

# OpenMP by Example

To copy all examples and exercises to your local scratch directory type:

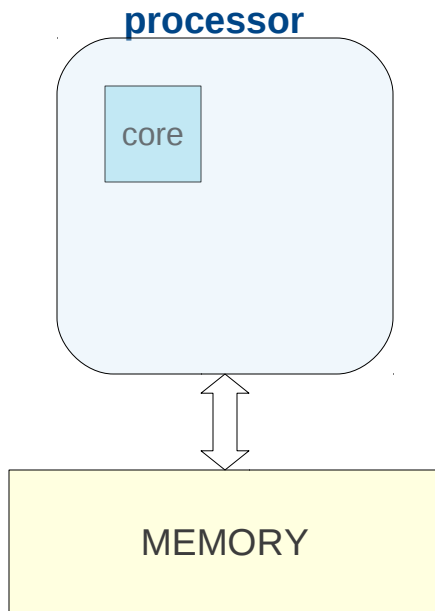
[/g/public/training/openmp/setup.sh](#)

To build one of the examples, type `"make <EXAMPLE.X>"` (where `<EXAMPLE>` is the name of file you want to build (e.g. make test.x will compile a file test.f)).

To build all examples at once, just type `"make"`

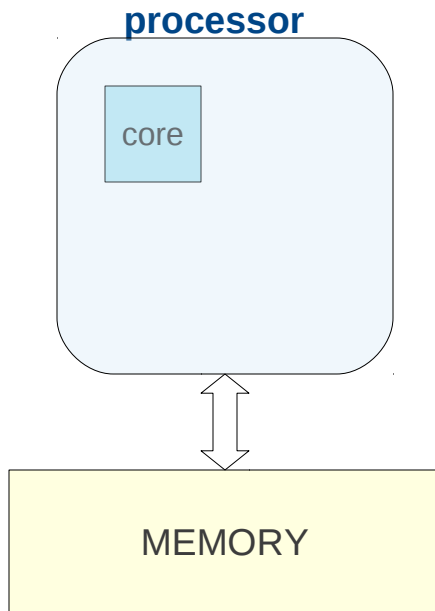
# Basic Architecture

Older processor had only one  
cpu core to execute instructions

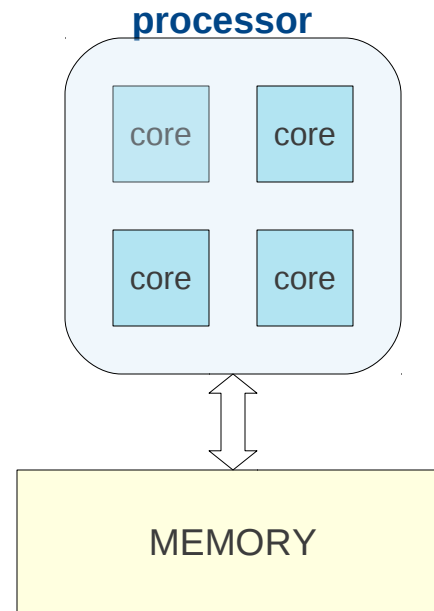


# Basic Architecture

Older processor had only one  
cpu core to execute instructions



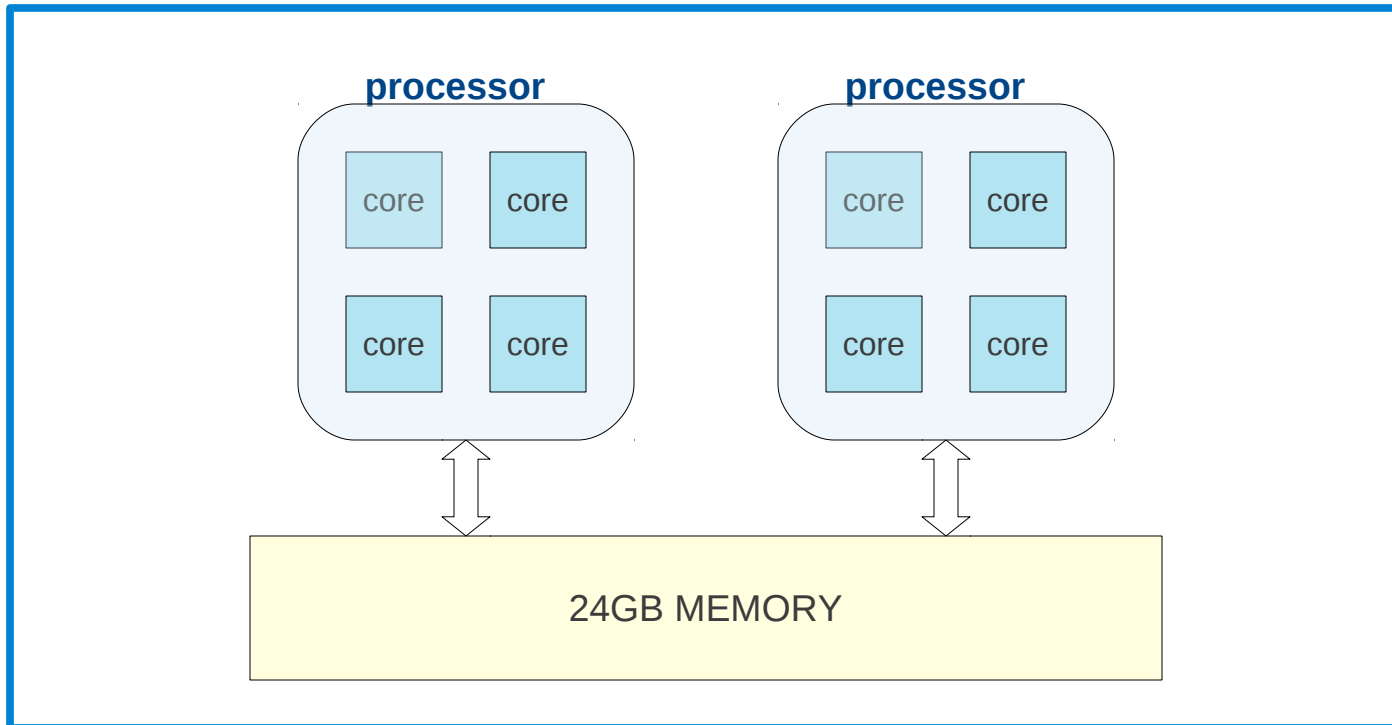
Modern processors have 4 or more  
independent cpu cores to execute instructions



# Basic Architecture (EOS)

An EOS node consists of two processors (4 cpu cores each) and total memory of 24GB

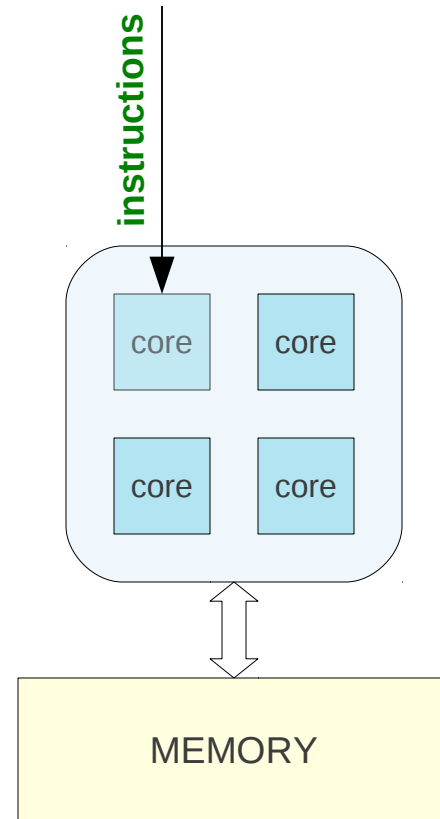
## NODE



# Sequential Program

## When you run sequential program

- Instructions executed on 1 core
- Other cores are idle



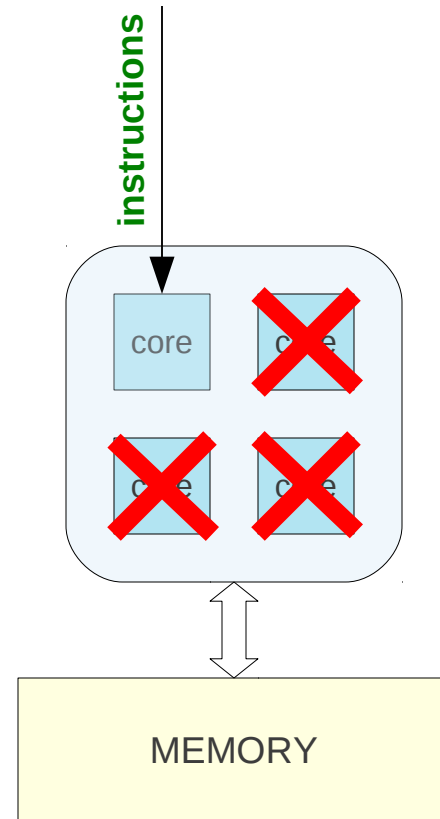
# Sequential Program

## When you run sequential program

- Instructions executed on 1 core
- Other cores are idle

**Waste of available resources. We want all cores to be used to execute program.**

**HOW?**





# What is OpenMP?

Defacto standard API for writing [shared memory](#) parallel applications in C, C++, and Fortran

OpenMP API consists of:

- Compiler Directives
- Runtime subroutines/functions
- Environment variables

# HelloWorld

```
PROGRAM HELLO
!$OMP PARALLEL
PRINT *, "Hello World"
!$ OMP END PARALLEL
STOP
END
```

```
#include <iostream>
#include "omp.h"
int main() {
#pragma omp parallel
{
    std::cout << "Hello World\n"
}
return 0;
}
```

```
intel: ifort -openmp -o hi.x hello.f
pgi: pgfortran -mp -o hi.x hello.f
gnu: gfortran -fopenmp -o hi.x hello.f
```

```
intel: icc -openmp -o hi.x hello.f
pgi: pgcpp -mp -o hi.x hello.f
gnu: g++ -fopenmp -o hi.x hello.f
```

```
Export OMP_NUM_THREADS=4
./hi.x
```

# HelloWorld

```
PROGRAM HELLO
!$OMP PARALLEL
PRINT *, "Hello World"
!$ OMP END PARALLEL
STOP
END
```

```
#include <iostream>
#include "omp.h"
int main() {
#pragma omp parallel
{
    std::cout << "Hello World\n"
}
return 0;
}
```

OMP COMPILER DIRECTIVES

```
intel: ifort -openmp -o hi.x hello.f
pgi:  pgfortran -mp -o hi.x hello.f
gnu:  gfortran -fopenmp -o hi.x hello.f
```

```
intel: icc -openmp -o hi.x hello.f
pgi:  pgcpp -mp -o hi.x hello.f
gnu:  g++ -fopenmp -o hi.x hello.f
```

```
Export OMP_NUM_THREADS=4
./hi.x
```

OMP ENVIRONMENTAL VARIABLE

NOTE: example hello.f90

# Fortran, C/C++

fortran directive format:

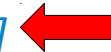
```
!$ OMP PARALLEL [clauses]
:
!$OMP END PARALLEL
```



Will discuss later!

C/C++ directive format is:

```
#pragma omp parallel [clauses]
{
:
}
```

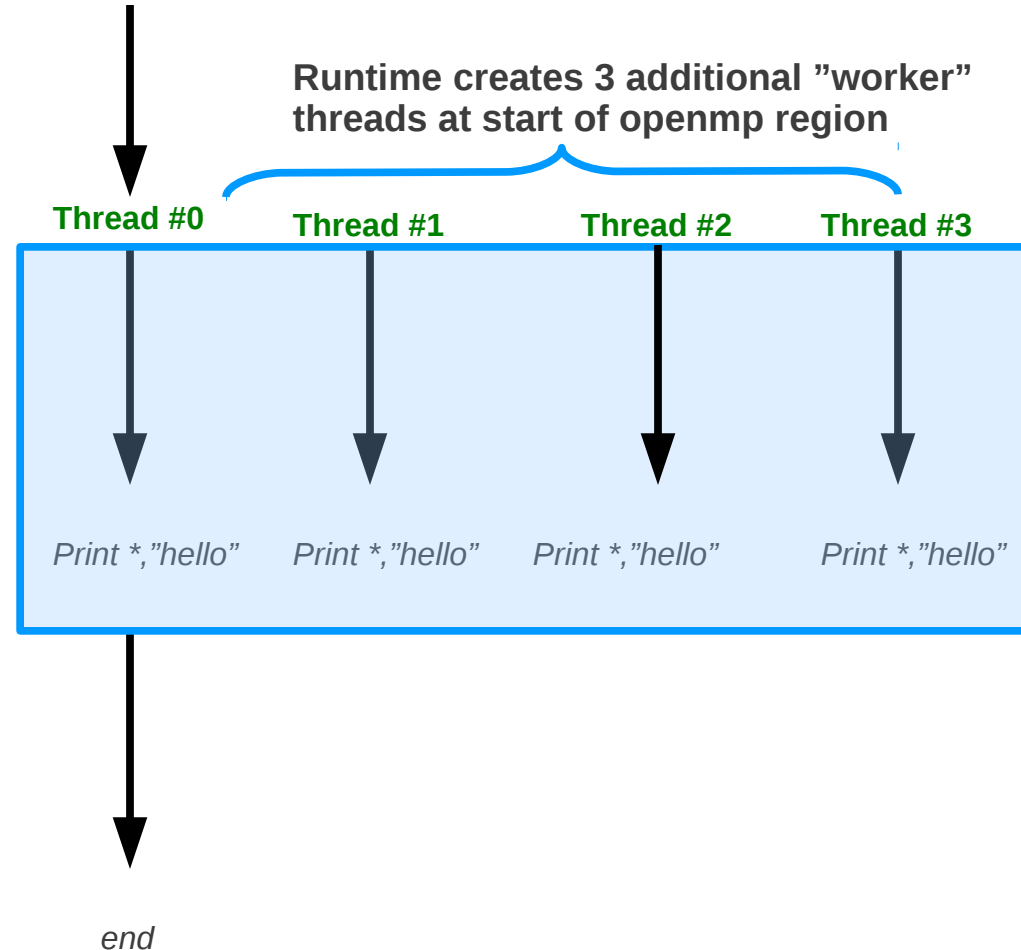


New line required

All OpenMP directives follow this format.

# At run time

Program Hello



OMP region, every thread executes all the instructions in the OMP block

# Fork/Join Model

OpenMP follows the fork/join model:

- ◆ OpenMP programs start with a single thread; the master thread (Thread #0)
- ◆ At start of parallel region master creates team of parallel "worker" threads (FORK)
- ◆ Statements in parallel block are executed in parallel by every thread
- ◆ At end of parallel region, all threads synchronize, and join master thread (JOIN)



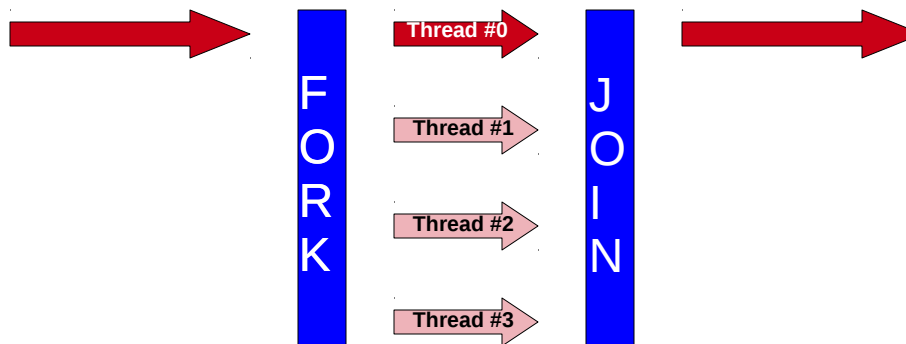
Implicit barrier. Will discuss  
synchronization later

# Fork/Join Model

OpenMP follows the fork/join model:

- ◆ OpenMP programs start with a single thread; the master thread
- ◆ At start of parallel region master creates team of parallel "worker" threads (FORK)
- ◆ Statements in parallel block are executed in parallel by every thread
- ◆ At end of parallel region, all threads synchronize, and join master thread (JOIN)

Implicit barrier. Will discuss  
synchronization later



# OpenMP Threads versus Cores

## What are threads, cores, and how do they relate?

- ◆ Thread is independent sequence of execution of program code
  - ◆ Block of code with one entry and one exit
- ◆ For our purposes a more abstract concept
- ◆ Unrelated to Cores/CPUs
- ◆ OpenMP threads are mapped onto physical cores
- ◆ Possible to map more than 1 thread on a core
- ◆ In practice best to have one-to-one mapping.



# More About OpenMP Threads

Number of openMP threads can be set using:

- ◆ Environmental variable **OMP\_NUM\_THREADS**
- ◆ Runtime function **omp\_set\_num\_threads(n)**

Other useful function to get information about threads:

- ◆ Runtime function **omp\_get\_num\_threads()**
  - ◆ Returns number of threads in parallel region
  - ◆ Returns 1 if called outside parallel region
- ◆ Runtime function **omp\_get\_thread\_num()**
  - ◆ Returns id of thread in team
  - ◆ Value between  $[0, n-1]$  // where  $n = \text{\#threads}$
  - ◆ Master thread always has id 0

# HelloThreads Exercise

Extend the program below to make it parallel where every thread prints out it's id and total number of threads

## FORTRAN

```
PROGRAM HELLOTHREADS
INTEGER THREADS , ID

PRINT *, "NUM THREADS:", THREADS
PRINT *, "hello from thread:",id," out of", threads

STOP
END
```

## C++

```
int main() {
    int threads = 100;
    int id = 100;

    std::cout << "hello from thread: ",id << "out of ";
    std::cout << threads << "\n";

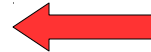
    return 0;
}
```

# Shared Memory Model

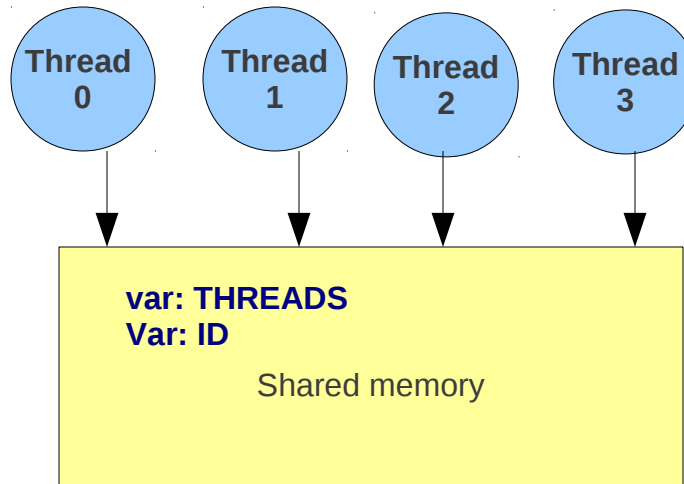
The memory is (logically) shared by all the cpu's

```

$OMP PARALLEL
THREADS = omp_get_num_threads()
ID = omp_get_thread_id()
:
!$ OMP END PARALLEL
  
```



Relevant piece of code  
from our example on  
previous page



*"All threads try to access the same variable (possibly at the same time). This can lead to a race condition. Different runs of same program might give different results because of race conditions"*

# Shared and Private Variables

OpenMP provides a way to declare variables private or shared within an OpenMP block. This is done using the following OpenMP *clauses*:

- ◆ SHARED ( *list* )
  - ◆ All variables in *list* will be considered shared.
  - ◆ Every openmp thread has access to all these variables
- ◆ PRIVATE ( *list* )
  - ◆ Every openmp thread will have it's own "private" copy of variables in list
  - ◆ No other openmp thread has access to this "private" copy

For example: `!$OMP PARALLEL PRIVATE(a,b,c)` (fortran) or  
`#pragma omp parallel private(a,b,c)` (C/C++)

*By default most variables are considered shared in OpenMP. Exceptions include index variables (Fortran, C/C++) and variables declared inside parallel region (C/C++)*

# HelloThreads Exercise (2)

Adapt the program below such that all relevant variables are correctly classified as OpenMP private/shared

## FORTRAN

```
PROGRAM HELLOTHREADS
INTEGER THREADS , ID
!$OMP PARALLEL
threads = omp_get_num_threads()
id = omp_get_thread_num()
PRINT *, "NUM THREADS:", THREADS
PRINT *, "hello from thread:",id," out of", threads
!$OMP PARALLEL END
STOP
END
```

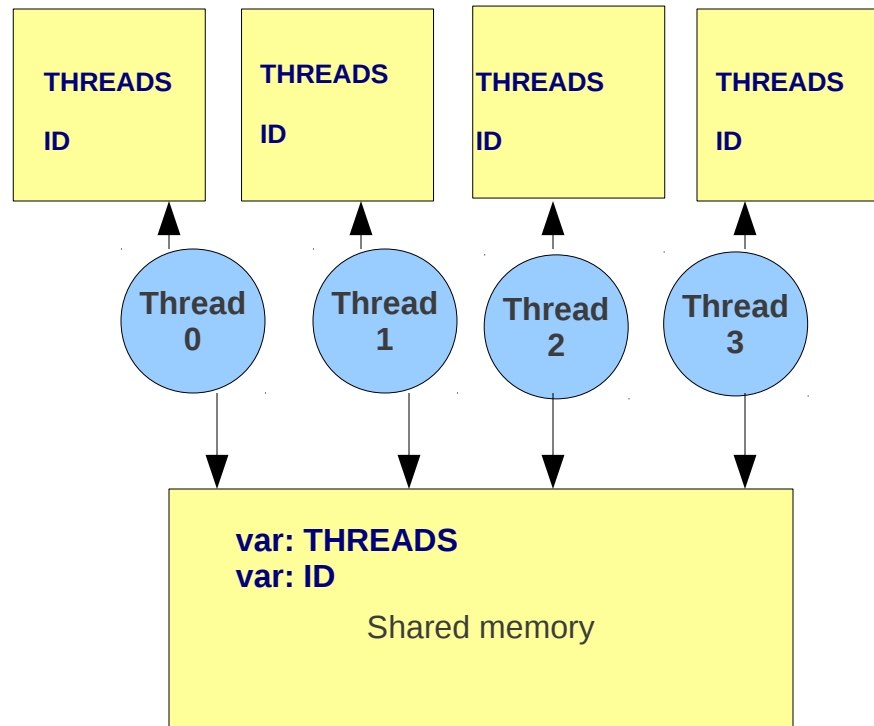
## C++

```
int threads = 100;
int id = 100;
#pragma omp parallel
{
    threads = omp_get_num_threads()
    id = omp_get_thread_num()
    std::cout << "hello from thread: ",id << "out of ";
    std::cout << threads << "\n";
}
return 0;
```

# Revisit Shared Memory Model

The memory is (logically) shared by all the cpu's

**There is also private memory for every openmp thread**



*shared variables "threads" and "id" still exist, but every thread also has a private copy of variables "threads" and "id". There will not be any race condition for these private variables*

# TIP: Practical Issue

- ◆ OpenMP creates separate data stack for every worker thread to store copies of private variables (master thread uses regular stack)
- ◆ Size of these stacks is not defined by OpenMP standards
  - ◆ Intel compiler: default stack is 4MB
  - ◆ gcc/gfortran: default stack is 2MB
- ◆ Behavior of program undefined when stack space exceeded
  - ◆ Although most compilers/RT will throw seg fault
- ◆ To increase stack size use environment var `OMP_STACKSIZE`, e.g.
  - ◆ `export OMP_STACKSIZE=512M`
  - ◆ `export OMP_STACKSIZE=1GB`
- ◆ To make sure master thread has large enough stack space use `ulimit -s` command (unix/linux).

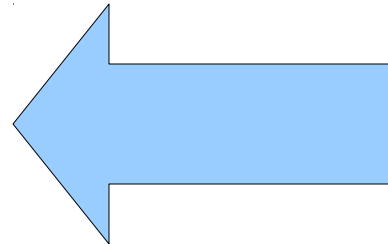
# Work Sharing (manual approach)

So far only discussed parallel regions that all did same work. Not very useful. What we want is to share work among all threads so we can solve our problems faster.

```

:
!$OMP PARALLEL PRIVATE(n,num,id,f,l)
id = omp_get_thread_num()
num = omp_get_num_threads()
f = id*(N/num)+1
l = (id+1)*(N/num)
DO n=f,l,1
  A(n) = A(n) + B
ENDDO
!$OMP END PARALLEL
:

```



Partition the iteration space manually, every thread computes  $N/\text{num}$  iterations

**Can be cumbersome and error prone!**



# Work Sharing

Suppose:  $N=100$  and  $\text{num}=4 \rightarrow N/\text{num}=25$

```
f = id*(N/num)+1  
l = (id+1)*(N/num)  
DO n=f,l,1  
    A(n) = A(n) + B  
ENDDO
```

<u>Thread 0</u>	<u>Thread 1</u>	<u>Thread 2</u>	<u>Thread 3</u>
$f=0*25+1 = 1$ $l=1*25 = 25$	$f=1*25+1=26$ $l=2*25 = 50$	$f=2*25+1=51$ $l=3*25 = 75$	$f=3*25+1=76$ $l=4*25 = 100$

Thread 0 computes elements from index 1 to 25, Thread 1 computes from index 26 to 50, etc.

The above implementation makes one big assumption. What is it?

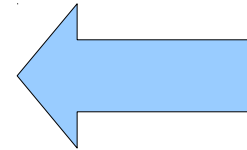
# Work Sharing (openmp approach)

## FORTRAN

```
!$OMP PARALLEL
!$OMP DO
DO I=1,100
  A(I) = A(I) + B
ENDDO
!$OMP END DO
!$ OMP END PARALLEL
```

## C++

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i<100;++i) {
    A(I) = A(I) + B
  }
}
```



OpenMP takes care of partitioning the iteration space for you. The only thing needed is to add the !\$OMP DO and !\$OMP END DO directives

*"even more compact by combining omp parallel/do directives"*

## FORTRAN

```
!$OMP PARALLEL DO
DO I=1,100
  A(I) = A(I) + B
ENDDO
!$ OMP END PARALLEL DO
```

## C++

```
#pragma omp parallel for
for (i=0;i<100;++i) {
  A(I) = A(I) + B
}
```

# Exercise

Create a program that computes a simple matrix vector multiplication  $b=Ax$ , either in fortran or C/C++. Use OpenMP directives to make it run in parallel.

# Reductions

A common type of computation is something like:

```
DO i=1,10
  a = a op expr
ENDDO
```

```
for (int i=0;i<10;++i) {
  a = a op expr
}
```

**a = a op expr** is called a reduction operation. Obviously, there is a loop carried flow dependence (will discuss later) for variable 'a' which prohibits parallelization.

For these kind of cases OpenMP provides the **REDUCTION(op:list)** clause. This clause can be applied when the following restrictions are met:

- **a** is a scalar variable in the list
- **expr** is a scalar expression that does not reference **a**
- Only certain kind of **op** allowed; e.g. +,\*,-
- For fortran **op** can also be intrinsic; e.g. **MAX,MIN,IOR**
- Vars in list have to be shared

# Exercise

Create a program that computes the sum of all the elements in an array A (in fortran or C/C++) or a program that finds the largest number in an array A (Fortran) . Use OpenMP directives to make it run in parallel.

# TIP: OpenMP Scoping Rules

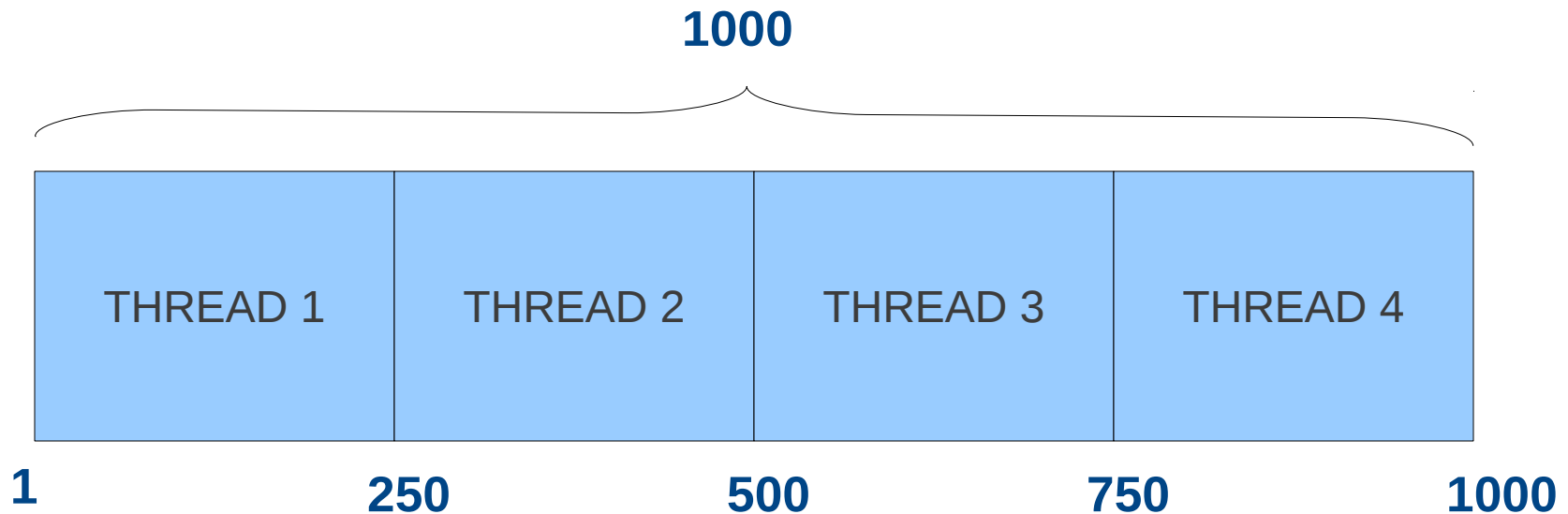
So far we have all the directives nested within the same Routine (!\$OMP PARALLEL outer most). However, OpenMP provides more flexible scoping rules. E.g. It is allowed to have routine with only !\$OMP DO In this case we call the !\$OMP DO an orphaned directive.

***Note: There are some rules (e.g. When an !\$OMP DO directive is encountered program should be in parallel section)***

# How OMP schedules iterations?

Although the OpenMP standard does not specify how a loop should be partitioned most compilers split the loop in  $N/p$  ( $N$  #iterations,  $p$  #threads) chunks by default. This is called a **static schedule** (with chunk size  $N/p$ )

*For example, suppose we have a loop with 1000 iterations and 4 omp threads. The loop is partitioned as follows:*

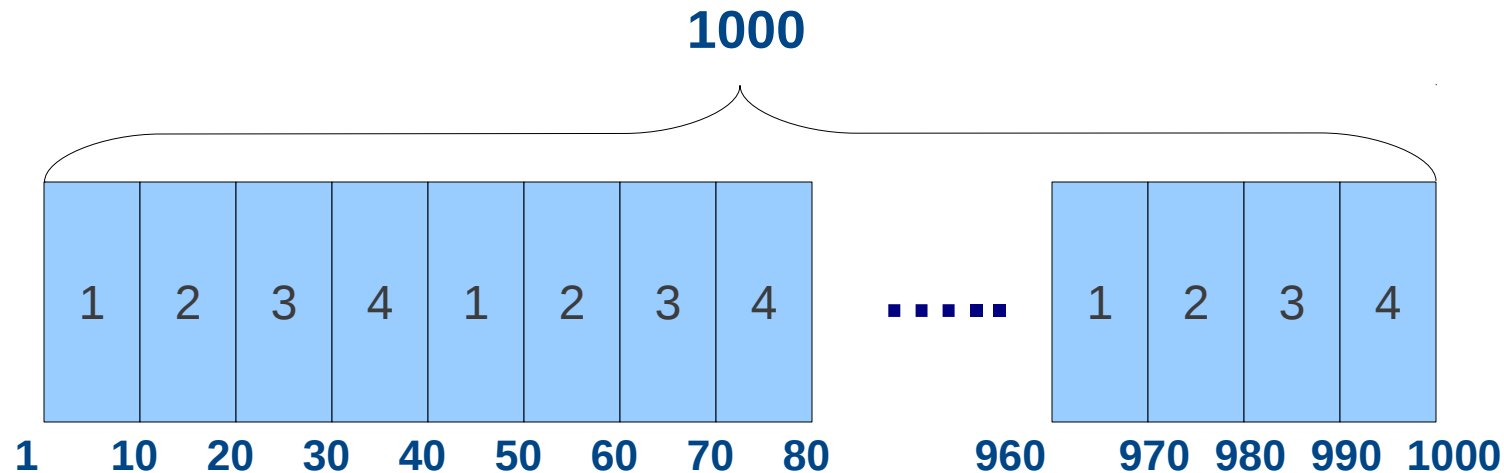


# Static Schedule

To explicitly tell the compiler to use a static schedule (or a different schedule as we will see later) OpenMP provides the SCHEDULE clause

**\$!OMP DO SCHEDULE (STATIC,n)** // (n is chunk size)

*For example, suppose we set the chunk size n=10*



NOTE: example static\_schedule, vary chunk size

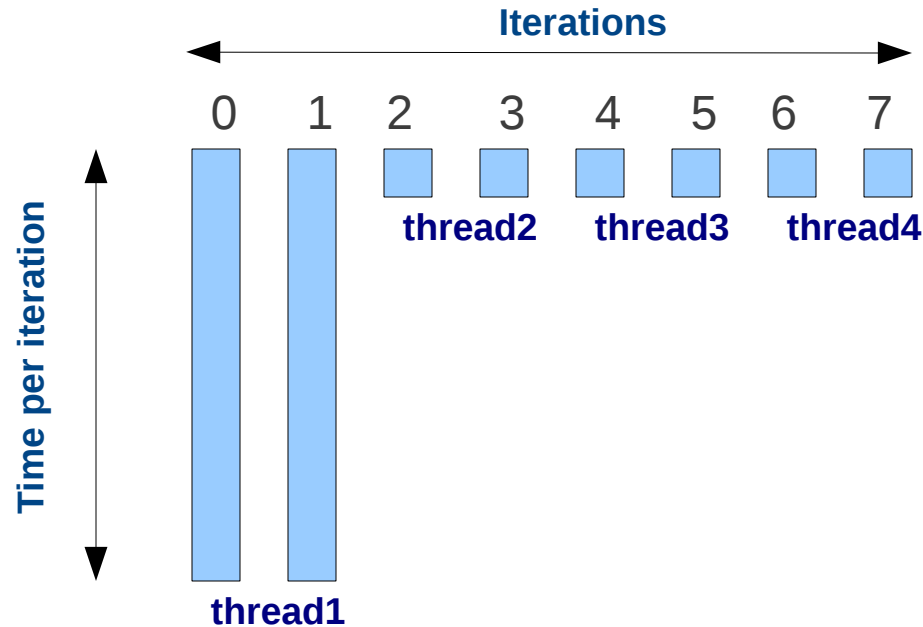


# Issues with Static schedule

With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations). This is not always the best way to partition. Why is This?

# Issues with Static schedule

With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations). This is not always the best way to partition. Why is This?



*This is called load imbalance. In this case threads 2,3, and 4 will be waiting very long for thread 1 to finish*

**How can this happen?**

# Dynamic Schedule

With a dynamic schedule new chunks are assigned to threads when they come available. OpenMP provides two dynamic schedules:

- ◆ `$!OMP DO SCHEDULE(DYNAMIC,n)` // n is chunk size  
Loop iterations are divided into pieces of size chunk. When a thread finishes one chunk, it is dynamically assigned another.
- ◆ `$!OMP DO SCHEDULE(GUIDED,n)` // n is chunk size  
Similar to DYNAMIC but chunk size is relative to number of iterations left.

*Keep in mind: although Dynamic scheduling might be the preferred choice to prevent load imbalance in some situations, there is a significant overhead involved compared to static scheduling.*

# OMP SINGLE

Another work sharing directive (although it doesn't really share work) OpenMP provides is **!\$OMP SINGLE**. When encountering a single directive only one member of the team will execute the code in the block

- ◆ One thread (not necessarily master) executes the block
- ◆ Other threads will wait
- ◆ Useful for thread-unsafe code
- ◆ Useful for I/O operations

We will use this directive in a later exercise

# Data Dependencies

Not all loops can be parallelized. Before adding OpenMP directives need to check for any dependencies:

We categorize three types of dependencies:

- ◆ Flow dependence: Read after Write (RAW)
- ◆ Anti dependence: Write after Read (WAR)
- ◆ Output dependence (Write after Write (WAW))

## FLOW

*X = 21*  
*PRINT \*, X*

## ANTI

*PRINT \*, X*  
*X = 21*

## OUTPUT

*X = 21*  
*X = 21*

# Data Dependencies (2)

*For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)*

Let's find the dependencies in the following loop?

```

S1: DO I=1,10
S2:   B(i) = temp
S3:   A(i+1) = B(i+1)
S4:   temp = A(i)
S5: ENDDO
    
```

# Data Dependencies (2)

*For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)*

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:  B(i) = temp
S3:  A(i+1) = B(i+1)
S4:  temp = A(i)
S5: ENDDO
```




1: S3 → S2 anti (B)

# Data Dependencies (2)

*For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)*

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:  B(i) = temp
S3:  A(i+1) = B(i+1)
S4:  temp = A(i)
S5: ENDDO
```



1: S3 → S2 anti (B) 2: S3 → S4 flow (A)
--

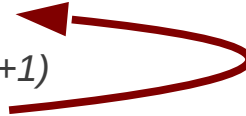


# Data Dependencies (2)

*For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)*

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:  B(i) = temp
S3:  A(i+1) = B(i+1)
S4:  temp = A(i)
S5: ENDDO
```



- 1: S3 → S2 anti (B)
- 2: S3 → S4 flow (A)
- 3: S4 → S2 flow (temp)

# Data Dependencies (2)

*For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)*

What are the dependencies in the following loop?

```

S1: DO I=1,10
S2:  B(i) = temp
S3:  A(i+1) = B(i+1)
S4:  temp = A(i) ←
S5: ENDDO
  
```

- 1: S3 → S2 anti (B)
- 2: S3 → S4 flow (A)
- 3: S4 → S2 flow (temp)
- 4: S4 → S4 output (temp)

# Data Dependencies (2)

*For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)*

What are the dependencies in the following loop?

```

S1: DO I=1,10
S2:   B(i) = temp
S3:   A(i+1) = B(i+1)
S4:   temp = A(i)
S5: ENDDO
  
```

1: S3 → S2 anti (B) 2: S3 → S4 flow (A) 3: S4 → S2 flow (temp) 4: S4 → S4 output (temp)
--

*Sometimes it helps to "unroll" part of the loop to see loop carried dependencies more clear*

```

S2: B(1) = temp
S3: A(2) = B(2)
S4: temp = A(1)
  
```

```

S2: B(2) = temp
S3: A(3) = B(3)
S4: temp = A(2)
  
```

# Data Dependencies (2)

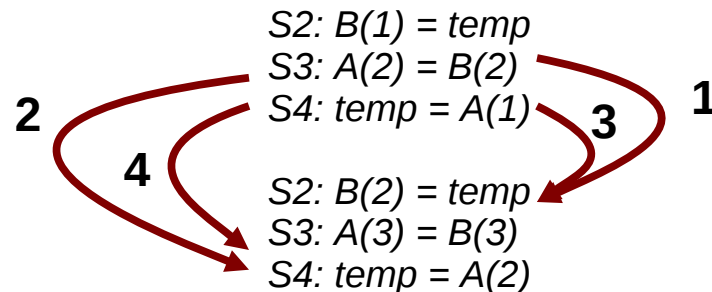
*For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)*

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:   B(i) = temp
S3:   A(i+1) = B(i+1)
S4:   temp = A(i)
S5: ENDDO
```

- |                          |
|--------------------------|
| 1: S3 → S2 anti (B)      |
| 2: S3 → S4 flow (A)      |
| 3: S4 → S2 flow (temp)   |
| 4: S4 → S4 output (temp) |

Sometimes it helps to "unroll" part of the loop to see loop carried dependencies more clear



# Case Study: Jacobi

Implement a parallel version of the Jacobi algorithm using OpenMP. A sequential version is provided.

# Data Dependencies (3)

Loop carried anti- and output dependencies are not true dependencies (re-use of the same name) and in many cases can be resolved relatively easily.

Flow dependencies are true dependencies (there is a flow from definition to its use) and in many cases cannot be removed easily. Might require rewriting the algorithm (if possible)

# Resolving Anti/Output Deps

## Use PRIVATE clause:

Already saw this in example hello\_threads

## Rename variables (if possible):

Example: in-place left shift

```
DO i=1,n-1
  A(i)=A(i+1)  →  ANEW(i) = A(i+1)  →
ENDDO          ENDDO
```

```
!$OMP PARALLEL DO
DO i=1,n-1
  ANEW(i) = A(i+1)
ENDDO
!$OMP END PARALLEL DO
```

If has to be in-place can do it in two steps:

```
!$OMP PARALLEL
!$OMP DO
  T(i) = A(i+1)
!$OMP END DO
!$OMP DO
  A(i) = T(i)
!$OMP END DO
!$OMP END PARALLEL
```

# More about shared/private vars

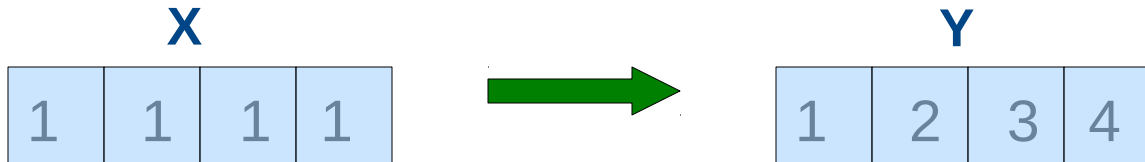
Besides the clauses described before OpenMP provides some additional datascope clauses that are very useful:

- ◆ **FIRSTPRIVATE ( *list* ):**  
Same as PRIVATE but every private copy of variable 'x' will be initialized with the original value (before the omp region started) of 'x'
- ◆ **LASTPRIVATE ( *list* ):**  
Same as PRIVATE but the private copies of the variables in list from the last work sharing will be copied to shared version. To be used with **!\$OMP DO** Directive.
- ◆ **DEFAULT (SHARED | PRIVATE | FIRSTPRIVATE | LASTPRIVATE ):**  
Specifies the default scope for all variables in omp region.



# Case Study: Removing Flow Deps

$$Y = \text{prefix}(X) \rightarrow Y(1) = X(1); Y(i) = Y(i-1) + X(i)$$

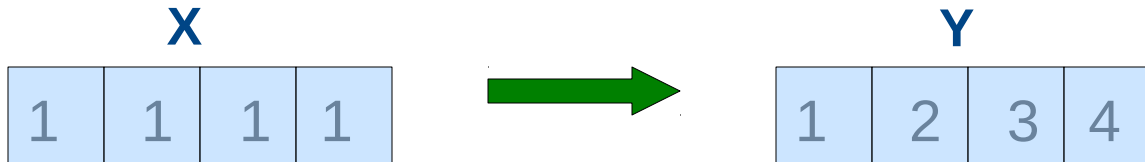


## SEQUENTIAL

```
Y[1] = X[1]
DO i=2,n,1
  Y[i] = Y[i-1] + X[i]
ENDDO
```

# Case Study: Removing Flow Deps

$Y = \text{prefix}(X) \rightarrow Y(1) = X(1); Y(i) = Y(i-1) + X(i)$



## SEQUENTIAL

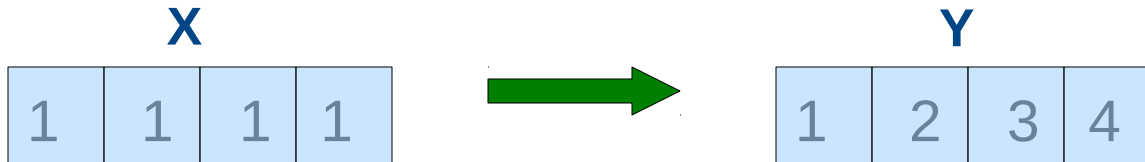
```
Y[1] = X[1]
DO i=2,n,1
  Y[i] = Y[i-1] + X[i]
ENDDO
```

## PARALLEL

```
Y[1] = X[1]
!$OMP PARALLEL DO
DO i=2,n,1
  Y[i] = Y[i-1] + X[i]
ENDDO
!$OMP END PARALLEL DO
```

# Case Study: Removing Flow Deps

$Y = \text{prefix}(X) \rightarrow Y(1) = X(1); Y(i) = Y(i-1) + X(i)$



## SEQUENTIAL

```
Y[1] = X[1]
DO i=2,n,1
  Y[i] = Y[i-1] + X[i]
ENDDO
```

## PARALLEL

```
Y[1] = X[1]
!$OMP PARALLEL DO
DO i=2,n,1
  Y[i] = Y[i-1] + X[i]
ENDDO
!$OMP END PARALLEL DO
```

**WHY?**

# Case Study: Removing Flow Deps

## REWRITE ALGORITHM

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

STEP 1: split X among threads; every thread computes its own (partial) prefix sum

1	2	3	4
---	---	---	---

1	2	3	4
---	---	---	---

1	2	3	4
---	---	---	---

1	2	3	4
---	---	---	---

STEP 2: create array T  $\rightarrow T[1]=0$ ,  $T[i] = X[(\text{length}/\text{threads}) * (i-1)]$ , perform simple prefix sum on T  
(will collect last element from every thread (except last) and perform simple prefix sum)

0	4	4	4
---	---	---	---

 $\rightarrow$ 

0	4	8	12
---	---	---	----

STEP 3: every thread adds T[threadid] to all its element

+0			
1	2	3	4

+4			
5	6	7	8

+8			
9	10	11	12

+12			
13	14	15	16

STEP 4: Finished; we rewrote prefix sum by removing dependencies.

# Prefix Sum Implementation

How to implement the algorithm on the previous slide?

- Three separate steps
- Steps 1 and 3 can be done in parallel
- Step 2 has to be done sequential
- Step 1 has to be performed before step 2
- Step 2 has to be performed before step 3

**NOTE: For illustration purposes we can assume array length is multiple of #threads**

NOTE: exercise prefix

# Case Study: Removing Flow Deps

This Case study showed an example of an algorithm with real (flow) dependencies

- Sometimes we can rewrite algorithm to run parallel
- Most of the time this is not trivial
- Speedup much less impressive (often)

# OpenMP Sections

Suppose you have blocks of code that can be executed in parallel (i.e. No dependencies). To execute them in parallel OpenMP provides the **!\$OMP SECTIONS** directive. The syntax will look something like:

```
!$OMP PARALLEL
!$OMP SECTIONS
```

```
!$OMP SECTION
// WORK 1
!$OMP SECTION
// WORK 2
```

```
!$OMP END SECTIONS
!$OMP END PARALLEL
```

Combined version



```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
// WORK 1
!$OMP SECTION
// WORK 2
!$OMP END PARALLEL SECTIONS
```

This will execute "WORK 1" and "WORK 2" in parallel

# Exercise

Create a program that computes the adjusted prefix sum below for 4 different arrays and after that adds all of them up. Use OpenMP directives to make it run in parallel.

$$\begin{aligned}A(1) &= A(1) \\ A(2) &= A(2) + A(1) \\ A(i) &= A(i-2) + A(i-1)\end{aligned}$$



# NOWAIT Clause

Whenever a thread in a work sharing construct (e.g. **!\$OMP DO** ) finishes its work faster than other threads it will wait until all threads participating have finished their respective work. All threads will synchronize at end of work sharing construct

For situations where we don't need or want to synchronize at the end OpenMP provides the **NOWAIT** clause

Fortran

```
!$OMP DO  
:  
!$OMP END DO NOWAIT
```

C/C++

```
#pragma omp do nowait  
{  
:  
}
```

NOTE: example nowait

# Work Sharing Summary

OMP work sharing constructions we discussed

- ◆ **!\$OMP DO**
- ◆ **!\$OMP SECTIONS**
- ◆ **!\$OMP SINGLE**

Useful clauses that can be used with these constructs ( **incomplete list**  
and not all clauses can be used with every directive)

- ◆ **SHARED (list)**
- ◆ **PRIVATE (list)**
- ◆ **FIRSTPRIVATE (list)**
- ◆ **LASTPRIVATE(list)**
- ◆ **SCHEDULE (STATIC | DYNAMIC | GUIDED, chunk)**
- ◆ **REDUCTION(op:list)**
- ◆ **NOWAIT**

# Synchronization

OpenMP programs use shared variables to communicate. We need to make sure these variables are not accessed at the same time by different threads (will cause race conditions, [WHY?](#)). OpenMP provides a number of directives for synchronization.

- ◆ !\$OMP MASTER
- ◆ !\$OMP CRITICAL
- ◆ !\$OMP ATOMIC
- ◆ !\$OMP BARRIER

# !\$OMP MASTER

This Directive ensures that only the master threads executes instructions in the block. There is no implicit barrier so other threads will not wait for master to finish

# !\$OMP MASTER

This Directive ensures that only the master threads executes instructions in the block. There is no implicit barrier so other threads will not wait for master to finish

***What is difference with !\$OMP SINGLE DIRECTIVE?***

# !\$OMP CRITICAL

This Directive makes sure that only one thread can execute the code in the block. If another threads reaches the critical section it will wait untill the current thread finishes this critical section. Every thread will execute the critical block and they will synchronize at end of critical section

- ◆ Introduces overhead
- ◆ Serializes critical block
- ◆ If time in critical block relatively large → speedup negligible

# Exercise

In the REDUCTION exercise we created a program that computes the sum of all the elements in an array A. Create another program that does the same, without using the REDUCE clause. Compare the two versions.

# !\$OMP ATOMIC

This Directive is very similar to the `!$OMP CRITICAL` directive on the previous slide. Difference is that `!$OMP ATOMIC` is only used for the update of a memory location. Sometimes `!$OMP ATOMIC` is also referred to as a mini critical section.

- ◆ Block consists of only one statement
- ◆ Atomic statement must follow specific syntax



# !\$OMP ATOMIC

This Directive is very similar to the `!$OMP CRITICAL` directive on the previous slide. Difference is that `!$OMP ATOMIC` is only used for the update of a memory location. Sometimes `!$OMP ATOMIC` is also referred to as a mini critical section.

- ◆ Block consists of only one statement
- ◆ Atomic statement must follow specific syntax

*"Can replace "**critical**" with "**atomic**" in previous example"*

# !\$OMP BARRIER

!\$OMP BARRIER will enforce every thread to wait at the barrier until all threads have reached the barrier. !\$OMP BARRIER is probably the most well known synchronization mechanism; explicitly or implicitly. The following omp directives we discussed before include an implicit barrier:

- ◆ !\$ OMP END PARALLEL
- ◆ !\$ OMP END DO
- ◆ !\$ OMP END SECTIONS
- ◆ !\$ OMP END SINGLE
- ◆ !\$ OMP END CRITICAL

# Potential Speedup

Ideally, we would like to have perfect speedup (i.e. speedup of  $N$  when using  $N$  processors). However, this is not really feasible in most cases for the following reasons:

- ◆ Not all execution time is spent in parallel regions (e.g. loops)
  - ◆ e.g. not all loops are parallel (data dependencies)
- ◆ There is an inherent overhead with using openmp threads

*Let's look at an example that shows how speedup will be affected because of non parallel parts of a program*

# TIP: IF Clause

OpenMP provides another useful clause to decide at run time if a parallel region should actually be run in parallel (multiple threads) or just by the master thread:

IF (logical expr)

For example:

<code>\$!OMP PARALLEL IF(n &gt; 100000)</code>	(fortran)
<code>#pragma omp parallel if (n&gt;100000)</code>	(C/C++)

This will only run the parallel region when  $n > 100000$

# Amdahl's Law

Every program consists of two parts:

- ◆ Sequential part
- ◆ Parallel part

Obviously, no matter how many processors, the sequential part will always be executed by exactly one processor. Suppose, program spends  $p$  ( $0 < p < 1$ ) part of the time in parallel region. running time (relative to sequential) will be:

$$\frac{(1-p) + \frac{p}{N}}{1}$$

This means that maximum speedup will be:

$$\frac{1}{(1-p) + \frac{p}{N}}$$

*e.g. Suppose 80% of program can be run in parallel and we have unlimited #cpus, maximum speedup will still be only 5*

# OpenMP overhead/Scalability

Starting a parallel OpenMP region does not come free. There is considerable overhead involved. Consider the following before placing openmp pragmas around a loop:

- ◆ Remember Amdahl's law
- ◆ Parallelize most outer loop possible (in some cases even if less iterations)
- ◆ Make sure speedup in parallel region enough to overcome overhead
  - ◆ Is number of iterations in loop large enough?
  - ◆ Is amount of work per iteration enough?
- ◆ Overhead can be different on different machine/OS/compiler

OpenMP programs don't always scale well. I.e. When number of openmp threads increases speedup will suffer

- ◆ More threads competing for available bandwidth
- ◆ Cache issues

# Nested Parallelism

OpenMP allows nested parallelism (e.g. `!$OMP DO` within `!$OMP DO`)

- ◆ env var `OMP_NESTED` to enable/disable
- ◆ `omp_get_nested()`, `omp_set_nested()` runtime functions
- ◆ Compiler can still choose to serialize nested parallel region (i.e. use team with only one thread)
- ◆ Significant overhead. **WHY?**
- ◆ Can cause additional load imbalance. **WHY?**

# OpenMP Loop collapsing

When you have perfectly nested loops you can collapse inner loops using the OpenMP clause `collapse(n)`. Collapsing a loop:

- ◆ Loop needs to be perfectly nested
- ◆ Loop needs to have rectangular iteration space
- ◆ Makes iteration space larger
- ◆ Less synchronization needed than nested parallel loops

Example:

```

!$OMP PARALLEL DO PRIVATE (i,j) COLLAPSE(2)
DO i = 1,2
  DO j=1,2
    call foo(A,i,j)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
  
```

NOTE: example collapse



# TIP: Number of Threads Used

OpenMP has two modes to determine how many threads will actually be used in parallel region:

- ◆ DYNAMIC MODE
  - ◆ Number of threads used can vary per parallel region
  - ◆ Setting number of threads only sets max number of threads (actual number might be less)
- ◆ STATIC MODE
  - ◆ Number of threads is fixed and determined by programmer
- ◆ Environmental Variable `OMP_DYNAMIC` to set mode
- ◆ Runtime functions `omp_get_dynamic/omp_set_dynamic`

# Math Libraries

Math libraries have very specialized and optimized version of many functions and many of them have been parallelized using OpenMP. On EOS we have the Intel Math Kernel Library (MKL)

For more information about MKL:

<http://sc.tamu.edu/help/eos/mathlib.php>

So, before implementing your own OpenMP Math function, check if there already is a version in MKL