



Compute•Calcul  
Ontario



<http://www.hpcvl.org>

# Shared-Memory Programming With OpenMP

2018 Ontario HPC Summer School  
Hartmut Schmider  
Centre for Advance Computing,  
Queen's University July/August 2017

# Course Requirements

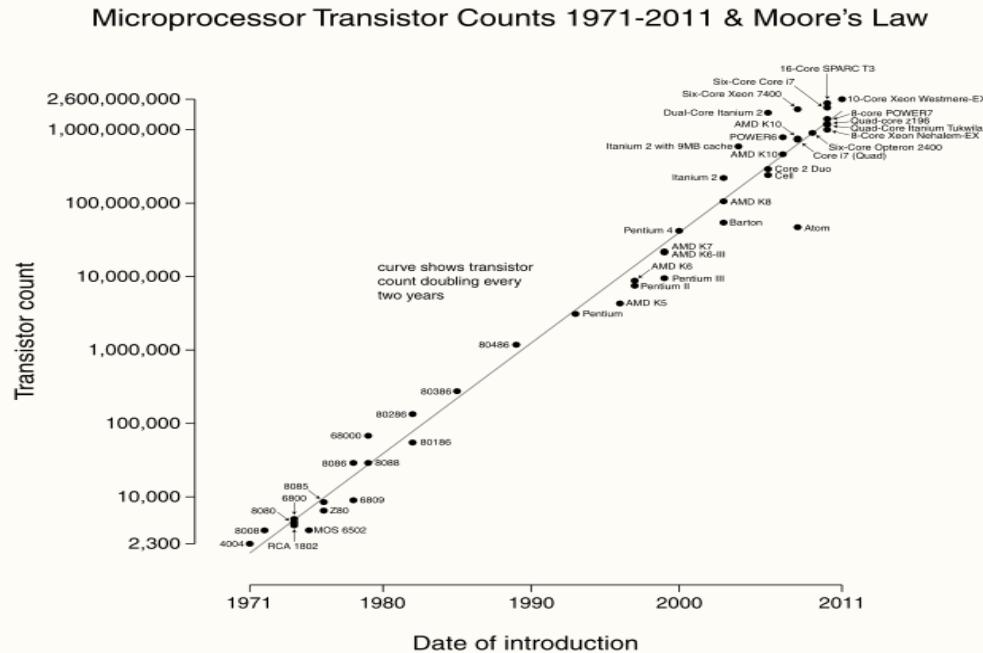
- ❑ No previous experience with parallel programming required
- ❑ Programming background with Fortran and/or C/C++ is useful
- ❑ Experience with Unix helps

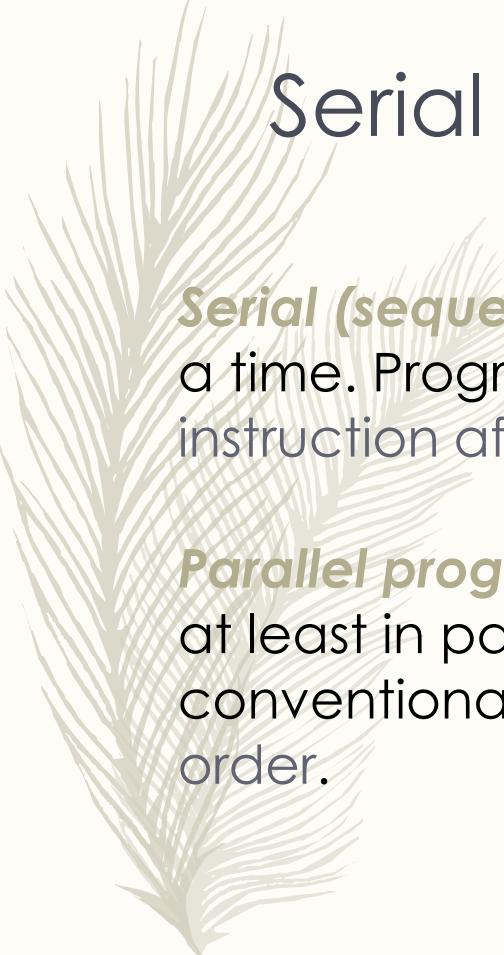
# Part 1 Outline

- Parallel Programming
- Shared Memory and Threads
- Explicit & Automatic Threading, OpenMP Directives
- OpenMP Directives, Clauses and Routine calls
- Loop Parallelism
- Shared and private variables, scoping
- Scheduling
- Fixing Dependences
- Usage of OpenMP on Unix

# Parallel Programming

- Exponential growth in speed (Moore's Law) of single CPUs is unsustainable
- Parallelism achieves performance increase (Moore's Law)
- Multiple processes run simultaneously
- Processes are static or dynamically created





# Serial and Parallel Programming

**Serial (sequential) program:** runs on one processor at a time. Program structure is conventional, one instruction after the other in a predictable order.

**Parallel program:** runs on several processors at a time, at least in part. Program structure might be non-conventional, instructions do not imply a specific order.

## Serial Code



P

Process



C

CPU

## Parallel Code

User

P

P

P

.....

P

System

C

C

.....

C



# Number of Processes

For efficiency, always choose the number of processes smaller than the number of available CPU's. This ensures that every process occupies one CPU exclusively, i.e. is executed on a dedicated processor.

# Parallelism & Concurrency

## Parallelism:

- More than one process is present and executing at a given time.
- Usually requires separate hardware, “cores” or CPU's.
- Used to scale programs, i.e. reduce execution time by a given factor.

## Concurrency:

- More than one process is present and active, but not always executed at the same time.
- Can be achieved with single core and CPU that “switches”.
- Increases flexibility and responsiveness.

# Instruction-Level Parallelism

- ILP appears on **local level** even in serial code.
- Usually, ILP is **exploited by the compiler**, using techniques such as pipelining, out-of-order execution, speculative execution, and branch-prediction.
- **Hardware** may **support** ILP, for instance through “superscalar” CPU's and pipelines.

...  
a+=c\*c;  
b+=d-e;  
g=a\*a+b;

...

Could be done simultaneously if CPU allows  
more than one floating-point operation

# Speedup, Scaling, Efficiency

- Speedup is the ratio between serial and parallel execution times:

$$S_p = T_1/T_p$$

- If the speedup is equal to the number of processors in the parallel case, the program is said to scale linear.
- In most (but not all) cases, the speedup will be smaller than the number of processors (sub-linear scaling).
- Efficiency is the ratio between the Speedup and the number of processors:

$$E_p = S_p/p$$

# Strong and Weak Scaling

## Strong Scaling:

How does the execution time vary as a function of the number of processors, given a fixed problem size. Linear best-case scenario:  $N$  times the processors,  $1/N$  times the execution time.

## Weak Scaling:

How does the execution time vary as a function of the number of processors if the problem size scales with the latter, i.e. given a fixed workload per processor. Linear best-case scenario: Execution time stays constant.

# Amdahl's Law

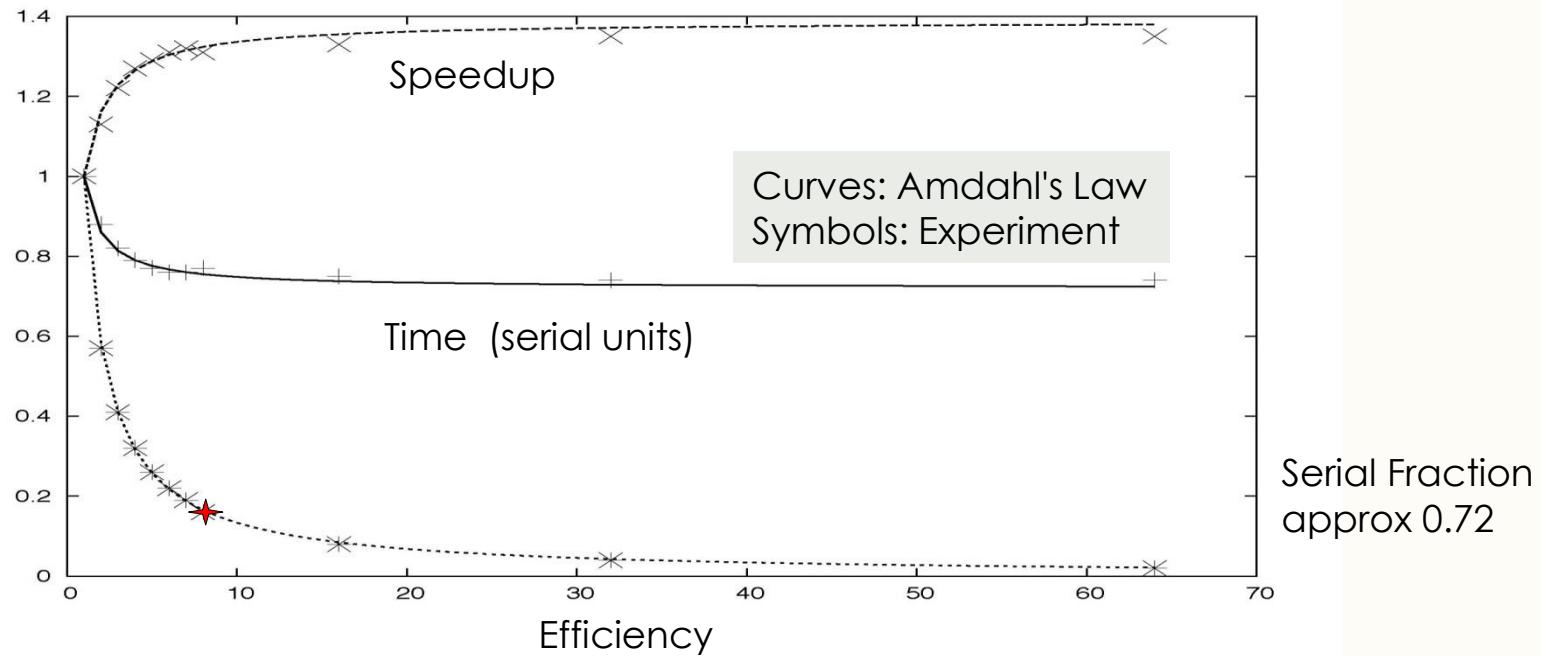
Amdahl's Law: The speedup for a parallel program is limited by the fraction of time spent for execution of the serial portion of the program,  $F_s$ . It is

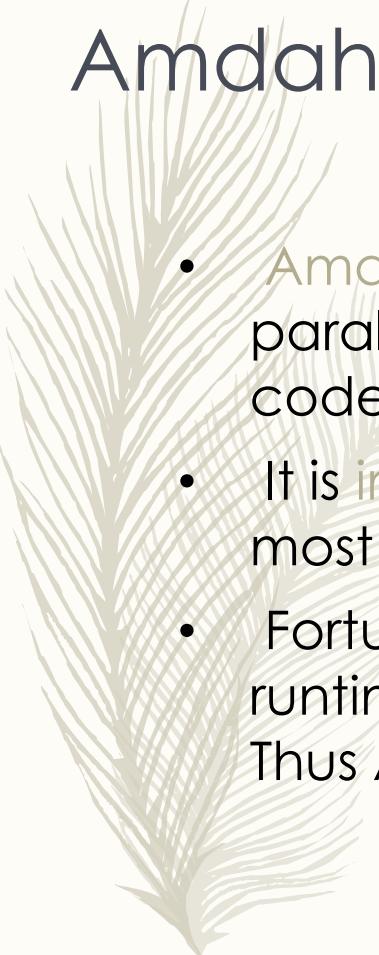
$$S_{max} = \left[ F_s + \frac{1 - F_s}{p} \right]^{-1} \leq \frac{1}{F_s}$$

This means that no matter how many processors, the speedup cannot exceed the inverse of the serial fraction.

# Amdahl's Law: Example

Here is the (pretty bad) scaling behaviour of a multithreaded dot-product code





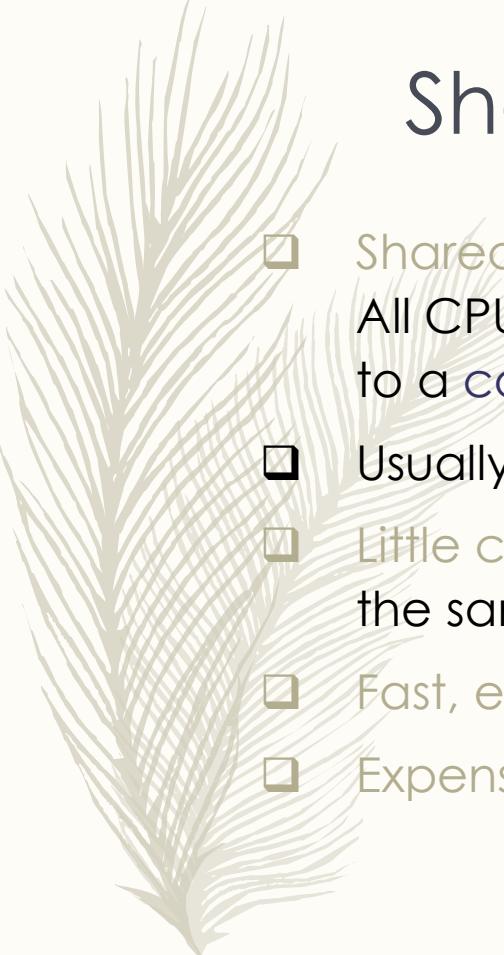
# Amdahl's Law (cont)

- Amdahl's Law is very relevant for shared-memory parallelization, because often only parts of the code are “parallelized”.
- It is important to parallelize those portions where most time is spent in a serial run.
- Fortunately, often the parallel portion of the runtime increases with increasing problem size. Thus AL may be overly pessimistic.

# Load Balancing

- It is important to make sure that all processes do useful work at any given time
- If a workload is distributed among processes, one needs to make the subtasks as equal as possible

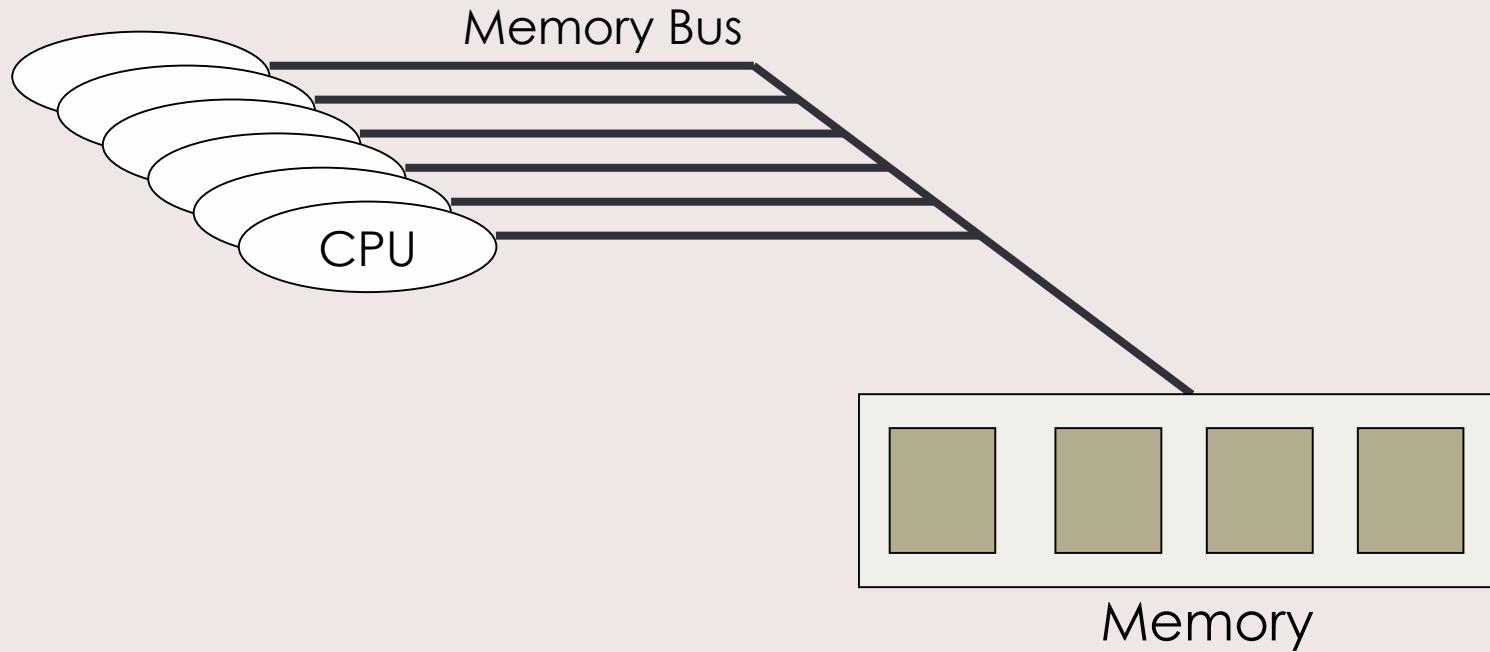


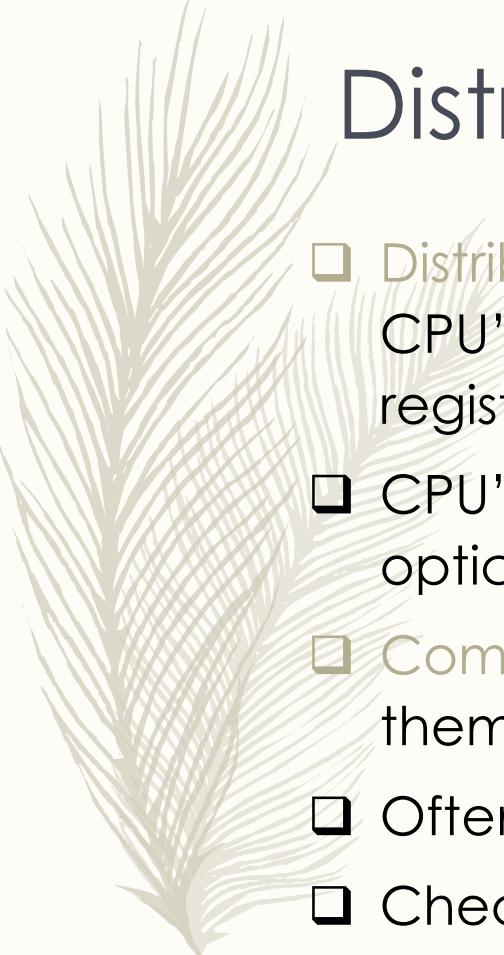


# Shared Memory

- ❑ Shared Memory:
  - All CPU's are connected via a memory bus to a **common memory** pool
  - ❑ Usually, each CPU has **its own** register and cache
  - ❑ Little **communication** between CPU's is needed as all work on the same memory space
  - ❑ Fast, **efficient**, and often easy to program but
  - ❑ Expensive and of limited expandability

# Shared Memory (cont)

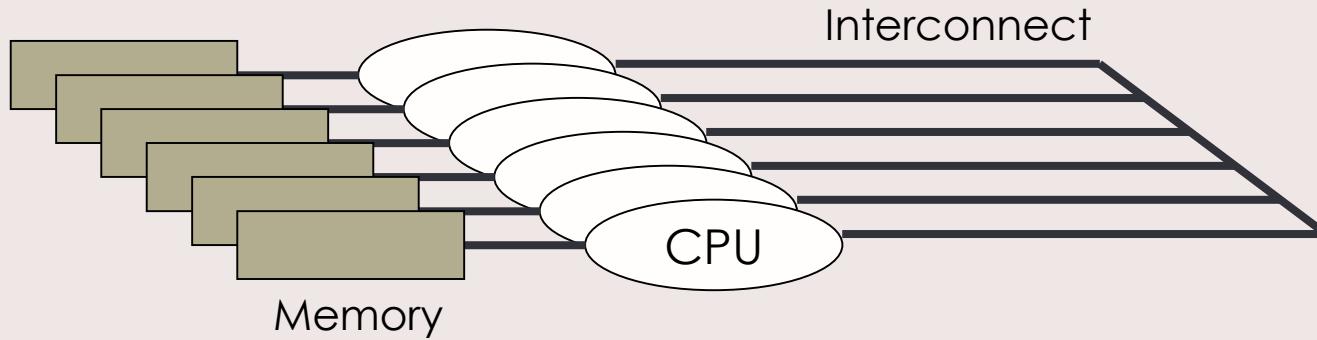




# Distributed Memory

- ❑ Distributed Memory:  
CPU's are independent and have their own memory, register and cache
- ❑ CPU's are interconnected via Ethernet, fast switches, optics etc.
- ❑ Communication between CPU's is necessary to make them work together: bottleneck
- ❑ Often hard to program for but
- ❑ Cheap and expandable

# Distributed Memory (cont)



# Shared vs Distributed Memory

## Shared Memory

## Distributed Memory

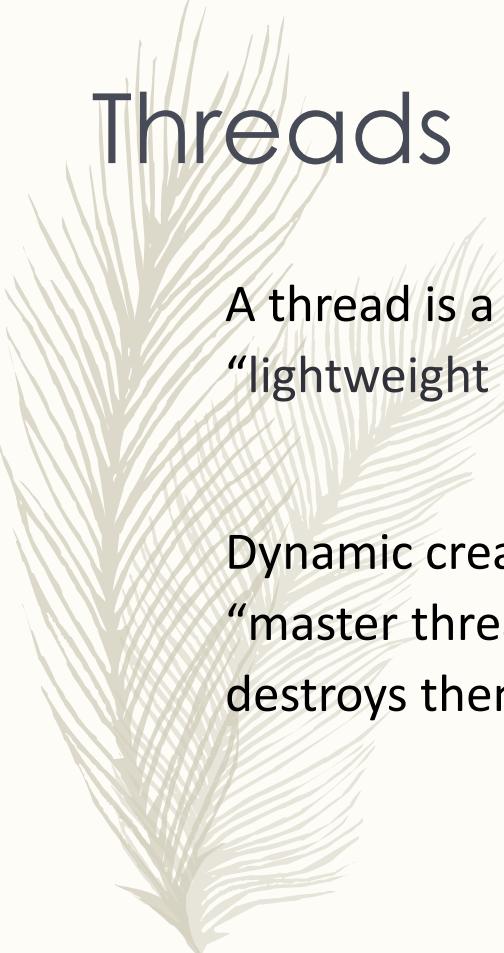
<b>Pro</b> <ul style="list-style-type: none"><li>✓ Easy to program and convert</li><li>✓ Auto-parallelization possible</li><li>✓ Little communication overhead</li><li>✓ Fast</li><li>✓ Works with DM programs</li></ul>	<b>Pro</b> <ul style="list-style-type: none"><li>✓ Cheap</li><li>✓ Easily extended</li><li>✓ Good scaling (&gt;1000 CPUs)</li><li>✓ Good control by user</li></ul>
<b>Con</b> <ul style="list-style-type: none"><li>✓ Expensive</li><li>✓ No expandability, fixed size</li><li>✓ Scaling limited if simple approach is taken</li><li>✓ Hidden complexities</li></ul>	<b>Con</b> <ul style="list-style-type: none"><li>✓ Communication, slow</li><li>✓ Often more difficult</li><li>✓ Conversion non-trivial</li><li>✓ Explicit parallelization</li><li>✓ SHM programs don't work</li></ul>



# Which is Better ?

Shared memory  
if (and that's a big if)  
You can afford it

# Threads



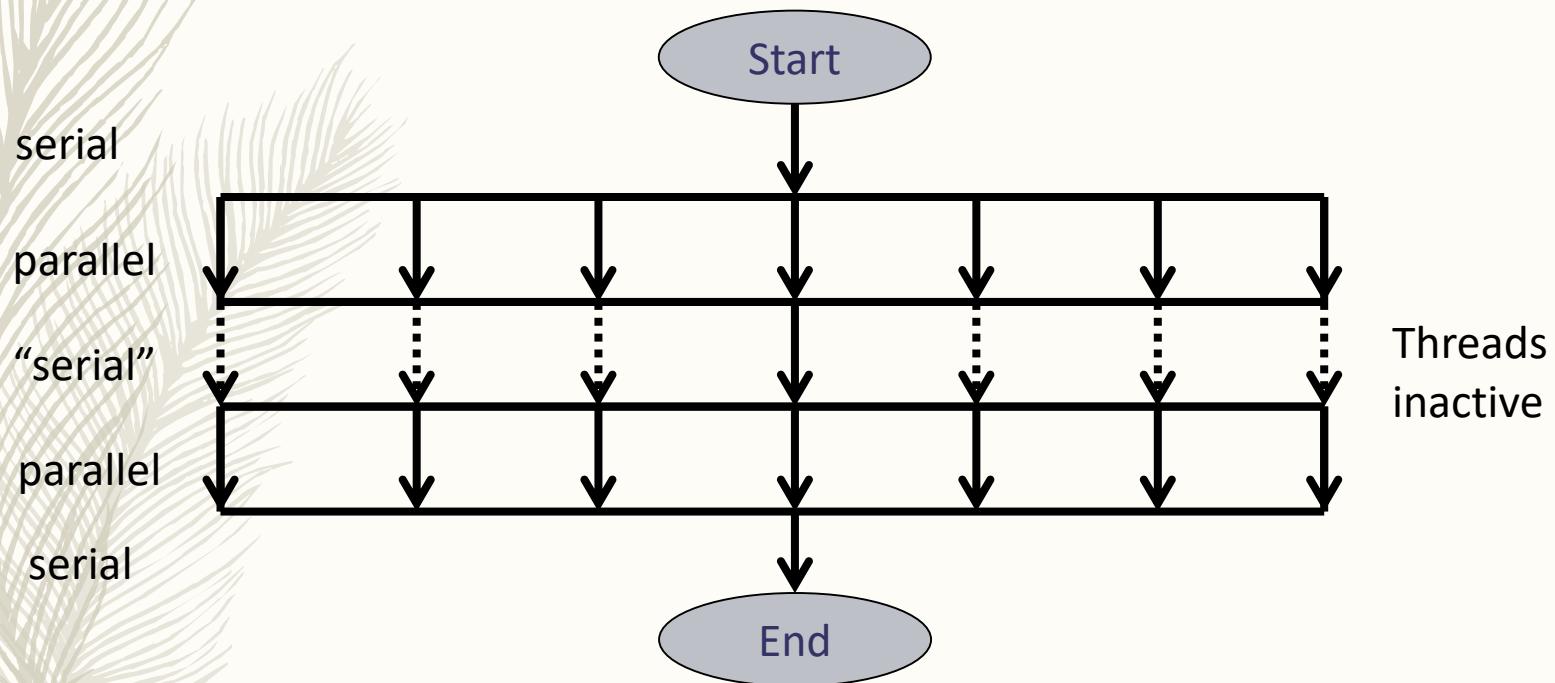
A thread is a **dynamically created process**, sometimes also called a “lightweight process”.

Dynamic creation means that the original process (often called “master thread”) spawns additional processes (threads) and destroys them when they are not needed anymore.

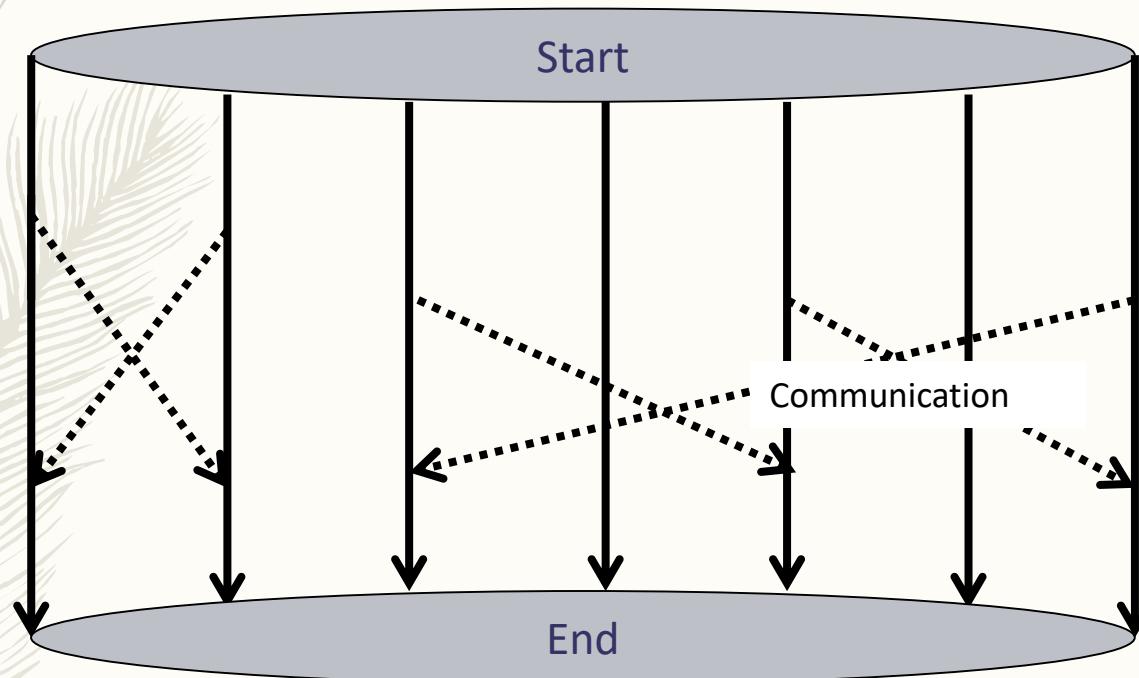
# Multithreading: Shared Memory

- Shared memory supports **multithreading** over multiple processors
- The program is started “in serial mode”
- Temporary “light-weight” processes = **threads** are created dynamically
- Can be done
  - explicitly (e.g. Posix threads)
  - automatically at compile time
  - via **directives** (e.g. **OpenMP**)
- This technique is often used to create a flexible program structure, even if only one CPU is available (serial, e.g. OS)

# Multithreading



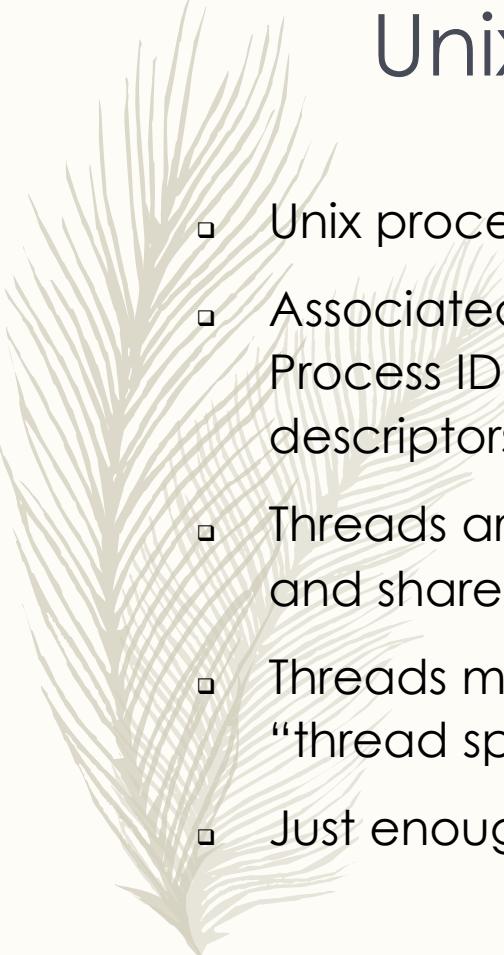
# Multiprocessing





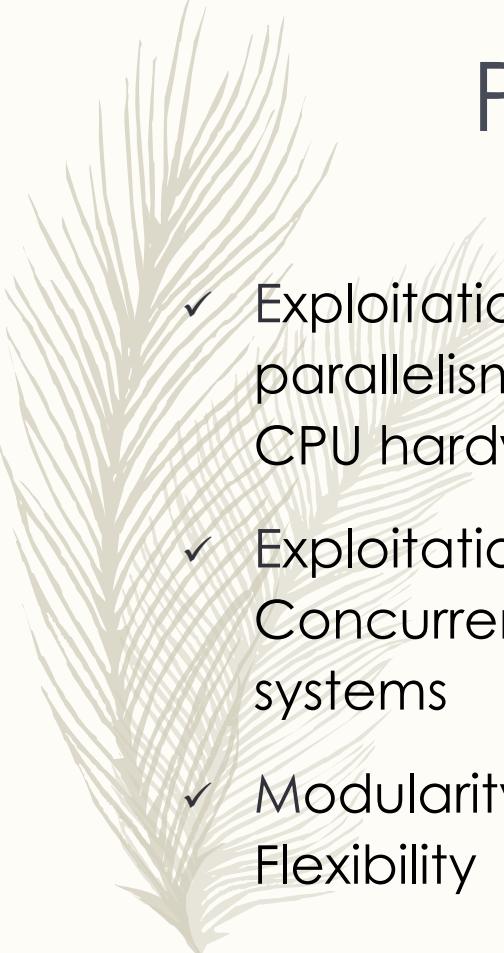
# Multithreading (cont'd)

- Often used for “task parallelism”
- If several independent task are to be performed in a loop they can be “distributed” among threads
- Also often: “loop parallelism”



# Unix Procs & Threads

- ❑ Unix processes are created by the OS
- ❑ Associated Information and overhead:  
Process ID, instructions, registers, stack with pointer, heap, file descriptors, signal, libraries ...
- ❑ Threads are created by a main process  
and share its resources, bringing down overhead and latency
- ❑ Threads maintain their own registers, stack, block signals, and  
“thread specific” data
- ❑ Just enough to run threads independently



# Pros & Cons of MT

- ✓ Exploitation of parallelism on multi-CPU hardware
  - ❖ Computing overhead, largely to synchronization
  - ❖ Increased complexity and programming discipline
  - ❖ Libraries may not be thread-safe
  - ❖ Harder to debug
- ✓ Exploitation of Concurrency on all systems
- ✓ Modularity and Flexibility



# Posix Threads

- Explicit creation and handling of threads
- Used from C-programs, using library `libpthread.so`
- Available for all Unix platforms (e.g. Solaris, Linux, etc)
- High degree of control, but difficult in practice

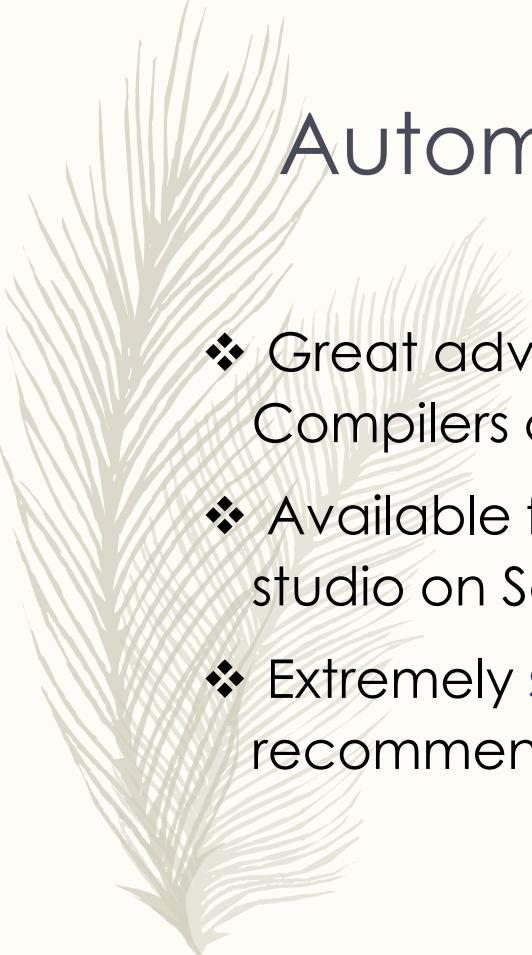
# A Posix Example

```
#include <stdio.h>
#include <stdlib.h>
#include "pthread.h"

void *output(void *arg);

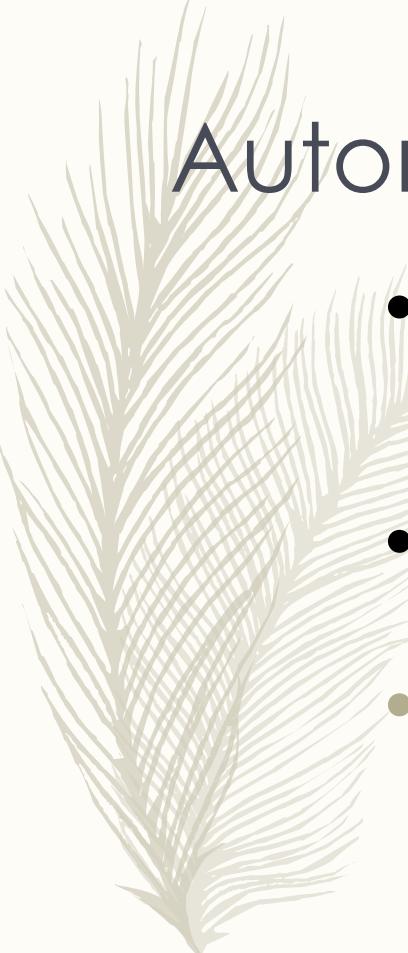
int main(int argc, char *argv[]){
    int id,rv,nt=atoi(argv[1]);
    pthread_t* thread=(pthread_t*)malloc(nt*sizeof(pthread_t));
    int* ids=(int*)malloc(nt*sizeof(int));
    for (id=0;id<nt;id++){ /** Create threads **/
        ids[id]=id;
        rv(pthread_create(&thread[id], NULL, output, (void*)&ids[id]));
    }
    for (id=0;id<nt;id++) rv(pthread_join(thread[id],NULL));
    return(0);
}

void output(void arg){ /** Hello world function ***/
    printf("Hello from thread Number %d\n", (int*)arg);
    return 0;
}
```



# Automatic Parallelization

- ❖ Great advantage of multithreading:  
Compilers can “auto-parallelize” serial code
- ❖ Available for some compilers, for instance  
studio on Solaris, intel on Linux
- ❖ Extremely simple to use, but caution is  
recommended, as compilers are “conservative”



# Automatic MT (cont)

- Only a `compiler option` is required:
  - `parallel` for intel/Linux,
  - `xautopar` for studio/Solaris
- May need `optimization` to work:
  - `xO3` for studio/Solaris
- Reduction operations involving all threads help:
  - `xreduction` for studio/Solaris

# Example: Automatic MT

```
subroutine test(a,b,c,n,sum)
    integer :: i,n
    real*8 :: a(n),b(n),c(n),sum
    a=b+c ! Line 4: Easily parallelized
    do i=1,n-1 ! Line 5: Loop dependence
        a(i+1)=a(i)+b(i)
    end do
    sum=0
    do i=1,n ! Line 9: Requires Reduction
        sum=sum+a(i)
    end do
end subroutine test
```

# Example (cont)

(on SUN system for demonstration, no reduction on Linux)

**Without reduction:**

```
bash 2.05$ f90 -c -xO3 -xautopar -xloopinfo autotest.f90
"autotest.f90", line 4: PARALLELIZED, and serial version generated
"autotest.f90", line 5: not parallelized, unsafe dependence (a)
"autotest.f90", line 9: not parallelized, unsafe dependence (sum)
```

**With reduction:**

```
bash 2.05$ f90 -c -xO3 -xautopar -xloopinfo -xreduction autotest.f90
"autotest.f90", line 4: PARALLELIZED, and serial version generated
"autotest.f90", line 5: not parallelized, unsafe dependence (a), distributed
"autotest.f90", line 9: PARALLELIZED, reduction, and serial version generated
```

# Multithreading in OpenMP

- ▶ In OpenMP, the **parallel region** is a block of code which is executed simultaneously by a **Master Thread** (with an **ID=0**) and **Worker Threads** (**ID>0**)
- ▶ Work sharing is either done by special constructs (“**parallel do**”), or **explicitly** (“**parallel**”)

# OpenMP Compiler Directives

- To help the compiler parallelizing loops, we use compiler directives. These are like “local compiler flags” and are written into the source code.
- They are **not** function calls or other executable code lines.
- A common **standard** for these is **OpenMP**
- OpenMP Compiler directives are only interpreted if the **`-fopenmp`** compiler flag is issued.

# Example: Compiler Directives

```
subroutine test(a,b,c,n,sum)
    integer :: i,n
    real*8 :: a(n),b(n),c(n),sum,sumup
!$omp parallel do ! Compiler Directive: forces parallelization
    do i=1,n
        a(i)=sumup(b(i),c(i))          ! Possible dependency: no auto
    end do
end subroutine test
real*8 function sumup(x,y)           ! Sum hidden in a function
    real*8 :: x,y
    sumup=x+y
end function sumup
```

# Example (cont)

(on SUN system for demonstration, does not react the same way on Linux)

**Without compiler directives:**

```
bash-2.05$ f90 -c -xO3 -xautopar -xloopinfo testomp.f90  
"testomp.f90", line 5: not parallelized, call may be unsafe
```

**With compiler directives:**

```
bash-2.05$ f90 -c -xO3 -xautopar -xloopinfo -xopenmp  
testomp.f90  
"testomp.f90", line 5: PARALLELIZED, user pragma used
```



# Issues With Shared Memory Programming

Shared-Memory Programming is usually simpler than Distributed-Memory Programming.

However, there are some pitfalls:

- ▶ Data Dependencies
- ▶ Race Conditions
- ▶ False Sharing

# Data Dependency

```
fact(1)=1
do i=2,n
    fact(i)=fact(i-1)*i
end do
```

Loop cannot be parallelized by distributing iterations among threads, because each iteration depends on the previous one.

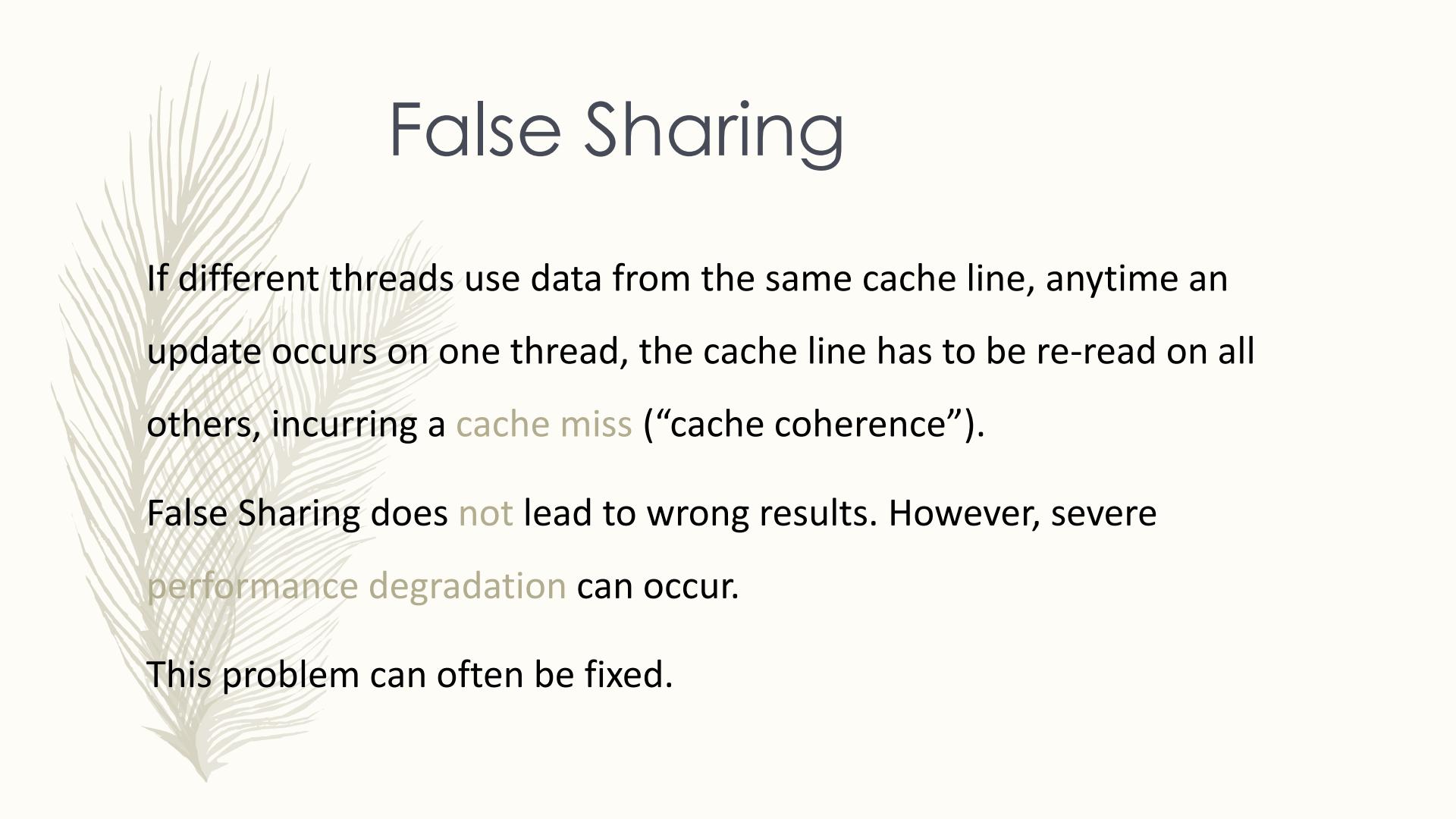
If the compiler refuses to auto-par code because of dependences, it is necessary to investigate if there actually are any. Only if you are sure there are not, proceed.

# Race Conditions

```
...  
do i=1,100  
    total = total + b(i)*c(i)  
end do  
...
```

In this loop, there may be a problem, because multiple threads may be updating total at the same time. The **result depends on “who comes last”**. Thus, **“race condition”**. The compiler will assume the worst and refuse to parallelize this.

Race conditions can be very **hard to detect**, and their result may be subtle. If you receive **“inaccurate”** results depending on the number of processors, and seemingly the weather, you might have a race condition. These can be resolved by the use of “critical regions” or locks.



# False Sharing

If different threads use data from the same cache line, anytime an update occurs on one thread, the cache line has to be re-read on all others, incurring a **cache miss** (“cache coherence”).

False Sharing does **not** lead to wrong results. However, severe **performance degradation** can occur.

This problem can often be fixed.

# Practical Stuff

Most modern compilers are OpenMP enabled  
OpenMP works with Fortran, C, and C++

Basic compiler option on Linux (intel):

**-qopenmp** [enables OpenMP]

This compiler option may imply a minimum optimization level  
that is automatically enforced even if not specified

# Most Important Environment Variable

Commonly used to set conditions for program execution

**OMP\_NUM\_THREADS=n**

number of threads,

default sometimes 1, sometimes number of cores.

# OpenMP

- A set of compiler directives for declaration of parallelism in source code
- Also includes a supporting library of functions/routines
- Works with Fortran, C and C++
- Requires enabled compilers which are available for most platforms that support shared-memory parallelism
- Information on website

<http://www.openmp.org>



# OpenMP: Some History

- 1980's: SHM compiler directives proprietary & platform specific
- Early efforts at standardization (CMFortran, C\*, HPF) failed
- 1996: OpenMP Architecture Review Board, industry standard
- Original members: ASCI, DEC, HP, IBM, Intel, KAI, SGI
- Later joined by SUN and Compaq
- 1997 Fortran v1.0, 1998 C/C++ v1.0
- Presently non-profit, ongoing development
- More recently (May 2008): [OpenMP 3.0](#)
- July 2013: [OpenMP 4.0](#)
- In preparation (2018): OpenMP 5.0



# Why OpenMP ?

- Most platforms support it, industry standard
- Small and simple
- Good for latest multicore architectures
- Parallelization can be done incrementally
- Newer software/libraries use it
- Standardization makes code largely platform independent

# OpenMP Directives: Basic Format

**! \$omp ...**

Fortran free format

**! \$omp ... &**

Requires continuation line

**#pragma omp ...** C and C++

The first symbol is either interpreted as a comment symbol (!,Fortran) or indicates a preprocessor construct (#,C/C++) if no OpenMP compiler flag is issued. If OpenMP is enabled, it will be interpreted as OMP directive.



# OpenMP Routines

- OpenMP also supplies supporting routines
- If compilation/linking is done with OpenMP enabled, the proper libraries are linked in automatically
- For Fortran
  - `use omp_lib`
- For C and C++, use
  - `#include <omp.h>`
- The routines are functions or subroutines with names that start with `omp_`

# Conditional Compilation (Fortran)

A code line that starts with:

**! \$ ...** (free format, two spaces)

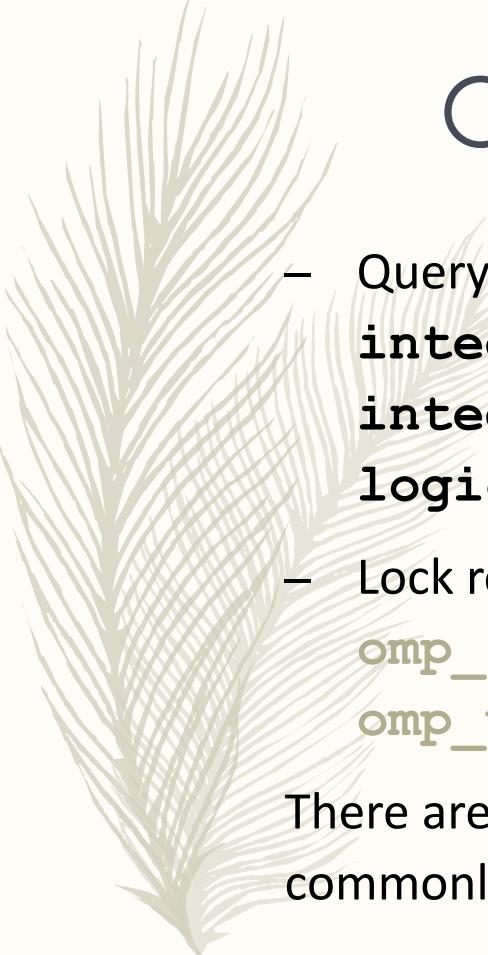
is only recognized as a line of Fortran  
code if OpenMP is enabled, otherwise it is interpreted as a  
comment.

# Conditional Compilation (C/C++):

For C and C++, Pre-processor constructs are used:

```
#ifdef _OPENMP  
...  
#endif
```

encloses code lines that are only retained if OpenMP is enabled,  
otherwise they are skipped. Do not define this keyword explicitly.



# OpenMP Routines

- Query routines, e.g.

```
integer omp_get_num_threads()
```

```
integer omp_get_thread_num()
```

```
logical omp_in_parallel()
```

- Lock routines, e.g.

```
omp_set_lock(addr)
```

```
omp_unset_lock(addr)
```

There are many others, but these are the most commonly used.

# OpenMP: Example (FORTRAN)

OpenMP directives mark and enclose  
a parallel region

```
program helloworld
  !$ use omp_lib
  write (*,*)' Here is the main thread (serial) ...'
  !$omp parallel
    !$ write (*,*)' ... and here is thread number '&
    !$ ,omp_get_thread_num(),' (parallel) ...'
  !$omp end parallel
  write (*,*)' ... and now it is serial again.'
end program helloworld
```

This line  
calls an OpenMP function and is only compiled  
conditionally

# OpenMP: Example (C)

OpenMP directive and {} mark and enclose a parallel region

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("Here is the main thread (serial) ... \n");
#ifndef _OPENMP
#pragma omp parallel
    → {printf(" ... and here is thread number %d %s \n",
              omp_get_thread_num(), "(parallel) ..."); }
#endif
    printf(" ... and now it is serial again.\n");
    return 0;
}
```

This line  
calls an OpenMP function and is only compiled  
conditionally

# Example: Serial

Setting # of processors

Compiling (no OpenMP)

```
$ ifort -o hello_s.exe -O5 hello.f90  
OMP_NUM_THREADS=4 ./hello_s.exe  
Here is the main thread (serial) ...  
... and now it is serial again.
```

Execution proceeds in **serial**, although number of procs was set

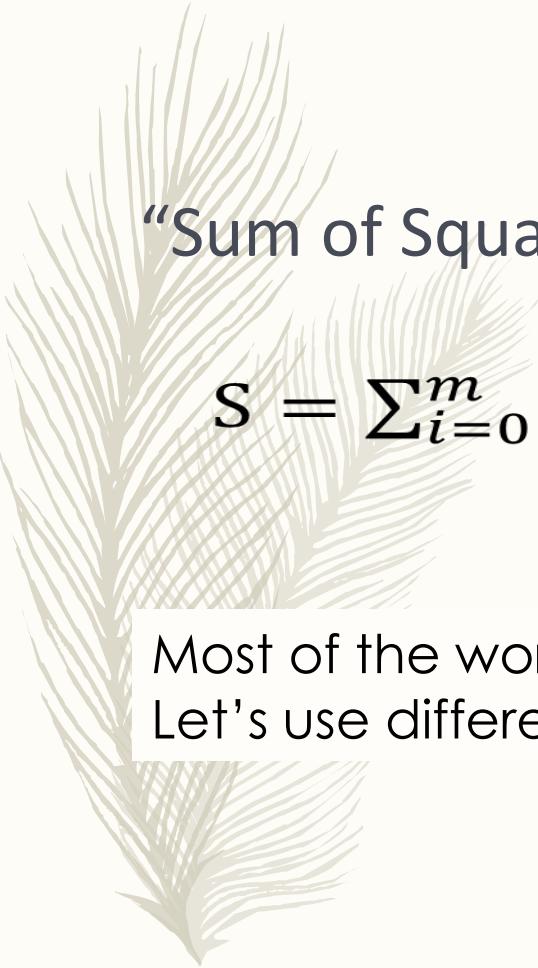
# Example: Parallel

# of threads

Compiling (OpenMP)

```
$ ifort -o hello_p.exe -O5 -openmp -openmp-report=2 hello.f90
hello.f90(4): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
$ OMP_NUM_THREADS=4 ./hello_p.exe
Here is the main thread (serial) ...
... and here is thread number          0 (parallel) ...
... and here is thread number          2 (parallel) ...
... and here is thread number          1 (parallel) ...
... and here is thread number          3 (parallel) ...
... and now it is serial again.
```

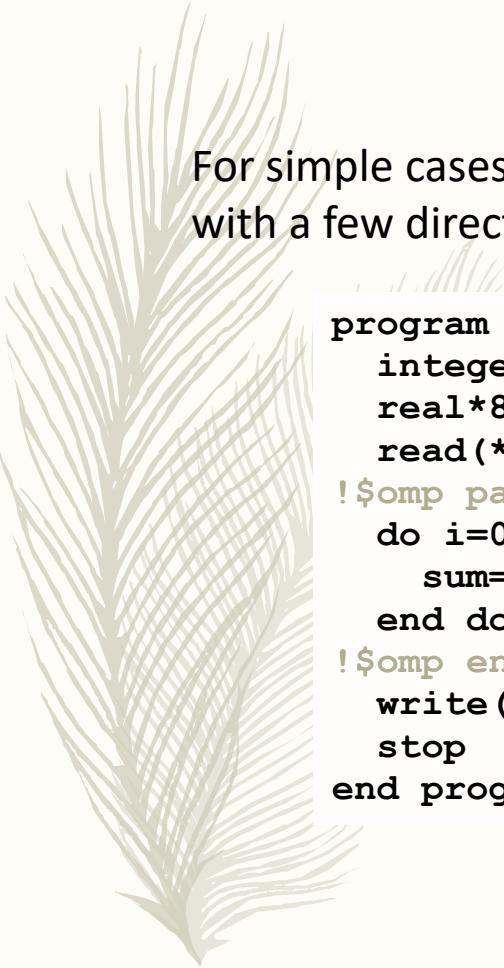
Execution proceeds in parallel



## “Sum of Square Roots” Example

$$S = \sum_{i=0}^m \sqrt{i} = \sqrt{0} + \sqrt{1} + \sqrt{2} + \cdots + \sqrt{m}$$

Most of the work is in the evaluation of the square roots.  
Let's use different thread for different square roots.



For simple cases, an OpenMP program is just the serial code with a few directives “thrown in”:

```
program rootsum ! Sum of squareroots of integers
    integer :: i,m
    real*8 :: sum=0.d0
    read(*,*)m
!$omp parallel do reduction(+:sum)
    do i=0,m
        sum=sum+sqrt(dfloat(i))
    end do
!$omp end parallel do
    write(*,*)" Result =",sum
    stop
end program rootsum
```

# Example (cont'd)

```
$ cat rootsum.in  
1234567890
```

```
$ ifort -O3 rootsum.f90  
$ time -p ./a.out < rootsum.in  
  Result = 28918862541603.5  
real 8.77  
user 8.76  
sys 0.00
```

```
$ ifort -O3 -openmp -openmp-report=2 rootsum.f90  
rootsum.f90(5): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.  
$ OMP_NUM_THREADS=8 time -p ./a.out < rootsum.in  
  Result = 28918862541603.0  
real 1.23  
user 9.17  
sys 0.02
```

$$\text{Speedup} = \frac{T_s}{T_p}$$
$$= \frac{8.77}{1.23} = 7.1$$

**BINGO**

# Loop Parallelism: **PARALLEL DO**



The simplest OpenMP directives are concerned with loop parallelism. In many cases these are sufficient for some parallel performance

The simplest form is

**PARALLEL DO** (Fortran)

**parallel for** (C/C++)

This **must** be followed by a do (for) **loop** and extends to the end of that loop. It distributes the iterations across threads to achieve parallelism.

# PARALLEL DO (for)

The parallel do (for) directive causes a do (for) loop that follows it to be executed **in parallel**, even if the loop has data dependencies. Afterwards the threads are “destroyed”.

```
!$omp parallel do  
do i=2,n-1  
    a(i)=b(i+1)+c(i-1)  
end do
```

```
#pragma omp parallel for  
for (i=2;i<n;i++){  
    a[i]=b[i+1]+c[i-1];  
}
```

By default, variables are shared, with the exception of **loop indices** which are **private**, i.e. each thread has its own value.



# Data Scopes

We need to declare if the variables inside a parallel region (loop) are

**shared** (list...)

i.e. all threads see the same value and the variable is accessible to all threads, or

**private** (list...)

i.e. each thread has its own copy of the variable and they can not access each others values.

This is done through a “clause” that follows the `omp` directive.



# Default Scoping Rules

- By default, all variables are shared
- Exceptions:
  - ❖ All loop indices must be private by default, i.e. both for parallel loops, and loops inside (Caution: not in C, only parallel loops)
  - ❖ Local variables inside functions that are called in parallel loops are private



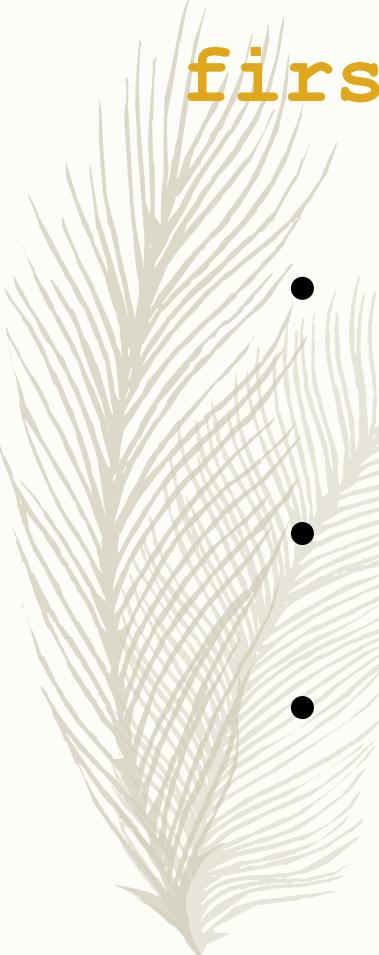
# Private Variables

- ❑ Private variables are allocated in new and separate memory locations
- ❑ Each thread has its own copy in memory, different from the others, and from the “serial” variable
- ❑ private variables are not initialized on loop entry
- ❑ The serial variable is effectively invisible in a parallel loop
- ❑ The serial variable does not have a specific value on loop exit, i.e. do not rely on consistency of private variables after the loop.

# Private Variables

```
x=12  
...  
!omp parallel do private(x)  
do i=1,100  
...  
    x=i*120+1  
...  
end do  
...
```

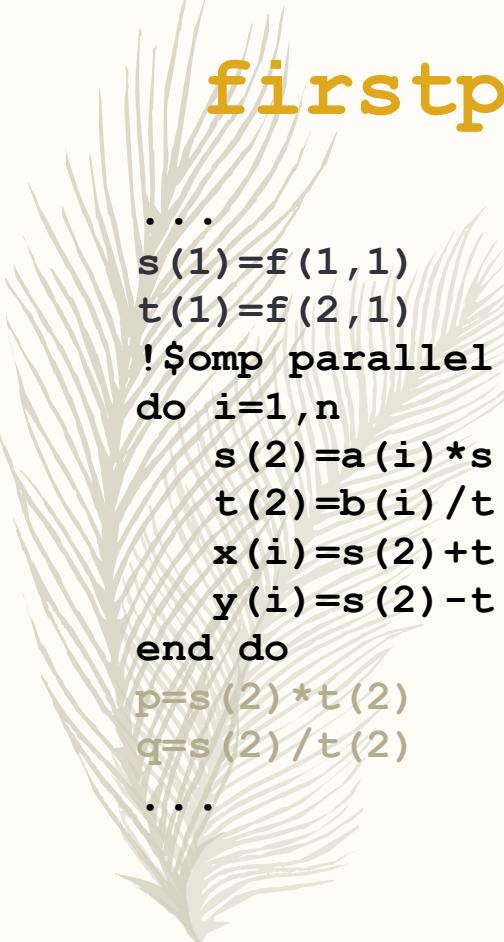
What is x here ?  
Certainly **not** 12001.  
Possibly still 12, but don't rely on it.  
Better to re-initialize.



# **firstprivate** and **lastprivate**

- If you need to initialize a private variable with the “sequential value”, use the **firstprivate** declaration
- If you want to re-initialize a sequential value with a private variable, use the **lastprivate** declaration
- These are not necessary if the private variable is temporary, but can be useful in other cases

# firstprivate and lastprivate

```
...  
s(1)=f(1,1)  
t(1)=f(2,1)  
!$omp parallel do firstprivate(s,t) lastprivate(s,t)  
do i=1,n  
    s(2)=a(i)*s(1)  
    t(2)=b(i)/t(1)  
    x(i)=s(2)+t(2)  
    y(i)=s(2)-t(2)  
end do  
p=s(2)*t(2)  
q=s(2)/t(2)  
...  

```

Initialized with serial value because of **firstprivate**

Retains value for i=n because of **lastprivate**

# Default Scoping Rules

By default, all variables are shared

Exceptions:

- All loop indices (even contained ones) must be private by default
- Caution:** not in C, only parallel loops
- Local variables inside functions that are called in parallel loops are private

# Example:

```
program default
    integer :: i,j,k=10
    real*8 :: x(10)
    do i=1,10
        x(i)=sqrt(dfLOAT(i))
    end do
!$omp parallel do
    do i=1,10
        do j=1,10
            call sub(x(i),k,j)
        end do
        write (*,*) i,x(i)
    end do
end program default
```

Private: i, j, index, ic  
Shared: x, a, k, in

```
subroutine sub(a,in,index)
    real*8 :: a
    integer :: in,index, isub,ic=5
    do isub=1, index
        a=a+dfloat(in+ic)
    end do
    return
end subroutine sub
```

# In Practice: Large Loops

Sometimes it is best to put the loop content into a routine to keep most local variables private and to minimize chances for errors and memory conflicts

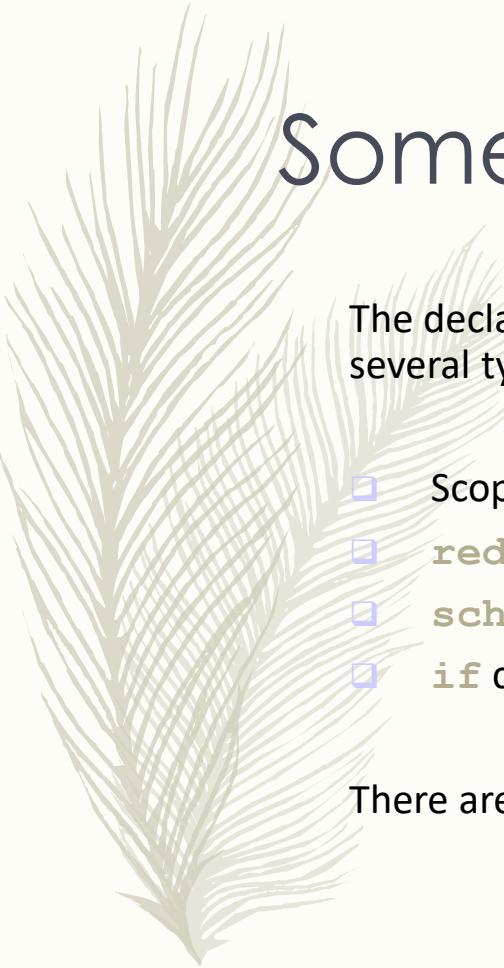
```
do i=1,n  
    x=...  
    a= ... x ...  
    b(i) = ...a....  
    do j=1,m  
        ...  
    end do  
    ...  
end do
```

```
do i=1,n  
    call sub(i,b,m,...)  
end do  
  
subroutine sub(i,b,m,...)  
    x=...  
    a= ... x ...  
    b(i) = ...a....  
    do j=1,m  
        ...  
    end do  
    ...  
    return  
end subroutine
```

```
!$omp parallel do shared(b,m,...)  
do i=1,n  
    call sub(i,b,m,...)  
end do  
  
subroutine sub(i,b,m,...)  
    x=...  
    a= ... x ...  
    b(i) = ...a....  
    do j=1,m  
        ...  
    end do  
    ...  
    return  
end subroutine
```

# Changing the Default

- ❑ With a `default()` clause, you can change the default setting for variable
- ❑ Takes one of `shared`, `private`, and `none` as argument (no `private` in C and C++).
- ❑ Used when most of the variables need to be private (in Fortran)
- ❑ `default(None)` might be a good idea as it forces declaration of all variables

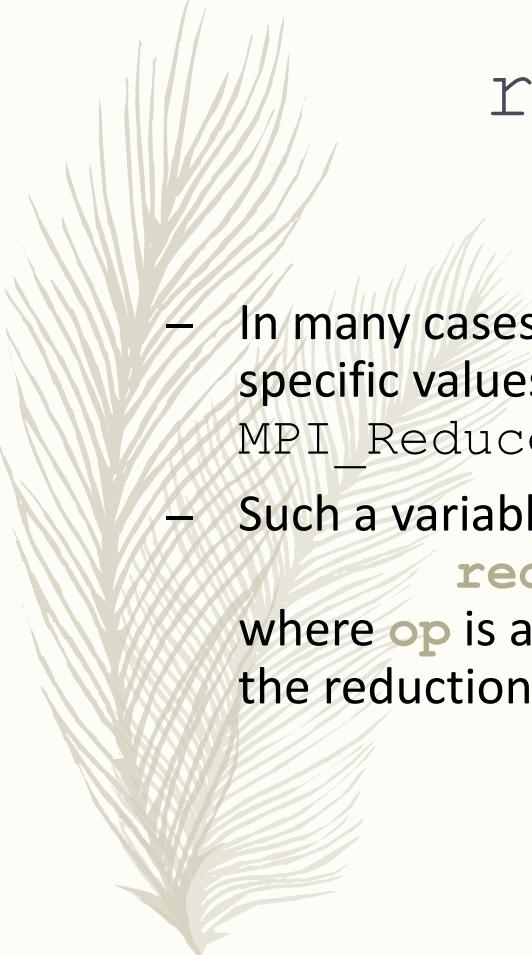


# Some Other Clauses

The declaration of private or shared variables is an example for clauses. There are several types:

- ❑ Scoping clauses (`private`, `shared`, `default`, etc)
- ❑ `reduction` clause (actually, also a scoping clause)
- ❑ `schedule` clauses, assigning iterations to threads
- ❑ `if` clause for conditional parallelism

There are many others



# reduction

- In many cases, loops involve operations, where iteration specific values are “reduced” to a single variable. (see MPI\_Reduce).
- Such a variable should be declared with a reduction clause:  
**reduction (op : var)**  
where **op** is an operation (+, \*, max, min,...) and **var** is the reduction variable

# reduction (cont)

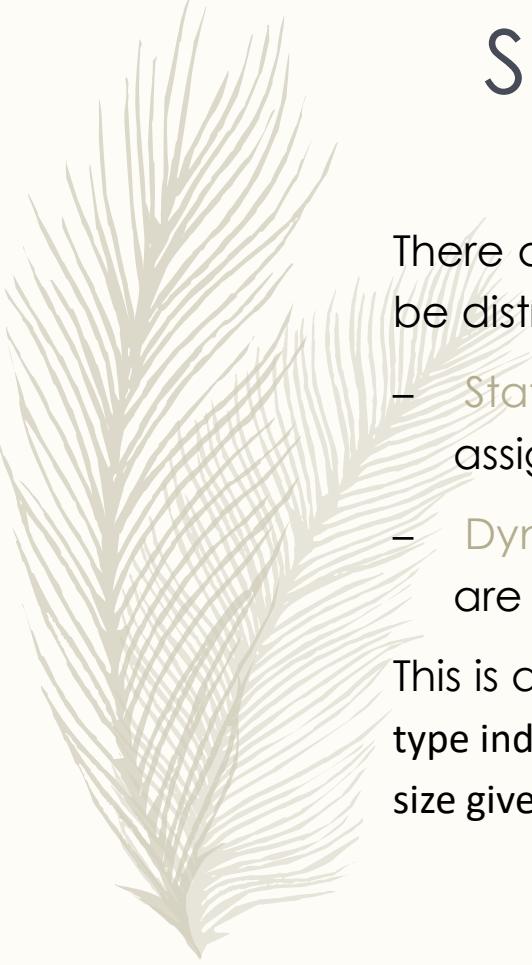
```
do i=0,m  
sum=sum+sqrt(dfloat(i))  
end do
```

```
!$omp parallel do reduction(+:sum)  
do i=0,m  
    sum=sum+sqrt(dfloat(i))  
end do  
 !$omp end parallel do
```

No problem, since the order  
of summation does not matter

```
do i=0,m  
sum=sqrt(sum+dfloat(i))  
end do
```

Problem, since the order  
of this operation matters

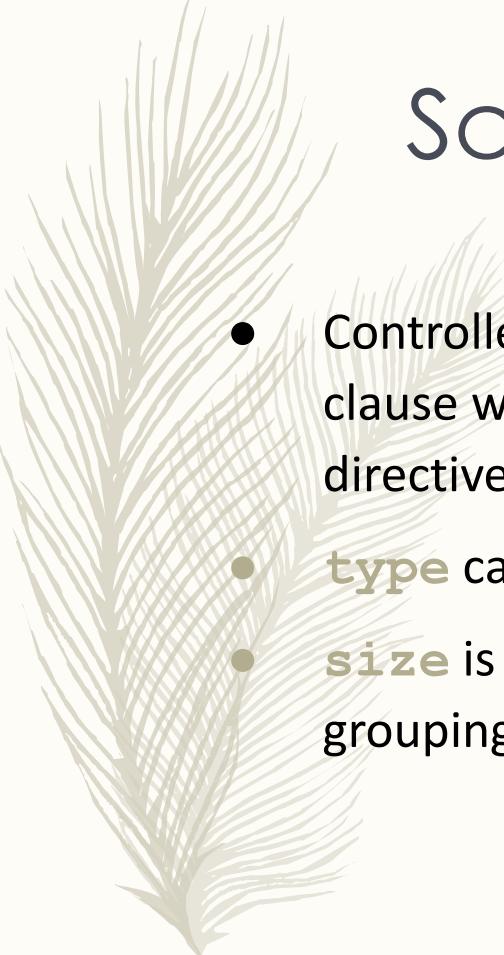


# Scheduling

There are two ways in which iterations in a parallel loop may be distributed among threads:

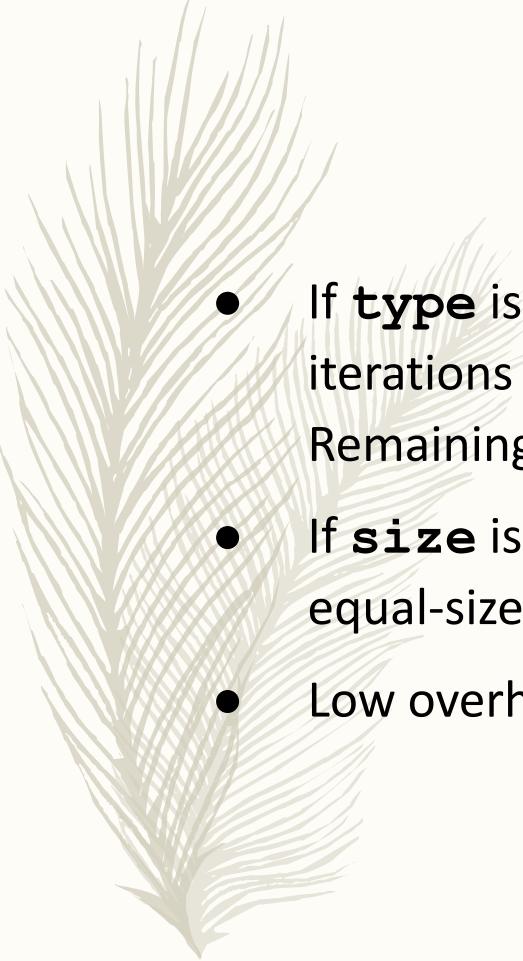
- **Static** schedules: determined beforehand, iterations are assigned according to fixed schedule. Fast but inflexible.
- **Dynamic** schedules: determined at runtime, iterations are assigned to idle threads. Flexible but overhead.

This is done using the **schedule(type, size)** clause, where type indicates the schedule type (static, dynamic, guided, runtime) and size gives the size of the iteration “chunks” involved.



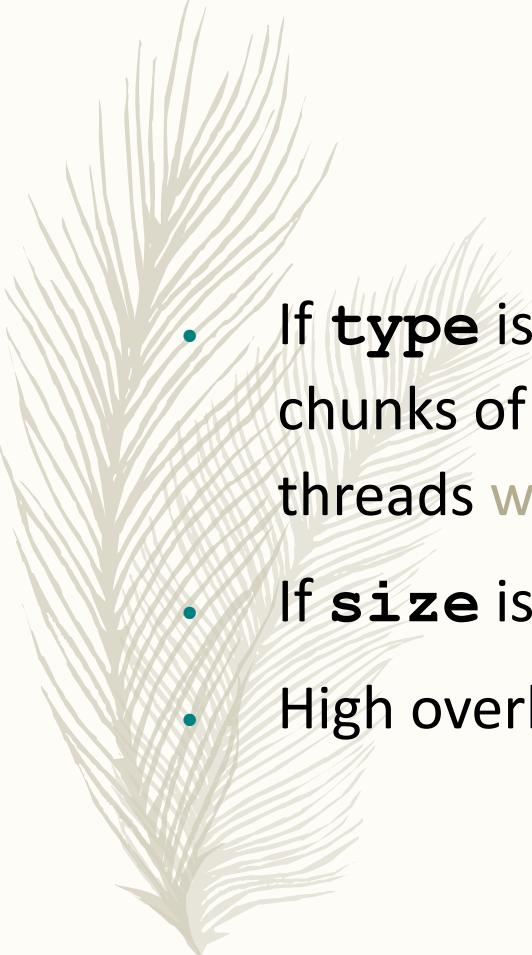
# Scheduling (cont'd)

- Controlled by the `schedule(type, size)` clause which is issued after the `parallel do` directive.
- `type` can be `static`, `dynamic`, `guided`, or `runtime`
- `size` is a chunk size that is used to create work loads by grouping iterations



# static

- If **type** is **static**, each thread gets assigned chunks of iterations of **fixed size** **size** in a **round-robin** fashion.  
Remaining iterations are distributed by the system.
- If **size** is omitted, it is chosen such that all chunks are equal-sized, and there is one per thread.
- Low overhead



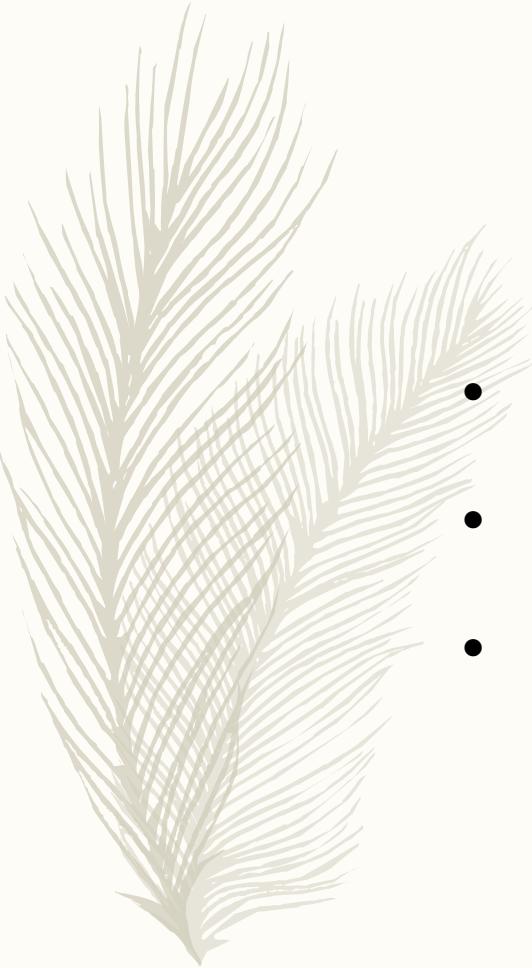
# dynamic

- If **type** is **dynamic** the iterations are divided into chunks of fixed size **size** and then assigned to threads whenever a thread is **idle**.
- If **size** is omitted it is set to 1
- High overhead



# guided

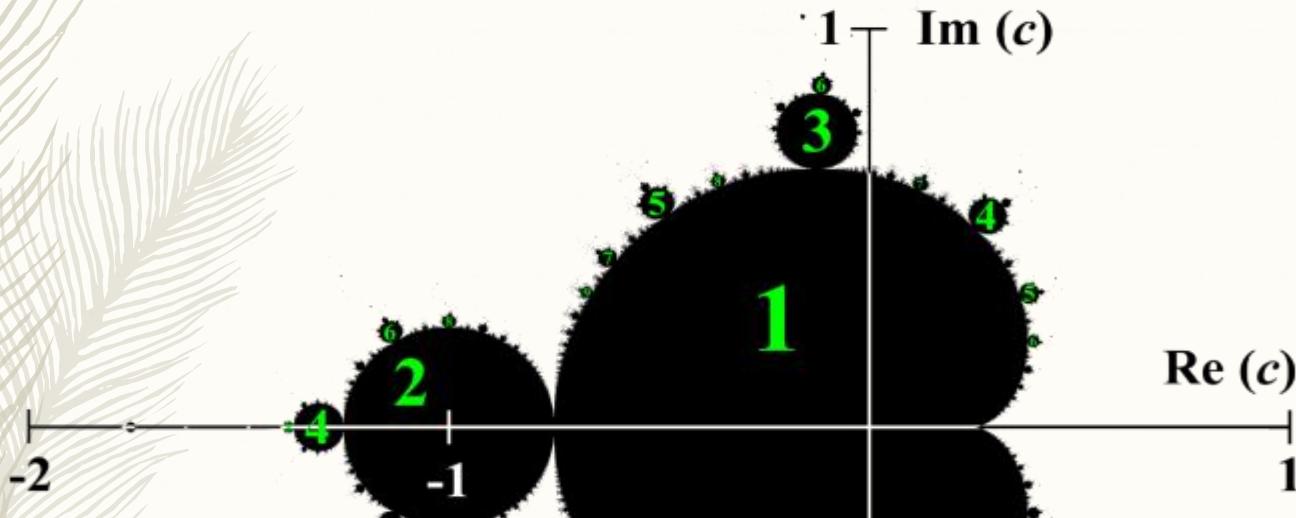
- If **type** is **guided**, iterations are divided into chunks of exponentially decreasing **size**. The smallest chunk size is **size**.
- Details are implementation specific.
- If **size** is omitted it is set to **1**.
- The chunks are assigned dynamically, i.e. a thread gets one when it's idle.
- Very high overhead



# runtime

- If **type** is **runtime**, the schedule is determined by the environment variable **OMP\_SCHEDULE**
- **OMP\_SCHEDULE** is of the same format as the arguments of **schedule**.
- If **OMP\_SCHEDULE** is not set, the choice of schedule is implementation dependent

# Example: Mandelbrot Set

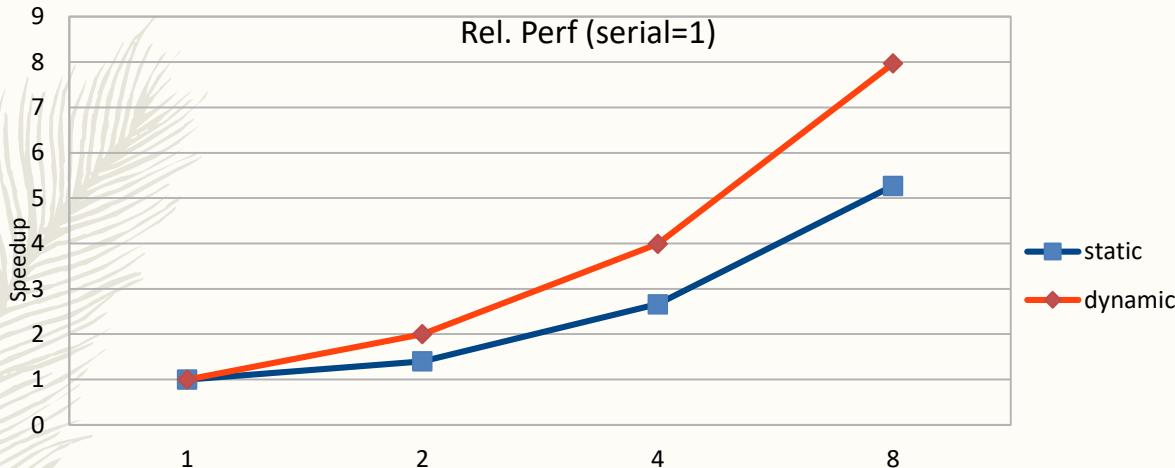


If  $z \rightarrow z^2 + c$  stays below  $|z| < 2$  after n iterations: black

Do this for  $-1.5 < \text{Re}(c) < 0.5$ ;  $0 < \text{Im}(c) < 1$

Problem: lower half much blacker than upper half

# Mandelbrot Set (cont'd)

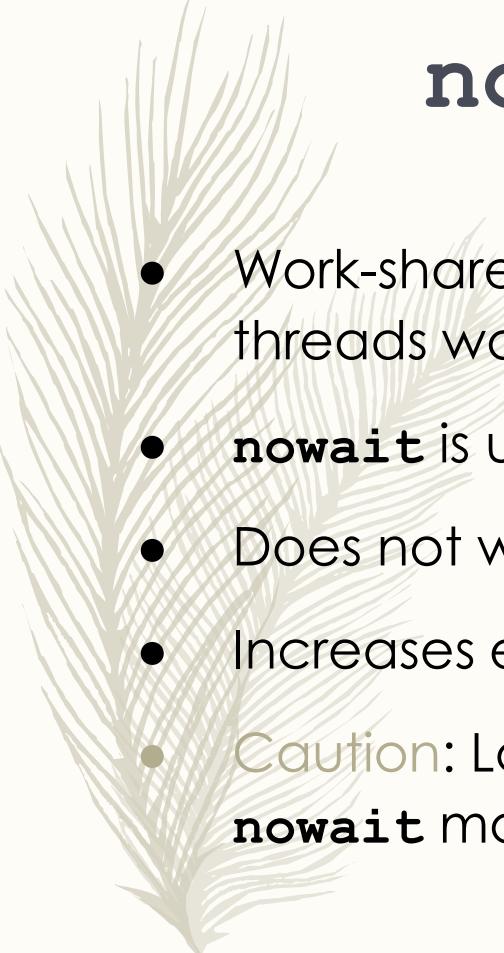


Sometimes the type of scheduling makes a big difference. Here a loop iteration corresponds to a line with constant imaginary part. The dynamic scheduling scales but the static one doesn't. This is similar to "Master-Slave" model in MPI. We will encounter another version of it again.

# if ()

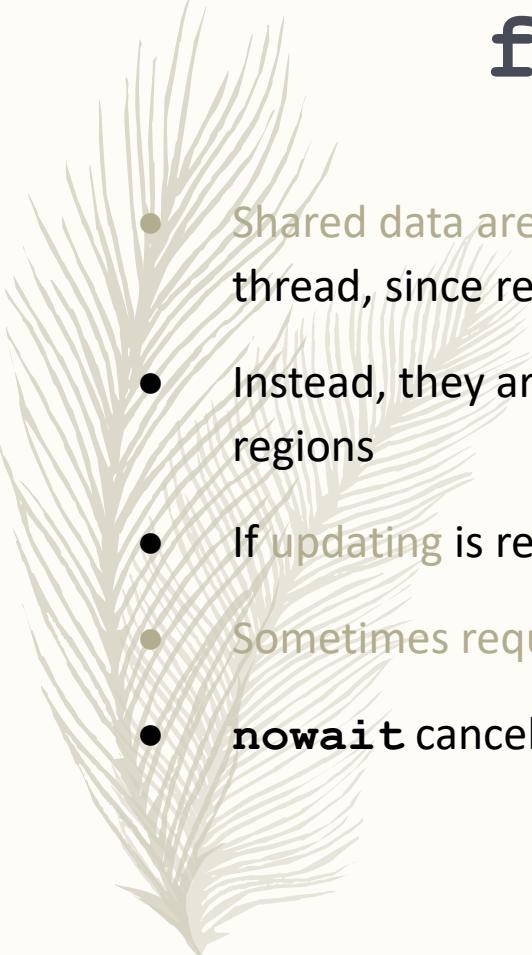
- Sometimes necessary to make use of directive dependent on runtime situation
- Argument: logical expression
- OMP directive only used if argument is TRUE  
(conditional parallelization)
- For instance, loop only parallel if minimum number of iterations:

```
!$omp parallel do if (n.gt.minn)
```



# **nowait**

- Work-share directives usually imply a **barrier**, i.e. threads wait until all threads finished
- **nowait** is used to **override** that barrier
- Does not work with end of parallel region
- Increases efficiency and load balance
- **Caution:** Later code may depend on results, **nowait** may improve speed, but also break code



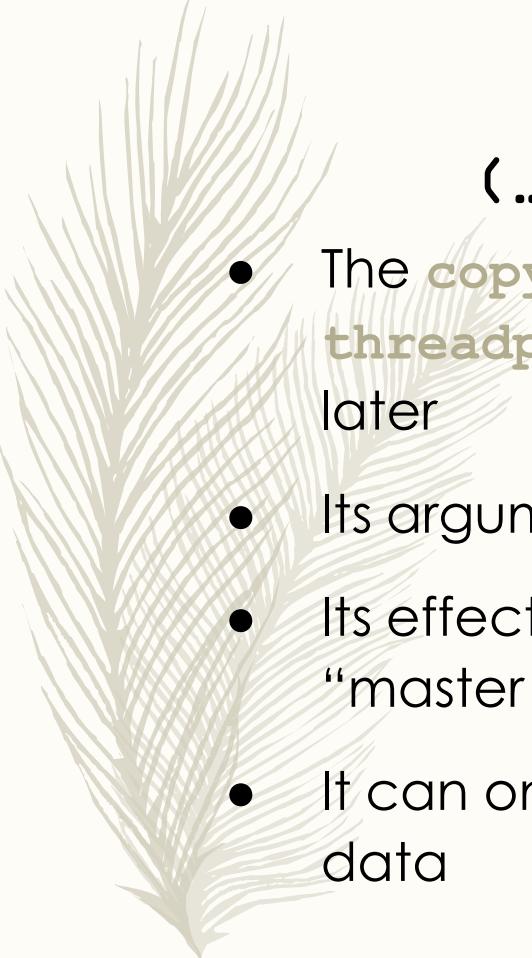
# **flush**

- Shared data are not immediately updated in memory when written by a thread, since registers, caches etc. serve as buffers
- Instead, they are updated at barriers, e.g. at exit from parallel or critical regions
- If updating is required **in between**, use **flush**
- Sometimes required with locks
- **nowait** cancels barriers and therefore implicit updating



# ordered (...later)

- Used to mark loops that may contain **ordered** sections
- **ordered** sections are discussed later; they are declared by an **ordered** directive
- In those sections, things are done in serial order, i.e. they limit parallel execution.



# copyin

(...later)

- The **copyin** clause is used **together with the `threadprivate`** declaration, and will be discussed later
- Its argument is a list of variables
- Its effect is to copy the value of a variable of the “master thread” onto a “slave thread”
- It can only be used with special **threadprivate** data



# More About Parallel Loops

- Must be **static**, i.e. the number of iterations is fixed (**do** / **for** loops). Dynamic loops such as “**while** loops” are not allowed (see OMP3 for an exception).
- **Dynamic loops** (e.g. **while** loops) are intrinsically **dependent**, as it depends on the data if an iteration is executed.
- **Nested loops**: only one may be parallel, the others (inside or outside) are performed sequentially, even within a thread
- Newer implementations allow nested-loop parallelism.

**Caution!**

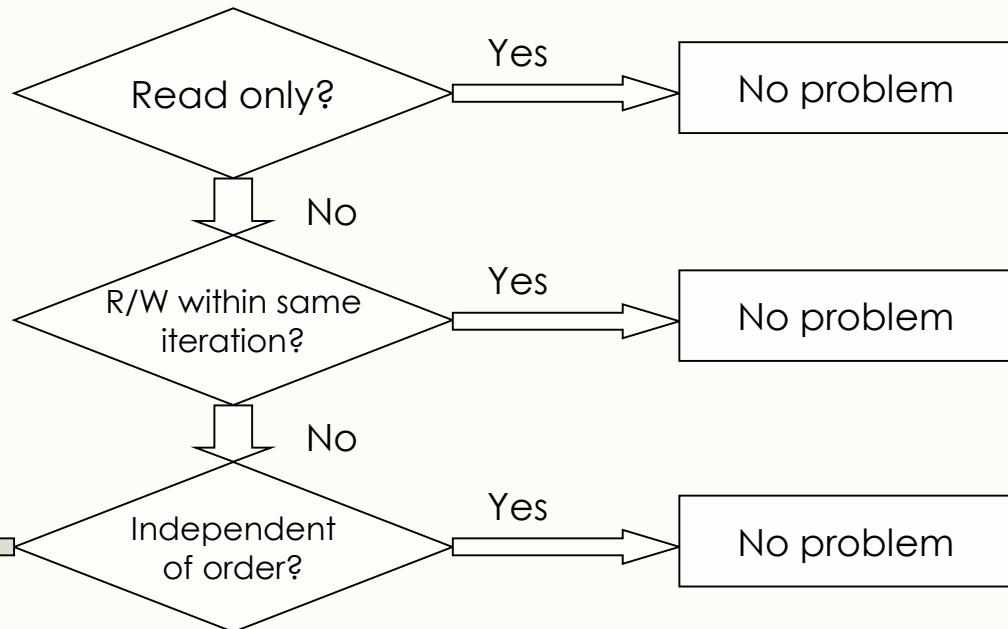


# Dependencies:

- Find them
- Identify them
- Resolve them if possible

# Finding Dependencies

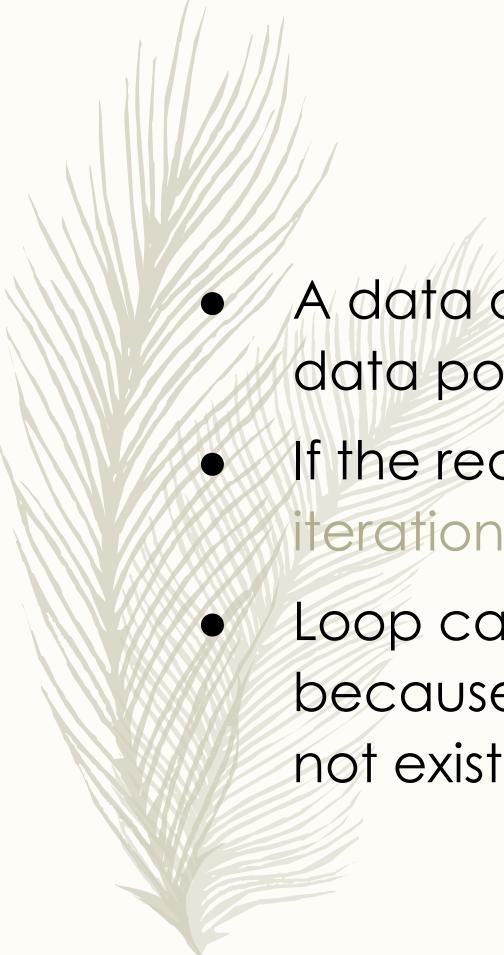
Is a variable in a loop ...



Problem

# Types of Dependencies

Type	Serious?
Loop Carried Flow Dependence	Yes, often prevents parallelization
Loop Carried Anti Dependence	Can be handled
Loop Carried Output Dependence	Can be handled
Non Loop Carried	Not a problem



# Loop Carried

- A data dependence exists if the computation of one data point requires **previously computed** other data.
- If the required data are computed in **another loop iteration**, the dependence is called “loop carried”.
- Loop carried dependence are often a **problem** because they assume an execution order that does not exist in a parallel loop.

# Flow Dependence

This is the “classic” data dependence.

Executing one iteration requires data from a previous one, thus forcing an order.

Flow dependences range from blatant ...

...to hidden and can often not be removed.

```
do i=2,n  
  x(i)=(x(i)+x(i-1))/2  
end do
```

```
do i=2,n  
  if(step(i).eq.1) y=i  
  x(i)=y  
end do
```

Hint:  
If **step(i).ne.1**  
the **y** value  
from previous  
iteration is used)

# Anti Dependence

This is a “backwards” data dependence in that one iteration requires data that would be modified “later” in the serial case, implying an order that is not there in the parallel case.

Anti dependences look a bit like flow dependences, but can usually be handled much easier.

```
do i=1,n-1  
    x(i)=(y(i)+x(i+1))/2  
end do
```

# Output Dependence

This is a data dependence that implies a serial loop order, usually by relying on a specific loop iteration being executed **last**, and using a variable from inside the loop outside of it.

Output dependences occur when assumptions are made about which iteration changes a variable last. They are easy to handle.

```
do i=1,n  
  a=(x(i)+y(i))/i  
end do  
f=sqrt(a+b)
```

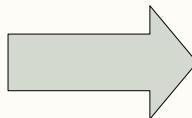
# Removing Dependencies

Type	Removal Techniques
Loop Carried Flow Dependence	<code>reduction()</code> clause loop skewing induction variable elimination
Loop Carried Anti Dependence	auxiliary array
Loop Carried Output Dependence	<code>lastprivate()</code> clause
Non Loop Carried	Not necessary

# Flow Dependences: reduction ()

As we have seen before, flow dependences can be removed by reduction if the operation that causes it does not depend on order

```
do i=0,m  
sum=sum+sqrt(dfloat(i))  
end do
```



```
!$omp parallel do reduction(+:sum)  
do i=0,m  
    sum=sum+sqrt(dfloat(i))  
end do  
 !$omp end parallel do
```

No problem, since the order  
of summation does not matter

Only in special cases

# Flow Dependencies:

## loop skewing

If the computation of one array element in one iteration depends on an element of another array from a “previous” iteration, shifting computations to another iteration (“loop skewing”) solves the problem.

```
do i=2,n  
  x(i)=(x(i)+y(i-1))/2  
  y(i)=y(i)+z(i)  
end do
```

“Loop Skewing”

```
x(2)=x(2)+y(1)  
 !$omp parallel do  
 do i=2,n-1  
   y(i)=y(i)+z(i)  
   x(i+1)=(x(i+1)+y(i))/2  
 end do  
 y(n)=y(n)+z(n)
```

Order reversed!

Can't be parallelized because iteration  $i$  needs  $y$  element from iteration  $i-1$

Regrouping one line makes dependency non-loop carrying

Only in special cases

# Flow Dependencies: elimination of induction variables

In some cases, variables that establish a data dependence can be eliminated by reference to the loop index.

```
factor=1  
  
do i=1,n  
    x(i)=factor*y(i)  
    factor=factor/2  
  
end do
```

“Elimination”

```
!$omp parallel do  
  
do i=1,n  
    x(i)=y(i)*0.5**(i-1)  
  
end do  
  
factor=0.5**n
```

**factor** establishes an unnecessary dependence ...

.. and might as well be kicked out of the loop. If it is used later, we may compute it outside.

Warning: This works only in special cases

# Anti Dependencies: auxiliary array

Anti dependences can be resolved by copying the needed data into a new array that contains the needed elements as they were before the parallel loop was executed.

```
do i=1,n-1  
  x(i)=(y(i)+x(i+1))/2  
end do
```

“Auxiliary” xp

```
xp=x  
!$omp parallel do  
do i=1,n-1  
  x(i)=(y(i)+xp(i+1))/2  
end do
```

Since nothing has happened to x(i+1) when it is needed in serial...

.. we can save the unaltered x in xp before the loop and eliminate the dependency

# Output Dependences: **lastprivate()**

Output dependences occur when the value of a variable that is used inside and outside of the loop depends on a specific iteration being executed last. The **lastprivate()** clause takes care of this.

```
do i=1,n  
  a=(x(i)+y(i))/i  
  z(i)=a  
end do  
f=sqrt(a+b)
```

"lastprivate"

```
!$omp parallel do lastprivate(a)  
do i=1,n  
  a=(x(i)+y(i))/i  
  z(i)=a  
end do  
f=sqrt(a+b)
```

The implicit assumption is that iteration **n** is last to alter **a** ...

... which is exactly the effect of **lastprivate(a)**



# Outline

- ❑ Parallel regions: out of the loop
- ❑ Work sharing in parallel regions
- ❑ `threadprivate` and `copyin`
- ❑ `critical` regions and synchronization
- ❑ What to do about false sharing



# Parallel Regions

- ❑ Not all OpenMP parallelism is “loop parallelism”
- ❑ It is possible to define a “stand-alone” parallel region using

```
parallel
end parallel
```

in Fortran or

```
parallel
{ }
```

in C
- ❑ The effect of this is that a set of threads is created and all of them work through the enclosed block of code separately, just like in MPI
- ❑ This style of OpenMP programming requires the use of routines.

# Hello World

OpenMP directives enclose a  
parallel region

```
program helloworld
  !$ use omp_lib
  write (*,*) ' Here is the main thread (serial) ...'
  !$omp parallel
  !$  write (*,*) ' ... and here is thread number '&
  !$      ,omp_get_thread_num(),' (parallel) ...'
  !$omp end parallel
  write (*,*) ' ... and now it is serial again.'
end program helloworld
```

These enclosed lines are compiled “conditionally” (!\$ followed by a blank), i.e. only if OpenMP is enabled with the `-xopenmp` flag. They are calling the OpenMP supporting routine `omp_get_thread_num()` to determine their “ID” (like rank in MPI).

# Using parallel/end parallel

- ❑ The workload must be allocated explicitly
- ❑ The techniques used are similar to the ones used in distributed-memory (MPI) programming
- ❑ In many cases, threads work on separate portions of one or several (shared) arrays

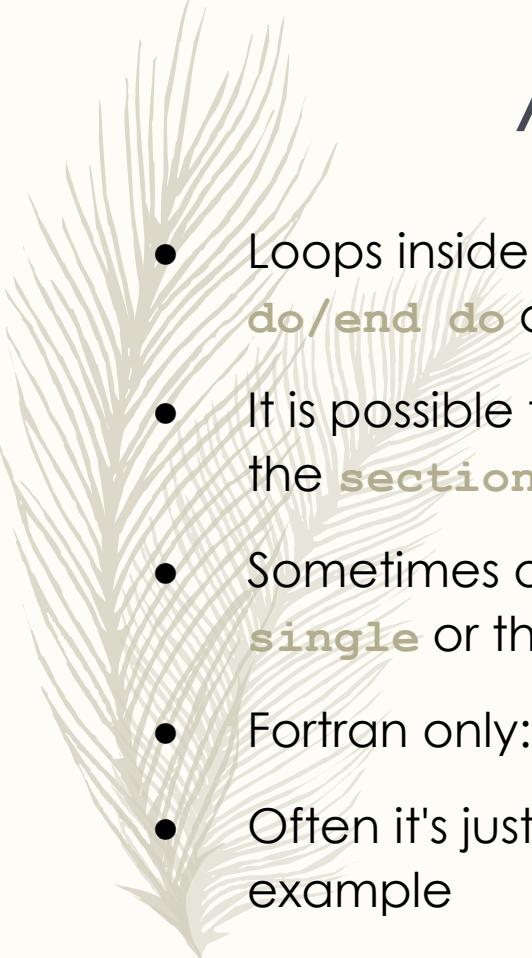
# Sum of Square Roots (inefficient)

Number of threads and thread number ()  
(like size and rank in MPI)

```
nthr = omp_get_num_threads()
sub = (m-1)/nthr+1
 !$omp parallel private(ithr,from,to)
    ithr = omp_get_thread_num()
    from = ithr*sub+1
    to = min(from+sub-1,m)
    do i=from,to
        sqrs(i)=sqrt(dfloat(i))
    end do
 !$omp end parallel
```

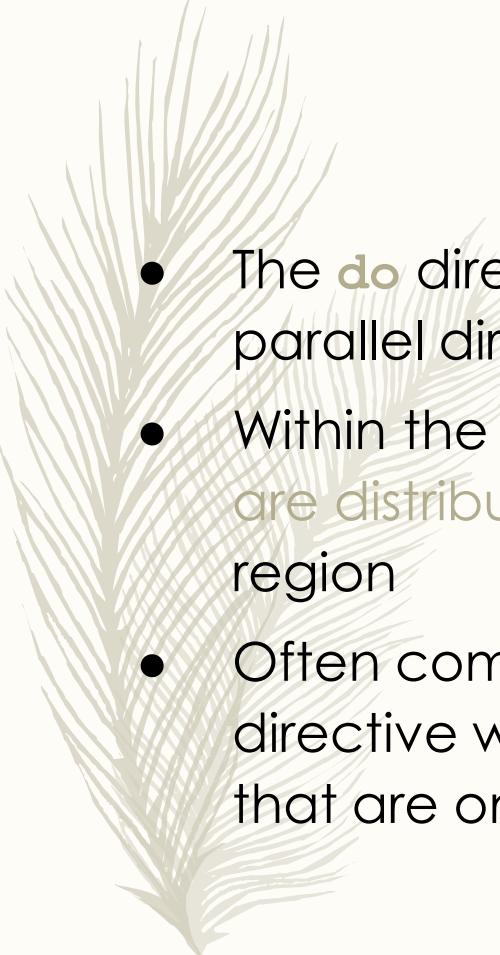
Work load computed explicitly. Each thread does part of the work (**from..to** are private)

The sum over the array is done sequentially afterwards



# Assigning Work

- Loops inside a parallel regions can be handled using the `do/end do` directive(s)
- It is possible to designate sections for different threads by the `section` directive
- Sometimes only one thread is needed:  
`single` or the `master` directives
- Fortran only: `workshare`
- Often it's just done "manually", just as in the previous example



# The **do** Directive

- The **do** directive is called **within** a pre-defined (via parallel directive) **parallel region**.
- Within the loop enclosed by **do** and **end do** iterations are distributed the same way as in a **parallel do** region
- Often combined with the **single/end single** directive which marks region inside a parallel region that are only executed by one thread

# do/end do

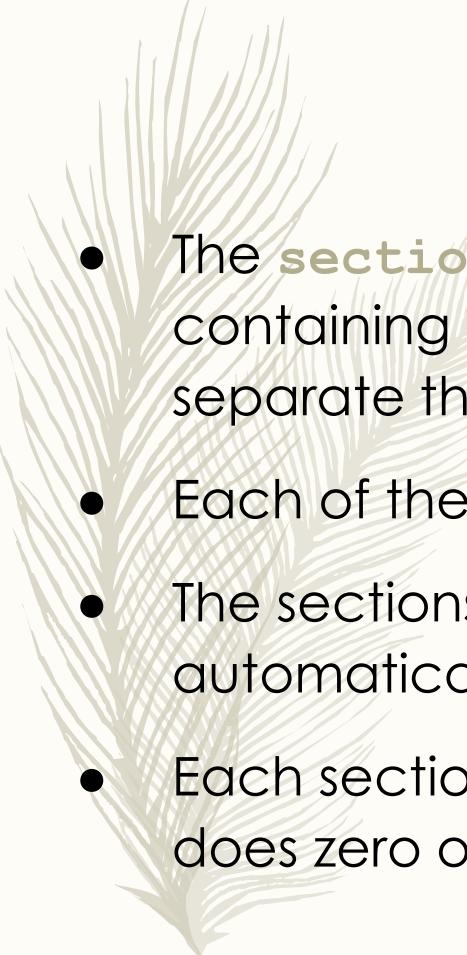
```
!$omp parallel
```

(replicated)

```
!$omp do  
do i=1,n  
...  
end do
```

(shared as in  
parallel do)

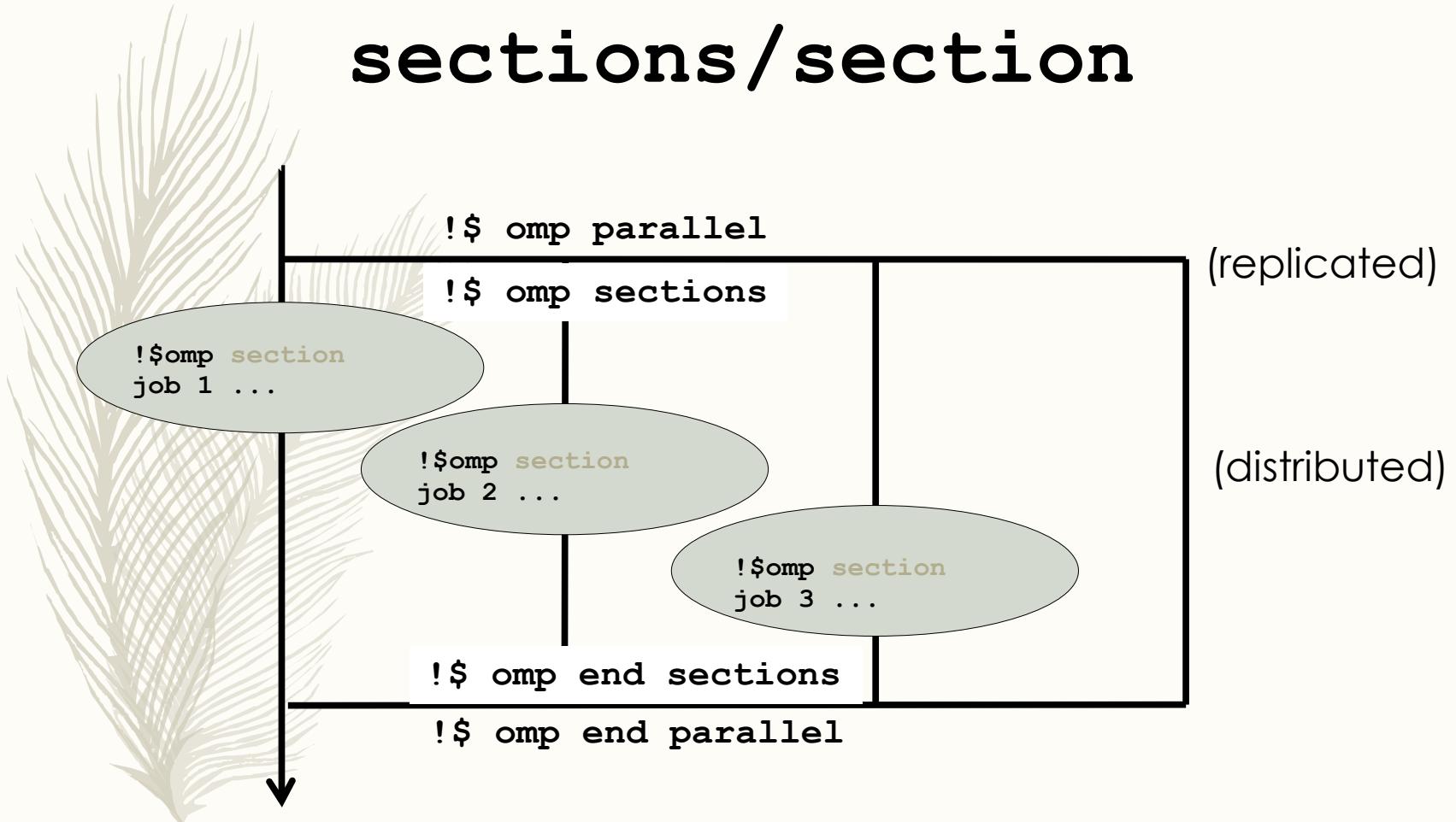
```
!$omp end parallel
```

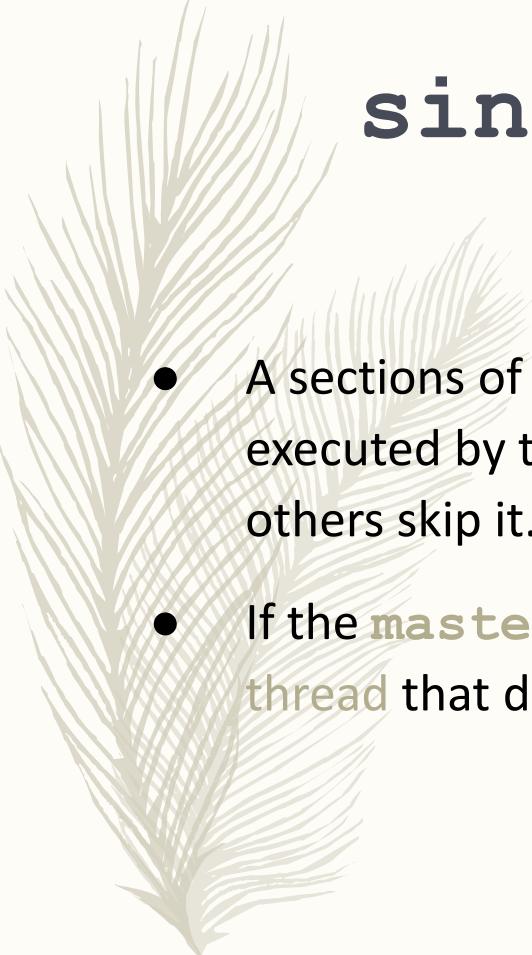


# **section** Directive

- The **sections** directive declares part of the code as containing “chunks” of work that are executed by separate threads
- Each of the chunks start with a **section** directive
- The sections are then distributed among threads automatically.
- Each section is executed by one thread, each thread does zero or more sections.

# sections/section





# single and master

- A sections of code labeled by the **single** directive is **only** executed by the **first thread** that encounters them. The others skip it.
- If the **master** directive is used instead, it is the **master thread** that does it. The others skip it.

# single and master directives

Thread 0 (master)

```
!$omp master  
... master job ...  
!$omp end master
```

```
!$omp parallel
```

```
!$omp single  
...single job...  
!$omp end single
```

(replicated)

(only one thread)

```
!$omp end parallel
```

# workshare (Fortran only)

- Fortran offers special array syntax that lets you assign and manipulate arrays and **array sections** simple statements
- **workshare** “splits” these statements into units and assigns **blocks** of such units to multiple threads
- Also works with **forall** and **where** statements
- The assignment to threads is implementation dependent

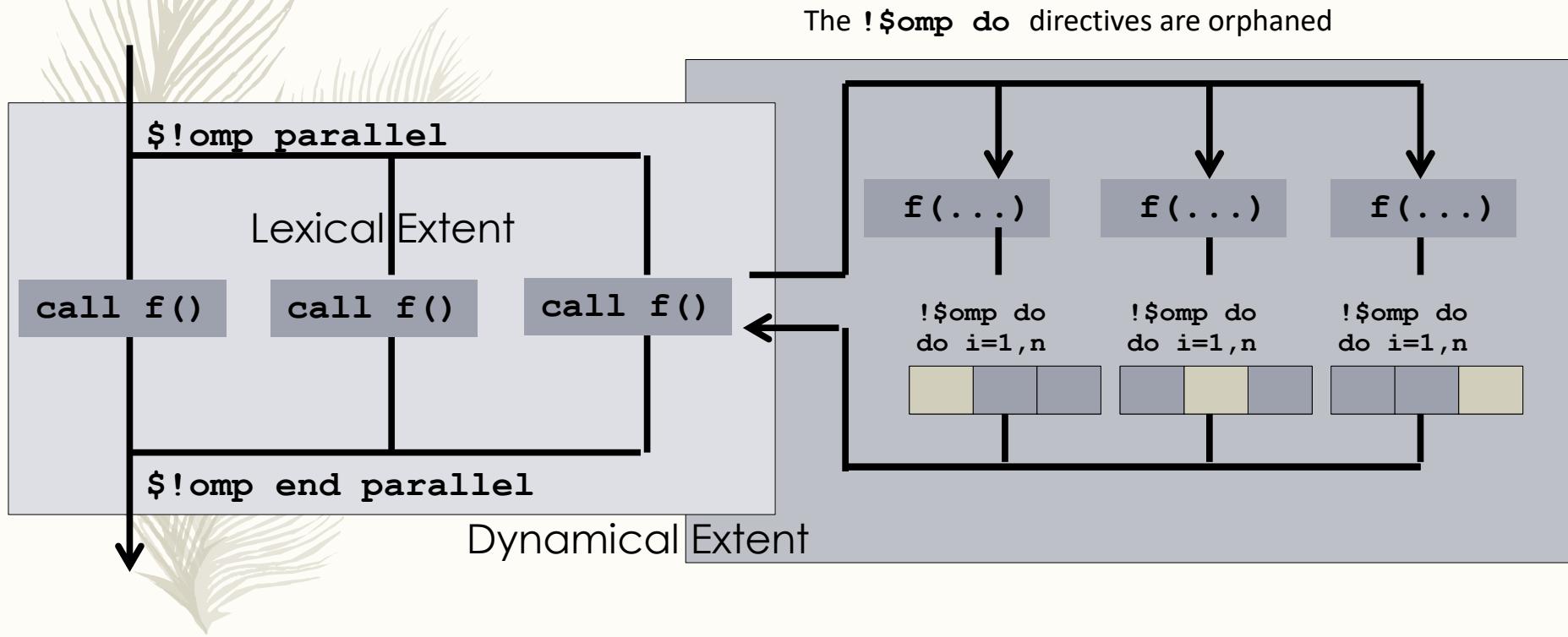
```
!$omp parallel
  !$omp workshare
    a(:,1)=b(:)*x(:,2)
  !$omp end workshare
 !$end parallel
```

Fortran allows sections of arrays;  
(:) stands for full range;  
**workshare** splits up computations and  
assigns them to the threads in the  
parallel region

# Lexical and Dynamic Extents, Orphaning

- The block of code that appears between the **parallel/end parallel** directives is called the **lexical extent** of a parallel region
- If we include the code in all routines that are called, we obtain the **dynamic extent**
- Directives that appear in those routines, i.e. in the **dynamic** but not the **lexical** extent, are called **orphaned**
- Orphaning directives is frequently necessary, e.g. with **do** directives

# Lexical and Dynamic Extents, Orphaning



# Usage of `threadprivate`

Sometimes global data cause race conditions:

```
program main
common /problem/ w(1000)
...
!$omp parallel do
do i=1,n
    call sub(i)
end do
...
end
subroutine sub(j)
    common /problem/ w(1000)
    ...
    do i=1,1000
        w(i)=...j...
    end do
    ...
    return
end subroutine
```

Either expand the common block or global array...

```
common /problem/ w(1000,nt)
...
    it=omp_get_thread_num()+1
    do i=1,1000
        w(i,it)=...j...
    end do
    ...

```

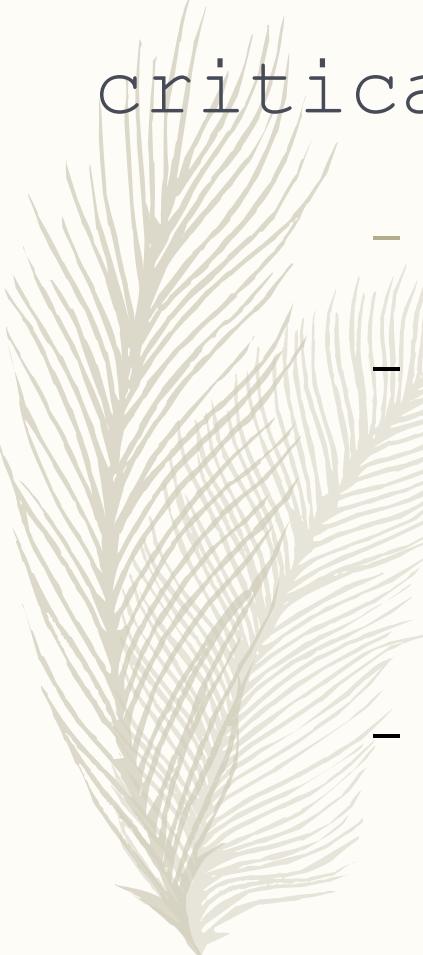
... or declare it `threadprivate`

```
common /problem/ w(1000)
!$omp threadprivate(/problem/)
...
    do i=1,1000
        w(i)=...j...
    end do
    ...

```

# Important: Synchronizing Threads

- Often threads need to be **synchronized** to keep them from getting in each other's way.
- Synchronization helps **resolve race conditions**
- The simplest way is the **critical/end critical** directive
- There are others:  
**barrier** directive  
**atomic** directive  
**ordered/end ordered** directive  
**lock** routines in the runtime library
- Synchronizations might have the effect of slowing things down (by forcing threads to “wait”)

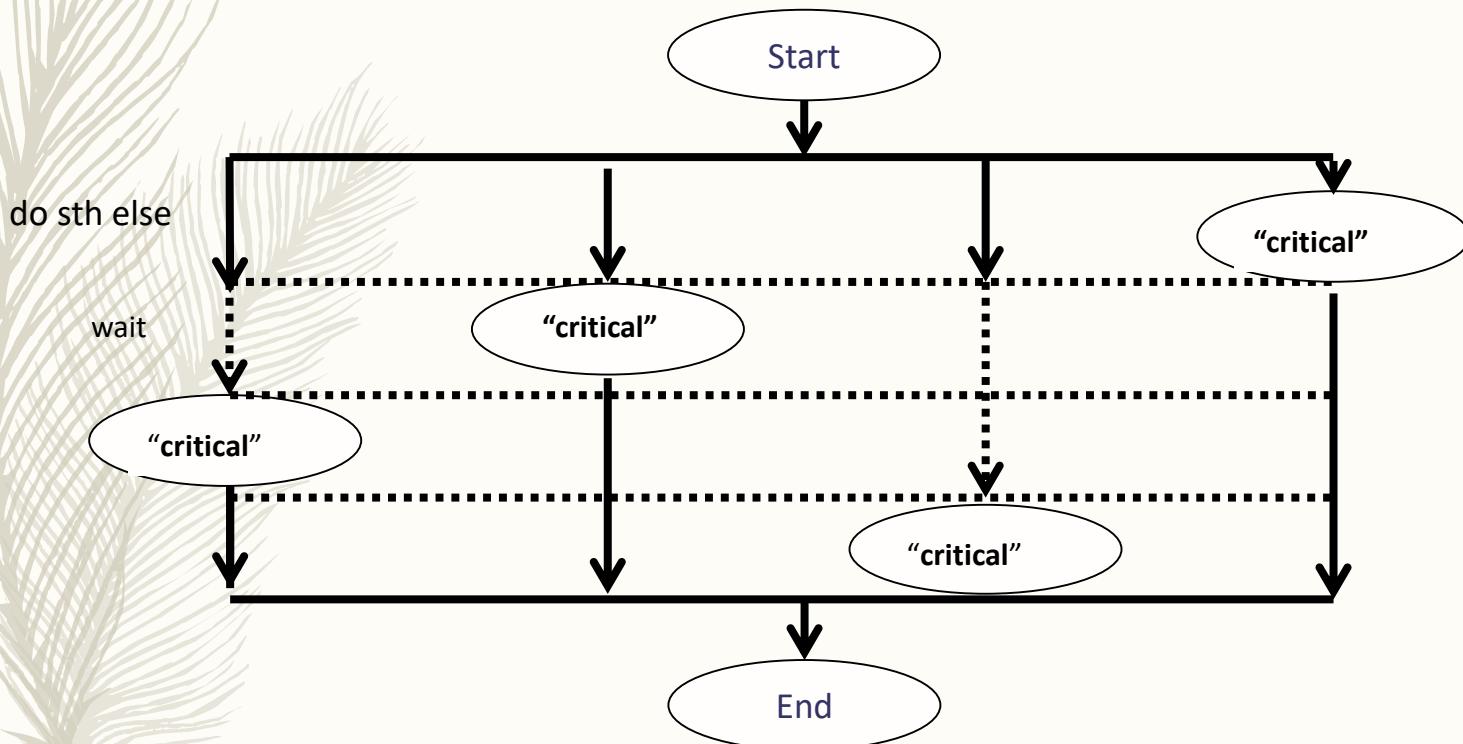


# critical Regions

- Critical regions are executed by only one thread at a time, although all threads execute them
- They are created by a **critical** directive:

```
!$omp critical  
...block...  
!$omp end critical
```
- While one thread executes a critical region the others wait if they have nothing else to do

# critical Regions



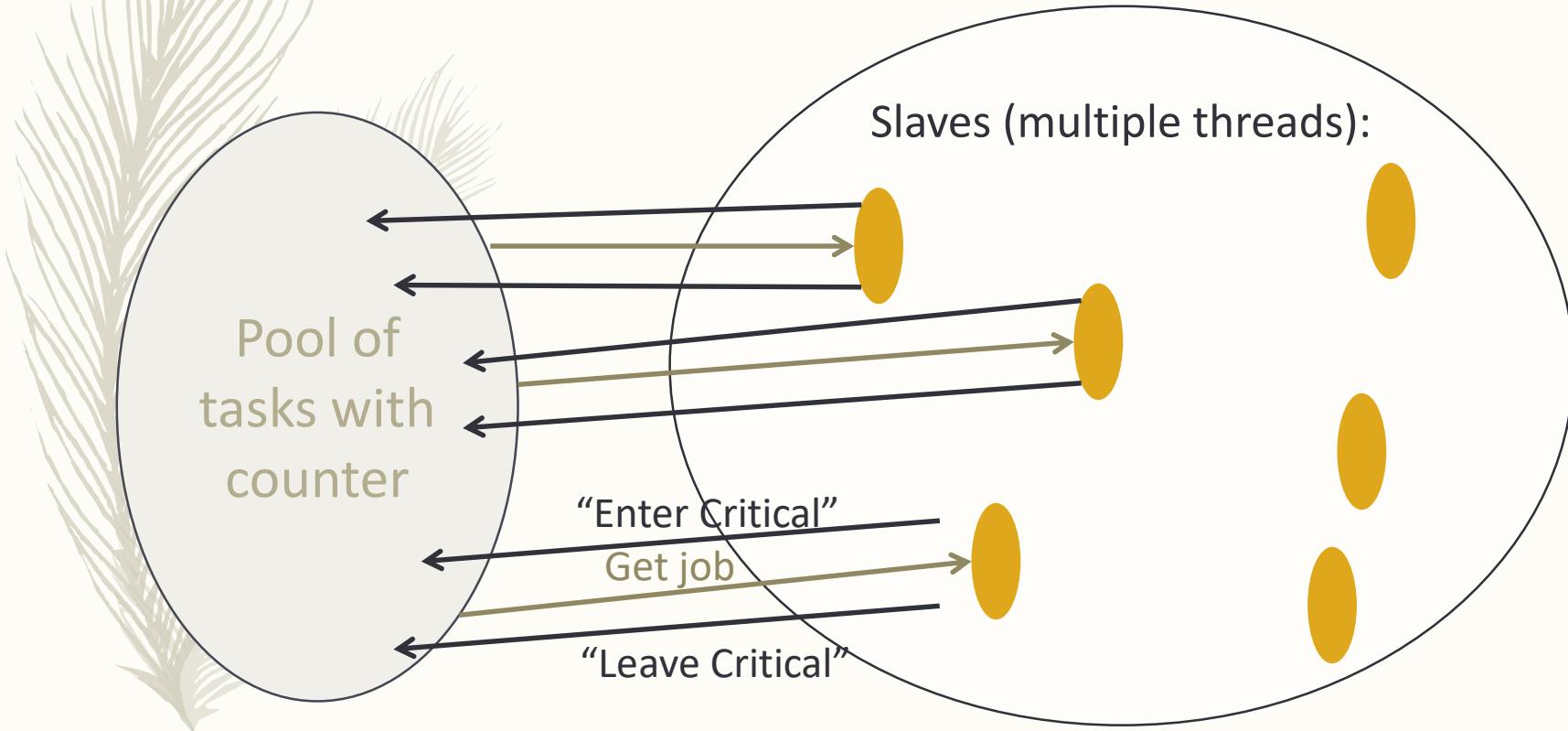
# Example: “All Slaves”

- In the “all slaves” model, a pre-determined number of tasks is given to threads whenever they are idle
- The distribution has to be done “one at a time”, using critical regions. The work itself is done in parallel
- A very similar effect is achieved by the dynamic scheduling within a parallel do loop.
- This is the shared-memory equivalent of the Master-Slave model in MPI, but because of the shared memory no master is needed.

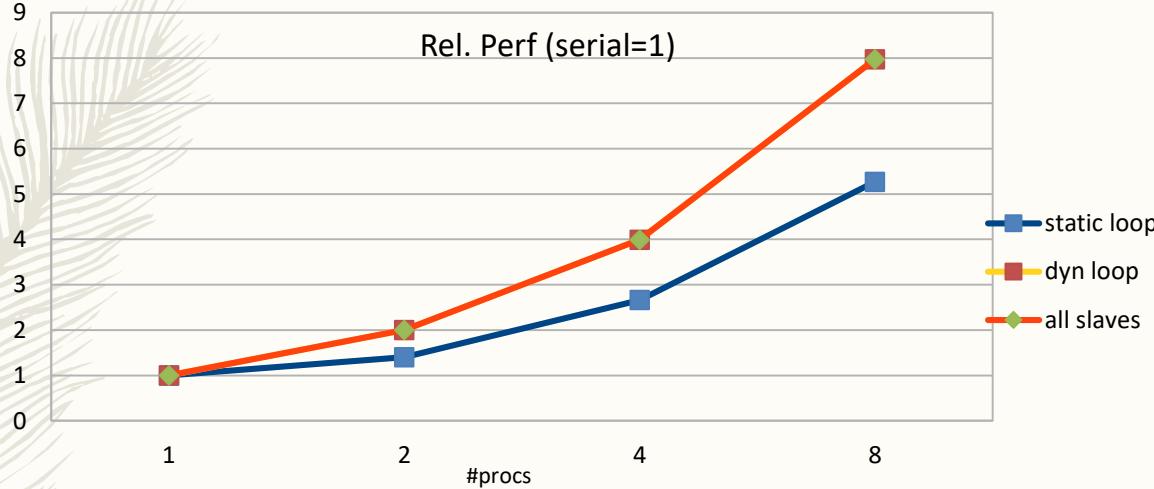
**Fortran** [Code](#)

**Example** [Runs](#)

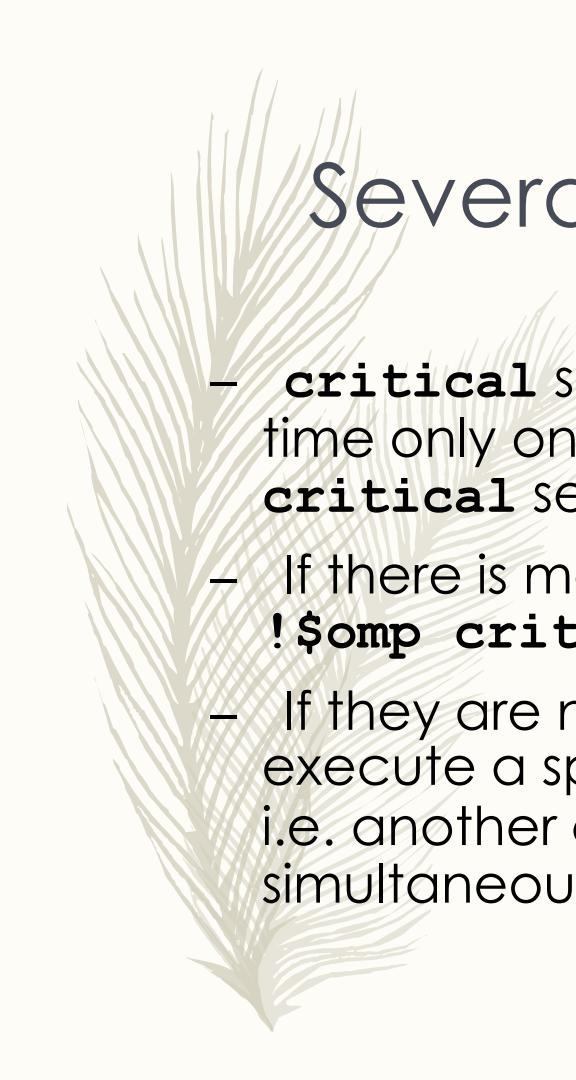
# “All Slaves” parallel model



# The Return of Mandelbrot



The timings in an all-slaves model, where each task corresponds to a given imaginary part, are virtually identical to a loop schedule (dynamic,1). Scaling is almost perfectly linear.



# Several critical Sections

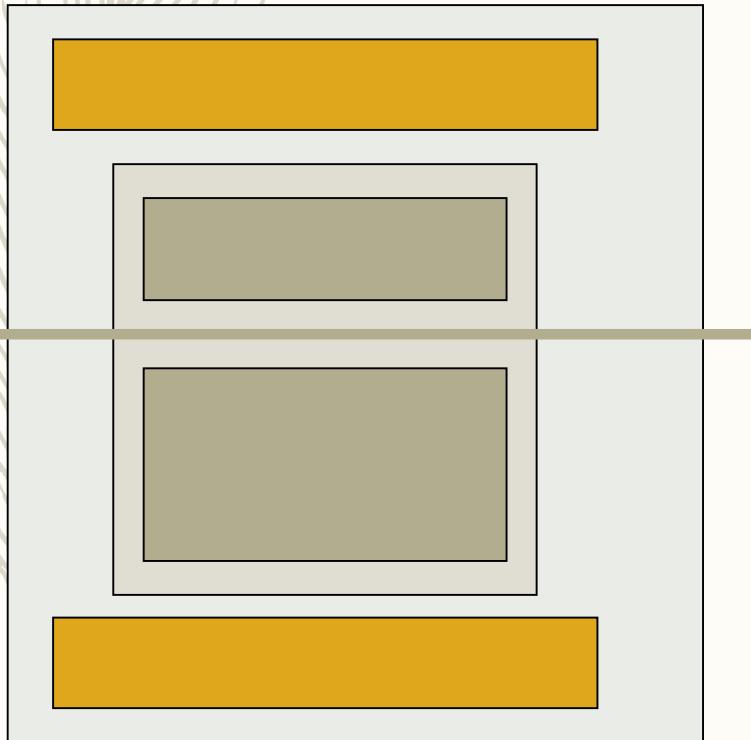
- **critical** sections are **global**, i.e. at any one time only one thread can execute only one **critical** section
- If there is more than one, they can be **named**:  
`!$omp critical (NAME)`
- If they are named, only one thread can execute a specific **critical** section at a time, i.e. another can execute another simultaneously



# barrier

- Works like a **barrier** in MPI: all threads must go pass it for anyone to continue
- Only makes sense within a **parallel** region
- Usually placed between two separate sub-regions, one of which depends on the other

# barrier “Pseudo Example”



Serial part, e.g. reading in data

Parallel part, with two distinct sections,  
e.g. `!omp parallel`

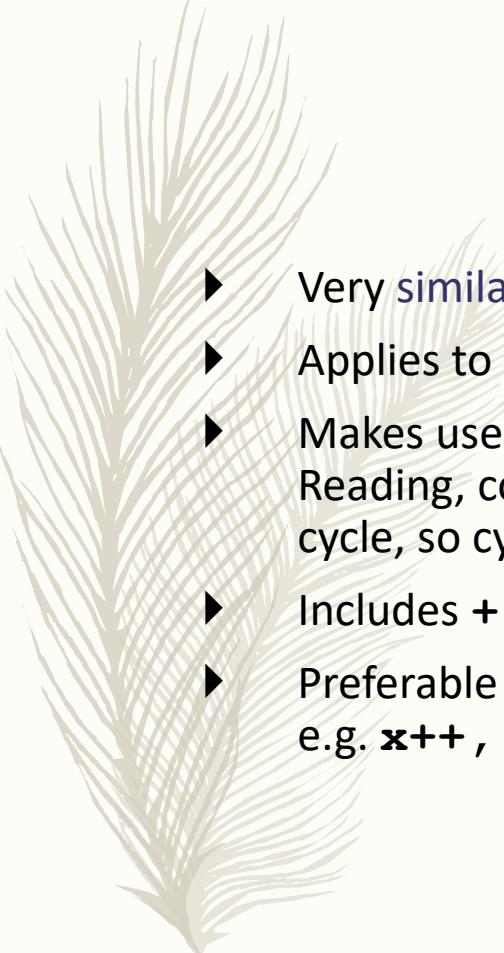
Computing elements of an array...

...synchronizing with `!$omp barrier`...

...and multiplying each element  
of the array with all the others.

Parallel part ends,  
e.g. `!omp end parallel`

Serial part, e.g. output of results



# atomic

- ▶ Very similar in effect to **critical** region
- ▶ Applies to only one simple update of a scalar variable
- ▶ Makes use of hardware:  
Reading, computing, writing are done within single clock cycle, so cycle is blocked
- ▶ Includes + - \* / &(and) |(or)
- ▶ Preferable if expression is very simple  
e.g. `x++`, `y*=5`, `x(i)=x(i)/3`

# Using `atomic`

```
...  
!$omp parallel do  
do i=1,n  
  
...  
!$omp atomic  
x(index(i))=x(index(i))+1  
...  
end do  
...
```

Fixing a race condition:

The loop is executed in parallel

If `index` is not unique,  
one thread might update `x` while  
another is using the old version

The `atomic` directive fixes that, making sure  
that only one thread refers to `x` at a time.  
Almost no overhead.

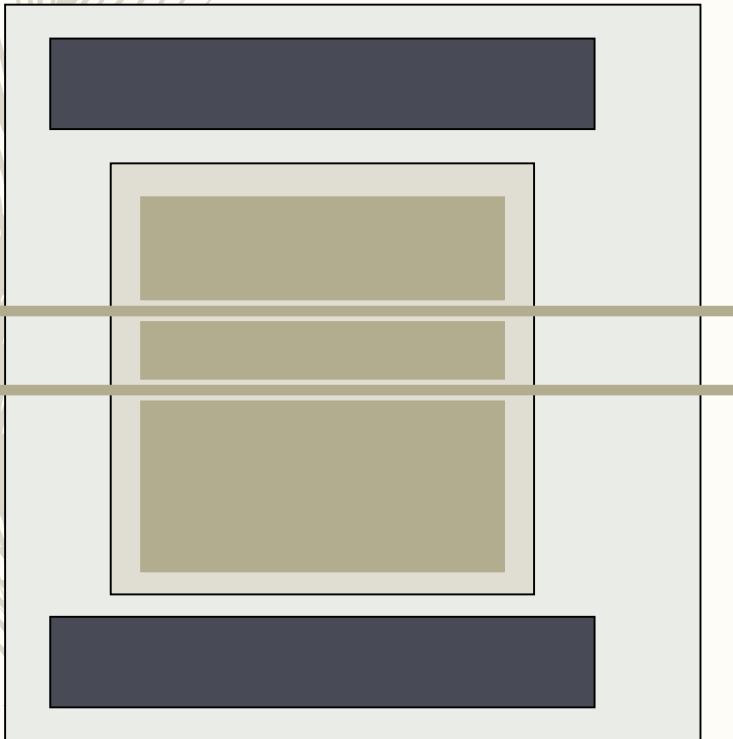
# ordered / end ordered

- Sometimes it is necessary that operations in a parallel region are performed in the “original”, i.e., **serial** order
- Such operations can be enclosed in

```
!$omp ordered  
...block...  
!$omp end ordered
```
- This directive has the potential of adversely affecting the efficiency of the parallel region



# ordered “Pseudo Example”



Serial part, e.g. reading in data

Parallel part begins, e.g.

`!$omp parallel do ordered`

Computing elements of a vector...

...starting region: `!$omp ordered`

...printing the vector out properly...

...ending region: `!$omp end ordered`

...and doing some other stuff with the vector in parallel.

Parallel part ends, e.g.

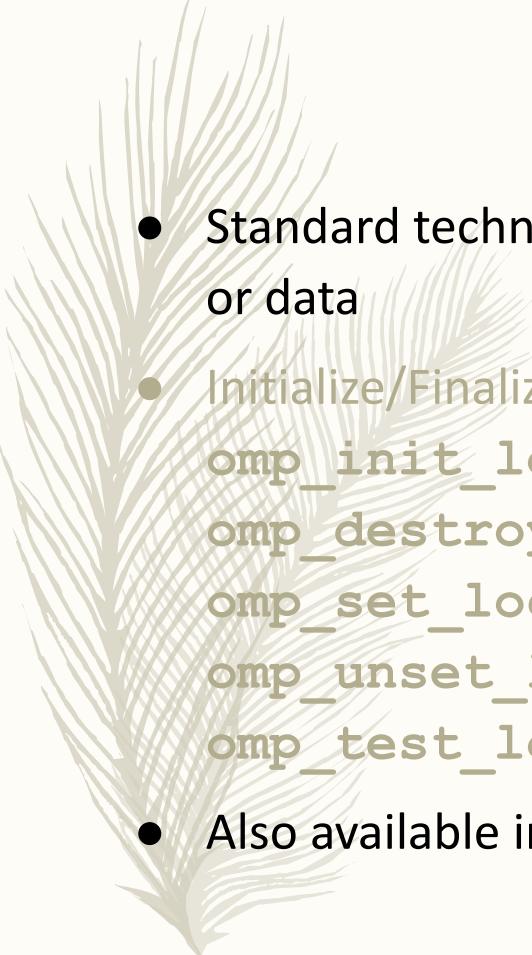
`!$omp end parallel do`

Serial part, e.g. output of final results



# ordered Regions

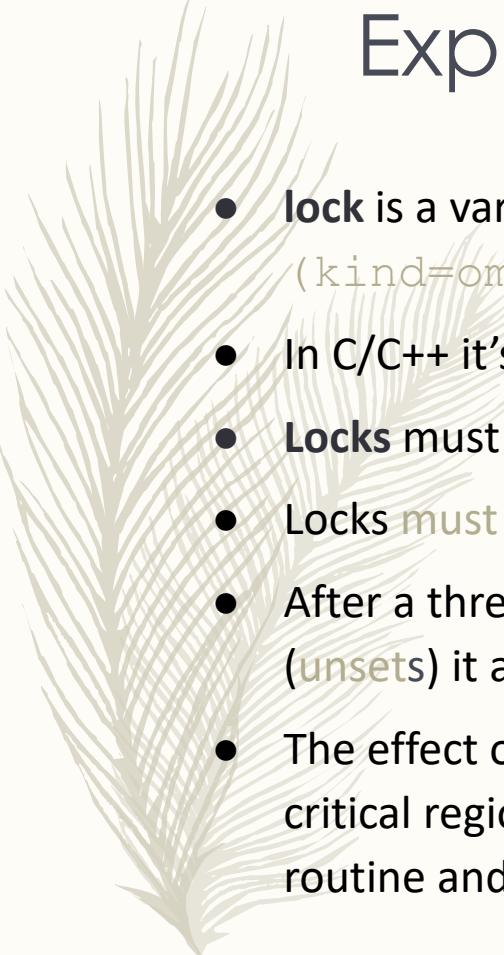
- Ordering **only** applies to the **enclosed block**, not relative to statements outside of it, i.e. the block statements are partly executed in parallel with others
- To minimize the impact of this construct, it is best to **keep the enclosed blocks small**, and preferably near the end of a **parallel** region
- Often **ordered** regions are used for I/O that needs to be done in a specific order



# Explicit Locks

- Standard technique if several processes might want to access files or data
- Initialize/Finalize, Acquire/Release, and Test routines are available:

<code>omp_init_lock(lock)</code>	[Initialize lock]
<code>omp_destroy_lock(lock)</code>	[Finalize lock]
<code>omp_set_lock(lock)</code>	[Acquire lock]
<code>omp_unset_lock(lock)</code>	[Release lock]
<code>omp_test_lock(lock)</code>	[Test lock]
- Also available in `omp_*_nest_lock` variety



# Explicit Locks (cont'd)

- **lock** is a variable that can hold an address (Fortran), best `integer (kind=omp_lock_kind)`
- In C/C++ it's a pointer of type `*omp_lock_t`
- **Locks** must be initialized/finalized outside a parallel region
- Locks must be shared
- After a thread has acquired (`set`) a lock, all others wait until it releases (`unset`s) it again
- The effect on the region between `set/unset` is similar to a named critical region, but the use is more flexible (for instance set in one routine and unset in another).

# Explicit Locks (cont'd)

- Locks are often used if the setting and unsetting needs to happen in different areas of the code, e.g. in different routines.
- They are more flexible than critical regions, but harder to program, as they require code alteration.
- Use them only if you need to, in most cases critical regions are easier.
- In the following example we use them anyway (although a critical would do). In a later advanced example they must be used.

# Explicit Locks: Example

Finding the smallest element in an array

```
call omp_init_lock(mini)
 !$omp parallel do
 do i=1,n
   if(array(i).lt.smallest) then
     call omp_set_lock(mini)
     if (array(i).lt.smallest) &
       smallest=array(i)
     call omp_unset_lock(mini)
   end if
 end do
 !$omp end parallel do
 call omp_destroy_lock(mini)
```

Creating a lock

Starting the parallel region

Saving time: only if element is smaller  
do we need to acquire the lock

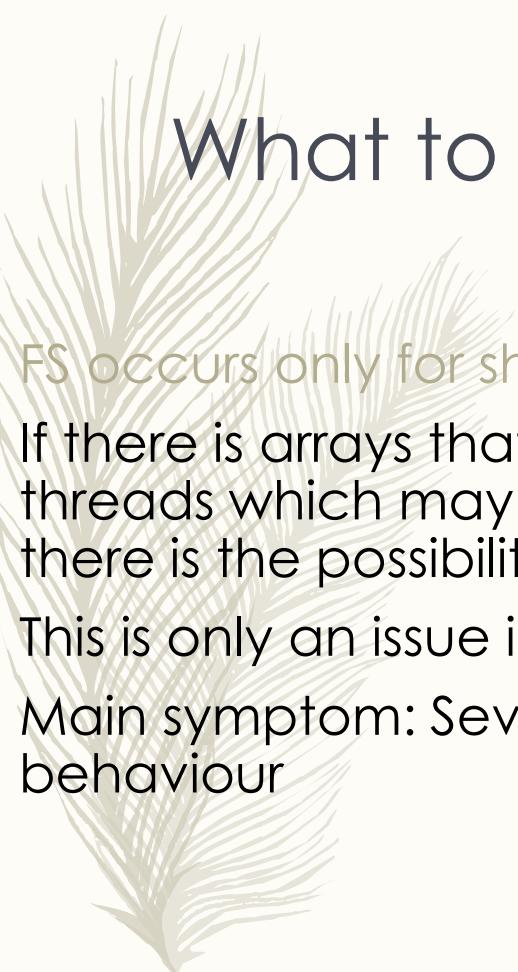
Check again (might have changed),  
and release the lock

End of parallel region,  
Lock is not required anymore

Fortran Code

Run with lock

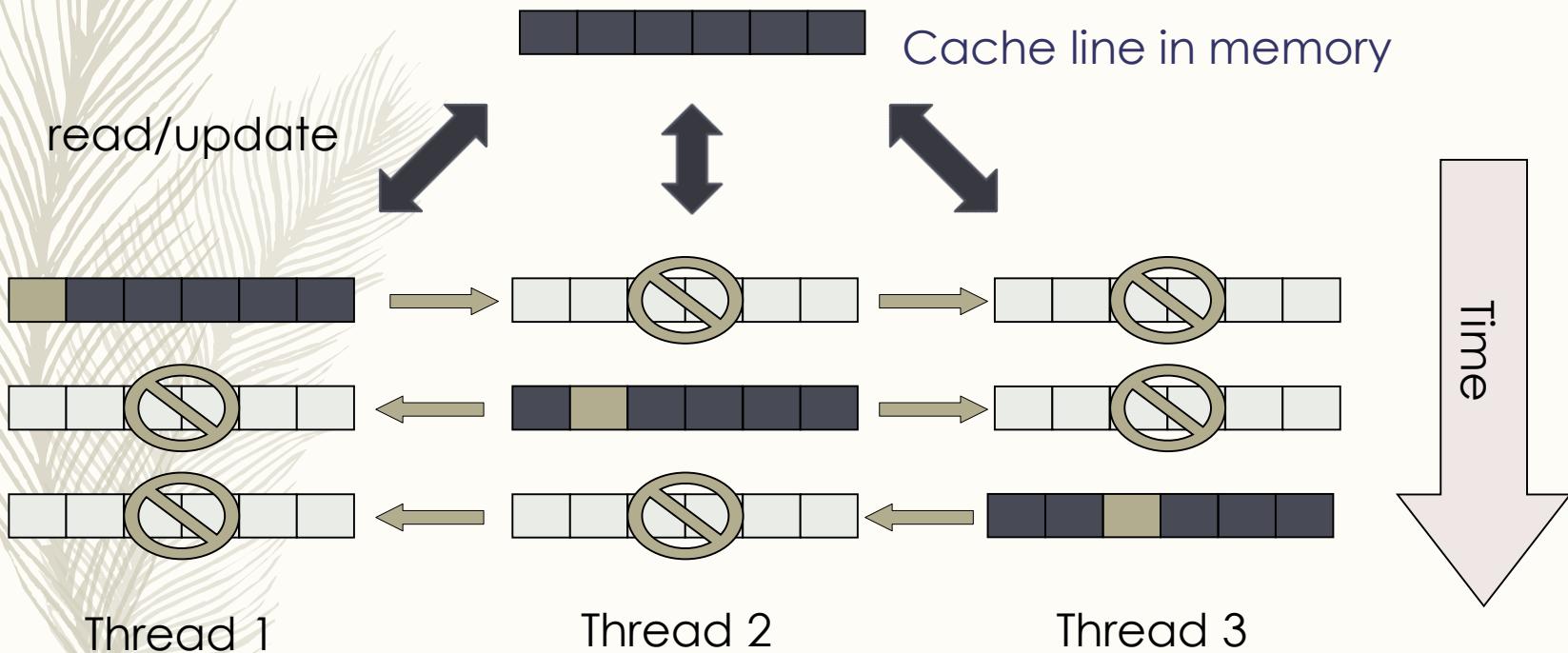
Run without lock



# What to Do about False Sharing ?

- FS occurs only for shared arrays that are read/write
- If there are arrays that are modified by multiple threads which may have the same cache line, there is the possibility of FS
- This is only an issue if data updates are frequent
- Main symptom: Severely restricted scaling behaviour

# False Sharing





# Remedies

Antidotes for False Sharing include:

- “Privatization”
- “Think Big”
- Optimization
- Rescheduling
- Others

None of these is a silver bullet, although they can be very effective in some cases

# “Privatization”

- Arguably the easiest way to alleviate the issue
- To reduce the number of times that a variable in an array needs to be updated, **temporary private variables** can be introduced
- This does not eliminate false sharing completely, but can greatly reduce it.

# Example (from Sun Application Tuning Seminar example)

```
do i=1,100000  
  do j=1,100000  
    sum=sum+a(j,i)  
  end do  
end do
```

Version 1 (serial):  
Sum over all matrix elements

Parallelization needs **column sum vector** to avoid race condition on sum.  
Sum over columns is later done in serial.

OpenMP

```
!$omp parallel do
do i=1,100000
  col(i)=0.
  do j=1,100000
    col(i)=col(i)+a(j,i)
  end do
end do
do i=1,n
  sum=sum+col(i)
end do
```

Version 2 (**parallel**):  
Summing over column into  
vector `col()` in parallel

Single sum over `col` can be  
done in serial

Elements of `col` get hit too often,  
causing **False Sharing**

Privatization

```
!$omp parallel do private(col)
do i=1,100000
  col=0.
  do j=1,100000
    col=col+a(j,i)
  end do
  col(i)=col
end do
 !$omp end parallel do
do i=1,n
  sum=sum+col(i)
end do
```

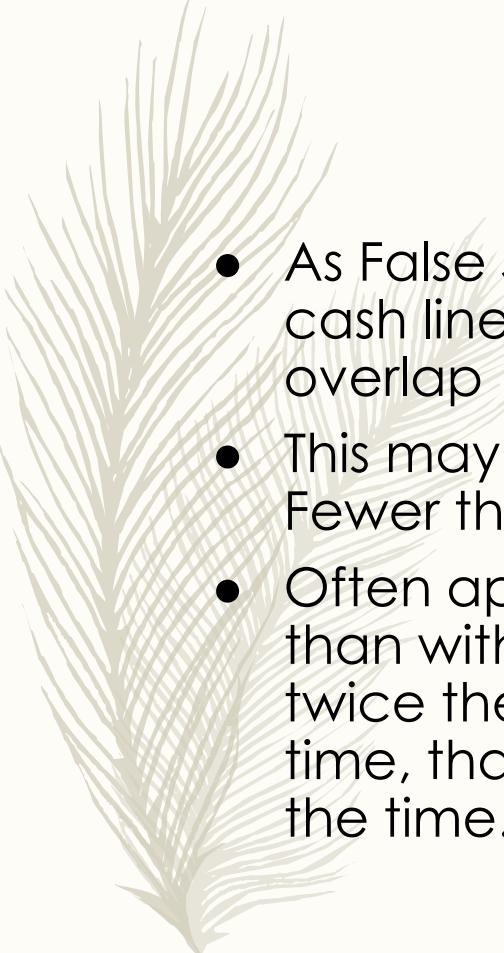
private variable **col** reduces  
number of updates from **n** to **1**,  
thus reducing false sharing

Version 3 (parallel, improved):

Vector **col** temporarily replaced  
by private variable **col**

Almost no false sharing

Finally, **col** gets hit only once



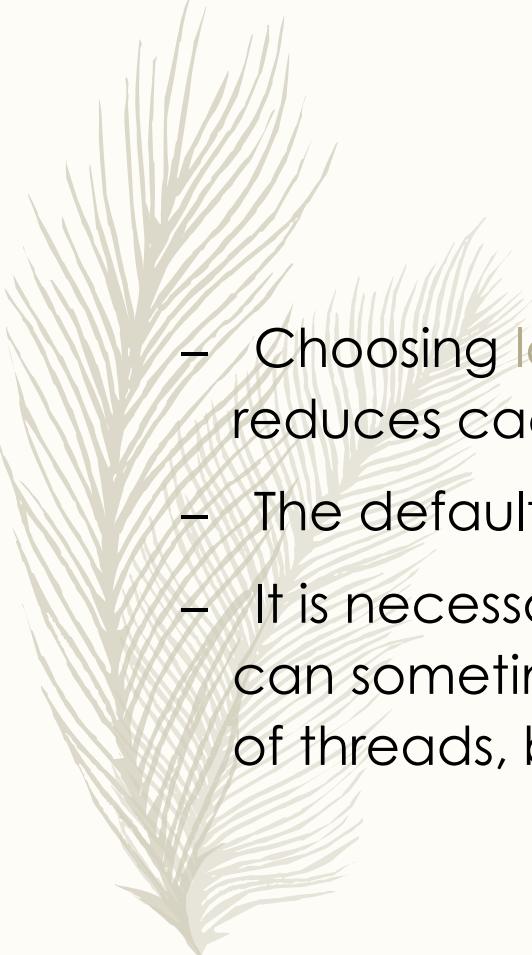
# “Think Big”

- As False Sharing happens only when threads share cash lines, FS can be alleviated by reducing cache overlap
- This may be done in two ways:  
Fewer threads (duh !) or larger data structures
- Often application scale better with problem size than with number of processors, i.e. it is easier to get twice the work done with two processors in the same time, than the same work with two processors in half the time.



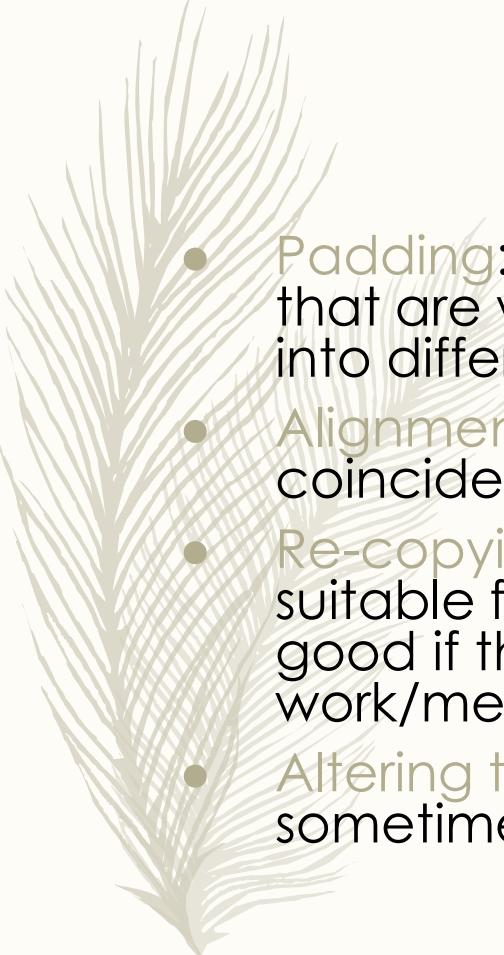
# Optimization

- Serial optimization often has the effect of alleviating false sharing
- For some machines, minimum optimization is enforced by the `-openmp` option anyway
- Forcing the alignment of data along “natural” boundaries improves cache coherence and is an optimization option for many compilers



# Rescheduling

- Choosing larger chunk sizes when scheduling loops reduces cache overlap and therefore false sharing
- The default schedule might not be good
- It is necessary to experiment, as different schedules can sometimes yield better results for large numbers of threads, but worse for small numbers



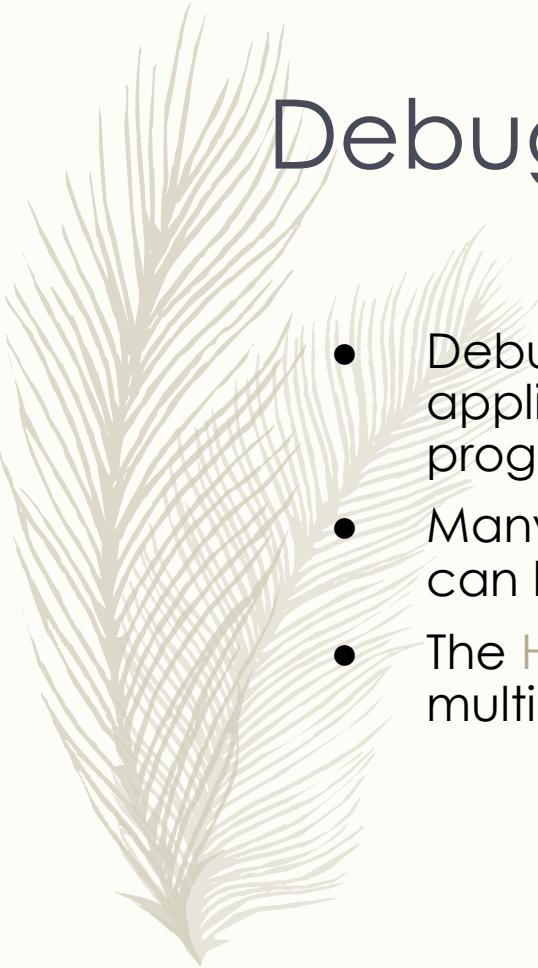
# Others

- Padding: Inserting “blank” data will separate data that are written in different threads, i.e. force them into different cache lines
- Alignment of data such that boundaries for threads coincide with boundaries for caches
- Re-copying data onto another structure that is more suitable for the access in the parallel loop. This is only good if the copying is much cheaper than the work/memory access in the loop
- Altering the loop, for instance by “blocking” will sometimes reduce false sharing



# If Time Allows:

- Some General Considerations
- OpenMP 3.0
- Tasking
- Further Reading



# Debugging/Profiling

- Debugging and profiling OpenMP applications is harder than with serial programs
- Many modern debuggers (e.g. SUN **xdb**) can handle multiple threads
- The **HPCVL Working Template** handles multiple threads.



# Parallel Principles

- Minimize repetition of heavy computations.
- Distribute simultaneous tasks among processes as evenly as possible, to reduce waiting time.
- Minimize memory conflicts, as they require protective regions which serialize the code. Use private or thread-private data if possible.
- Avoid close-by access to data because of the danger of false sharing. Common performance bottleneck.  
Don't overdo: data locality is key to serial performance.
- If necessary, copy data into local (private) variables to avoid memory problems.
- Parallelize outer loop rather than inner ones. This tends to space out memory access and reduces overhead.

# Parallelizing Serial Code

- Optimize serial code.
- Profile the code to determine which sections need parallelization (system tools, development tools, HWT).
- Introduce OpenMP framework into the code (header, compilation flags).
- Handle I/O: move I/O operations into serial regions, preferably Input at beginning, Output at end.
- Choose parallel method and parallelize “profitable” sections (new algorithm might be needed).
- Profile the code to determine scaling.
- Repeat the last two steps until meeting performance requirements.

# Some Features of OpenMP 3.0

- OpenMP 3.0 was introduced in 2008
- Designed “by committee” with user input
- Support for previously unsupported types of parallelism, prominently “tasking” in while loops.
- Complete Specification at  
<http://www.openmp.org/mp-documents/spec30.pdf>
- Quick reference at  
<http://openmp.org/wp/2009/03/openmp-30-fortran-summary-card/>

# OMP 3: General Features

- **Tasking**
- Waiting threads policies
- **Loop collapse** and nested parallelism
- Storage reuse
- Stack size control
- Multiple internal control variables



# OMP 3: Language Features

- Fortran:
  - Handling allocatable arrays
- C:
  - Unsigned and pointer loop control variables
- C++:
  - Constructors and destructors
  - Iterator loops
  - Enhanced `threadprivate` inside classes

# Tasking

Remember “impossible while loop” earlier ?

Some of those can now be handled.

## Important example: Linked lists of tasks

```
task = first_task;  
while (task != NULL) {  
    execute(task);  
    task=new(task);  
}
```

**while** Loop is implicitly dependent as it cannot be predicted when **NULL** will turn up.

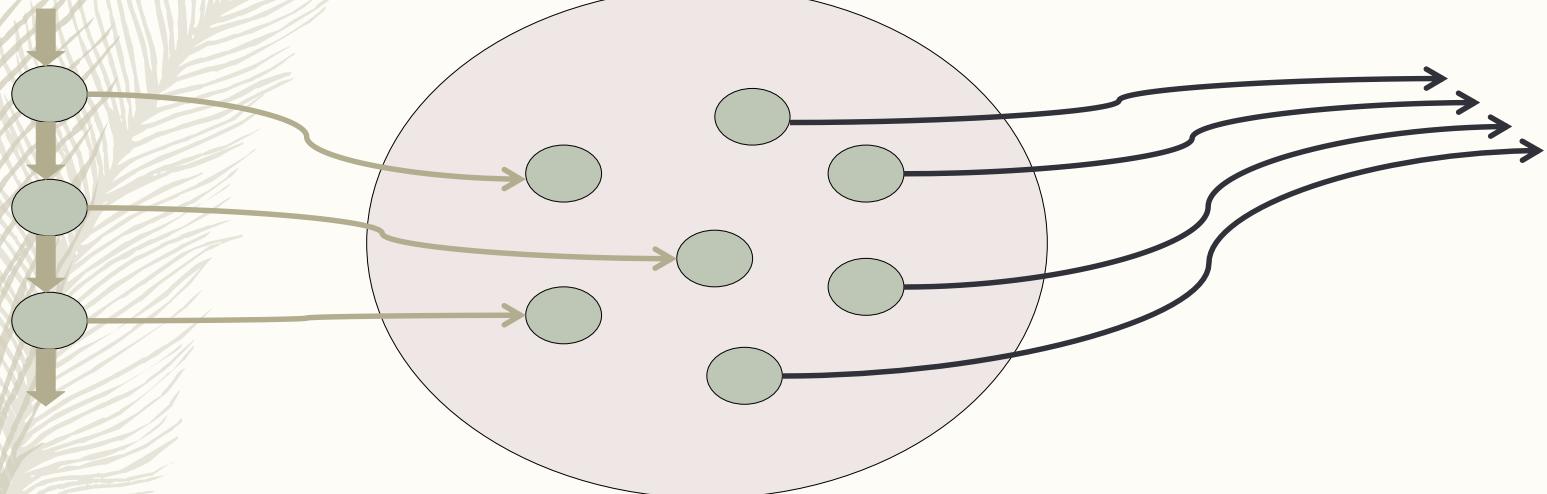
However:  
Often **new()** workload is very small compared to **execute()** workload.

Why not let one thread make a list of tasks while the others work on it?

# Tasking

Encountering Thread adds task to Pending Pool.

All threads work down the tasks from the Pending Pool.



# Tasking

```
task = first_task;
while (task != NULL) {
    execute(task);
    task=new(task);
}
```

Back to the example:  
Insert some OMP directives

```
#pragma omp parallel
{
    #pragma omp single private(task)
    {
        task = first_task;
        while (task != NULL) {
            #pragma omp task
            {execute(task);}
            task=new(task);
        }
    }
}
```



# Tasking

**Fortran:**

```
!$omp task  
!$omp end task
```

**C/C++:**

```
#pragma omp task
```

Designates a block of code that constitutes a task.

If used inside a parallel/single region, causes the encountering thread to add a “possibly deferred” task to a pool that can be worked on by all threads in any order.



# Tasking

**Fortran:**

**`!$omp taskwait`**

**C/C++:**

**`#pragma omp taskwait`**

Current **task** is suspended until all tasks generated within it are done (**task barrier**). Implicit or explicit thread barriers also have this effect.

# Collapsing Loops

**Fortran:**

```
!$omp parallel do collapse(n)
```

**C/C++:**

```
#pragma omp parallel for collapse(n)
```

Sometimes nested loops are very simple, and may be “collapsed”, i.e. turned into a single loop which is then parallelized.  $n$  denotes the number of loop levels that are eliminated.

# Combining MPI and Multithreading

- New chip architectures:  
Multi-core & multi-threaded allow a single core (CPU) to execute multiple threads
- If used in cluster setting, makes use of OpenMP or Posix threads combined with MPI desirable
- MPI library should be thread-safe, but don't rely on it.
- Each of  $n$  MPI processes dynamically creates dynamically  $m$  threads per process for a total of  $N=n \times m$
- Will be discussed in more detail in MPI course

# “Sum-of-Square-Roots” MPI/OpenMP Hybrid

- Remember the square-root example?
- Each process goes through different elements of the loop (MPI)
- Loop could be further distributed among threads, using OpenMP

Code in Fortran

Code in C

Code in C++

# Further Reading

- OpenMP website: [www.OpenMP.org](http://www.OpenMP.org)
- Chapman et al. [Using OpenMP](#)
- Chandra et al. [Parallel Programming in OpenMP](#)
- MJ Quinn [Parallel Programming in C with MPI and OpenMP](#)



**Thanks for  
Your Attention**