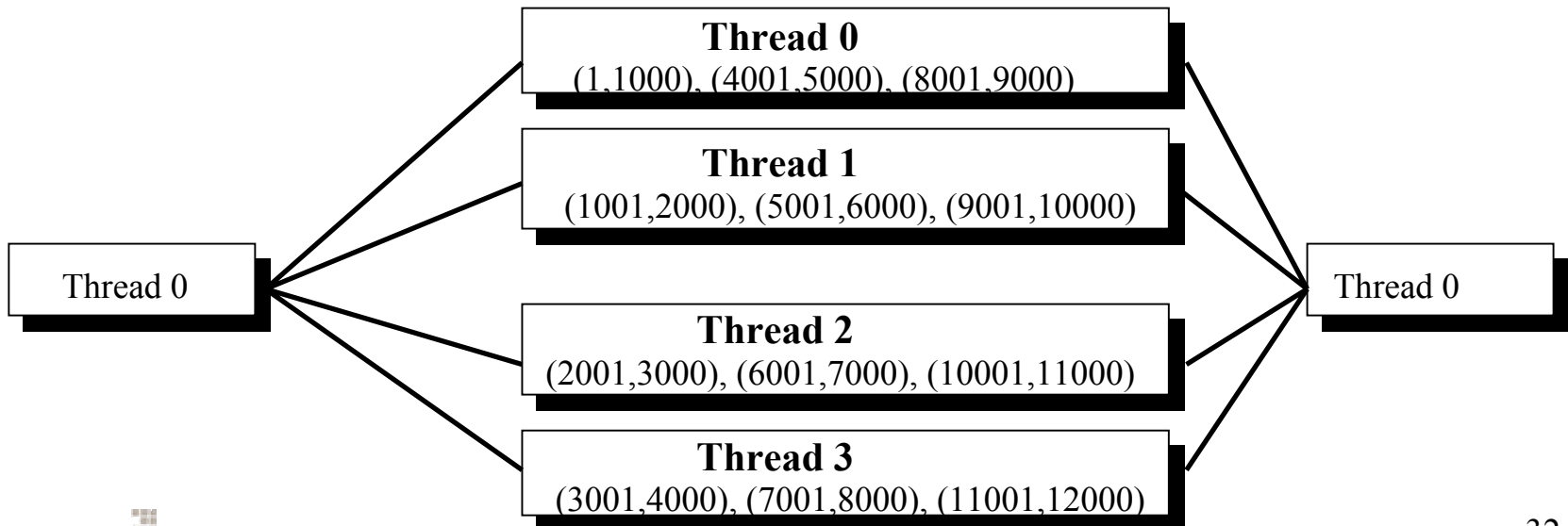
```
c$omp do shared(x) private(i)
c$omp& schedule(static)
          do i = 1, 1000
            x(i)=a
          enddo
```

thread 0 (i = 1,250)

thread 1 (i = 251,500)

thread 0

thread 2 (i = 501,750)

thread 0

thread 3 (i = 751,1000)

schedule(static,chunk)

- Divides the work load in to `chunk` sized parcels
- If there are N threads, each thread does every Nth `chunk` of work

```
c$omp   do shared(x)private(i)
c$omp&  schedule(static,1000)
        do i = 1, 12000
          … work …
        enddo
```

Thread 0

Thread 0
(1,1000), (4001,5000), (8001,9000)

Thread 1
(1001,2000), (5001,6000), (9001,10000)

Thread 2
(2001,3000), (6001,7000), (10001,11000)

Thread 3
(3001,4000), (7001,8000), (11001,12000)

Thread 0

Parallel Programming with OpenMP

OSC

`schedule(dynamic,chunk)`

- Divides the workload into `chunk` sized parcels.
- As a thread finishes one `chunk`, it grabs the next available `chunk`.
- Default value for `chunk` is 1.
- More overhead, but potentially better load balancing.

```
c$omp  do shared(x) private(i)
c$omp& schedule(dynamic,1000)
   do i = 1, 10000
     … work …
   end do
```
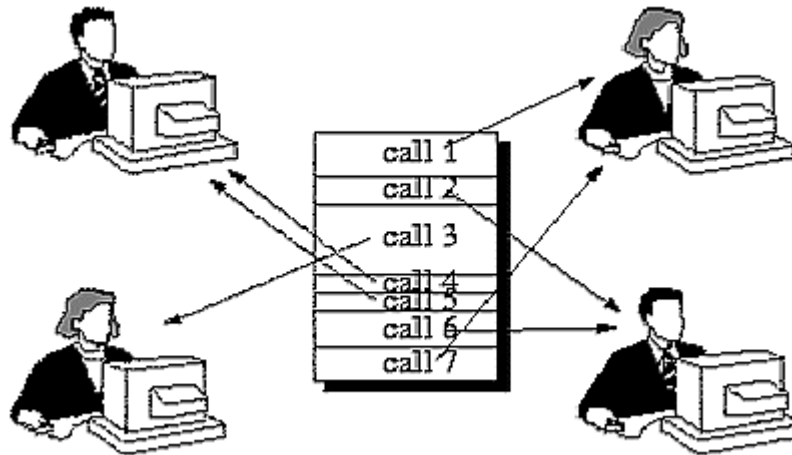
schedule(guided,chunk)

- Like `dynamic` scheduling, but the chunk size varies dynamically.
- Chunk sizes depend on the number of unassigned iterations.
- The chunk size decreases toward the specified value of `chunk`.
- Achieves good load balancing with relatively low overhead.
- Insures that no single thread will be stuck with a large number of leftovers while the others take a coffee break.

```
c$omp  do shared(x) private(i)
c$omp& schedule(guided,55)
       do i = 1, 12000
              … work …
       end do
```

`schedule(runtime)`

- Scheduling method is determined at runtime.
- Depends on the value of environment variable **OMP_SCHEDULE**
- This environment variable is checked at runtime, and the method is set accordingly.
- Scheduling method is `static` by default.
- Chunk size set as (optional) second argument of string expression.
- Useful for experimenting with different scheduling methods without recompiling.

```
[mck-login1]$ setenv OMP_SCHEDULE static,1000
[mck-login1]$ setenv OMP_SCHEDULE dynamic
```

Parallel Programming with OpenMP

`lastprivate`

- Like `private` within the parallel construct - each thread has its own copy.
- The value corresponding to the last iteration of the loop (in serial mode) is saved following the parallel construct.

```
c$omp   do shared(x)
c$omp&  lastprivate(i)
        do i = 1, N
          x(i)=a
        enddo

        n = i
```

- When the loop is finished, `i` is saved as the value corresponding to the last iteration in serial mode (i.e., `n = N + 1`).
- If `i` is declared `private` instead, the value of `n` is undefined!

`reduction(operator|intrinsic:var1[,var2])`

- Allows safe global calculation or comparison.
- A private copy of each listed variable is created and initialized depending on `operator` or `intrinsic` (e.g., 0 for +).
- Partial sums and local mins are determined by the threads in parallel.
- Partial sums are added together from one thread at a time to get gobal sum.
- Local `mins` are compared from one thread at a time to get `gmin`.

```
c$omp do shared(x) private(i)
c$omp&   reduction(+:sum)
      do i = 1, N
          sum = sum + x(i)
      enddo


c$omp do shared(x) private(i)
c$omp&   reduction(min:gmin)
      do i = 1,N
          gmin = min(gmin,x(i))
      end do
```

`reduction(operator|intrinsic:var1[,var2])`

- Listed variables must be `shared` in the enclosing parallel context.
- In Fortran
  - `operator` can be **+, *, -, .and., .or., .eqv., .neqv.**
  - `intrinsic` can be **max, min, iand, ior, ieor**
- In C
  - `operator` can be **+, *, -, &, ^, |, &&, ||**
  - pointers and reference variables are not allowed in reductions!

OSC

OpenMP Work-Sharing Constructs - `sections`

```
c$omp parallel
c$omp sections

c$omp section
      call computeXpart()
c$omp section
      call computeYpart()
c$omp section
      call computeZpart()

c$omp end sections
c$omp end parallel
      call sum()
```

- Each parallel `section` is run on a separate thread
- Allows functional decomposition
- Implicit barrier at the end of the `sections` construct
  - Use the `nowait` clause to suppress this

OpenMP Work-Sharing Constructs - `sections`

- Fortran syntax:

```
c$omp sections [clause[,clause]...]
c$omp section
      code block
[c$omp section
      another code block
[c$omp section
      …]]
c$omp end sections [nowait]
```

- Valid clauses:
  - `private(`*list*`)`
  - `firstprivate(`*list*`)`
  - `lastprivate(`*list*`)`
  - `reduction(`*operator|intrinsic:list*`)`

OpenMP Work Sharing Constructs - `sections`

- C syntax:

```
#pragma omp sections [clause [clause...]]
{
    #pragma omp section
        structured block
  [#pragma omp section
        structured block
   …]
}
```

- Valid clauses:
  - private(*list*)
  - firstprivate(*list*)
  - lastprivate(*list*)
  - reduction(*operator:list*)
  - nowait

OpenMP Work Sharing Constructs - `single`

- Ensures that a code block is executed by only one thread in a parallel region.
- The thread that reaches the `single` directive first is the one that executes the `single` block.
- Equivalent to a `sections` directive with a single section - but a more descriptive syntax.
- All threads in the parallel region must encounter the single directive.
- Unless `nowait` is specified, all non-involved threads wait at the end of the `single` block

```
c$omp parallel private(i) shared(a)
c$omp do
      do i = 1, n
         …work on a(i) …
      enddo

c$omp single
   … process result of do …
c$omp end single

c$omp do
      do i = 1, n
      … more work …
    enddo
c$omp end parallel
```

OSC

OpenMP Work Sharing Constructs - `single`

- Fortran syntax:

```
c$omp single [clause [clause…]]
        structured block
c$omp end single [nowait]
```

where `clause` is one of
   - `private(list)`
   - `firstprivate(list)`

OpenMP Work Sharing Constructs - `single`

- C syntax:

> **#pragma omp single** [clause [clause…]]
> *structured block*

where `clause` is one of

- `private(`*list*`)`
- `firstprivate(`*list*`)`
- `nowait`

OSC

Combined Parallel Work-Sharing Constructs

- Short cuts for specifying a parallel region that contains only one work sharing construct (a parallel `for/DO` or parallel `sections`).

- Semantically equivalent to declaring a parallel section followed immediately by the relevant work-sharing construct.

- All clauses valid for a parallel section and for the relevant work-sharing construct are allowed, except `nowait.`

  – The end of a parallel section contains an implicit barrier anyway.

Parallel `DO`/`for` Directive

```
c$omp parallel do [clause [clause…]]
        do loop
[c$omp end parallel do]
```

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

Parallel `sections` Directive

```
c$omp parallel sections [clause [clause…]]
[c$omp section]
        structured block
[c$omp section
        structured block]

…
c$omp end parallel sections
```

```
#pragma omp parallel sections [clause [clause…]]
{
    [#pragma omp section]
        structured block
    [#pragma omp section
        structured block
     …]
}
```

OpenMP Environment Variables

- **OMP_NUM_THREADS**
  - Sets the number of threads requested for parallel regions.

- **OMP_SCHEDULE**
  - Set to a string value which controls parallel loop scheduling at runtime.
  - Only loops that have schedule type RUNTIME are affected.

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads actually used in a parallel region (due to system load).
  - Default value is implementation dependent.

- **OMP_NESTED**
  - Enables or disables nested parallelism.
  - Default value is FALSE (nesting disabled).

OpenMP Environment Variables

- Examples:

```
[mck-login1]$ export OMP_NUM_THREADS=16

[mck-login1]$ setenv OMP_SCHEDULE "guided,4"

[mck-login1]$ export OMP_DYNAMIC=false

[mck-login1]$ setenv OMP_NESTED TRUE
```

Note: values are case-insensitive!

Parallel Programming with OpenMP

OpenMP Runtime Environment Routines

- **(void) omp_set_num_threads(int *num_threads*)**
  - Sets the number of threads to be requested for subsequent parallel regions.

- **int omp_get_num_threads()**
  - Returns the number of threads currently in the team.

- **int omp_get_max_threads()**
  - Returns the maximum value that may be returned by `omp_get_num_threads`.
  - Generally used to allocate data structures that have a maximum size per thread when `OMP_DYNAMIC` is set to `TRUE`.

- **int omp_get_thread_num()**
  - Returns the thread number, an integer from `0` to the number of threads minus `1`.

- **int omp_get_num_procs()**
  - Returns the number of physical processors available to the program.

OSC

OpenMP Runtime Environment Routines

- **`(int/logical) omp_in_parallel()`**
  - Returns "true" (logical `.TRUE.` in Fortran; a non-zero integer in C) if called from a parallel region, "false" (logical `.FALSE.` in Fortran, 0 in C) otherwise.
- **`(void) omp_set_dynamic(`*`expr`*`)`**
  - Enables (*`expr`* is "true") or disables (*`expr`* is "false") dynamic thread allocation.
- **`(int/logical) omp_get_dynamic()`**
  - Returns "true" or "false" if dynamic thread allocation is enabled/disabled, respectively.
- **`void omp_set_nested(int/logical `*`expr`*`)`**
  - Enables (*`expr`* is "true") or disables (*`expr`* is "false") nested parallelism.
- **`(int/logical) omp_get_nested()`**
  - Returns "true" or "false" if nested parallelism is enables/disabled, respectively.

Parallel Programming with OpenMP

OpenMP Runtime Environment Routines

- In Fortran, routines that return a value (integer or logical) are functions, while those that set a value (*i.e.*, take an argument) are subroutines.

- In Fortran, functions must be declared as the appropriate datatype

- In C, be sure to **#include <omp.h>**

- Changes to the environment made by function calls have precedence over the corresponding environment variables.
  - For example, a call to **omp_set_num_threads()** overrides any value that **OMP_NUM_THREADS** may have.

Interlude: Data Dependencies

- In order for a loop to parallelize, the work done in one loop iteration cannot depend on the work done in any other iteration.

- In other words, the order of execution of loop iterations must be irrelevant.

- Loops with this property are called data independent.

- Some data dependencies may be broken by changing the code.
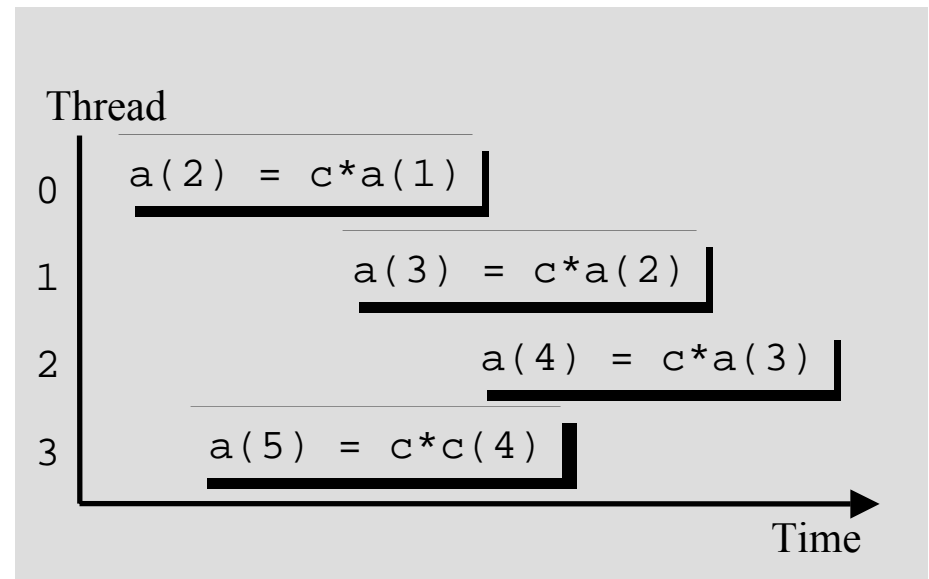
OSC

Data Dependencies (cont.)

- Only variables that are **written** in one iteration and **read** in another iteration will create data dependencies.
- A variable cannot create a dependency unless it is **shared**.
- Often data dependencies are difficult to identify. Compiler tools can help by identifying the dependencies automatically.

Recurrence:

```
do i = 2,5
    a(i) = c*a(i-1)
enddo
```

Is there a dependency here?

```
do i = 2,N,2
  a(i) = c*a(i-1)
enddo
```

Thread

0	a(2) = c*a(1)
1	a(3) = c*a(2)
2	a(4) = c*a(3)
3	a(5) = c*c(4)

Time

Parallel Programming with OpenMP

Data Dependencies (cont.)

- Unless declared as `private`, a temporary variable may be shared, and will cause a data dependency.

Temporary Variable Dependency

```
do i = 1,n
    x = cos(a(i))
    b(i) = sqrt(x * c)
enddo
```

Function Calls

```
do i = 1,n
  call myroutine(a,b,c,i)
enddo

subroutine myroutine(a,b,c,i)
…
a(i) = 0.3 * (a(i-1)+b(i)+c)
…
return
```

- In general, loops containing function calls can be parallelized.
- The programmer must make certain that the function or subroutine contains no dependencies or other side effects.
- In Fortran, make sure there are no `static` variables in the called routine.
- Intrinsic functions are safe.

Data Dependencies (cont.)

Reductions

- Similar to the temporary variable dependency, a reduction dependency is eliminated simply by using the `reduction` clause to the `parallel do` directive.

```
do i = 1,n
 xsum = xsum + a(i)
 xmul = xmul * a(i)
 xmax = max(xmax,a(i))
 xmin = min(xmin,a(i))
enddo
```

Indirect Indexing

```
do i = 1,n
  a(i) = c * a(idx(i))
enddo

do i = 1,n
  a(ndx(i)) = b(i)+c (i)
enddo
```

- If `idx(i)` not equal to `i` on every iteration, then there is a dependency.
- If `ndx(i)` ever repeats itself, there is a dependency.

Data Dependencies (cont.)

Conditional Loop Exit

- Loops with conditional exits should not be parallelized. Requires ordered execution.

```
do i = 1,n
    a(i) = b(i) + c(i)
    if (a(i).gt.amax) then
      a(i) = amax
      goto 100
    endif
enddo

100 continue
```

Nested Loop Order

```
do k = 1, n
 do j = 1, n
  do i = 1, n
   a(i,j)=a(i,j)+b(i,k)*c(k,j)
  enddo
 enddo
enddo
```

- If the k-loop is parallelized, then there is a dependency related to a(i,j)
- This can be fixed by making the k-loop the innermost loop

Parallel Programming with OpenMP

Minimizing the Cost of a Recurrence

- Move the dependency into a separate loop.
- Parallelize the loop without the dependency.
- Make sure benefits outweigh the cost of loop overhead.

```
do i = 1, NHUGE
   a(i) = ...lots of math...
 &        + a(i-1)
   enddo
```

$\longrightarrow$

```
c
c   Parallel Loop
c
c$omp  parallel do shared(junk)
c$omp& private(i)
      do i = 1, NHUGE
       junk(i) = ...lots of math...
      enddo

c
c   Serial Loop
c
      do i = 1, NHUGE
       a(i) = junk(i) + a(i-1)
      enddo
```

OSC

Loop Nest Parallelization Possibilities

All examples shown run on 8 threads with `schedule(static)`

- Parallelize the outer loop:

```
!$omp parallel do private(i,j) shared(a)
      do i=1,16
        do j=1,16
          a(i,j) = i+j
        enddo
      enddo
```

- Each thread gets two values of `i` (T0 gets `i`=1,2; T1 gets `i`=3,4, etc.) and *all* values of `j`

Loop Nest Parallelization Possibilities

- Parallelize the inner loop:

```
        do i=1,16
!$omp parallel do private(j) shared(a,i)
          do j=1,16
            a(i,j) = i+j
          enddo
        enddo
```

- Each thread gets two values of j (T0 gets j=1,2; T1 gets j=3,4, etc.) and *all* values of i

OpenMP Synchronization Constructs

- [critical](#)
- [atomic](#)
- [barrier](#)
- [master](#)
- [ordered](#)
- [flush](#)

OpenMP Synchronization - `critical` Section

- Ensures that a code block is executed by only one thread at a time in a parallel region

- Syntax:

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

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

- When one thread is in the critical region, the others wait until the thread inside exits the critical section.

- *name* identifies the critical region.

- Multiple critical sections are independent of one another unless they use the same name.

- All unnamed critical regions are considered to have the same identity.

OSC

OpenMP Synchronization - `critical` Section Example

```
integer :: cnt1, cnt2

c$omp parallel private(i)
c$omp& shared(cnt1,cnt2)

c$omp do
   do i = 1, n
      …do work…
      if(condition1)then
c$omp critical (name1)
         cnt1 = cnt1+1
c$omp end critical (name1)
      else
c$omp critical (name1)
         cnt1 = cnt1-1
c$omp end critical (name1)
      endif
      if(condition2)then
c$omp critical (name2)
         cnt2 =cnt2+1
c$omp end critical (name2)
      endif
   enddo
c$omp end parallel
```

OSC

OpenMP - Critical Section Problem

Is this correct?

What about this?

```
…

c$omp parallel do
      do i = 1,n
        if (a(i).gt.xmax) then
c$omp critical
        xmax = a(i)
c$omp end critical
        endif
      enddo
…
```

```
…

c$omp parallel do
      do i = 1,n
c$omp critical
        if (a(i).gt.xmax) then
        xmax = a(i)
        endif
c$omp end critical
      enddo
…
```

OpenMP Synchronization - `atomic` Update

- Prevents a thread that is in the process of (1) accessing, (2) changing, and (3) restoring values in a shared memory location from being interrupted at any stage by another thread.

- Syntax:

```
#pragma omp atomic
        statement
```

```
!$omp atomic
        statement
```

- Alternative to using the `reduction` clause (it applies to same kinds of expressions).

- Directive in effect only for the code statement immediately following it.

OSC

OpenMP Synchronization - `atomic` Update

```fortran
integer, dimension(8) :: a,index
data index/1,1,2,3,1,4,1,5/

c$omp parallel private(i),shared(a,index)
c$omp do
  do i = 1, 8
c$omp  atomic
    a(index(I)) = a(index(I)) + index(I)
  enddo
c$omp end parallel
```

OpenMP Synchronization - `barrier`

- Causes threads to stop until all threads have reached the barrier.
- Syntax:

```
!$omp barrier
```

```
#pragma omp barrier
```

- A red light until all threads arrive, then it turns green.
- Example:

```
c$omp parallel
c$omp do
      do i = 1, N
        <assignment>
c$omp barrier
        <dependent work>
      enddo
c$omp end parallel
```

OSC

OpenMP Synchronization - `master` Region

- Code in a `master` region is executed only by the master thread.

- Syntax:

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

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

- Other threads skip over entire master region (no implicit barrier!).

OpenMP Synchronization - `master` Region

```
!$omp parallel shared(c,scale) &
!$omp private(j,myid)
      myid=omp_get_thread_num()
!$omp master
   print *,'T:',myid,' enter scale'
   read *,scale
!$omp end master
!$omp barrier
!$omp do
   do j = 1, N
      c(j) = scale * c(j)
   enddo
!$omp end do
!$omp end parallel
```

OpenMP Synchronization - `ordered` Region

- Within an `ordered` region, loop iterations are forced to be executed in sequential order.

- Syntax:

```
c$omp ordered
      structured block
c$omp end ordered
```

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

- An `ordered` region can only appear in a parallel loop.

- The parallel loop directive must contain the `ordered` clause (new).

- Threads enter the ordered region one at a time.

OpenMP Synchronization - `ordered` Region

```
   integer, external :: omp_get_thread_num
   call omp_set_num_threads(4)
c$omp parallel private(myid)
   myid=omp_get_thread_num()
c$omp do private(i) ordered
   do i = 1, 8
c$omp ordered
      print *,'T:',myid,' i=',i
c$omp end ordered
   enddo
c$omp end parallel
--------------------
T:0 i=1
T:0 i=2
T:1 i=3
T:1 i=4
T:2 i=5
T:2 i=6
T:3 i=7
T:3 i=8
```

OpenMP Synchronization - `flush` Directive

- Causes the present value of the named shared variable to be immediately written back ("flushed") to memory.

- Syntax:

```
c$omp flush(var1[,var2]…)
```

```
#pragma omp flush(var1[,var2]…)
```

- Enables signaling between threads by using a shared variable as a semaphore.

- When other threads see that the shared variable has been changed, they know that an event has occurred and proceed accordingly.

OSC

Sample Program: `flush` Directive

```fortran
program flush
   integer, parameter :: M=1600000
   integer, dimension(M) :: c
   integer :: stop,sum,tid
   integer, dimension(0:1) :: done
   integer, external :: omp_get_thread_num

   call omp_set_num_threads(2)
   c=1
   c(345)=9
!$omp parallel default(private) shared(done,c,stop)
   tid=omp_get_thread_num()
   done(tid)=0
   if(tid==0) then
      neigh=1
   else
      neigh=0
   end if
!$omp barrier
```

Sample Program: `flush` Directive (cont.)

_____

```fortran
 if (tid==0) then
     do j=1,M
        if(c(j)==9) stop=j
     end do
  end if
  done(tid)=1

!$omp flush(done)
  do while(done(neigh).eq.0)
!$omp flush(done)

  end do

  if (tid==1) then
     sum=0
     do j=1,stop-1
        sum=sum+c(j)
     end do
  end if
!$omp end parallel

end program flush
```

Some Advanced Features of OpenMP

- Advanced data scoping: the `threadprivate` directive
- "Orphaning" OpenMP directives
- Advanced synchronization: lock functions

Advanced Data Scoping - `threadprivate` Directive
(Fortran)

- Can a thread keep its own private variables throughout every parallel section in a program? Yes!

- Put the desired variables in a common block and declare that common block to be **threadprivate.**

- Makes common blocks private to individual threads but global within each thread

- Syntax:

```
c$omp threadprivate(/cb/[,/cb2/…])
```

- `threadprivate` directive must appear after the common block declaration.

- `threadprivate` variables may only appear in the `copyin` clause.

- For `threadprivate` variables to persist over several parallel regions, must use static execution mode and the same number of threads in every region.

Advanced Data Scoping - `threadprivate` Directive (C/C++)

- In C, `threadprivate` applies to file-scope and static variables

- Makes them private to individual threads, but global within each thread.

- Syntax:

  `#pragma omp threadprivate(var1,var2,…)`

- The `threadprivate` directive must appear after the declarations of the specified variables but before any references to them, and must itself be at file (or namespace) scope.

- Threadprivate variables can only appear in the `copyin`, `schedule` and `if` clauses.

- For `threadprivate` variables to persist over several parallel regions, must use static execution mode and the same number of threads in every region.

Sample Program: `threadprivate`

```fortran
program region
  integer,external :: omp_get_thread_num
  integer :: tid,x
  common/mine/x

!$omp threadprivate(/mine/)
  call omp_set_num_threads(4)

!$omp parallel private(tid)
  tid=omp_get_thread_num()
  x=tid*10+1
  print *,"T:",tid," inside first parallel region x=",x
!$omp end parallel

  print *,"T:",tid," outside parallel region x=",x

!$omp parallel private(tid)
  tid=omp_get_thread_num()
  print *,"T:",tid," inside next parallel region x=",x
!$omp end parallel

end program region
```

Sample Program: `threadprivate`

```
[jimg@mck-login1 mck]$ cat threadprivate.job
#PBS -N threadprivate
#PBS -l walltime=20:00
#PBS -l ncpus=4
#PBS -j oe
cd $PBS_O_WORKDIR
efc -openmp threadprivate.f90 -o threadprivate
./threadprivate


[jimg@mck-login1 mck]$ qsub threadprivate.job
46188.nfs1.osc.edu


[jimg@mck-login1 mck]$ cat threadprivate.o46188
efc: Command line warning: openmp requires C style preprocessing; fpp level is reset to 2
   program REGION
threadprivate.f90(10) : (col. 0) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
threadprivate.f90(18) : (col. 0) remark: OpenMP DEFINED REGION WAS PARALLELIZED.


27 Lines Compiled
 T:          0  inside first parallel region x=          1
 T:          1  inside first parallel region x=         11
 T:          2  inside first parallel region x=         21
 T:          3  inside first parallel region x=         31
 T:          0  outside parallel region x=          1
 T:          0  inside next parallel region x=          1
 T:          2  inside next parallel region x=         21
 T:          3  inside next parallel region x=         31
 T:          1  inside next parallel region x=         11
```

Parallel Programming with OpenMP

Initializing `threadprivate` Variables - The `copyin` Clause

- Causes `threadprivate` variables to be given the master thread's values at the onset of parallel code.

- Fortran syntax:

```
copyin(/cb/[,/cb2/…])
```

  C syntax:

```
copyin(var1,var2,…)
```

- Note: `copyin` is also a valid clause for `parallel do` loops and the `parallel sections` construct.

OSC

Sample Program: The `copyin` Clause

```fortran
  integer :: x,tid
  integer, external :: omp_get_thread_num()
  common/mine/x
!$omp threadprivate(/mine/)
  x=33
  call omp_set_num_threads(4)
!$omp parallel private(tid) copyin(/mine/)
  tid=omp_get_thread_num()
  print *,'T:',tid,' x=',x
!$omp end parallel
----------------------
T:1 i=33
T:2 i=33
T:0 i=33
T:3 i=33
```

"Orphaning" OpenMP Directives

- Parallel work initiated in a parallel region does not have to be actually performed within the region's "lexical" scope.

- Work can be "orphaned" out of the parallel region via a subroutine/function call.

Sample Program: Orphaned `parallel do`

```fortran
program orphan
  integer, parameter :: M=8
  integer, dimension(M) :: x
  integer :: myid,i
  common/global/x,myid,i

  call omp_set_num_threads(4)
!$omp parallel shared(x)
  call work()
!$omp end parallel
  write(6,*) x
end program orphan

subroutine work()
  integer, parameter :: M=8
  integer, dimension(M) :: x
  integer,external :: omp_get_thread_num
  common/global/x,myid,i
!$omp do private(i,myid)
  do i=1,M
     myid=omp_get_thread_num()
     write(6,*) "T:",myid," i=",i
     x(i)=myid
  end do
!$omp end do
  return
end subroutine work
```

Sample Program: Output

```
[jimg@mck-login1 mck]$ qsub orphan.job
46211.nfs1.osc.edu

[jimg@mck-login1 mck]$ more orphan.o46211
Warning: no access to tty (Bad file descriptor).
Thus no job control in this shell.
efc: Command line warning: openmp requires C style preprocessing; fpp
level is reset to 2
    program ORPHAN
orphan.f90(8) : (col. 0) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
    external subroutine WORK
orphan.f90(19) : (col. 0) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.

28 Lines Compiled
 T:           0  i=           1
 T:           0  i=           2
 T:           1  i=           3
 T:           1  i=           4
 T:           2  i=           5
 T:           2  i=           6
 T:           3  i=           7
 T:           3  i=           8
        0              0          1          1          2          2
        3              3
```

Advanced Synchronization: Lock Functions (C/C++)

- **`void omp_init_lock(omp_lock_t *lock);`**
  - Initializes the lock associated with the parameter `lock`
- **`void omp_destroy_lock(omp_lock_t *lock);`**
  - Ensures the lock variable **`lock`** is uninitialized
- **`void omp_set_lock(omp_lock_t *lock);`**
  - Blocks the thread executing the function until `lock` is available, then sets the lock and proceeds.
- **`void omp_unset_lock(omp_lock_t *lock);`**
  - Releases ownership of **`lock`**
- **`integer omp_test_lock(omp_lock_t *lock);`**
  - Tries to set the lock, but does not block the thread from executing.
  - Returns non-zero ("true") if the lock was successfully set.
- Must **`#include <omp.h>`**

OSC

Advanced Synchronization: Lock Functions (Fortran)

- **subroutine omp_init_lock(lock)**
    - Initializes the lock associated with the parameter **lock**
- **subroutine omp_destroy_lock(lock)**
    - Ensures the lock variable **lock** is uninitialized
- **subroutine omp_set_lock(lock)**
    - Blocks the thread executing the function until `lock` is available, then sets the lock and proceeds.
- **subroutine omp_unset_lock(lock)**
    - Releases ownership of **lock**
- **logical function omp_test_lock(lock);**
    - Tries to set the lock, but does not block the thread from executing
    - Returns **.TRUE.** if the lock was successfully set
- **lock** should be an integer of a `KIND` large enough to hold an address.

Lock Functions: Example

```c
#include <omp.h>
void main()
{
        omp_lock_t lock;
        int myid;
        omp_init_lock(&lock);
        #pragma omp parallel shared(lock) private(myid)
        {
          myid = omp_get_thread_num();
          omp_set_lock(&lock);
          printf("Hello from thread %d\n", myid);
          omp_unset_lock(&lock);

          while (! omp_test_lock(&lock)) {
                skip(myid);
          }
          do_work(myid);
          omp_unset_lock(&lock);
        }
        omp_destroy_lock(&lock);
}
```

OSC

Debugging OpenMP Code

- [Examples: race conditions](#)
- [Examples: deadlock](#)
- [Other danger zones](#)

Debugging OpenMP Code

- Shared memory parallel programming opens up a range of new programming errors arising from unanticipated conflicts between shared resources

- Race Conditions
  - When the outcome of a program depends on the detailed timing of the threads in the team.

- Deadlock
  - When threads hang while waiting on a locked resource that will never become available.

OSC

Examples: Race Conditions

```
c$omp parallel sections
        A = B + C
c$omp section
        B = A + C
c$omp section
        C = B + A
c$omp end parallel sections
```

- The result varies unpredictably depending on the order in which threads execute the sections.

- Wrong answers are produced without warning!

Examples: Race Conditions

```
c$omp parallel shared(x) private(tmp)
        id = omp_get_thread_num()
c$omp do reduction(+:x)
        do j=1,100
            tmp = work(j)
            x = x + tmp
        enddo
c$omp end do nowait
        y(id) = work(x,id)
c$omp end parallel
```

- The result varies unpredictably because the value of x isn't correct until the barrier at the end of the do loop is reached.

- Wrong answers are produced without warning!

- Be careful when using nowait!

Examples: Race Conditions

```fortran
  real :: tmp,x
c$omp parallel do reduction(+:x)
  do j=1,100
    tmp = work(j)
    x = x + tmp
  enddo
c$omp end do
  y(id) = work(x,id)
```

- The result varies unpredictably because access to the shared variable `tmp` is not protected.
- Wrong answers are produced without warning!
- Probably want to make `tmp` private.

Examples: Deadlock

```
        call OMP_INIT_LOCK(lcka)
        call OMP_INIT_LOCK(lckb)
c$omp parallel sections
        call OMP_SET_LOCK(lcka)
        call OMP_SET_LOCK(lckb)
        call useAandB(res)
        call OMP_UNSET_LOCK(lckb)
        call OMP_UNSET_LOCK(lcka)
c$omp section
        call OMP_SET_LOCK(lckb)
        call OMP_SET_LOCK(lcka)
        call useBandA(res)
        call OMP_UNSET_LOCK(lcka)
        call OMP_UNSET_LOCK(lckb)
c$omp end parallel sections
```

- If A is locked by one thread and B by another, you have deadlock.
- If both are locked by the same thread, you have a race condition!
- Avoid nesting different locks.

OSC

Examples: Deadlock

```
        call OMP_INIT_LOCK(lcka)
c$omp parallel sections
        call OMP_SET_LOCK(lcka)
        ival = work()
        if (ival.eq.tol) then
          call OMP_UNSET_LOCK(lcka)
        else
          call error(ival)
        endif
c$omp section
        call OMP_SET_LOCK(lcka)
        call useBandA(res)
        call OMP_UNSET_LOCK(lcka)
c$omp end parallel sections
```

- If A is locked in the first section and the if statement branches around the unset lock, then threads in the other section will deadlock waiting for the lock to be released.

- Make sure you release your locks!

Other Danger Zones

- Are the libraries you are using thread-safe?
  - Standard libraries should always be okay.
- I/O inside a parallel region can interleave unpredictably.
- `private` variables can mask globals.
- Understand when shared memory is coherent.
  - When in doubt, use `FLUSH`
- `NOWAIT` removes implicit barriers.

OSC

Performance Tuning and OpenMP

- [Basic strategies](#)
- [Automatic parallelization](#)
- [Example 1](#)
- [Example 2](#)
- [The memory hierarchy](#)
- [Cache locality](#)
- [Data locality](#)

Basic Strategies

- If possible, use auto-parallelizing compiler as a first step
- Use profiling to identify time-consuming code sections (loops)
- Add OpenMP directives to parallelize the most important loops
- If a parallelized loop does not perform well, check for/consider
  - Parallel startup costs
  - Small loops
  - Load imbalances
  - Many references to shared variables
  - Low cache affinity
  - Unnecessary synchronization
  - Costly remote memory references (in NUMA machines)

Automatic Parallelization

- The major languages often have versions of their compilers which will automatically parallelize your code.

- The compiler stage that performs this is called the Automatic Parallelizer (AP).

- The AP will insert OpenMP directives into your code if a loop can be parallelized.   If not, it will tell you why.

- "Safe" parallel optimization implies there are no dependencies.

- Only loops can be parallelized automatically.

- Should be considered, at best, as a first step toward getting your code parallelized

- The next step should be inserting your own directives, and tuning the various parallel sections for optimum performance.

OSC

Strategy for Using Auto-Parallelization

- Run AP on source files, and examine the listing.
  - Convenient to break code up into separate source files (use `fsplit(1)` and `make(1))`.
- For loops that don't automatically parallelize, try to eliminate inhibiting dependencies by modifying the source code.
- Use the listing to implement parallelization by hand using OpenMP directives.
- Stop when you are satisfied with performance.

Performance Tuning: Example 1

- Original code:

```
c1 = x(1)>0
c2 = x(1:10)>0

DO i=1,n
   DO j=i,n
      if (c1) then r(1:100) = …

       …
      if (c2) then … = r(1:100)
      sum(j) = sum(j) + …
   ENDDO
ENDDO
```

Example 1 (cont.)

_____

- First, parallelize the loop.
  - Prefer to parallelize the outer loop - higher iteration count
  - Note $c2$ is never true unless $c1$ is also true - can make r private!
  - Also parallelize the reduction
- But, the loop is "triangular"! By default, iterations may be unbalanced between processors.
  - Use the schedule clause to enforce more efficient load balancing

Example 1 - Parallel Version

```
c1 = x(1)>0
c2 = x(1:10)>0
ALLOCATE(xsum(1:nprocs,n))

c$omp  parallel do private(i,j,r,myid)
c$omp& schedule(static,1)
DO i=1,n
   myid = omp_get_thread_num()
   DO j=i,n
      if (c1) then r(1:100) = …

       …
      if (c2) then … = r(1:100)
      xsum(myid,j) = sum(myid,j) + …
   ENDDO
ENDDO

c$omp parallel do
DO i=1,n
   sum(i) = sum(i) + xsum(1:nprocs,i)
ENDDO
```

Performance Tuning: Example 2

- Increasing parallel loop granularity using the `nowait` clause:

```fortran
!$omp parallel private(ld1,ld2,ldi,j,ld,k)
      do k = 2,ku-2
!$omp do
        do j = jlo, jhi
            ld2 = a(j,k)
            ld1 = b(j,k)+ld2*x(j,k-2)
            ld  = c(j,k)+ld1*x(j,k-1)+ld2*y(j,k-1)
            ldi = 1./ld
            f(j,k,1) = ldi*(f(j,k,1)-f(j,k-2,1)*ld2
            f(j,k,1) = ldi*(f(j,k,2)-f(j,k-2,2)*ld1
            x(j,k) = ldi*(d(j,k)-y(j,k-1)*ld1
            y(j,k) = e(j,k)*ld
          enddo
!$omp end do nowait
      end do
!$omp end parallel
```

The Memory Hierarchy

- Most parallel systems are built from CPUs with a memory hierarchy
  - Registers
  - Primary cache
  - Secondary cache
  - Local memory
  - Remote memory - access through the interconnection network
- As you move down this list, the time to retrieve data increases by about an order of magnitude for each step.
- Therefore:
  - Make efficient use of local memory (caches)
 - Minimize remote memory references

OSC

Performance Tuning - Cache Locality

- The basic rule for efficient use of local memory (caches):

  Use a memory stride of one

- This means array elements are accessed in the same order they are stored in memory.

- Fortran: "Column-major" order
  - Want the leftmost index in a multi-dimensional array varying most rapidly in a loop

- C: "Row-major" order
  - Want rightmost index in a multi-dimensional array varying most rapidly in a loop

- Interchange nested loops if necessary (and possible!) to achieve the preferred order.

OSC

Performance Tuning - Data Locality

- On NUMA ("non-uniform memory access") platforms, it may be important to know
    - Where threads are running
    - What data is in their local memories
    - The cost of remote memory references
- OpenMP itself provides no mechanisms for controlling
    - the binding of threads to particular processors
    - the placement of data in particular memories
- Designed with true (UMA) SMP in mind
    - For NUMA, the possibilities are many and highly machine-dependent
- Often there are system-specific mechanisms for addressing these problems
    - Additional directives for data placement
    - Ways to control where individual threads are running

Altix 3000: Architecture
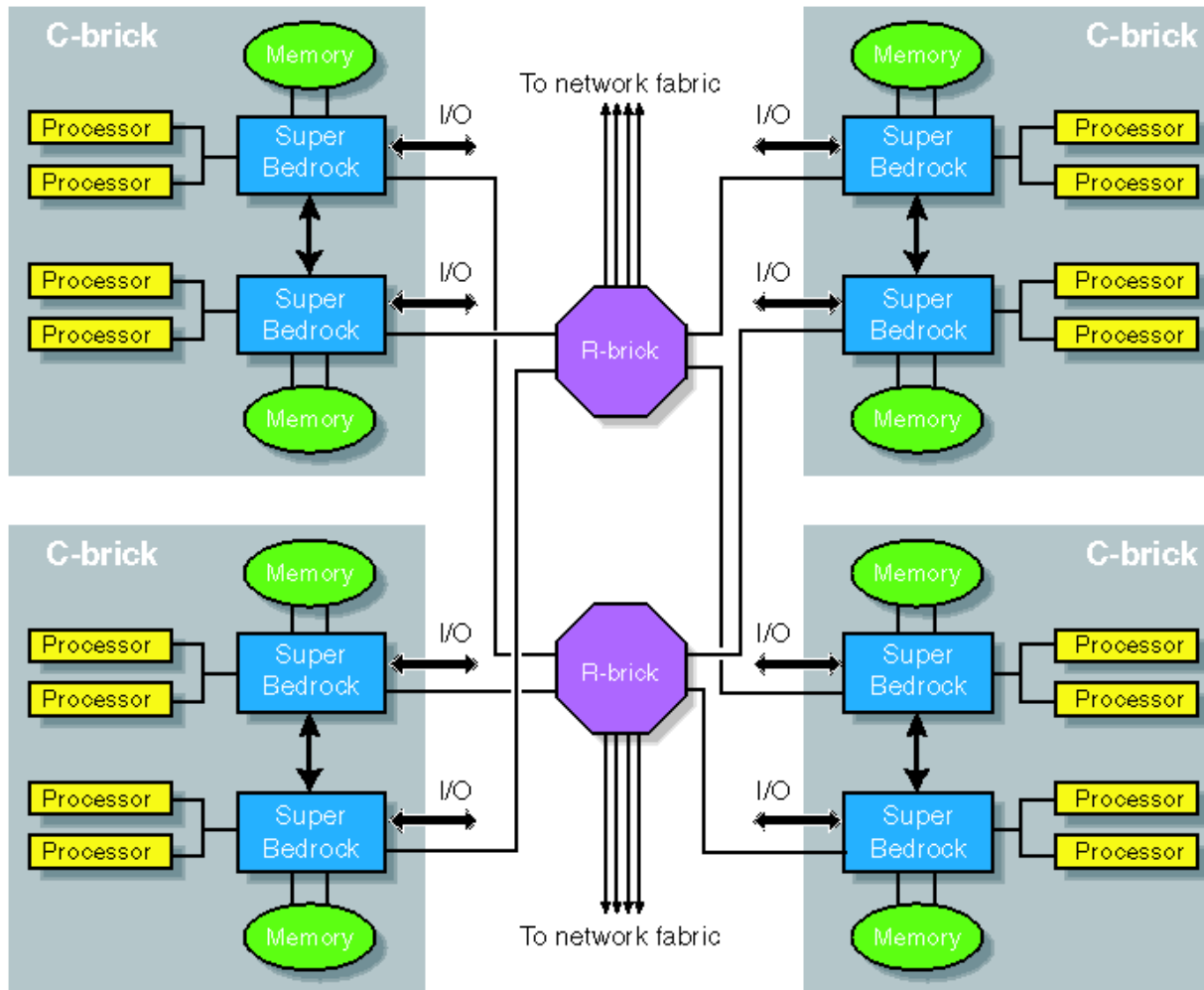
References

- Official OpenMP site: www.openmp.org
  - Contains the complete OpenMP specifications for Fortran and C/C++
  - News of future developments
  - Sample programs
  - Links to other sites of interest

OpenMP Problem Set

✦ Write a program where each thread prints the message 'Hello World!', along with its thread ID number and the total number of threads used. Run with 8 threads and run your program several times. Does the order of the output change? Repeat using 4,16, 33 and 50 threads

✦ Modify your solution to Problem 1 so that only even-numbered threads print out the information message.

✦ Write a program that declares an array A to have 16000 integer elements and initialize `A` so that each element has its index as its value. Then create a real array `B` which will contain the running average of array `A`. That is,

$$B(I) = (A(I-1) + A(I) + A(I+1)/3.0$$

except at the end points. Your code should do the initialization of `A` and the running average in parallel using 8 threads. Experiment with all four of scheduling types for the running average loop by timing the loop with different schedules.

❋ Write a program so that the parallel threads print out 'Backwards' and their thread ID number in **reverse order** of thread number. That is, each time your program is run the last thread prints out first, then the second to last and so on. There are at least five different ways to solve this problem. Find as many as you can.

⊕ Compile the code `mystery.f` and run on 16 threads. What is wrong with this program? (You may have to run it several times). Fix the code so that it works correctly. As with problem 4 there are several ways to fix the code, try to find them all.

⊕ Write a program to read in the x,y,z coordinates from a file `points.dat` (which you will be given) and calculate the geometric center which is the average x value, the average y value, and the average z value. Do the calculation in parallel. Write two versions of your program: the first using loop-level parallelism, the next using functional decomposition. (The points data file is ASCII with one x,y,z triplet per line)

✧ Using the functional decomposition version of program 6, calculate the average coordinate value given by the equation

$$(\Sigma\, x_i + \Sigma\, y_i + \Sigma\, z_i)/3N$$

where N is the number of data points. Implement using a global sum and critical regions.

⌧ Write a program to multiply two large matrices together.

 a) Compile for single-processor execution. Time the program

 b) Compile for multiple processor execution (OpenMP directives) and time for 4,8 and 12 processors

◈ Compile the program `alias.f` and run on four threads. Can you see the inefficiency in the program? Write a new version that is more efficient.

Appendix A: Auto-Parallelization on the Altix 3000

- [Using the auto-parallelizer](#)

- [Auto-parallelizer files](#)

- [Examples](#)

Using the Auto-Parallelizer

- Syntax:

  efc –parallel –par_report2 prog.f –o prog

- What the options do:

  ```
  -parallel => enable the auto-parallelizer to generate
     multi-threaded code for loops that can be safely
     executed in parallel
  ```

  ```
  -par_report{0|1|2|3} => control the auto-parallelizer
     diagnostic level
  ```

- Other compiler options can also be used

OSC

Example Subroutine (successful)

- Original source code (`mysub.f`)

```
subroutine mysub(a,b,c)
  real, dimension(1000) :: a,b,c
  do i=1,1000
    a(i)=b(i)+c(i)
  end do
  return
end subroutine mysub
```

- AP command:

efc –parallel –par_report2 –c mysub.f90

- Performance report

```
external subroutine MYSUB
procedure: mysub
mysub.f90(3) : (col. 0) remark: LOOP WAS AUTO-PARALLELIZED.
parallel loop: line 3


7 Lines Compiled
```

Data Dependence Example - Indirect Indexing

- Original source code (`indirect.f`):

```fortran
subroutine indirect(a,b,c,idx)
   real, dimension(1000)::a,b,c
   integer, dimension(1000)::idx
   do i=1,1000
       a(idx(i))=a(idx(i))+c(i)
   end do
   return
end subroutine indirect
```

- Parallelization report:

```
[mck-login1]$ efc -parallel -par_report3 -c indirect.f90
   external subroutine INDIRECT
   procedure: indirect
   serial loop: line 4
       anti data dependence assumed from line 5 to line 5, due to "a"
       output data dependence assumed from line 5 to line 5, due to "a"
       flow data dependence assumed from line 5 to line 5, due to "a"
```

Parallel Programming with OpenMP

Data Dependence Example - Function Call

- Original source code (`func.f`):

```
subroutine func(a,b,c)
   real, dimension(1000)::a,b,c
   external xfunc
   do i=1,1000
       a(i)=xfunc(b(i),c(i))
   enddo
   return
end subroutine func
```

- Parallelization report:

```
[mck-login1]$ efc -parallel -par_report2 -c funct.f90
   external subroutine FUNC
   procedure: func
   serial loop: line 4: not a parallel candidate due to statement
   at line 5

 8 Lines Compiled
```