Programming Shared Memory Systems with OpenMP

Reinhold Bader (LRZ)

Georg Hager (RRZE)

What is OpenMP?

- **Directive-based Parallelization Method on Shared Memory Systems**
  - Implementations for DMS also exist
  - Some library routines are provided
- **Support for Data Parallelism**
- **“Base Languages”**
  - Fortran (77/90/95)
  - C (90/99)
  - C++

  Note: Java (JOMP, Java Threads based, is **not** a base language)
- **WWW Resources**
  - OpenMP Home Page:

    http://www.openmp.org
  - OpenMP Community Page:

    http://www.compunity.org

OpenMP Standardization

- **Standardized for Portability:**
  - Fortran Specification 1.0 Oct. 1997
  - Fortran Specification 1.1 Nov. 1999 (Updates)
  - Fortran Specification 2.0 Mar. 2000

    New Features:
    - Better support nested parallelism
    - Array reductions
    - Fortran Module and Array support
  - Combined Fortran, C, C++ Specification 2.5 May 2005
    - No changes in functionality
    - Clarifications (Memory Model, Semantics)
    - Some renaming of terms

Further OpenMP resources

- **OpenMP at LRZ:**

  http://www.lrz.de/services/software/parallel/openmp

- **OpenMP at HLRS (Stuttgart):**

  http://www.hlrs.de/organization/tsc/services/models/openmp/index.html

- **R. Chandra et al.: Parallel Programming in OpenMP**

  Academic Press, San Diego, USA, 2001, ISBN 1-55860-671-8

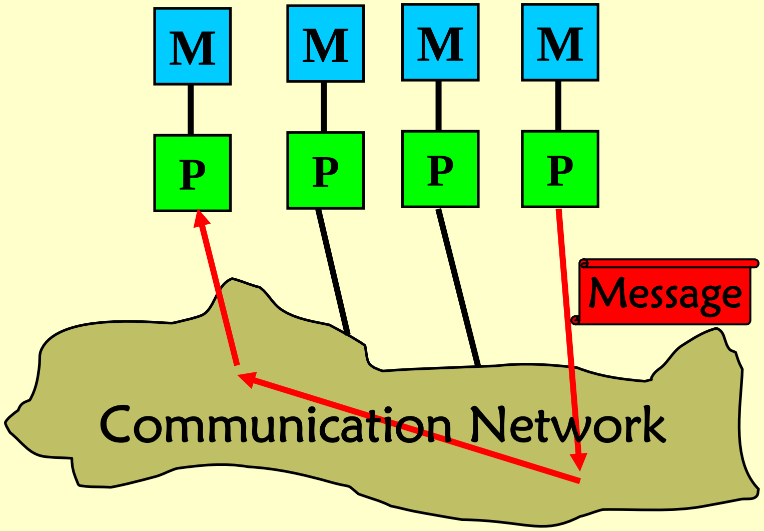
- **Acknowledgments are due to**
  - Isabel Loebich and Michael Resch (HLRS, OpenMP workshop, Oct., 1999)
  - Ruud van der Pas (Sun, IWOMP workshop, June 2005)

General Concepts

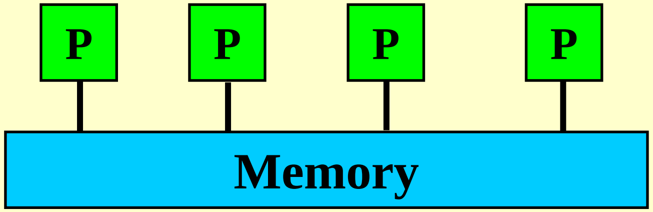
An abstract overview
of OpenMP terms
and usage context

Two Paradigms for Parallel Programming
as suggested (not determined!) by Hardware Design

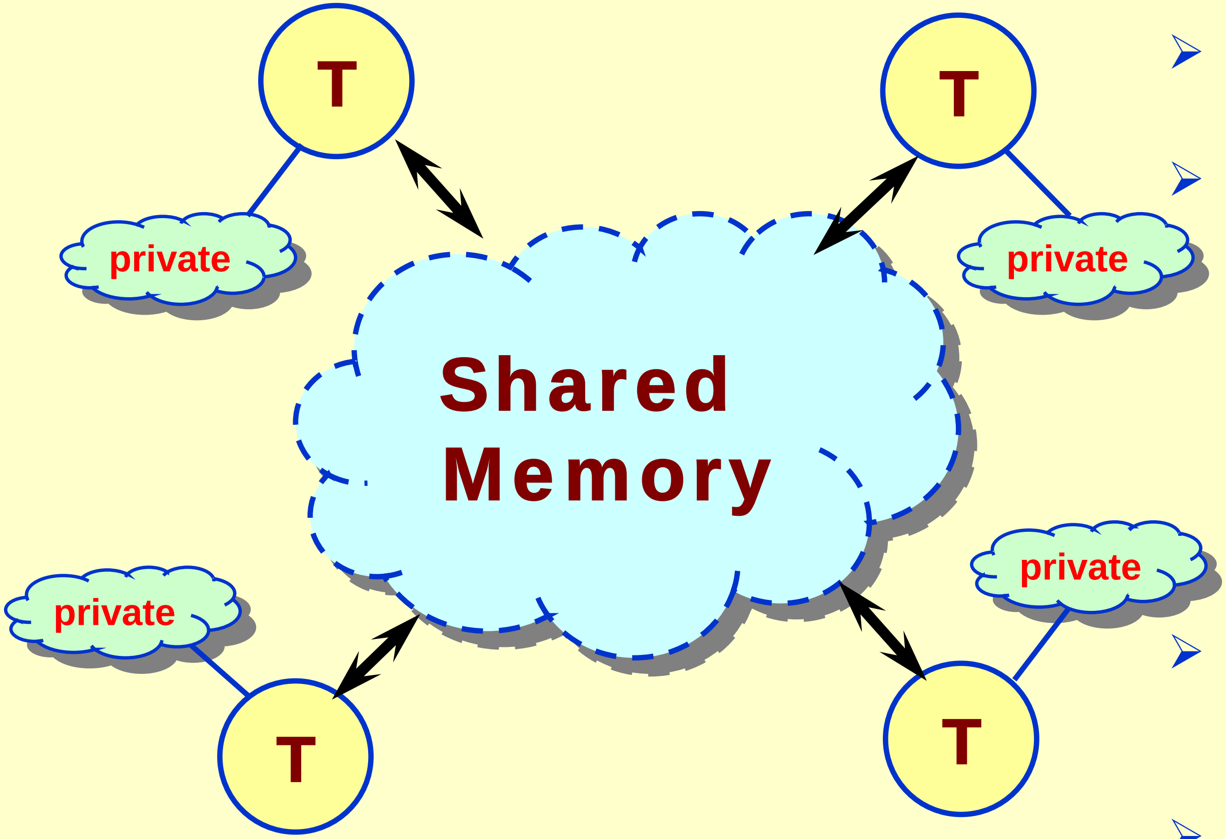
- **Distributed Memory**
  - Message Passing
  - explicit programming required



- **Shared Memory**
  - common address space for a number of CPUs
  - access efficiency may vary
    - SMP, (cc)NUMA
  - many programming models
  - potentially easier to handle
    - hardware and OS support!

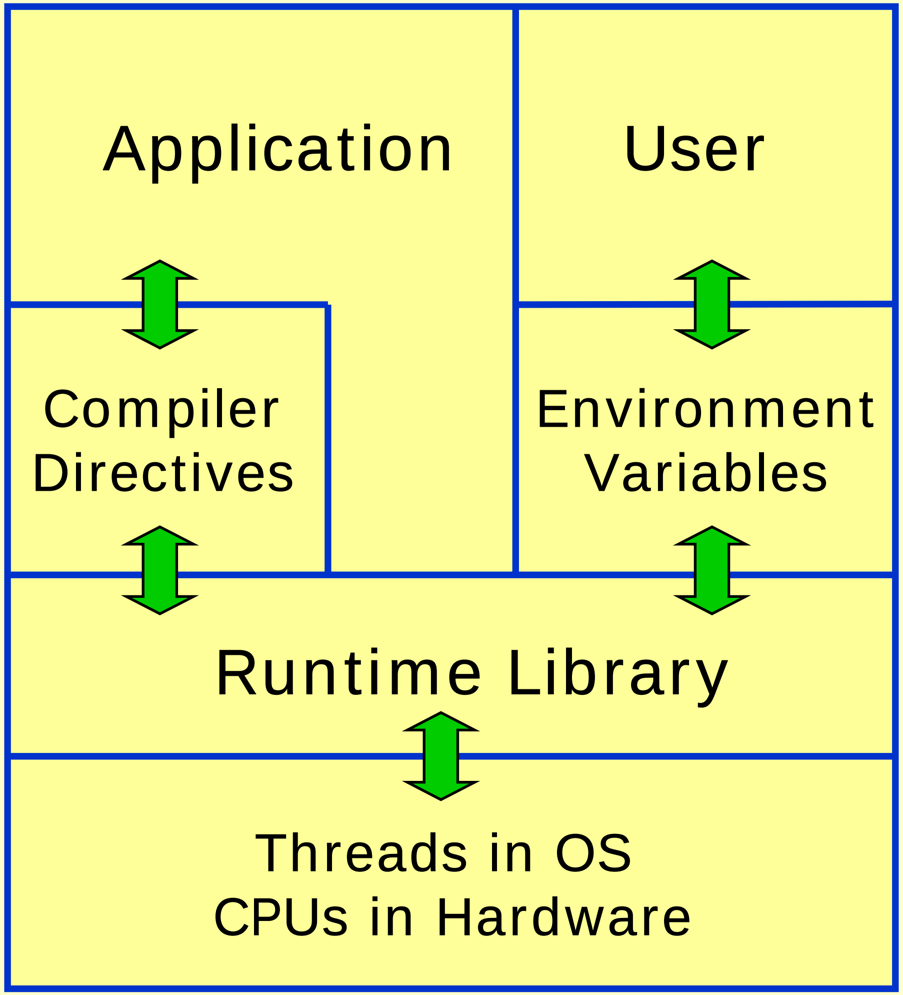


Shared Memory Model used by OpenMP



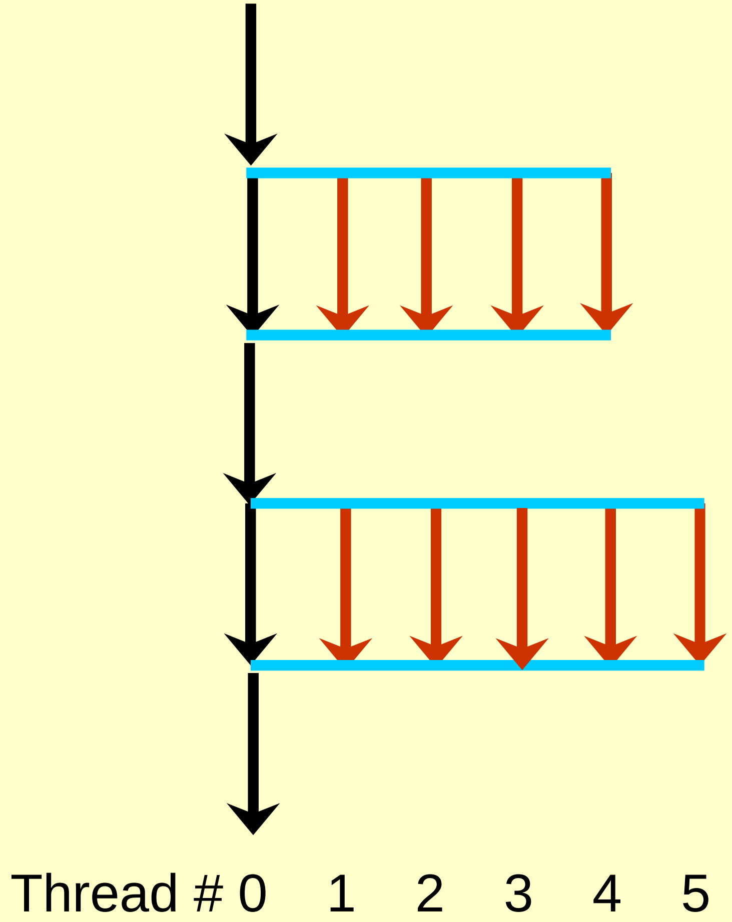
- Threads access globally shared memory
- Data can be shared or private
  - shared data available to all threads (in principle)
  - private data only to thread that owns it
- Data transfer transparent to programmer
- Synchronization takes place, is mostly implicit

OpenMP Architecture: Operating System and User Perspective



- **OS View:**
  - parallel work done by threads
- **Programmer's View:**
  - Directives (comment lines)
  - Library Routines
- **User's View**
  - Environment Variables (Resources, Scheduling)

OpenMP Program Execution
Fork and Join



- Program start: only master thread runs
- Parallel region: team of worker threads is generated (“fork”)
- synchronize when leaving parallel region (“join”)
- Only master executes sequential part
  - worker threads persist, but are inactive
- task and data distribution possible via directives
- Usually optimal:
  1 Thread per Processor

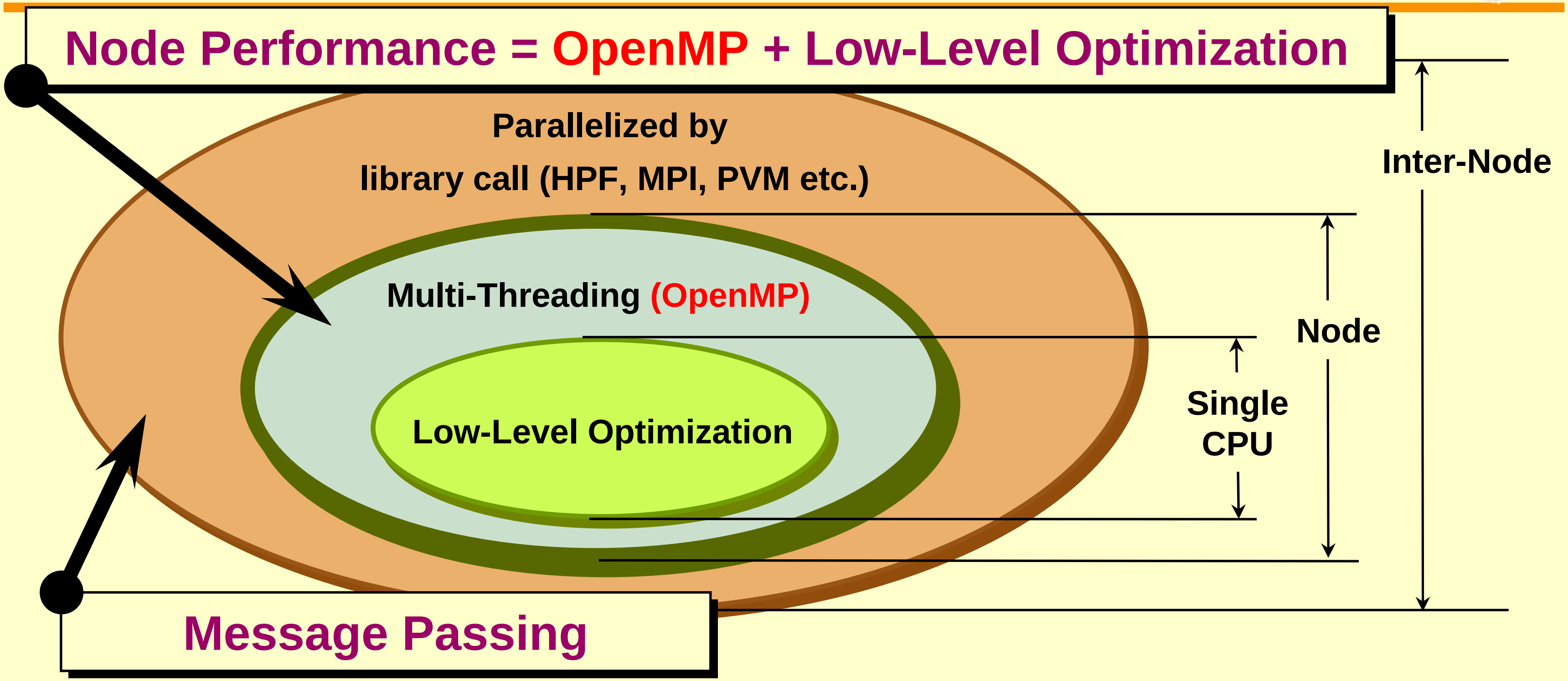
Retaining sequential functionality

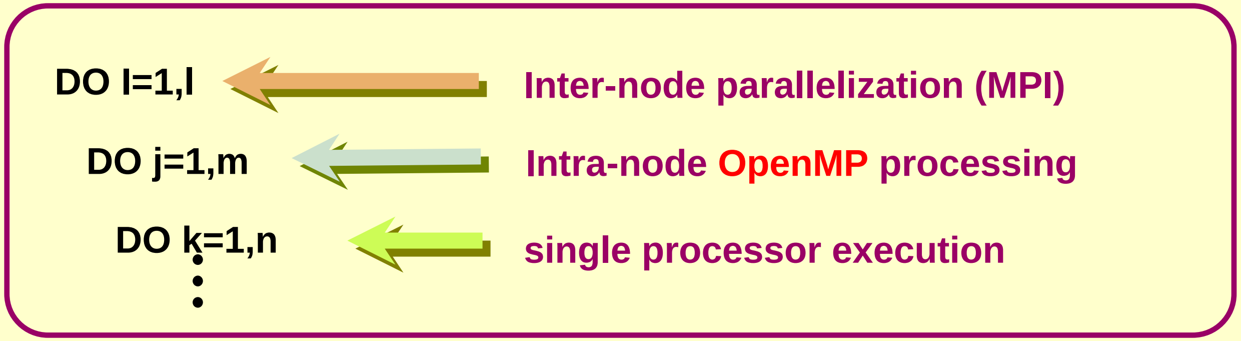
- **OpenMP**
  - enables to retain sequential functionality i.e.
  - by proper use of directives it is possible to compile code sequentially
  - and obtain correct results
- **No enforcement**
  - can also write conforming code in a way that omitting OpenMP functionality at compile time does not yield a properly working program
  - program documentation
- **Caveats**
  - non-associativity of numerical model number operations
  - parallel execution may reorder operations
  - and do so differently between runs and with varying thread numbers

OpenMP in the HPC context (1)
Comparing parallelization methods

	MPI (shared and distributed memory Systems)	OpenMP (shared memory Systems)	Proprietary parallelization Directives	High Performance Fortran
Portable?	Yes	Yes	No	Yes
Scalable?	Yes	Partially	Partially	Yes
Support for Data Parallelism?	No	Yes	Yes	Yes
Incremental Parallelization?	No	Yes	Yes	Partially
Serial Functionality unchanged?	No	Yes	Yes	Yes
Correctness verifiable?	No	Yes	?	?

OpenMP in the HPC context (2)
Hybrid parallelization on clustered SMPs





Inter-node parallelization (MPI)

Intra-node OpenMP processing

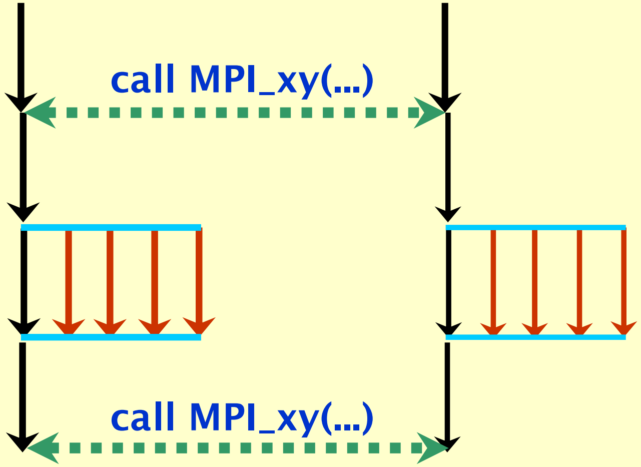
single processor execution

Levels of Interoperability between MPI and OpenMP (1)

- Call of MPI-2 threaded initialization

```
call MPI_INIT_THREAD(required, provided)
```

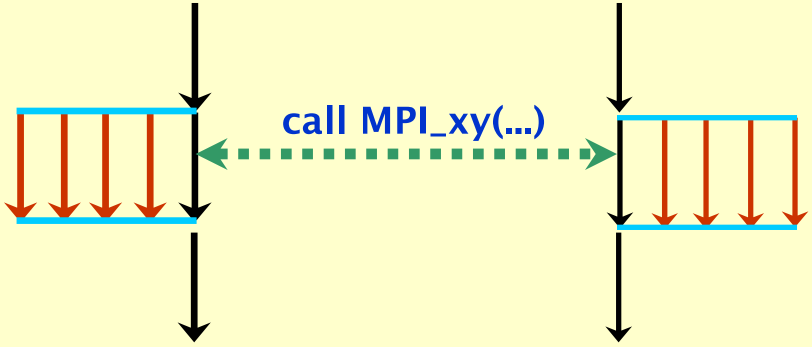
with parameters of default integer KIND replaces `MPI_INIT`



Base Level support:

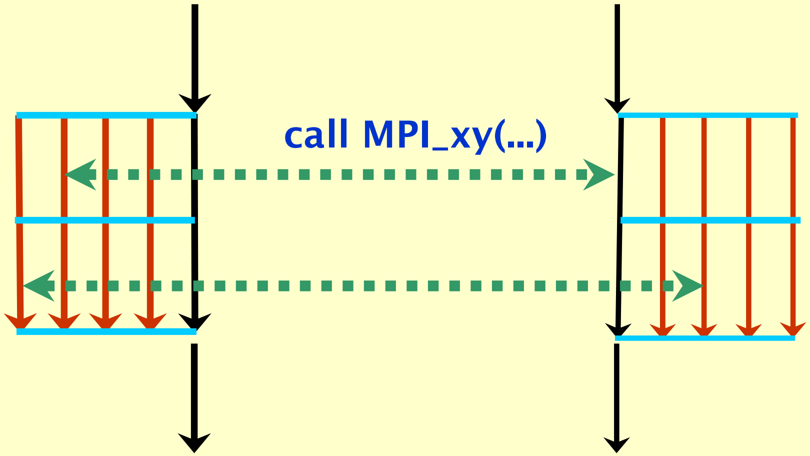
- Initialization returns `MPI_THREAD_SINGLE`
- MPI calls must occur in serial (i.e., non-threaded) parts of Program

Levels of Interoperability between MPI and OpenMP (2)



First Level support:

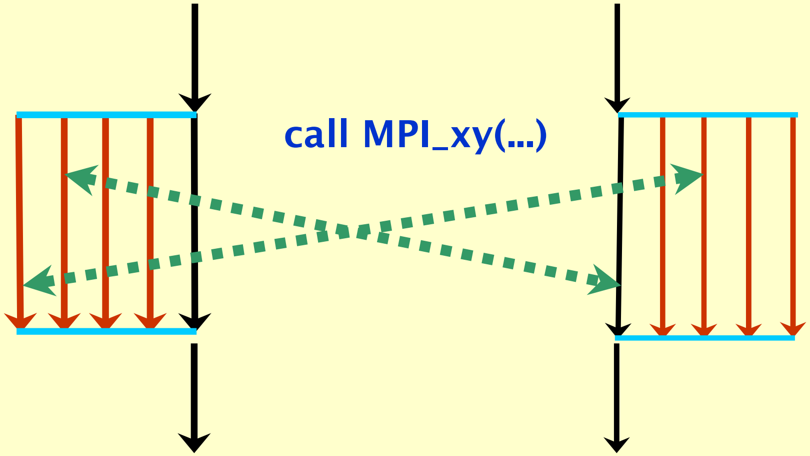
- Initialization returns `MPI_THREAD_FUNNELED`
- MPI calls allowed in threaded parts
- MPI calls only by master



Second Level support

- Initialization returns `MPI_THREAD_SERIALIZED`
- MPI calls allowed in threaded parts
- No concurrent calls
  - synchronization between calls required

Levels of Interoperability between MPI and OpenMP (3)



Third Level support

- Initialization returns **MPI_THREAD_MULTIPLE**
- MPI calls allowed in threaded parts
- No restrictions

Notes:

- Sometimes, a SINGLE implementation will also work in FUNNELED mode if no system calls (malloc → automatic buffering, file operations) are performed in connection with the MPI communication
- A fully threaded MPI implementation will probably have worse performance, especially for small message sizes
  - selection of thread level support by user at run time may help

OpenMP availability at LRZ

- **LRZ Linux Cluster: Intel Compilers**
  - IA32 and Itanium SMPs
  - sgi Altix 3700 (16 8-way bricks, ccNUMA)
  - sgi Altix 4700 (HLRB2)
- **Hitachi Fortran 90 and C Compilers:**
  - OpenMP maps to a subset of Hitachi's proprietary directives
  - Available within an 8-way node
  - C++ not supported

OpenMP availability at RRZE

- SGI R3400 (SGI Compiler)

  28-way system: 7 4-way bricks, ccNUMA

- SGI Altix (IA64-based, Intel Compiler)

  28-way system: 7 4-way bricks, ccNUMA

Programming with OpenMP

- **Not a coverage of complete OpenMP functionality**
- **Please read the Standard document!**
- **Give you a feel for how to use OpenMP**
  - a few characteristic examples
  - do-it-yourself: hands-on sessions
- **Give some hints on pitfalls when using OpenMP**
  - deadlock → hangs
  - livelock → never finishes
  - race conditions → wrong results

Basic OpenMP functionality

About Directives and Clauses

About Data

About Parallel Regions
and Work Sharing

A first example (1)
Numerical Integration

Approximate by a discrete sum

$$\int_0^1 f(t)\,dt \quad \approx \quad \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

where

$$x_i \quad = \quad \frac{i-0.5}{n} \qquad (i=1,\ldots,n)$$

We want

$$\int_0^1 \frac{4\,dx}{1+x^2} \quad = \quad \pi$$

→ solve this in OpenMP

```
program compute_pi
...   (declarations omitted)

! function to integrate
f(a)=4.0_8/(1.0_8+a*a)
```

```
w=1.0_8/n
sum=0.0_8

do i=1,n
   x=w*(i-0.5_8)
   sum=sum+f(x)
enddo
pi=w*sum
```

```
...    (printout omitted)
end program compute_pi
```

A first example (2):
serial and OpenMP parallel Code

```
use omp_lib
...
pi=0.0_8
w=1.0_8/n
!$OMP parallel private(x,sum)
sum=0.0_8
!$OMP do
do i=1,n
  x=w*(i-0.5_8)
  sum=sum+f(x)
enddo
!$OMP end do
!$OMP critical
pi=pi+w*sum
!$OMP end critical
!$OMP end parallel
```

Now let’s discuss the different bits we’ve seen here …

OpenMP Directives
Syntax in Fortran

- **Each directive starts with *sentinel*:**
  - fixed source: `!$OMP` or `C$OMP` or `*$OMP`
  - free source: `!$OMP`

  followed by a *directive* and, optionally, *clauses*.

in column 1

- **For function calls:**
  - conditional compilation of lines starting with `!$` or `C$` or `*$`

  Example:

```
   myid = 0
!$ myid = omp_get_thread_num()
```

  beware implicit typing!

  - use include file (or Fortran 90 module if available)

- Continuation line, e.g.:

```
!$omp directive  &
!$omp clause
```

OpenMP Directives
Syntax in C/C++

- Include file

```
#include <omp.h>
```

- *pragma* preprocessor directive:

```
#pragma omp [directive [clause ...]]
  structured block
```

- conditional compilation: switch sets preprocessor macro

```
#ifdef _OPENMP

... do something

#endif
```

- continuation line, e.g.:

```
#pragma omp directive \
    clause
```

OpenMP Syntax:
On Clauses

e.g., barrier

- Many (but not all) OpenMP directives support clauses
- Clauses specify additional information with the directive
- Integration example:
  - `private(x,sum)` appears as clause to the `parallel` directive
- The specific clause(s) that can be used depend on the directive

OpenMP Syntax:
Properties of “structured block”

- **Defined by braces in C/C++**
- **Requires a bit more care in Fortran**
  - code between begin/end of an OpenMP construct must be a complete, valid Fortran block

- **Single point of entry**
  - no GOTO into block (Fortran), no **setjmp()** to entry point (C)
- **Single point of exit**
  - no RETURN, GOTO, EXIT out of block (Fortran)
  - **longjmp()** and **throw()** may violate entry/exit rules (C, C++)
  - exception: STOP (exit () in C/C++) is allowed (error exit)

OpenMP parallel regions
How to generate a Team of Threads

- `!$OMP PARALLEL` and `!$OMP END PARALLEL`
  - Encloses a parallel region: All code executed between start and end of this region is executed by all threads.
  - This includes subroutine calls within the region (unless explicitly sequentialized)
  - Both directives must appear in the same routine.

- C/C++:

```
#pragma omp parallel
structured block
```

No **END PARALLEL** directive since block structure defines boundaries of parallel region

OpenMP work sharing for loops

Requires thread distribution directive

`!$OMP DO / !$OMP END DO` encloses a loop which is to be divided up if within a parallel region ("sliced").

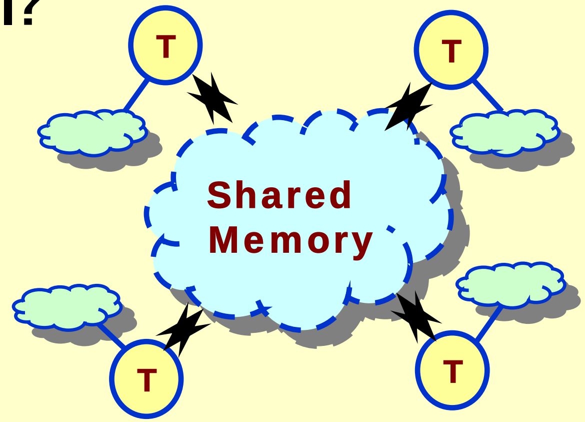
- all threads synchronize at the end of the loop body
- this default behaviour can be changed …

- **Only loop immediately following the directive is sliced**
- **C/C++:**

```
#pragma omp for [clause]
for ( ... )   {
          ...
    }
```

- **restrictions on parallel loops (especially in C/C++)**
  - trip count must be computable (no `do while`)
  - loop body with single entry and single exit point

Directives for Data scoping
shared and *private*



- **Remember the OpenMP memory model?**

  Within a parallel region,
  data can either be
- **private to each executing thread**

  → each thread has its own local copy of data

  or be
- **shared between threads**

  → there is only one instance of data available to all threads

  → this does not mean that the instance is always visible to all threads!
- **Integration example:**
  - shared scope not desirable for x and sum since values computed on one thread must not be interfered with by another thread.
  - Hence:

```
!$OMP parallel private(x,sum)
```

Defaults for data scoping

- **All data in parallel region are shared**
- **This includes global data (Module, COMMON)**
- **Exceptions:**
  1. **Local** data within enclosed subroutine calls are **private** (Note: Inlining must be treated correctly by compiler!) **unless** declared with SAVE attribute
  2. **Loop variables** of parallel ("sliced") loops are **private**
- **Due to stack size limits it may be necessary to give large arrays the SAVE attribute**
  - This presupposes it is safe to do so!
  - If not: convert to ALLOCATABLE
  - For Intel Compilers: **`KMP_STACKSIZE`** may be set at run time (increase thread-specific stack size)

Changing the scoping defaults

- Default value for data scoping can be changed by using the `default` clause on a parallel region:

```
!$OMP parallel default(private)
```

Not in C/C++

- Beware side effects of data scoping:

  Incorrect `shared` attribute may lead to race conditions and/or performance issues (“false sharing”).

  - Use verification tools.

- Scoping of local subroutine data and global data
  - is not (hereby) changed
  - compiler cannot be assumed to have knowledge

- Recommendation: Use

```
!$OMP parallel default(none)
```

  so as not to overlook anything

Storage association of private data

- **Private variables: undefined on entry and upon exit of parallel region**
- **Original value of variable (before parallel region) is undefined after exit from parallel region**
- **To change this:**
  - Replace `private` by `firstprivate` or `lastprivate`
  - To have both is presumably not possible
- **Private variable within parallel region has no storage association with same variable outside region**

Notes on privatization of dynamic data

C pointers:

```
int *p
!$omp parallel private(p)
```

- previous pointer association will be lost
- need to allocate memory for the duration of parallel region
- or point to otherwise allocated space

```
int *p
!$omp parallel private(*p)
```

- this is not allowed

Fortran pointers/allocatables

```
real, pointer, dimension(:) :: p
real, allocatable :: a(:)
!$omp parallel private(p)
```

- p: pointer association lost if previously established
  - re-point or allocate/deallocate
- a: must have allocation status "not currently allocated" upon entry and exit to/from parallel region

A first example (4):

Accumulating partial sums → critical directive

- After loop has completed: add up partial results
- Code needs to be sequentialized to accumulate to a shared variable:

  ```
  !$OMP CRITICAL / !$OMP END CRITICAL
  ```

  Only one thread at a time may execute enclosed code.

  However, all threads eventually perform the code.

  → potential performance problems for sequentialized code!
- Alternative 1: Single line update of one memory location via atomic directive (possibly less parallel overhead):

  ```
  !$OMP atomic
  x = x operator expr
  ```
- Alternative 2: Reduction operation (discussed later)

Compiling OpenMP Code on the SGI Altix

- **Options for Intel Fortran Compiler (ifort)**

  ```
  -O3 -openmp -openmp_report2
  ```

  - enables the OpenMP directives in your code
  - gives information about parallelization procedure
  - -auto is implied: all local variables (except those with SAVE attribute) on the stack

```
ifort -O3 -tpp2 -openmp -o pi.run pi.f90
```

Running the OpenMP executable on the SGI Altix

- **Prepare environment:**

  ```
  export OMP_NUM_THREADS=4
  ```

  (usually: as many threads as processors are available for your job)

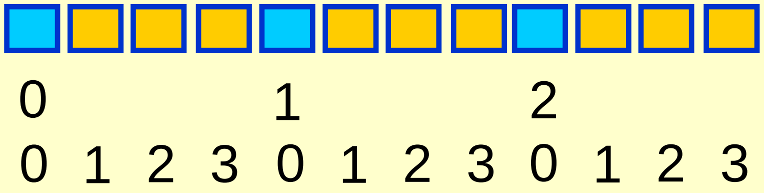
- **Start executable in the usual way (or use NUMA tools)**

  ```
  ./pi.run
  ```

- **If MPI is also used**

  ```
  export MPI_OPENMP_INTEROP=yes
  mpirun -np 3 ./myprog.exe
  ```

  to run on e.g., 12 CPUs



Idea:

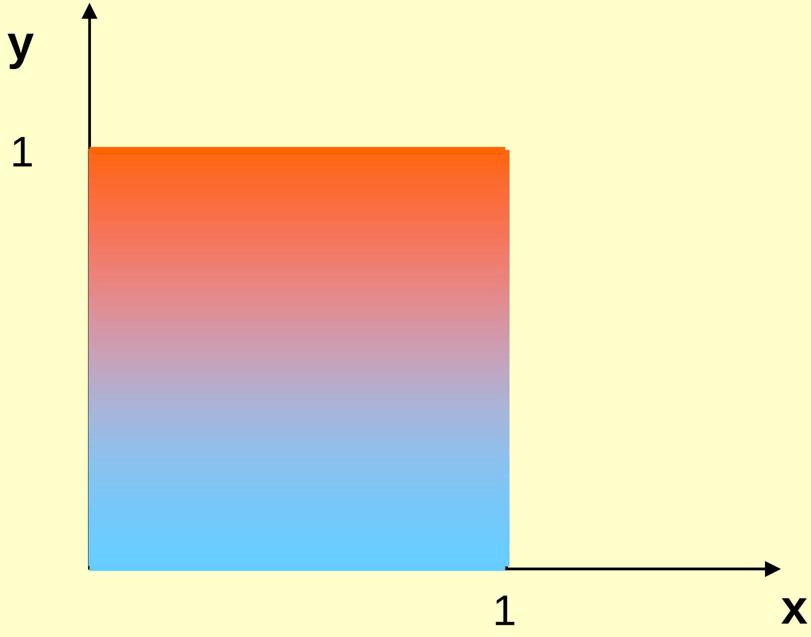
- space out MPI processes
- keep spawned threads as near to master as possible (minimize router hops)

New example:
Solving the heat conduction equation

- **Square piece of metal**
  - Temperature $\Phi(x,y,t)$
  - Boundary values:

    $\Phi(x,1,t) = 1$, $\Phi(x,0,t) = 0$,

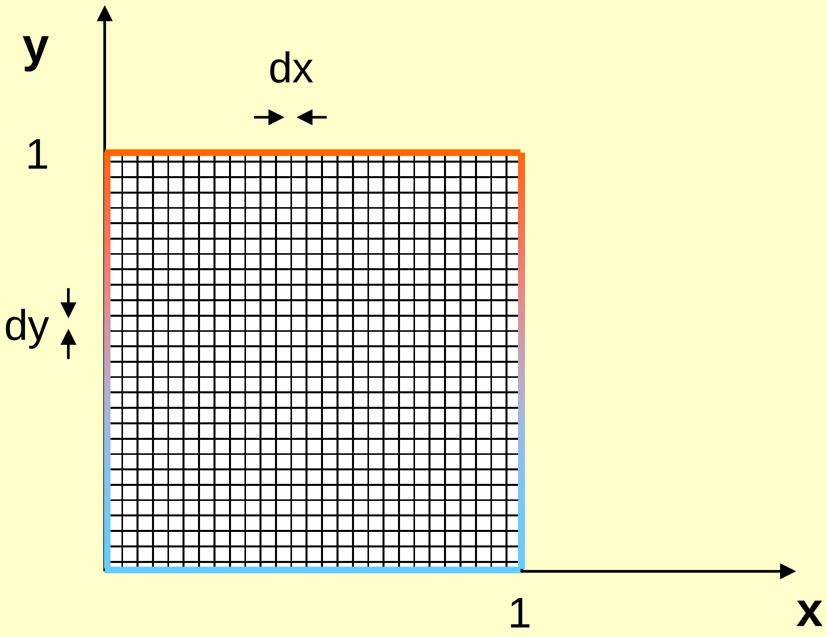
    $\Phi(0,y,t) = y = \Phi(1,y,t)$
  - Initial value within interior of square: zero
- **Temporal evolution:**
  - to stationary state
  - partial differential equation



$$\frac{\partial\Phi}{\partial t} = \frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2}$$

Heat conduction (2):
algorithm for solution of IBVP

- **Interested in stationary state**
  - discretization in space: x_i, y_i
    → 2-D Array Φ
  - discretization in time:
    → steps δt



repeatedly calculate increments

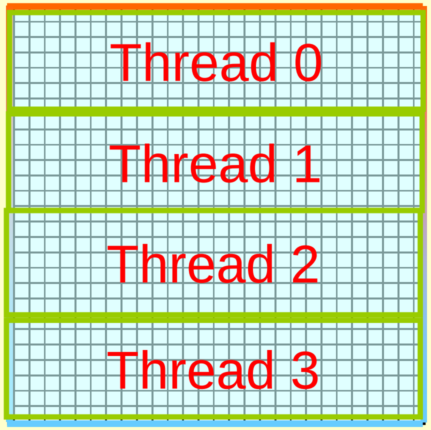
$$\delta\Phi(i,k) = \delta t \cdot \left[\frac{\Phi(i+1,k) + \Phi(i-1,k) - 2\Phi(i,k)}{dx^2} + \frac{\Phi(i,k+1) + \Phi(i,k-1) - 2\Phi(i,k)}{dy^2} \right]$$

until δΦ=0 reached.

Heat Conduction (3):

data structures

- 2-dimensional array `phi` for heat values
- equally large `phin`, to which updates are written
- Iterate updates until stationary value is reached
- Both arrays shared
  - since grid area is to be tiled to OpenMP threads



Heat Conduction (4):

code for updates

```
! iteration
do it=1,itmax
   dphimax=0.
!$OMP parallel do private(dphi,i) &
!$OMP reduction(max:dphimax)
   do k=1,kmax-1
   do i=1,imax-1
      dphi=(phi(i+1,k)+phi(i-1,k)-
    2.0_8*phi(i,k))*dy2i  &
          +(phi(i,k+1)+phi(i,k-1)-
    2.0_8*phi(i,k))*dx2i
      dphi=dphi*dt
      dphimax=max(dphimax,abs(dphi))
      phin(i,k)=phi(i,k)+dphi
   enddo
   enddo
!$OMP end parallel do
```

“parallel do”:
✓is a semantic fusion of “parallel” and “do”

```
!$OMP parallel do
   do k=1,kmax-1
   do i=1,imax-1
      phi(i,k)=phin(i,k)
   enddo
   enddo
!$OMP end parallel do
!required precision reached?
   if (dphimax.lt.eps) goto 10
enddo
10 continue
```

Reduction clause (1)

- `dphimax` has both **shared** and **private** characteristics, since maximum over all grid points required

  → new data attribute **reduction**,
  combined with an operation

- General form of reduction operation:

```
!$OMP do reduction (Operation : X)
   DO
      ...
      X = X Operation Expression     (*)
      ...
   END DO
!$OMP end do
```

The variable X is used as (scalar) reduction variable.

Reduction clause (2):

what can be reduced?

Operation	Initial Value	Remarks
+	0	
*	1	
-	0	X = Expression – X not allowed
.AND.	.TRUE.	
.OR.	.FALSE.	
.EQV.	.TRUE.	
.NEQV.	.FALSE.	
MAX	Smallest representable number	
MIN	Largest representable number	
IAND	All bits set	
IOR	0	
IEOR	0	

For function like e. g., MAX, can replace (*) by

```
X = MAX(X,Expression)
```

or

```
IF (X <= Expression) X = Expression
```

Reduction clause (3): reduction rules

- private copies of reduction variables exist during execution of parallel region
- private copies are initialized as shown in table above
- Reduction to shared reduction variable at synchronization point
  - beware nowait clause!
- More than one reduction variable: comma-separated List

```
!$OMP do reduction (+ : x, y, z)
```

- More than one reduction method:

```
!$OMP do reduction (+ : x, y) reduction(max : z)
```

- Operation specified in clause must be consistent with actually performed operation in Fortran code!
  - associativity and commutativity / ordering

Reduction clause (4): Array reductions

i.e., using an array in the reduction clause

- **are allowed since OpenMP 2.0**

- **Restrictions:**
  - no deferred shape or assumed size or allocatable arrays
    - size must be known at compile time
  - beware performance/scalability issues for large arrays!

Short break

10 Minutes

Controlling OpenMP execution

Loop Scheduling

Synchronization

Conditional Parallelism

Default scheduling of parallel loops

Division of work:

- default decided by vendor
- usually: static scheduling
- divide iteration space into largest possible chunks of equal size

Behaviour of Intel Compiler

- default is

`KMP_SCHEDULE="static,greedy"`

- optionally use

`KMP_SCHEDULE="static,balanced"`

non-standard runtime setting

```
!$omp do                  T 0          T 1          T 2
  do i=1,9              do i=1,3     do i=4,6     do i=7,9
   ...       ------>      ...          ...          ...
  end do                end do       end do       end do
!$omp end do
```

	Number of Iterations			
	Thr. 0	Thr. 1	Thr. 2	Thr. 3
	3	3	3	0
	3	2	2	2

User-determined scheduling (1)

Varying chunk sizes

- **What if we want to fix chunk size ourselves?**
  - use the schedule clause

```
!$OMP do schedule(static,chunk)
```

  - **chunk** is optional, of integer type, positive value, unchanged during loop execution
  - if omitted, one chunk of greatest possible size assigned to each thread
  - otherwise assignment of chunks to threads in round-robin order
- **Potentially beneficial effect:**
  - together with suitably inserted pre-fetches, non-maximal chunks may lead to improved overall data locality

User-determined scheduling (2)

Coping with load imbalances

- **How about this:**

```
!$omp do
do i=1,n
  if (iw(i) > 0) then
    call smallwork(...)
  else
    call bigwork(...)
  end if
 end do
!$omp end do
```

- **static scheduling will probably give a load imbalance**
  - idling threads

- **Fix this using a dynamic schedule**

```
!$OMP do &
!$OMP schedule(dynamic,chunk)
```

  - **chunk** is optional (as before)
  - if omitted, chunk is set to 1
  - each thread, upon completing its chunk of work, **dynamically** gets assigned the next one
  - in particular, the assignment may change from run to run of the program

- **Recommendations:**
  - sufficiently fat loop body
  - execution overhead much higher than for static scheduling (extra per-chunk synchronization required!)

User-determined scheduling (3)

Guided schedule

- **Number of chunks in simple dynamic scheduling**
  - too small → large overhead
  - too large → load imbalance
- **possible solution: dynamically vary chunk size**
  - guided schedule
- **If**
  - N = iteration count
  - P = thread count

start with chunk size $C_0 = \frac{N}{P}$

and dynamically continue with

$$C_k = \left(1 - \frac{1}{P}\right) \cdot C_{k-1}$$

- **This yields**
  - exponentially decreasing chunk size
  - and hence number of chunks may be greatly decreased (grows logarithmically with N!)
  - all iterations are covered
- **Syntax of guided clause:**

```
!$OMP do &
!$OMP schedule(guided,chunk)
```

  - if chunk is specified, it means the minimum chunk size
  - correspondingly, C_0 may need to be adjusted

User-determined scheduling (4)

Deferring the scheduling decision to run time

- Run time scheduling via

```
!$OMP do &
!$OMP schedule(runtime)
```

will induce the program to determine the scheduling at run time according to the setting of the

OMP_SCHEDULE

environment variable

- Disadvantage: chunk sizes are fixed throughout program

Possible values of OMP_SCHEDULE and their meaning	
“static,120”	static schedule, chunk size 120
“dynamic”	dynamic schedule, chunk size 1
“guided,3”	guided schedule, minimum chunk size 3

Synchronization (1)

Barriers

- **Remember: at the end of an OpenMP parallel loop all threads synchronize**
  - consistent access to all information in variables with shared scope is guaranteed to (parallel) execution flow after loop
- **This can also be explicitly programmed by the user:**

  !$OMP BARRIER

  - synchronization requirement:

    the execution flow of each thread blocks upon reaching the barrier until all threads have reached the barrier
  - barrier may not appear within !$omp single or !$omp do block (deadlock!)

Synchronization (2):

Relaxing synchronization requirements

- **end do (and: end sections, end single, end workshare)**
  - imply a barrier by default
  - this may be omitted if the nowait clause is specified
    - potential performance improvement
    - especially if load imbalance occurs within construct
  - Beware: race conditions!

```
            !$omp parallel
            !$omp do shared(a)
              ... (loop)
Thread 0 →    a(i) = ...
            !$omp end do nowait
              ... (some other parallel work)
            !$omp barrier
Thread 1 →    ... = a(i)
            !$omp end parallel
```

threads continue without waiting

Synchronization (3):

The “master” and “single” directives

Single directive:

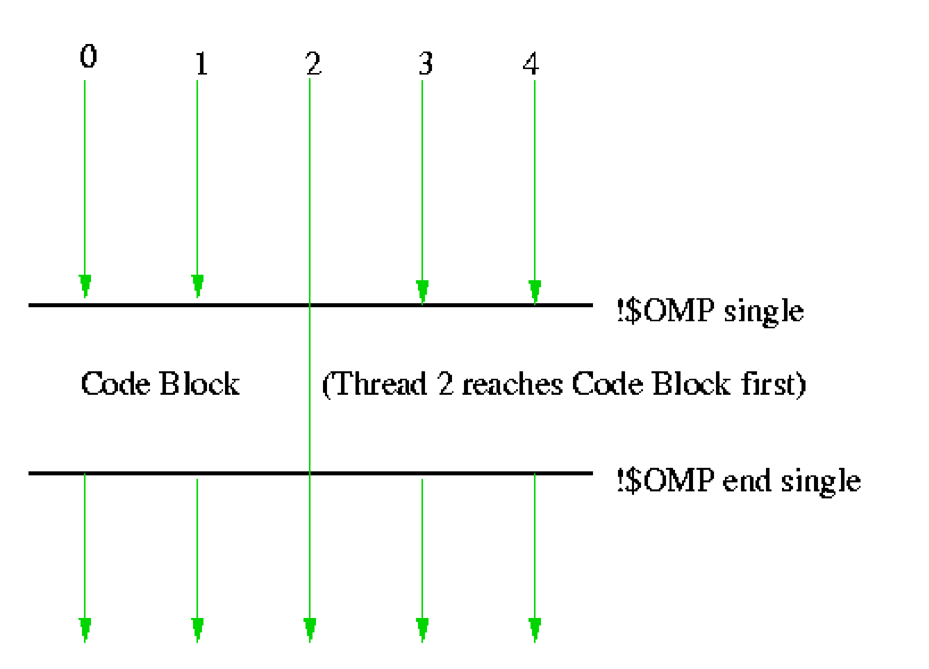
- only one thread executes
- others synchronize

Master directive

similar to single, but

- only thread 0 executes
- others continue
- binds only to current team

→ not all threads must reach code section



Single:

- may not appear within a parallel do (deadlock!)
- nowait clause after end single suppresses synchronization
- copyprivate(var) clause after end single provides value of private variable `var` to other threads in team (OpenMP 2.0)

Synchronization (4)

The “critical” and “atomic” directives

- **These have already been encountered**
  - each thread executes code (in contrast to single)
  - but only one at a time within code
  - with synchronization of each when exiting code block
  - atomic: code block must be a single line update

Fortran:

```
!$omp critical              !$omp atomic
block                       x = x <op> ...
!$omp end critical
```

C/C++:

```
# pragma omp critical       # pragma omp atomic
  block                       x = x <op> ... ;
```

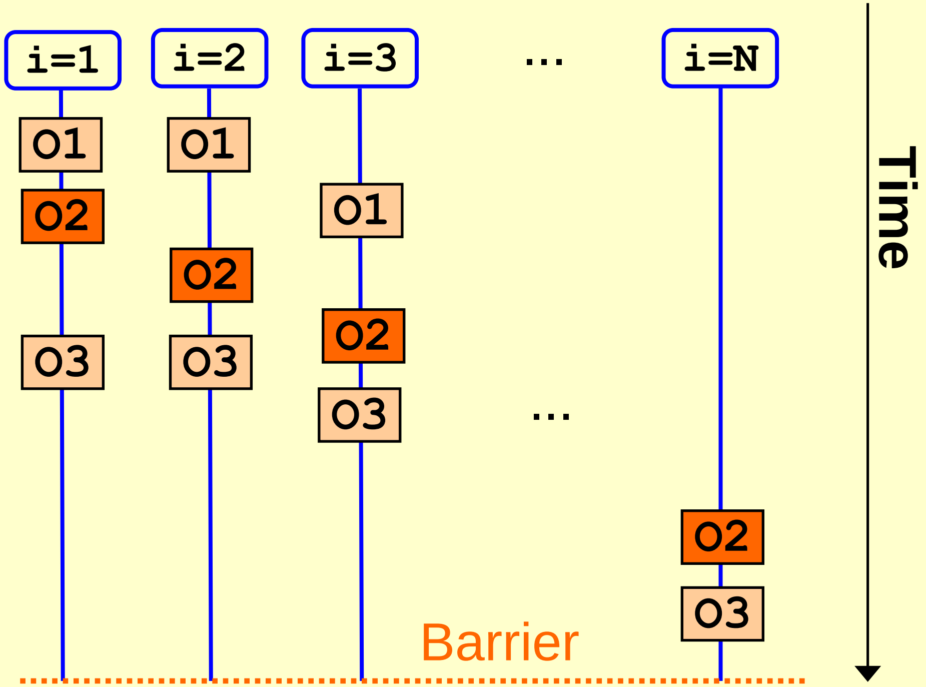
Synchronization (5)

The “ordered” directive

Statements must be within body of a loop

- Acts as single directive, threads do work ordered as in seq. execution
- Requires ordered clause to `$!OMP do`
- Only effective if code is executed in parallel
- Only one ordered region per loop
- Execution scheme:

```
!$OMP do ordered
do I=1,N
   O1
!$OMP ordered
   O2
!$OMP end ordered
   O3
end do
!$OMP end do
```



Two typical applications of “ordered”

- **Loop contains recursion**
  - not parallelizable
  - but should be only small part of loop

```
!$OMP do ordered
do I=2,N
  ... (large block)
!$OMP ordered
  a(I) = a(I-1) + ...
!$OMP end ordered
end do
!$OMP end do
```

- **Loop contains I/O**
  - results should be consistent with serial execution

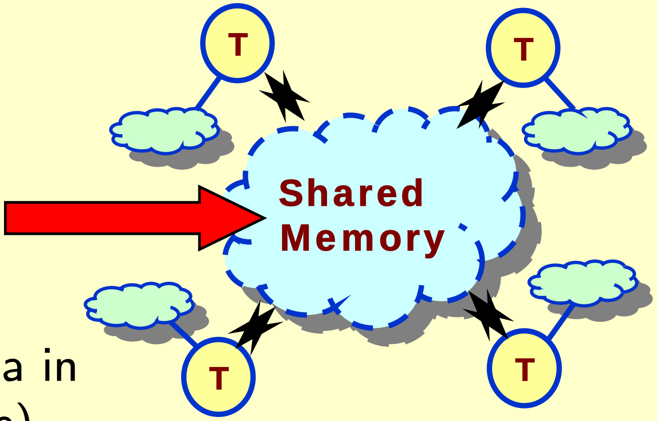
```
!$OMP do ordered
do I=1,N
  ... (calculate a(:,I))
!$OMP ordered
  write(unit,...) a(:,I)
!$OMP end ordered
end do
!$OMP end do
```

Synchronization (6)
Why do we need it?

Remember OpenMP Memory Model

- ✓ private (thread-local):
  - ➢ no access by other threads
- shared: two views
  - ➢ temporary view: thread has modified data in its registers (or other intermediate device)
  - ➢ content becomes inconsistent with that in cache/memory
  - ➢ other threads: cannot know that their copy of data is invalid

    Note: on the cache level, the coherency protocol guarantees this knowledge



Synchronization (7)
Consequences and Remedies

- **For threaded code without synchronization this means**
  - multiple threads writing to same memory location → resulting value is unspecified
  - one thread reading and another writing → result on (any) reading thread unspecified

- **Flush Operation**
  - performed on a set of (shared) variables

    → flush-set
  - discard temporary view
    - modified values forced to cache/memory (requires exclusive ownership)
    - next read access must be from cache/memory
  - further memory operations only allowed after all involved threads complete flush
    - restrictions on memory instruction reordering (by compiler)

Synchronization (8):
... and what must the programmer do?

- **Ensure consistent view of memory**
  - Assumption: Want to write something with first thread, read it with second
- **Order of execution required:**
  1. Thread 1 writes to shared variable
  2. Thread 1 flushes variable
  3. Thread 2 flushes same variable
  4. Thread 2 reads variable

- **OpenMP directive for explicit flushing**

```
!$OMP FLUSH [(var1,var2)]
```

applicable to all variables with shared scope including
  - SAVE, COMMON/Module globals
  - dummy arguments
  - pointer dereferences
- **If no variables specified, flush-set**
  - encompasses all shared variables
  - which are accessible in the scope of the FLUSH directive

Synchronization (9):
Example for explicit flushing

```
integer :: isync(0:nthrmax)
...
isync(0) = 1       ! dummy for
                   ! thread 0
!$omp parallel private(myid,neigh,...)
myid = omp_get_thread_num() + 1
neigh = myid - 1
isync(myid) = 0
!$omp barrier
   ... (work chunk 1)
isync(myid) = 1
!$omp flush(isync)
do while (isync(neigh) == 0)
  !$omp flush(isync)
end do
   ... (work chunk 2, dependency!)
!$omp end parallel
```

- to each thread its own flush variable + 1 dummy

- per-thread information
- Need to use OpenMP library function

Synchronization (10)
Implicit synchronization

- **Implicit barrier synchronization:**
  - at the beginning and end of parallel regions
  - at the end of critical, do, single, sections blocks unless a nowait clause is allowed and specified
    - all threads in the present team are flushed
- **Implicit flush synchronization:**
  - as a consequence of barrier synchronization
  - but note that flush-set then encompasses all accessible shared variables
  - hence explicit flushing (possibly only with a subset of threads in a team) may reduce synchronization overhead → improve performance

Conditional parallelism: The “if” clause

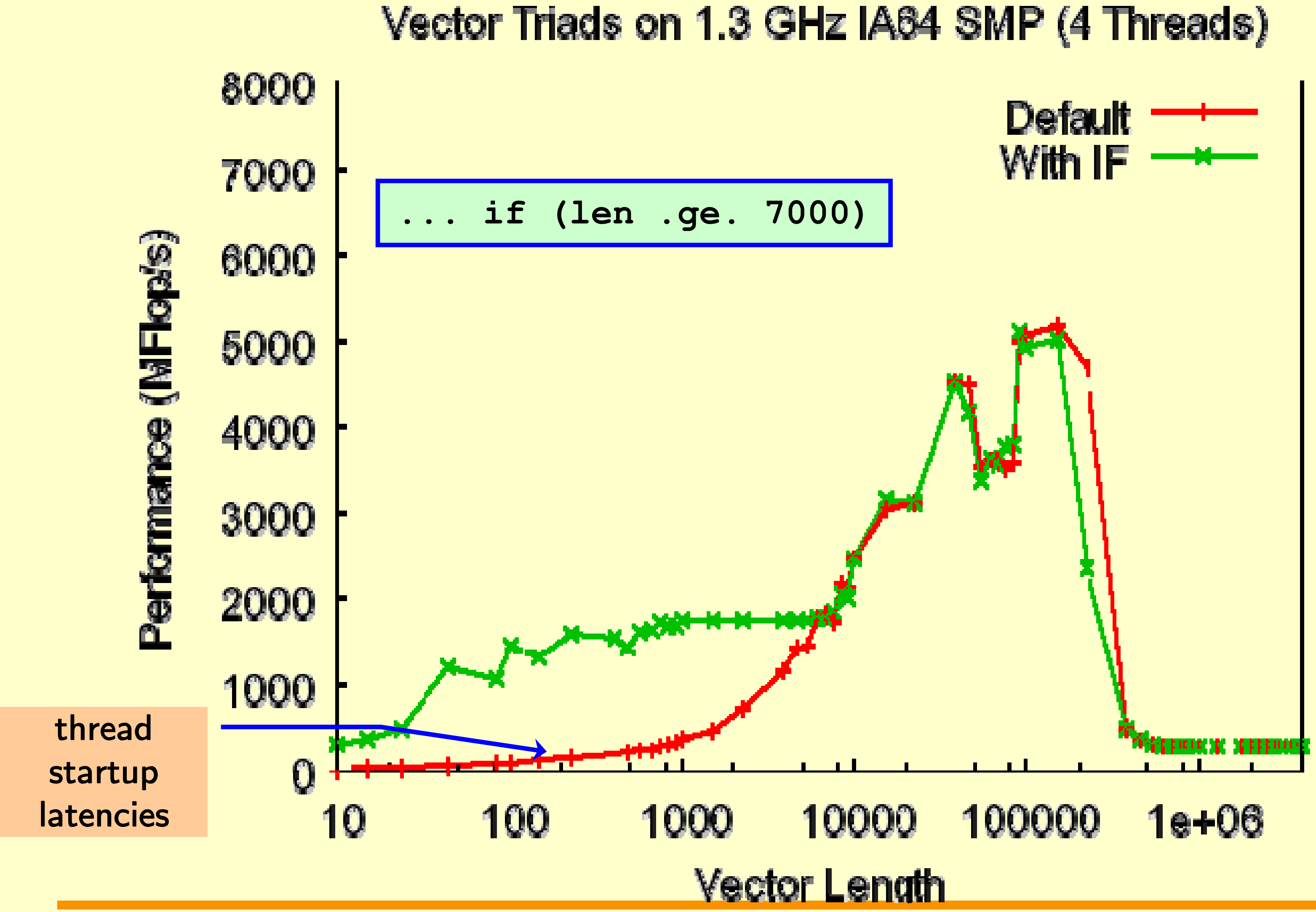
- **Syntax:**

```
!$omp parallel if (condition)
... (block)
!$omp end parallel
```

Fortran scalar logical expression

- **Usage: disable parallelism dynamically**
  - by using `omp_in_parallel()` library call to suppress nested parallelism
  - define crossover points for optimal performance
    - may require manual or semi-automatic tuning
    - may not need multi-version code

Example for crossover point: Vector triad with 4 threads on IA64



```
... if (len .ge. 7000)
```

thread startup latencies

Going beyond loop-level parallelism

Further work sharing constructs

OpenMP library routines

Global Variables

Further possibilities for work distribution

- parallel region is executed by all threads.
- what possibilities exist to distribute work?
  1. `!$OMP do`
  2. parallel sections
  3. workshare
  4. For hard-boiled MPI programmers: by thread ID
- parallel sections (within a parallel region):

```
!$OMP sections
!$OMP section
   code (thread #0)
!$OMP section
   code (thread #1)
   ...
!$OMP end sections
```

Parallel Sections: Ground rules

- **clauses:** `private, firstprivate, lastprivate, nowait` **and** `reduction`
- `section` **Directives allowed only within lexical extent of** `sections/end sections`
- **more sections than threads:**
  - last thread executes all excess sections sequentially (SR8000-specific)
  - Hence be careful about dependencies
- **more threads than sections:**
  - Excess threads synchronize unless nowait clause was specified
- **as usual: no branching out of blocks**

Handling Fortran 90 array syntax: the “workshare” directive

- **Replace loop by array expression**

```
do i=1,n
  a(i) = b(i)*c(i) + d(i)
end do
```

```
a(1:n) = b(1:n)*c(1:n) + d(1:n)
```

how do we parallelize this?

```
!$omp parallel
!$omp workshare
a(1:n) = b(1:n)*c(1:n) + d(1:n)
!$omp end workshare
!$omp end parallel
```

- an OpenMP 2.0 feature
- not available in C
- end workshare can have nowait clause

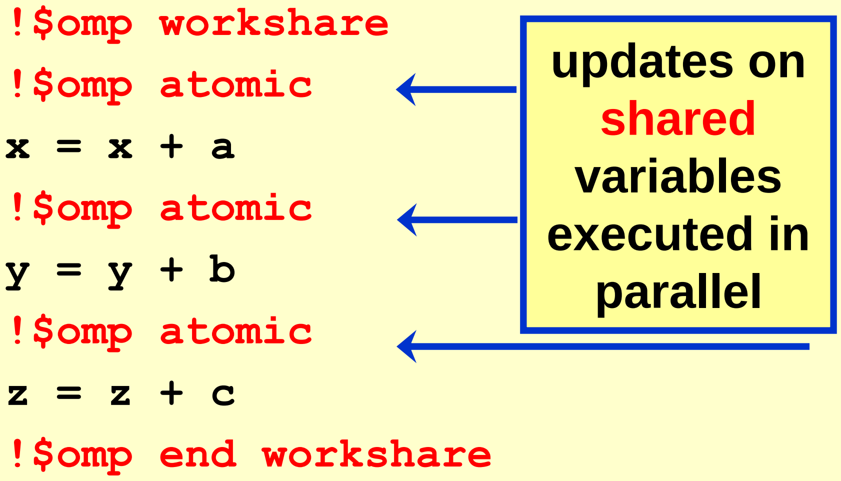
Intel Fortran Compiler:

- ➢ supports directive in 9.0 release
- ➢ but no performance increase registered for above example

Semantics of “workshare” (1)

- **Division of enclosed code block into units of work**
  - units are executed in parallel
- **Array expressions, Elemental functions**
  - each element a unit of work
- **Array transformation intrinsic (e.g., matmul)**
  - may be divided into any number of units of work
- **WHERE**
  - mask expr., then masked assignment workshared
- **FORALL**
  - WHERE + iteration space

- **OpenMP directives as units of work**



```
!$omp workshare
!$omp atomic
x = x + a
!$omp atomic
y = y + b
!$omp atomic
z = z + c
!$omp end workshare
```

updates on shared variables executed in parallel

- **also possible with:**
  - critical directive
  - parallel region → nested parallelism!

Semantics of “workshare” (2)

- implementation must add necessary synchronization points to preserve Fortran semantics

```
res = 0
n = size(aa)
!$omp parallel
!$omp workshare
aa(1:n) = bb(1:n) * cc(1:n)
!$omp atomic
res = res + sum(aa)
dd = cc * res
!$omp end workshare
!$omp end parallel
```

makes implementation difficult

sync

sync

Further remarks on “workshare”

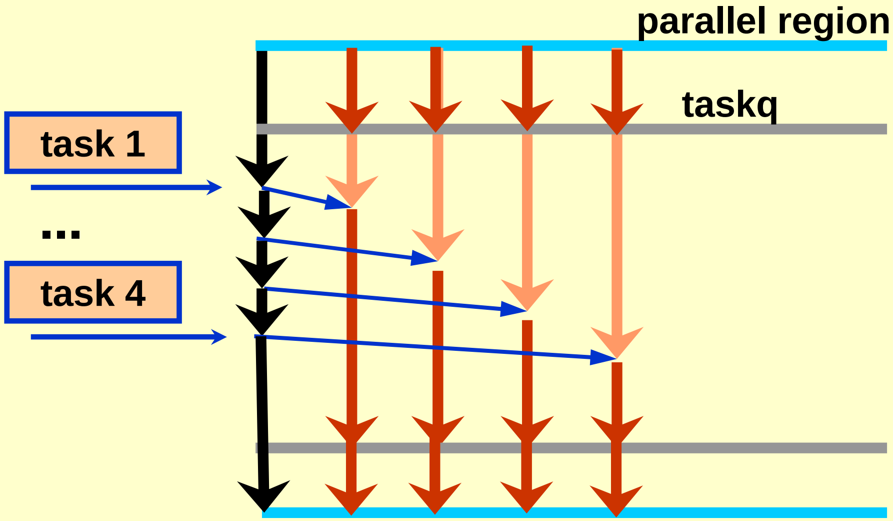
- **Referencing private variables**
  - should not be done (undefined value)
- **Assigning to private variables (in array expressions)**
  - should not be done (undefined values)
- **Calling user defined functions / subroutines**
  - should not be done unless ELEMENTAL

An extension to OpenMP: Task queuing

- This is an Intel-specific directive
  - presently only available for C/C++
  - submitted for inclusion in next OpenMP standard (3.0)
- Idea:
  - decouple work iteration from work creation
  - remember restrictions for `!$omp do` on loop control structures?
  - one thread administers the task queue
  - the others are assigned a task (=unit of work) at a time each

- This generalizes work sharing via
  - sections
  - loops

and can be applied to
  - while loops
  - C++ iterators
  - recursive functions



Task queuing directives and clauses

- **Setting up the task queue is performed via**

```
#pragma omp  parallel
{
#pragma intel omp taskq [cl.]
  { ...  // seq. setup code
#pragma intel omp task [cl.]
    {...
     // independent unit of work
    }
  }
}
```

sequential consistency

- **The taskq directive takes the clauses**
  - private, firstprivate, lastprivate, reduction, ordered, nowait
- **The task directive takes the clauses**
  - private: thread-local default-constructed object
  - captureprivate: thread-local copy-constructed object
  - all private, firstprivate and lastprivate variables on a taskq directive are by default captureprivate on enclosed task directives

Example for usage of task queuing

```
void foo(List *p)
{
#pragma intel omp parallel taskq shared(p)
  {
     while (p != NULL)
     {
#pragma intel omp task captureprivate(p)
        {
           do_work1(p);
        }
        p = p->next;
     }
  }
}
```

unit of work

Note on recursive functions:

- taskq directive can be nested
- will always use the team initially bound to

OpenMP library routines (1)

- **Querying routines**
  - how many threads are there?
  - who am I?
  - where am I?
  - what resources are available?
- **Controlling parallel execution**
  - set number of threads
  - set execution mode
  - implement own synchronization constructs

OpenMP library routines (2)

These function calls return type INTEGER

`num_th = OMP_GET_NUM_THREADS()`

- yields number of threads in present environment
- always 1 within sequentially executed region

`my_th = OMP_GET_THREAD_NUM()`

- yields index of executing thread (0, ...,num_th-1)

`num_pr = OMP_GET_NUM_PROCS()`

- yields number of processors available for multithreading

→ Always 8 for SR8000, number of processors in SSI for SGI (128 at LRZ)

How to reliably obtain the available number of threads

- e.g., at beginning of program
- with a shared **num_th**

```
!$omp parallel
!$omp master
num_th=omp_get_num_threads()
!$omp flush(num_th)
!$omp end master
...
!$omp end parallel
```

OpenMP library routines (3)

```
max_th = OMP_GET_MAX_THREADS()
```

maximum number of threads potentially available
e.g., as set by operating environment/batch system

The subroutine call (must be in sequential part!)

```
call OMP_SET_NUM_THREADS(nthreads)
```

sets number of threads to a definite value

```
0 < nthreads ≤ omp_get_max_threads()
```

- useful for specific algorithms
- dynamic thread number assignment must be deactivated
- overrides setting of `OMP_NUM_THREADS`

OpenMP library routines (4)

The logical function

am_i_par = OMP_IN_PARALLEL()

queries whether program is executed **in parallel or sequentially**

Timing routines (double precision functions):

ti = OMP_GET_WTIME()

returns elapsed wall clock time in seconds

- arbitrary starting point → calculate increments
- not necessarily consistent between threads

ti_delta = OMP_GET_WTICK()

returns precision of the timer used by **OMP_GET_WTIME()**

OpenMP library routines (5)
Dynamic threading

Alternative to user specifying number of threads:

- Runtime environment adjusts number of threads
- For fixed (batch) configurations probably not useful
- Activate this feature by calling

```
call omp_set_dynamic(.TRUE.)
```

- check whether enabled by calling the logical function

```
am_i_dynamic = omp_get_dynamic()
```

- If implementation does not support dynamic threading, you will always get `.FALSE.` here

OpenMP library routines (6)

Function/Subroutine calls for

- nested parallelism
- locking

will be discussed later

OpenMP library routines (7)

- **Library calls:**
  - destroy sequential consistence unless conditional compilation is used and some care is taken (e.g., default values for thread ID and numbers)
- **Fortran 77 `INCLUDE` file / Fortran 90 module**
  - correct data types for function calls!
- **Stub library**
  - for purely serial execution if `!$` construction not used
- **Intel Compiler**
  - include files, stub library and Fortran 90 module
  - replace `-openmp` switch by `-openmp_stubs`
- **SR8000 Compiler**
  - include files
  - stub library provided by LRZ. Link with

    `-L/usr/local/lib/OpenMP/ -lstub[_64]`
  - no Fortran 90 module (but can generate yourself from include file)

Using global variables in threaded programs

- **Numerical integration once more:**
  - use a canned routine (NAG: **D01AHF**)
  - do multiple integrations → why not in parallel?

```
!$omp parallel do
do i=istart,iend
   ... (prepare)
   call d01ahf(..., my_fun, ...)
end do
!$omp end parallel do
```

needs to be function of a single variable

- **Pitfalls:**
  - Is the vendor routine thread-safe? → documentation/tests
  - How are function calls (**my_fun**) treated? → discussed now

Using global variables (2)

Very typically, function values are provided by API call

```
call fun_std_interface(arg, par1, par2, ..., result)
```

so need to introduce globals e.g., via COMMON:

```
real function my_fun(x) result(r)
  double precision :: par1, par2, r, x
  common /my_fun_com/ par1, par2

  call fun_std_interface(x, par1, par2, ..., r)
end function my_fun
```

Using global variables (3)

Now, can we have

```
double precision :: par1, par2
common /my_fun_com/ par1, par2
...
!$omp parallel do private(par1,par2)
do i=istart,iend
   par1 = ...
   par2 = ...
   call d01ahf(..., my_fun, ...)
end do
!$omp end parallel do
```

?

will not work!
how can the compiler
know about what to do
elsewhere in the code?

will not work!
par1,par2 need
private scope

↔

COMMON is shared

Using global variables (4):
The “threadprivate” directive

Fix problem by declaring COMMON block threadprivate

```
double precision :: par1, par2
 common /my_fun_com/ par1, par2
 !$omp threadprivate ( /my_fun_com/ )
```

Notes:

- This must happen for **every** routine that references **/my_fun_com/** → if possible use INCLUDE to prevent mistakes
- Variables in **threadprivate** may **not** appear in **private, shared or reduction** clauses
- In serial region: values for thread 0 (master)
- In parallel region: copies for each thread created, with **undefined** value
- More than one parallel region:
  - no dynamic threading
  - number of threads must be constant for data persistence
- Only **named** COMMON blocks can be privatized

Using global variables (5):

The “copyin” clause

What if I want to use (initial) values calculated in a sequential part of the program?

```
par1 = 2.0d0
 !$omp parallel do copyin(par1)
 do i=istart,iend
   par2 = ...
   call d01ahf(..., my_fun, ...)
   par1 = ... (may depend on integration result)
 end do
 !$omp end parallel do
```

→ `par1` value for thread 0 is copied to all threads at beginning of parallel region

(Alternative: DATA initialization. Not supported e.g. on SR8000 ...)

Using global variables (6):

... and how about module variables?

The following will work

```
module my_fun_module
  double precision, save :: par1, par2
!$omp threadprivate (par1,par2)
contains
  function my_fun(x) result(r)
    double precision :: r, x

    call fun_std_interface(x, par1, par2, ..., r)
  end function my_fun
end module my_fun_module
```

only necessary for purely serial program

- and is much more elegant – if an OpenMP 2.0 conforming implementation is available

Advanced OpenMP concepts

Binding of Directives

Nested Parallelism

Programming with Locks

Binding of Directives (1)

Which parallel region does a directive refer to?

- `do, sections, single, master, barrier:`

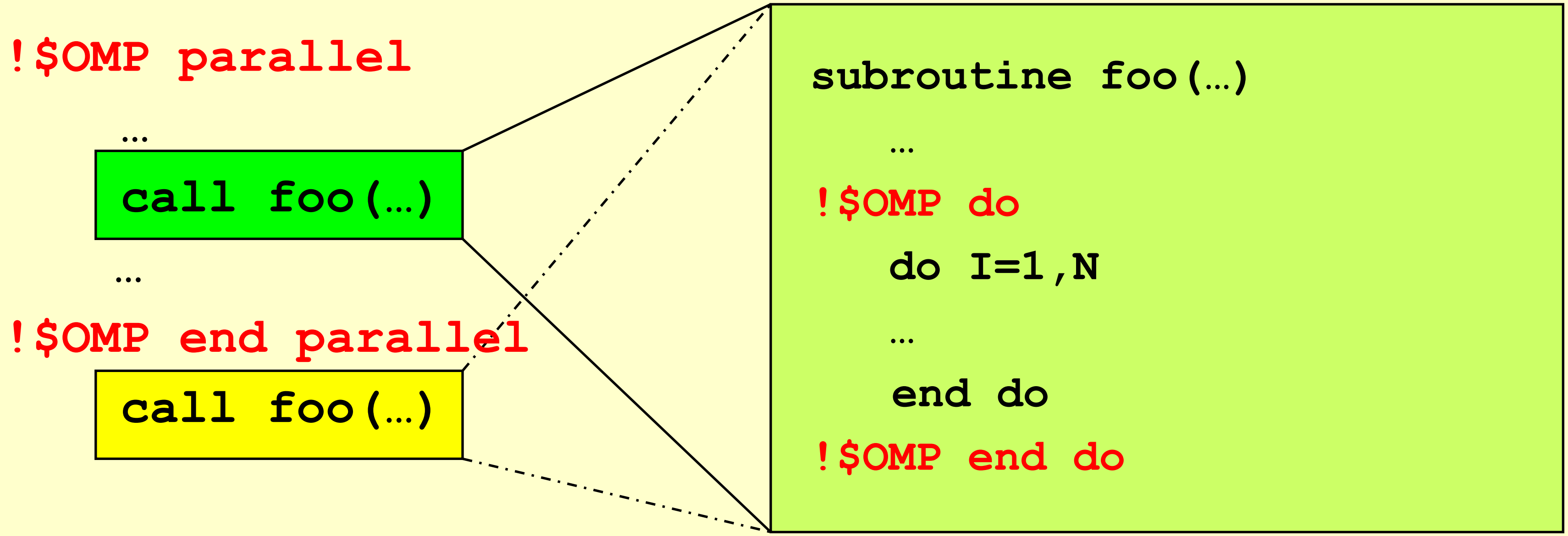
  to (dynamically) closest enclosing parallel region, if one exists

  "orphaning":

  only one thread if not bound to a parallel region

  Note: close nesting of `do, sections` not allowed
- `ordered`: binds to dynamically enclosing `do`
- `ordered`: not in dynamical extent of `critical` region.
- `atomic,critical`: exclusive access for all threads, not just current team

Binding of Directives (2)
Orphaning



```
!$OMP parallel
     …
     call foo(…)
     …
!$OMP end parallel
     call foo(…)
```

```
subroutine foo(…)
     …
!$OMP do
     do I=1,N
     …
     end do
!$OMP end do
```

Inside parallel region:

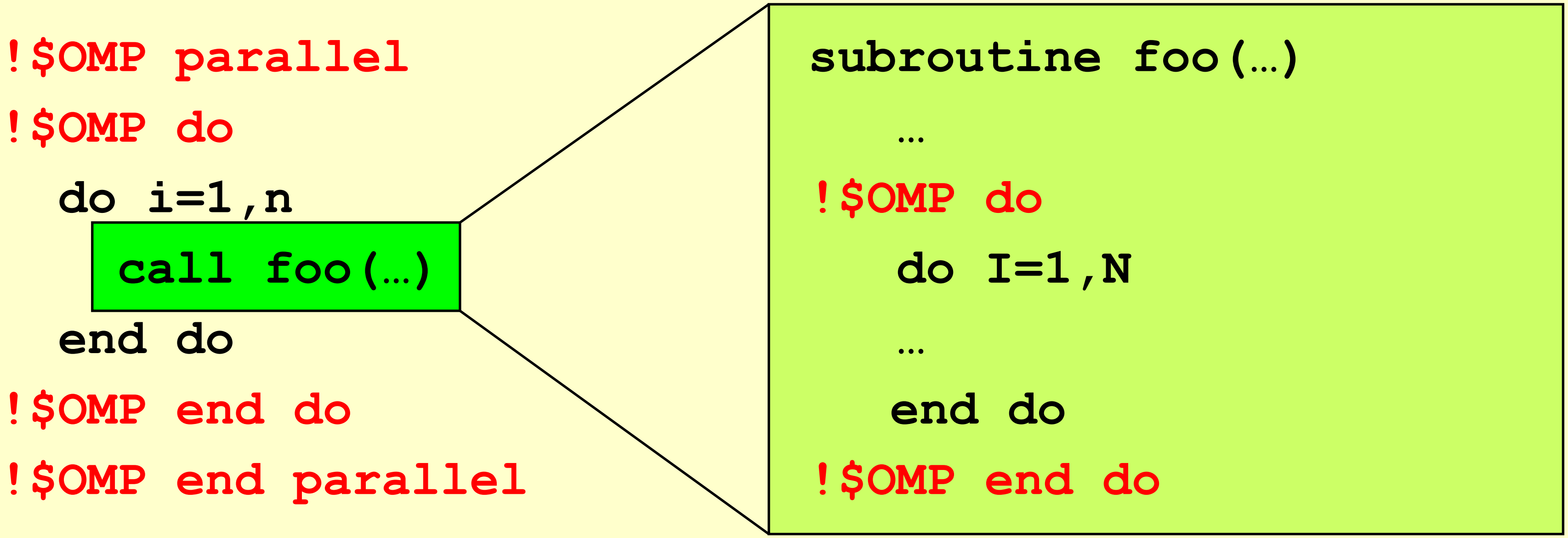
foo called by all threads

Outside parallel region:

foo called by one thread

- **OpenMP directives in foo are orphaned**
  - since they may or may not bind to a parallel region
  - decided at runtime
  - in both cases executed correctly

Binding of directives (3)
Example for incorrect nesting



```
!$OMP parallel
!$OMP do
  do i=1,n
    call foo(…)
  end do
!$OMP end do
!$OMP end parallel
```

```
subroutine foo(…)
    …
!$OMP do
    do I=1,N
    …
   end do
!$OMP end do
```

Not allowed:

do **nested within a** do

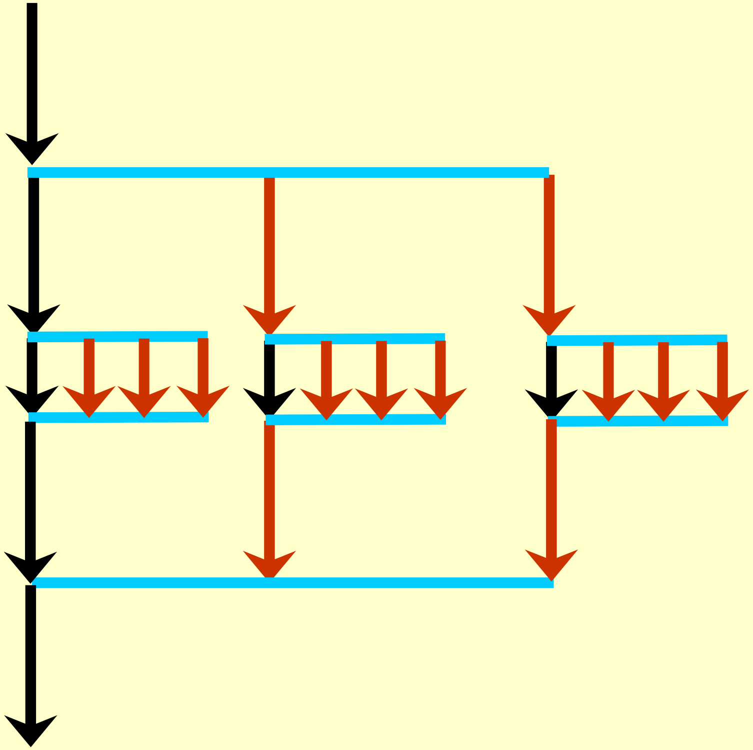
Nested parallelism (1)

```
!$OMP parallel
      code_1
!$OMP parallel
      code_2
!$OMP end parallel
      code_3
!$OMP end parallel
```

- code_1 and code_3 executed by team of threads
- code_2: each thread does work in serial by default
- nested parallelism enabled: additional threads may be created → behaviour is implementation-dependent

what could we wish for?

assumption: have 12 threads



Nested parallelism (2)

Controlling the number of threads:

- `omp_set_num_threads(n)` only callable in serial region
- `num_threads(n)` clause on parallel region directive
  - ➢ OpenMP 2.0

Environment Variable:

`OMP_NESTED`

- ❑ unset or set to “false”: **disable** nested parallelism
- ❑ set to “true”: **enable** nested parallelism **if supported** by implementation

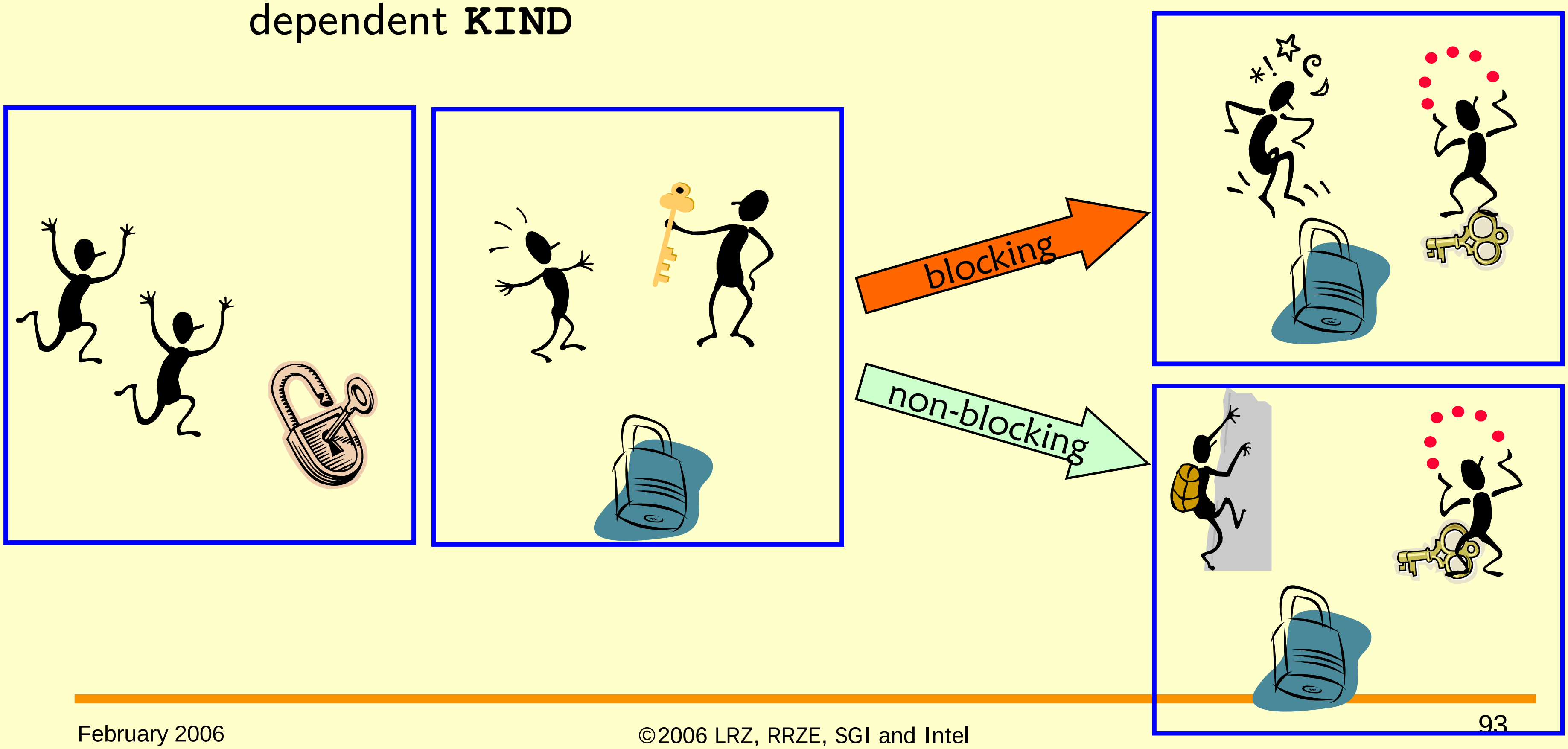
Run time check/control via service functions:

```
supp_nest=omp_get_nested()
call omp_set_nested(flag)
```

Need to re-check whether nesting supported before disposing thread distribution

Lock routines (1)

- **A shared lock variable can be used to implement specifically designed synchronization mechanisms**
  - In the following, **`var`** is an **`INTEGER`** of implementation-dependent **`KIND`**



Lock routines (2)

- `OMP_INIT_LOCK(var)`

  initialize a lock

  - lock is labeled by **`var`**
  - objects protected by lock: defined by **programmer** (red balls on previous slide)

  initial state is unlocked

  `var` not associated with a lock before this subroutine is called

- `OMP_DESTROY_LOCK(var)`

  disassociate `var` from lock

  `var` must have been initialized (see above)

Lock routines (3)

For all following calls: lock `var` must have been initialized

- **`OMP_SET_LOCK(var)`** :

  blocks if lock not available

  set ownership and continue execution if lock available

- **`OMP_UNSET_LOCK(var)`** :

  release ownership of lock

  ownership must have been established before

- logical function

  `OMP_TEST_LOCK(var)` :

  does not block, tries to set ownership

  → thread receiving failure can go away
  and do something else

Lock routines (4)

nestable locks:

- replace **`omp_*_lock(var)`** by **`omp_*_nest_lock(var)`**
- thread owning a nestable lock may re-lock it multiple times

  put differently:

  a nestable lock is available if

  - either it is unlocked

  or

  - it is owned by the thread executing **`omp_get_nest_lock(var)`**

    or **`omp_test_nest_lock(var)`**
- re-locking increments nest count
- releasing the lock decrements nest count
- lock is unlocked once nest count is zero

nestable locks are an OpenMP 2.0 feature!

Final remarks

- **Con: Automatic parallelization?**
  - use toolkits? (not available for SR8000)
  - some compilers also offer support for automatic parallelization
- **Con: Only a subset of proprietary functionality**
  - e. g., SR8000 (COMPAS) no pipelining in OpenMP (implement using barrier)
- **Performance: Beware of thread startup latencies!**
- **Pro: Portability**
- **Mixing OpenMP and MPI on SR8000:**
  - only one thread should call MPI
  - even then: OS calls not necessarily thread-safe, hence the other threads should not do anything sensitive
- **Mixing OpenMP and MPI on Altix:**
  - choose suitable threading level
  - in future, full multi-threading will be available (performance tradeoff?)

This ends the basic OpenMP stuff

... and we continue with practical considerations