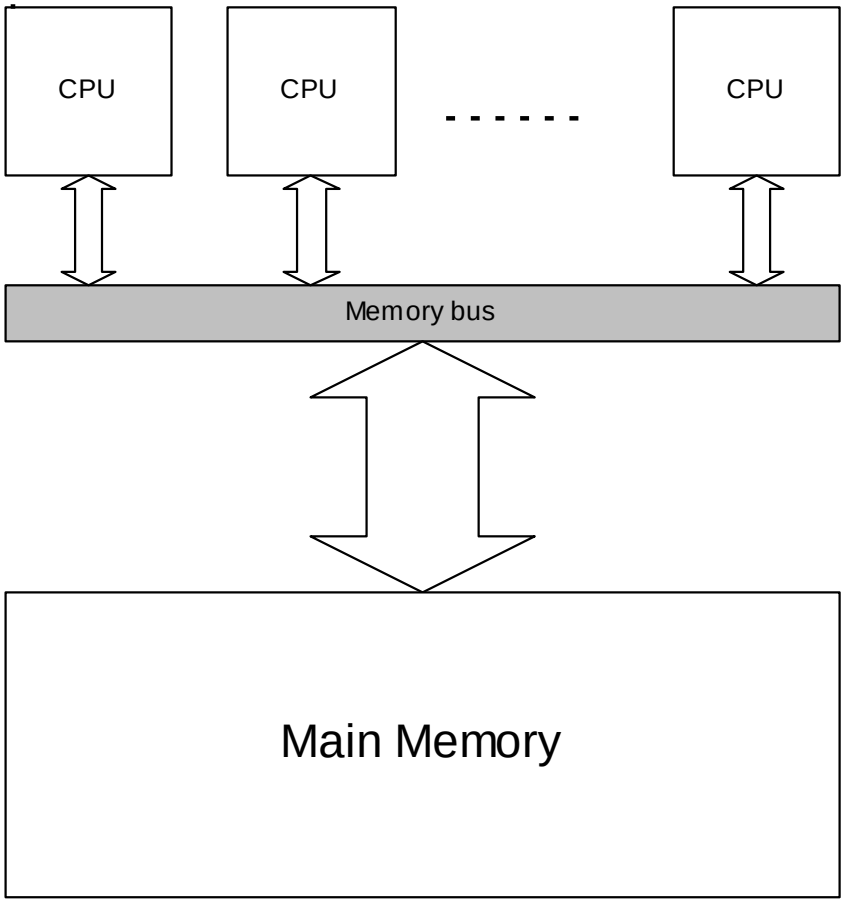
Shared Memory Multiprocessors

Shared Memory Multiprocessor



Programming Model

- Primary programming model
  - Parallel *threads* of control
    - each running on a separate processor of the SMP computer
    - sharing memory with other threads in the framework of the same process
- A *multi-threaded* (MT) program
  - All threads share the same process-level structures and data (file descriptors, user ID, etc.)
  - => Threads have access to all the same functions, data, open files, etc.

Programming Model (ctd)

- MT program
  - Starts up with one initial *main* thread
  - Each thread may create new threads
    - by calling the create routine, passing a routine for that new thread to run
    - the new thread now runs the routine and provides another stream of instructions operating on data in the same address space
  - Threads coordinate the usage of shared data via synchronization variables such as a *mut*ual *ex*clusion lock (a *mutex*)
  - Another way for threads to synchronize their work is to direct signals internally to individual threads

Programming Model (ctd)

- Secondary programming model
  - Parallel *processes*
    - each running on a separate processor
    - not sharing main memory with other processes
    - using message passing to communicate with the others in order to coordinate their work
  - Is a primary one for the distributed-memory multiprocessor architecture

Shared Memory Multiprocessor (ctd)

- SMP architecture
  - Provides more parallelism than VPs and SPs
  - Adds parallel streams of instructions to the instruction-level parallelism
- How significant is the performance potential of the SMP architecture?
  - One might expect a *n*-processor SMP computer to provide *n*-fold speedup compared to its 1-processor configuration
  - The real picture is different

Shared Memory Multiprocessor (ctd)

- What do you see in reality
  - Let you start from a 1-processor configuration and add processors one by one
  - Each next processor is adding only a fraction of the performance that you got from the first
  - The fraction is becoming smaller and smaller
  - Eventually, adding one more processor will just decrease performance
- This cannot be explained by the Amdahl law
  - "Normal" programs see other limitations far before they ever hit this one

Shared Memory Multiprocessor (ctd)

- The real bottleneck is the memory bus
- Heavy matrix multiplication programs come up against the limited memory bandwidth very quickly
  - Let 30 possible processors average one main memory reference every 90 bus cycles (which is quite possible for such programs)
  - Then there will be a reference every third cycle
  - If a request/reply occupies the bus for ~6 cycles, then that will be already twice what the bus can handle

Optimising Compilers

Optimising Compilers

- The main specific optimisation is *loop parallelization*
  - Different iterations are simultaneously executed by different parallel threads
- The most difficult problem is the recognition of parallelizable loops
  - Based on the analysis of data dependencies in loops
  - Optimising C and Fortran 77 compilers for SMPs use the same methods and algorithms as optimising C and Fortran 77 compilers for VP and SPs
- Advantages and disadvantages of the use of optimising C and Fortran 77 compilers for parallel programming are generic

Optimising Compilers (ctd)

- Can optimising C and Fortran 77 compiler be used to port legacy serial code to SMP computers?
- A good efficient serial algorithm
  - Tries to minimize redundant computations
    - Tries to maximize re-use of information, computed at each loop iteration, by all following iterations to minimize redundant computations
  - This leads to strong inter-iteration data dependences
    - Optimising compilers cannot parallelize the most principal and time-consuming loops
- Serial legacy code must be re-designed for SMPs

Thread Libraries

Thread Libraries

- Thread libraries directly implement the thread parallel programming model
  - The basic paradigm of multithreading is the same in different thread libraries (POSIX, NT, Solaris, OS/2, etc.)
  - The libraries just differ in details of implementation
- We outline the POSIX thread library, aka Pthreads
  - In 1995 Pthreads became a part of the IEEE POSIX standard => considered standard for Unix systems
  - Most hardware vendors offer Pthreads in addition to their proprietary thread libraries.

Pthreads

- Pthreads are defined as a C language library
  - A standard Fortran interface is not yet complete
- Pthreads introduce 3 classes of objects (and operations on the objects)
  - Threads
  - Mutexes
  - Condition variables

Pthreads (ctd)

- Thread
  - Represented by its ID
  - ID is a reference to an opaque data object holding full information about the thread
  - This information is used and modified by operations on threads
- Operations on threads
  - Create threads, terminate threads, join threads, etc.
  - Set and query thread attributes

Pthreads (ctd)

- Mutex
  - The primary means of thread synchronization
    - normally used when several threads update the same global data
  - Acts as a lock protecting access to the shared data resource
    - Only one thread can lock a mutex at any given time
    - If several threads simultaneously try to lock a mutex, only one will succeed and start owning that mutex
    - Other threads cannot lock that mutex until the owning thread unlocks it
  - Mutexes serialize access to the shared resource

Pthreads (ctd)

- Operations on mutexes
  - Creating
  - Destroying
  - Locking
  - Unlocking
  - Setting and modifying attributes associated with mutexes

Pthreads (ctd)

- Condition variable
  - Another way for threads to synchronize their work
  - Is a global variable shared by several threads
  - Used by threads to signal each other that some condition is satisfied
- A thread, willing to wait for some condition to be satisfied
  - Blocks itself on a condition variable, and
  - Does nothing until some other thread unblocks it
    - by performing a corresponding operation on the same condition variable as soon as the condition is satisfied

Pthreads (ctd)

- Condition variables
  - Allow the programmer to use the “cold” waiting
  - Otherwise, the thread would be constantly polling to check the condition out making the processor busy
- Operations on condition variables
  - Creating condition variables
  - Waiting and signalling on condition variables
  - Destroying condition variables
  - Set and query condition variable attributes

Operations on Threads

- An MT program starts up with one initial thread running the function `main`
  - All other threads must be explicitly created

- *Creation* of a new thread

```
int pthread_create(pthread_t *thread,
                   const  pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg)
```

Operations on Threads (ctd)

- The newly created thread
  - Runs concurrently with the calling thread
  - Executes the function `start_routine`
    - Only one argument may be passed to this function via `arg`
    - If multiple arguments must be passed, a structure may be created, which contains all of the arguments
  - `attr` specifies attributes to be applied to the new thread
    - can be `NULL` (default attributes are used)
  - ID of the thread is returned in `thread`
  - May create other threads (once created, threads are peers)
- On error, a non-zero error code is returned

Operations on Threads (ctd)

- *Learning* its own ID

```
pthread_t pthread_self(void)
```

- *Comparing* thread IDs

```
int pthread_equal(pthread_t t1, pthread_t t2)
```

- A thread can *terminate*
  - Explicitly
    - By calling function `void pthread_exit(void *status)`
  - Implicitly
    - by returning from the `start_routine` function
    - equivalent to calling `pthread_exit` with the result returned by `start_routine` as exit code

Operations on Threads (ctd)

- The exit status specified by `status`
  - Is made available to any join with the terminating thread
    - If the thread is not detached
- *Joining*
  - A synchronization operation on threads implemented by function

    ```
    int pthread_join(pthread_t t, void **status)
    ```
  - Blocks the calling thread until the thread `t` terminates
  - Returns successfully when thread `t` terminates
  - The value passed to `pthread_exit` by thread `t` will be placed in the location referenced by `status`

Operations on Threads (ctd)

- There are two types of threads – *joinable* and *detached*
  - It is impossible to join a detached thread
  - A detached or joinable state of the thread is set by using the `attr` argument in the `pthread_create` function
    - `attr` points to a variable of the `pthread_attr_t` type (an attribute variable)
- The attribute variable is an opaque data object holding all attributes of the thread

Operations on Threads (ctd)

- The attribute variable is first initialised with function

```
int pthread_attr_init(pthread_attr_t *attr)
```

  that sets the default value for all attributes

- Then the *detached status* attribute is set with function

```
int pthread_attr_setdetachstate(
        pthread_attr_t *attr, int detstat)
```

  - `detstat` can be either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE` (default value)

Operations on Threads (ctd)

- Function

```
int pthread_attr_getdetachstate(
    const pthread_attr_t *attr, int *pdetstat)
```

  retrieves the *detached status* attribute

- Function `int pthread_detach(pthread_t *t)` explicitly detaches thread t
- Using detached threads reduces the overhead
- A single attribute object can be used in multiple simultaneous calls to function `pthread_create`

Operations on Threads (ctd)

- Other attributes specify
  - the address and size for a thread's stack (allocated by system by default)
  - priority of the thread (0 by default)
  - etc
- Function

```
int pthread_attr_destroy(pthread_attr_t *attr)
```

releases resources used by the attribute object `attr`

Operations on Mutexes

- A typical sequence of operations on a mutex includes
  - Creation of a mutex
  - Initialisation of the mutex
  - Locking the mutex by one of several competing threads
    - The winner starts owning the mutex
    - The losers act depending on the type of lock operation used
      - The blocking lock operation blocks the calling thread
      - The non-blocking lock operation terminates even if the mutex has been already locked by another thread. On completion, the operation informs the calling thread whether or not it has managed to lock the mutex

Operations on Mutexes (ctd)

- A typical sequence of operations on a mutex (ctd)
  - Unlocking the mutex by its current owner
    - The mutex becomes available for locking by other competing threads
  - The mutex is destroyed
- *Creation* of a mutex <=> declaration of a variable of the `pthread_mutex_t` type
  - The variable is an opaque data object that contains all information about the mutex

Operations on Mutexes (ctd)

- *Initialisation* of the mutex
  - Dynamic
  - Static
- Dynamic initialisation
  - With function

```
int pthread_mutex_init(pthread_mutex_t *mutex,
          const pthread_mutexattr_t *mutexattr)
```

  - `mutexattr` is typically `NULL` (accept default attributes)
  - Upon successful initialisation, the state of the mutex becomes *initialised* and *unlocked*

Operations on Mutexes (ctd)

- Static initialisation
  - With the macro `PTHREAD_MUTEX_INITIALIZER` when the mutex variable is declared
  - Example.

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

  - Equivalent to dynamic initialization with `mutexattr` specified as `NULL`
- Function

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

destroys the mutex object

Operations on Mutexes (ctd)

- Function

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

locks the mutex object referenced by `mutex`

  - The calling thread blocks if the mutex is already locked
  - Returns with the mutex in the locked state with the calling thread as its owner

Operations on Mutexes (ctd)

- Function

```
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

  - Implements non-blocking lock
    - Returns immediately anyway
  - Returns 0 if it succeeds to lock the mutex
  - Returns a non-zero if the mutex is currently locked
- Function

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

releases the mutex object

Example

- Example. The dot product of two real *m*-length vectors *x* and *y* on a *n*-processor SMP computer.
- The MT application divides *x* and *y* into *n* sub-vectors
  - the first *n*-1 sub-vectors are of the same length *m/n*
  - the last *n*-th subvector may be shorter if *m* is not a multiply of *n*
- This application uses *n* parallel threads
  - *i*-th thread computes its fraction of the total dot product by multiplying sub-vectors x_i and y_i

Example (ctd)

- The *n* parallel threads
  - Share a data object accumulating the dot product
  - Synchronize their access to the data object with a mutex
- The main thread
  - Creates the *n* threads
  - Waits for them to complete their computations
    - via joining with each of the threads
  - Outputs the result
- <For the source code see handout>

Parallel Languages

Parallel Languages

- Thread libraries support efficiently portable parallel programming the SMP architecture
- But some people find their power redundant for the goals of parallel programming
  - The multi-threaded programming model underlying the libraries is universal
  - It supports both parallel and diverse distributed computing technologies

Parallel Languages (ctd)

- Most of these design approaches are mainly used in distributed programming
  - In particular, when implementing servers in server/client applications
- The *producer/consumer* design
  - Some threads create work and put it on a queue
  - Other threads take work items off the queue and execute them

Parallel Languages (ctd)

- The *pipeline* design
  - Each thread performs some part of an entire work and passes the partially completed work to the next thread
- The *personal servant* design
  - A thread is dedicated to each client, serving only that client
- The *master/slave* design
  - One thread does the main work of the program, occasionally creating other threads to help in some portion of the work

Parallel Languages (ctd)

- In mutli-threaded *parallel* programming, the master/slave design is predominantly used
  - If you want to use the MT libraries only for parallel programming, their programming model is seen too powerful and complicated (and, hence, *error-prone*)
- Parallel extensions of Fortran 77 and C
  - Provide a simplified multi-threaded programming model based on the master/slave strategy
  - Aimed specifically at *parallel computing* on SMPs
- We outline Fortran 95 and OpenMP

Fortran 95

- Fortran 95 is the Fortran standard released by ISO in December 1997
  - A minor enhancement of Fortran 90
    - Consists of a small number of new features
- The `FORALL` statement and construct
  - The major new feature added to support parallel programming SMP computers

The FORALL statement

- The FORALL statement
  - An alternative to the DO-loop
  - Its content can be executed in any order
    - It therefore can be implemented with parallel threads

- Example 1. The FORALL statement defining a Hilbert matrix of order N

```
FORALL (I = 1:N, J = 1:N) H(I,J) = 1./REAL(I+J-1)
```

The FORALL statement

- Example 2. The FORALL statement inverts the elements of a matrix, avoiding division with zero

```
FORALL(I=1:N,J=1:N, Y(I,J).NE.0.) X(I,J)=1/Y(I,J)
```

- The general form of the FORALL statement is

FORALL (v_1=l_1:u_1:s_1,..., v_n=l_n:u_n:s_n, *mask*)
a(e_1,..., e_m) = *expr*

- First the triplets are evaluated in any order
  - All possible pairings of indices form the set of combinations

The `FORALL` statement (ctd)

- Example. The set of combinations of `I` and `J` for

  ```
  FORALL (I=1:3, J=4:5) A(I,J) = A(J,I)
  ```

  is `{ (1,4), (1,5), (2,4), (2,5), (3,4), (3,5) }`

- Secondly, *mask* is evaluated, in any order, producing a set of *active combinations*

- Example. If mask `(I+J.NE.6)` is applied to the above set, the set of active combinations is:

  ```
  { (1,4), (2,5), (3,4), (3,5) }
  ```

The `FORALL` statement (ctd)

- Then, expressions e_1, …, e_m, and *expr* are evaluated in any order for all active combinations of indices

- Finally, the computed values of *expr* are assigned to the corresponding elements of *a* in any order for all active combinations of indices

The FORALL construct

- The FORALL construct is a multi-line extension of the single line FORALL statement
- Example.

```
REAL, DIMENSION(N, N) :: A, B
...
FORALL (I = 2:N-1, J = 2:N-1)
 A(I,J) = 0.25*(A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))
 B(I,J) = A(I,J)
END FORALL
```

PURE and ELEMENTAL functions

- FORALL statements and constructs by design require that all referenced functions be free of side effects
  - A procedure can be specified as side effect free, or PURE
- A restricted form of a PURE procedure is called ELEMENTAL
  - A procedure that operates elementally
  - A single procedure to operate on scalars and on arrays of different ranks
    - In Fortran 90, you must provide a separate version for scalars and for each rank from one to seven

OpenMP

- OpenMP is an API for writing multi-threaded applications based on the master/slave design strategy
- Supports parallel programming in C, C++ and Fortran on all SMP architectures
  - Including Unix platforms and Windows NT platforms
- Intel, HP, SGI, IBM, SUN, Compaq, KAI, PGI, NAG, and many others took part
- First specifications appeared in late 1998

OpenMP (ctd)

- OpenMP is a set of language extensions
  - Compiler directives
  - Library procedures
  - Environmental variables
- OpenMP is semantically the same between Fortran and C/C++
- We use the C/C++ API to present OpenMP

OpenMP (ctd)

- An OpenMP compiler directive is a C/C++ preprocessing `pragma` directive of the form:

  `#pragma omp directive-name list-of-clauses`

  - C/C++ compilers that do not support OpenMP can compile an OpenMP program
- OpenMP also specifies stubs for the run-time library
  - Enables portability to platforms not supporting OpenMP
    - OpenMP programs must be linked with the stub library
    - The stub functions emulate serial semantics assuming that the directives in the OpenMP program are ignored

OpenMP (ctd)

- As a rule, an OpenMP directive applies to a *structured block*
  - A statement (single or compound) that has a single entry and a single exit
- The directive and the structured block, which the directive applies to, make up an OpenMP *construct*
  - Some directives are not part of a construct

OpenMP (ctd)

- A typical sequence of OpenMP's constructs and single directives do the following:
  - Define a structured block, which is to be executed by multiple threads in parallel, and create the parallel threads
    - that structured block is called a *parallel region*
  - Distribute the execution of the parallel region among the parallel threads

OpenMP (ctd)

- A typical sequence of OpenMP's constructs and single directives do the following (ctd):
  - Synchronize the work of the parallel threads during the execution of the parallel region
  - Control the data environment during the execution of the parallel region
    - via specification of shared data, private data, accumulating variables, etc.

Parallel Regions

- A `parallel` construct starts parallel execution of a parallel region

  ```
  #pragma omp parallel list-of-clauses
   structured-block
  ```

- When a thread encounters a `parallel` construct
  - A team of threads is created
  - This thread becomes the master thread of the team, with a thread number of 0
  - All threads, including the master thread, execute the region in parallel

Parallel Regions (ctd)

- By default, the number of threads that are requested is implementation-defined
- To explicitly determine the number of threads
  - Use the **`num_threads`** clause in the **`parallel`** directive
  - Call the **`omp_set_num_threads`** function before activization of the parallel region

Parallel Regions (ctd)

- Example. Parallel initialisation of an array:

```
int a[5][5];
omp_set_num_threads(5);
#pragma omp parallel
{
    int thrd_num = omp_get_thread_num();
    zero_init(a[thrd_num], 5);
}
```

  - **omp_get_thread_num** returns the thread number (within its team)
  - The thread number lies between 0 and 4 (the master is thread 0)
  - Array **a** is shared between all the threads
  - Variable **thrd_num** is private to each thread

Parallel Regions (ctd)

- There is a barrier at the end of a parallel region
  - Each thread waits until all threads in the team arrive at this point
- Only the master thread of the team continues execution at the end of the parallel region
- A thread encountering another **`parallel`** construct
  - Creates a new team, and it becomes its master
  - By default, nested parallel regions are serialized
    - a nested parallel region is executed by a team of one thread

Work-Sharing Constructs

- A work-sharing construct
  - Does distribute the execution of the associated statement among the members of the team that encounter it
  - Does not launch new threads
  - There is no implicit barrier on entry to a work-sharing construct.
- Every thread in a team must encounter the same sequence of work-sharing constructs and **`barrier`** directives

Work-Sharing Constructs (ctd)

- A **`for`** construct:

  ```
  #pragma omp for list-of-clauses
  for-loop
  ```

- The iterations of the loop will be executed in parallel
- The **`for`** loop must be of the *canonical* form that allows the number of loop iterations to be computed upon entry to the loop

  ```
  for(var = lb; var logical-op rb; incr-expr)
  ```

The `for` Work-Sharing Construct

- The **`schedule`** clause of the `for` construct

  `schedule(`*`kind [, chunk-size]`*`)`

  - Specifies how iterations are divided among threads
  - *kind* may be **`static`**, **`dynamic`**, **`guided`**, or **`runtime`**
  - If *kind* is **`static`**, iterations are divided into chunks of a size specified by *chunk-size*
    - The chunks are statically assigned to threads in a round-robin fashion in the order of the thread number
    - When *chunk-size* is omitted, the iteration space is divided into approximately equal chunks (one chunk assigned to each thread)

The `for` Work-Sharing Construct (ctd)

- Example. Parallel initialisation of an array:

```
int a[10][10], k;
omp_set_num_threads(5);
#pragma omp parallel
#pragma omp for schedule(static)
for(k=0; k<10; k++)
 zero_init(a[k], 10);
```

  - Each of 5 threads will execute two successive iterations of the loop
  - Variable *k* will be private to each of the threads

The **`for`** Work-Sharing Construct (ctd)

- The **`schedule`** clause of the **`for`** construct (ctd)
  - If *kind* is **`dynamic`**, the iterations are divided into a series of chunks, each containing *chunk-size* iterations
    - Each chunk is assigned to a thread that is waiting for an assignment
    - The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned
    - When no *chunk-size* is specified, it defaults to 1

The **`for`** Work-Sharing Construct (ctd)

- Example. Parallel initialisation of an array:

```
int a[10][10], k;
omp_set_num_threads(5);
#pragma omp parallel
#pragma omp for schedule(dynamic,2)
for(k=0; k<10; k++)
zero_init(a[k], 10);
```

  - Each two successive iterations of the loop will be executed by one of the 5 threads
  - *Which thread will execute which pair of iterations is decided at run time*

The **`for`** Work-Sharing Construct (ctd)

- The **`schedule`** clause of the **`for`** construct (ctd)
  - If *kind* is **`guided`**, the iterations are assigned to threads in chunks with decreasing sizes
    - When a thread finishes its assigned chunk of iterations, it is dynamically assigned another chunk, until none remains
    - If *chunk-size* = 1, the size of each chunk is approximately the number of unassigned iterations divided by the number of threads. These sizes decrease approximately exponentially to 1
    - If *chunk-size* > 1, the sizes decrease approximately exponentially to *chunk-size* (the last chunk may be fewer than *chunk-size* iterations)
    - When no *chunk-size* is specified, it defaults to 1

The **`for`** Work-Sharing Construct (ctd)

- The **`schedule`** clause of the **`for`** construct (ctd)
  - If *kind* is **`runtime`**, the decision regarding scheduling is deferred until runtime
    - The schedule kind and size of the chunks can be chosen at run time by setting the environment variable **`OMP_SCHEDULE`**
    - If **`OMP_SCHEDULE`** is not set, the resulting schedule is implementation–defined
    - No *chunk-size* must be specified

The **`for`** Work-Sharing Construct (ctd)

- If no **`schedule`** clause is explicitly defined, the default **`schedule`** is implementation-defined
- There is an implicit barrier at the end of a **`for`** construct unless a **`nowait`** clause is specified

Work-Sharing Constructs (ctd)

- A **`sections`** construct:

```
#pragma omp sections list-of-clauses
{
   [#pragma omp section]
    structured-block
   [#pragma omp section
    structured-block]
   ...
}
```

- Specifies a set of sections to be divided among threads
  - Each section is executed once by a thread in the team
  - Each section must be preceded by a **`section`** directive (except for the first one)
  - An implicit barrier at the end of the **`sections`** construct

The `sections` Construct

- Example. Parallel initialisation of an array

```
int a[4][10];
omp_set_num_threads(2);
#pragma omp parallel
#pragma omp sections
{
  {
    zero_init(a[0], 10);
    zero_init(a[1], 10);
  }
  #pragma omp section
  {
    zero_init(a[2], 10);
    zero_init(a[3], 10);
  }
}
```

Work-Sharing Constructs (ctd)

- A **`single`** construct:

  `#`**`pragma`** `omp single` *`list-of-clauses`*
  `structured-block`

- Specifies that the associated structured block is executed by only one thread in the team
  - Not necessarily the master thread
- There is an implicit barrier after the **`single`** construct
  - Unless a **`nowait`** clause is specified

Work-Sharing Constructs (ctd)

- A **`task`** construct:

```
...
#pragma omp single
{
   #pragma omp task
     f1();
   #pragma omp task
     f2();
 }
```

Work-Sharing Constructs (ctd)

- A **`master`** construct:

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

- Specifies that the associated structured block is executed by the master thread in the team
  - Other threads of the team do not execute this statement
- There is no implicit barrier on entry to or exit from the **`master`** section

Work-Sharing Constructs (ctd)

- OpenMP provides shortcuts for specifying a parallel region that contains only one work-sharing construct:

```
#pragma omp parallel for list-of-clauses
for-loop
```

and

```
#pragma omp parallel sections list-of-clauses
{
   [#pragma omp section]
    structured-block
   [#pragma omp section
    structured-block]
   ...
}
```

Synchronization Directives and Constructs

- Implicit barriers are implied on some OpenMP constructs
- OpenMP provides a number of directives and constructs for explicit synchronization of threads
  - The **`barrier`** directive
  - The **`ordered`** construct
  - The **`critical`** construct
  - The **`atomic`** construct
  - The **`flush`** directive

Synchronization Directives and Constructs (ctd)

- The **`barrier`** directive

```
#pragma omp barrier
```

  - When encountered, each thread of the team waits until all of the others have reached this point

- An **`ordered`** construct

```
#pragma omp ordered list-of-clauses
structured-block
```

  - Specifies that the structured block is executed in the order in which iterations would be executed in a sequential loop

The **`critical`** Construct

- The **`critical`** construct

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

  - Synchronizes access to shared data
    - Restricts execution of the structured block to a single thread at a time
  - A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name *name*
  - All unnamed critical regions have the same unspecified name

The `critical` Construct (ctd)

- Example.

```
int a[100], ind[100], b[100], k;
...
omp_set_num_threads(5);
#pragma omp parallel for
for(k=0; k<100; k++)
{
  #pragma omp critical
   a[ind[k]] += f(k);

   b[k] += g(k);
 }
```

  - Different iterations may update the same element of a => the serialization is needed to avoid race conditions

The **`atomic`** Construct

- The **`atomic`** construct

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

  - May be seen as a special case of a critical section
  - Does not prevent simultaneous execution by multiple threads of the associated statement
  - Ensures that a specific memory location is updated atomically
    - prevents simultaneous access to the memory location by multiple threads

The **`atomic`** Construct (ctd)

- The **`atomic`** construct (ctd)
  - *expression-statement* is *x op*= *expr*, *x*++, ++*x*, *x*−−, or −−*x*
    - *x* is an lvalue of scalar type
    - *expr* is an expression of scalar type not referencing the object designated by *x*
    - *op* is one of the binary operators +, *, –, /, &, ^, |, <<, or >>
  - Only the load and store of *x* are atomic
    - The evaluation of *expr* is not atomic
  - To avoid race conditions, all updates of the location in parallel should be protected with the **`atomic`** directive

The `atomic` Construct (ctd)

- Example.

```
int a[100], ind[100], b[100], k;
...
omp_set_num_threads(5);
#pragma omp parallel for
for(k=0; k<100; k++) {
#pragma omp atomic
 a[ind[k]] += f(k);
 b[k] += g(k);
}
```

  - Allows updates of two different elements of **a** to occur in parallel
    - the **critical** directive leads to serial updates
  - More efficient than the **critical** directive if supported by hardware

The **`flush`** Directive

- The **`flush`** directive

```
#pragma omp flush [(variable-list)]
```

  - Specifies a sequence point at which all threads in a team must have a consistent view of memory
    - all memory operations (both reads and writes) specified before the sequence point must complete
    - all memory operations specified after the point must have not yet begun
  - An explicit mechanism to ensure consistency of shared data not depending on cache management in SMP computers

The **`flush`** Directive (ctd)

- The **`flush`** directive (ctd)
  - Without a *variable-list* synchronizes all shared objects
  - With a *variable-list* synchronizes only the objects in the *variable-list*
  - It is implied
    - for the **`barrier`** directive
    - at entry to and exit from **`critical`** and **`ordered`** constructs
    - at exit from **`parallel`**, **`for`**, **`sections`**, and **`single`** constructs
    - not implied if a **`nowait`** clause is present

Data Environment

- Two kinds of variable in an OpenMP program
  - Shared
  - Private
- A *shared* variable
  - Designates a single region of storage
  - All threads in a team that access this variable will access this single region of storage

Data Environment (ctd)

- By default,
  - A variable defined outside a **`parallel`** construct and visible when the **`parallel`** construct is encountered, is shared within the **`parallel`** construct
  - Static variables declared within a parallel region are shared
  - Heap allocated memory is shared
- A *private* variable
  - Designates a region of storage that is unique to the thread making the reference

Data Environment (ctd)

- Automatic variables declared within a **`parallel`** construct are private
- Other ways to specify that a variable is private
  - A **`threadprivate`** directive
  - A **`private`**, **`firstprivate`**, **`lastprivate`**, or **`reduction`** clause
  - Use of the variable as a **`for`** loop control variable immediately following a **`for`** or **`parallel for`** directive

Data Environment (ctd)

- The **`threadprivate`** directive

  `#pragma omp threadprivate [(variable-list)]`

  - Makes each variable in *variable-list* private to a thread
    - The variable must be global or static block-scope
    - Each copy of the variable is initialised once, before the first reference to that copy, and in the usual manner

- Example.

```
int count()
{
        static int counter = 0;
        #pragma omp threadprivate(counter)
        counter++;
        return counter;
}
```

Data Environment (ctd)

- The **`parallel`** directive and most of work-sharing directives accept clauses explicitly specifying whether a variable is private or shared, how the variable is initialised, etc.
- A **`copyin`** clause

  `copyin(`*`variable-list`*`)`

  - A mechanism to assign the same value to **`threadprivate`** variables for each thread in the team executing the parallel region
  - For each variable in *`variable-list`*, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region

Data Environment (ctd)

- A **`private`** clause

  `private(`*`variable-list`*`)`

  - Declares the variables in *variable-list* to be private to each thread in a team
  - Creates private automatic variables (with a construct scope) overriding the original (possibly, global or local static) variable
- A **`firstprivate`** clause

  `firstprivate(`*`variable-list`*`)`

  - A special case the **`private`** clause
  - Additionally initialises each new private object by the value of the original object that exists immediately before the **`parallel`** or work-sharing construct for the thread that encounters it

Data Environment (ctd)

- A **`lastprivate`** clause

  `lastprivate(`*`variable-list`*`)`

  - A special case of the **`private`** clause
  - Additionally determines the original object upon exit from the work-sharing construct
    - If a **`lastprivate`** variable is assigned a value by the sequentially last iteration of the associated loop, or the lexically last section, the value is assigned to the variable's original object
    - Other **`lastprivate`** variables have indeterminate values after the construct

Data Environment (ctd)

- A **`shared`** clause

`shared(`*`variable-list`*`)`

  - Shares variables in *variable-list* among all the threads in a team
    - All threads within a team access the same region of storage for **`shared`** variables

Data Environment (ctd)

- A **`copyprivate`** clause

  `copyprivate(`*`variable-list`*`)`

  - May only appear on the **`single`** directive
  - The thread that executed the associated structured block broadcasts the value of each `copy`**`rivate`** variable to other threads
    - Logically, it happens after the execution of the structured block and before any of the threads in the team have left the barrier at the end of this **`single`** construct

Data Environment (ctd)

- A **`reduction`** clause

  `reduction(`*`op`*`:`*`variable-list`*`)`

  - Specifies how to perform a reduction operation with operator *op* on a scalar variable in *variable-list*
    - It is supposed that the corresponding parallel or work-sharing construct must implement the reduction operation

Data Environment (ctd)

- A **`reduction`** clause (ctd)
  - Inside this construct a private copy of each reduction variable is created, one for each thread (as if the **`private`** clause had been used), and initialised depending on the *op*
    - `0` for `+`, `-`, `|`, `^`, and `||`
    - `1` for `*` and `&&`
    - `~0` for `&`
  - Then each thread updates its private copy of the variable
  - Then, the original object is updated by combining its original value with the final value of each of the private copies using the operator *op*

Data Environment (ctd)

- Example. Efficient computing the sum of elements of an array by parallel threads:

```
int k;
double sum, a[1000];
...
sum = 0.;
#pragma omp parallel for reduction(+: sum)
for(k=0; k<1000; k++)
        sum += a[k];
```

Data Environment (ctd)

- A **`default`** clause

```
default(none)
```

  - May be used in the **`parallel`** directive
  - Repeals all defaults for implicit specification of shared variables
    - requires explicit specification of their shared status

Run-Time Library Functions

- Two classes of the run-time library functions
  - To control and query the parallel execution environment
  - Implementing a mutex-like explicit synchronization mechanism
- The control-and-query functions
  - `void omp_set_num_threads(int num_threads)`
    - sets the default number of threads in a team
  - `int omp_get_num_threads(void)`
    - returns the number of threads currently in the team executing the parallel region from which it is called

Run-Time Library Functions (ctd)

- The control-and-query functions (ctd)
  - **`int`** `omp_get_max_threads(`**`void`**`)`
    - returns the maximum value that can be returned by calls to **`omp_get_num_threads`**
  - **`int`** `omp_get_thread_num(`**`void`**`)`
    - returns the thread number, within its team, of the calling thread
  - **`int`** `omp_get_num_procs(`**`void`**`)`
    - returns the maximum number of processors that could be assigned to the program

Run-Time Library Functions (ctd)

- A part of the OpenMP run-time library provides a mutex-like synchronization mechanism
  - Declares two types of *lock* variables (an analog of mutex variables of Pthreads)
  - Declares functions for
    - **Initialising a lock variable**
    - **Destroying the lock variable**
    - **Setting the lock variable (an analog of locking a mutex in Pthreads)**
    - **Unsetting the lock variable (an analog of unlocking a Pthreads mutex)**
    - **Testing the lock variable (an analog of the Pthreads's `trylock`)**

OpenMP (ctd)

- Example. Consider an OpenMP application that computes the dot product of two real vectors *x* and *y* implementing the same parallel algorithm as the Pthreads application presented earlier.
  - <For the source code see handout>
  - One can see that this OpenMP code is significantly simpler and more compact than the Pthreads code

SMP Architecture: Summary

- SMPs provide higher level of parallelism than VPs and SPs via multiple parallel streams of instructions
  - Still not a scalable parallel architecture
    - The speedup is limited by the bandwidth of the memory bus
- Multithreading is the primary programming model
- C / Fortran 77 + optimizing compilers may be used to write efficient programs for SMPs
  - Only a quite limited and simple class of MT algorithms can be implemented this way

SMP Architecture: Summary (ctd)

- Thread libraries directly implement the MT paradigm
  - Explicit MT programming
  - Use of ordinary compilers
  - Pthreads are standard for Unix platforms
  - Too powerful, complicated and error-prone if used for parallel programming only
- OpenMP is a high-level parallel extension of Fortran and C/C++
  - Supports a simplified master/slave design strategy of MT programming
  - Aimed specifically at parallel programming SMPs