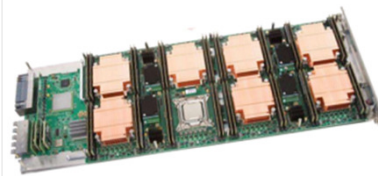


OpenMP

Michael Bane
HIGH END COMPUTE



SHARED MEMORY

OPENMP IN PRACTICE

OPENMP FORK-JOIN

REGIONS & WORK DISTRIBUTION

GOTCHYA?

TOP TIPS

UPDATES

1. Threads .v. hyperthreading

- Hyperthreading is physical: set by BIOS, whether the operating sees 2 (not 1) logical cores for each physical core.
- Threads: lightweight processes running. Typically put one thread per CPU core.
- For x86, best to have HT turned off, and 1 thread per physical core.
- For GPU, different story. Threading is good and used to mask high cost of memory access – the GPU cores are designed specifically to cope with lots of threads
- For MIC, leave their HT turned on, and best performance may be 2 or 3 or 4 threads per *physical* core

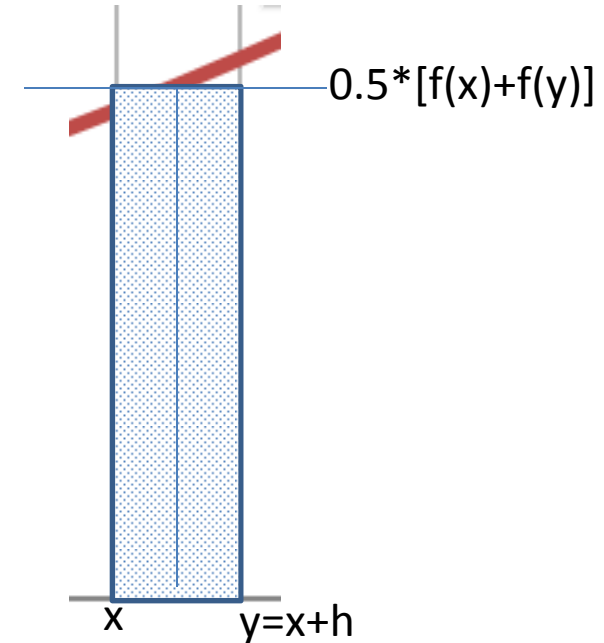
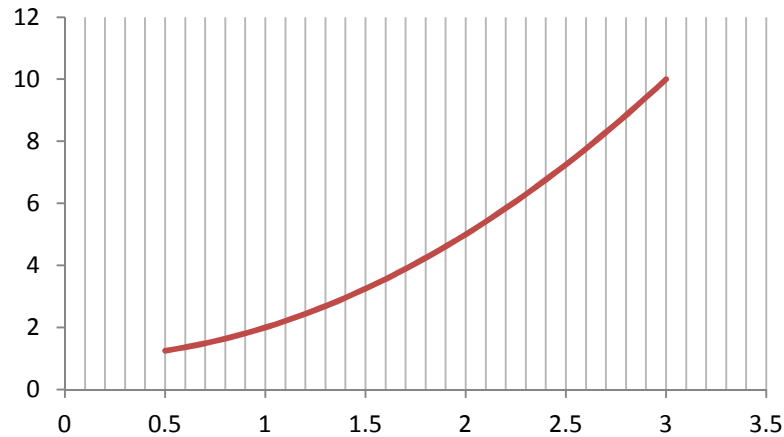
OpenMP

Dr. Michael K. Bane

HIGH END COMPUTE

OPENMP IN PRACTICE

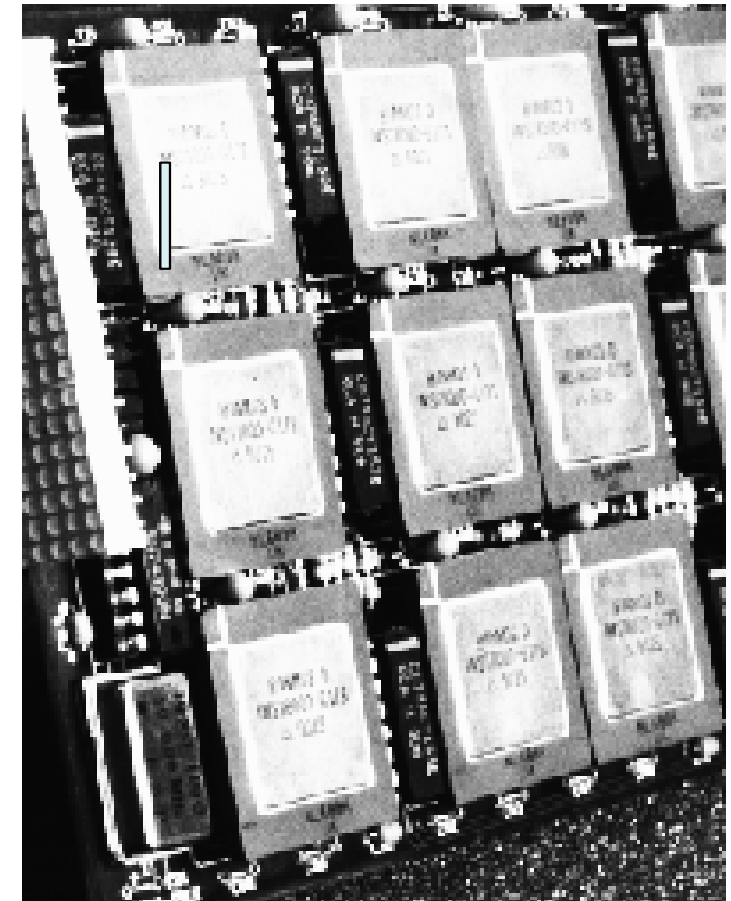
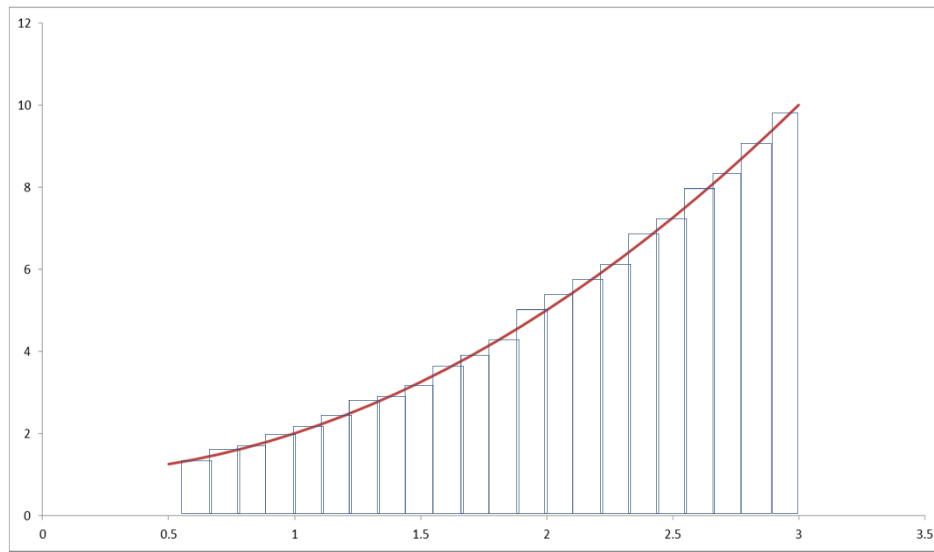
Quadrature



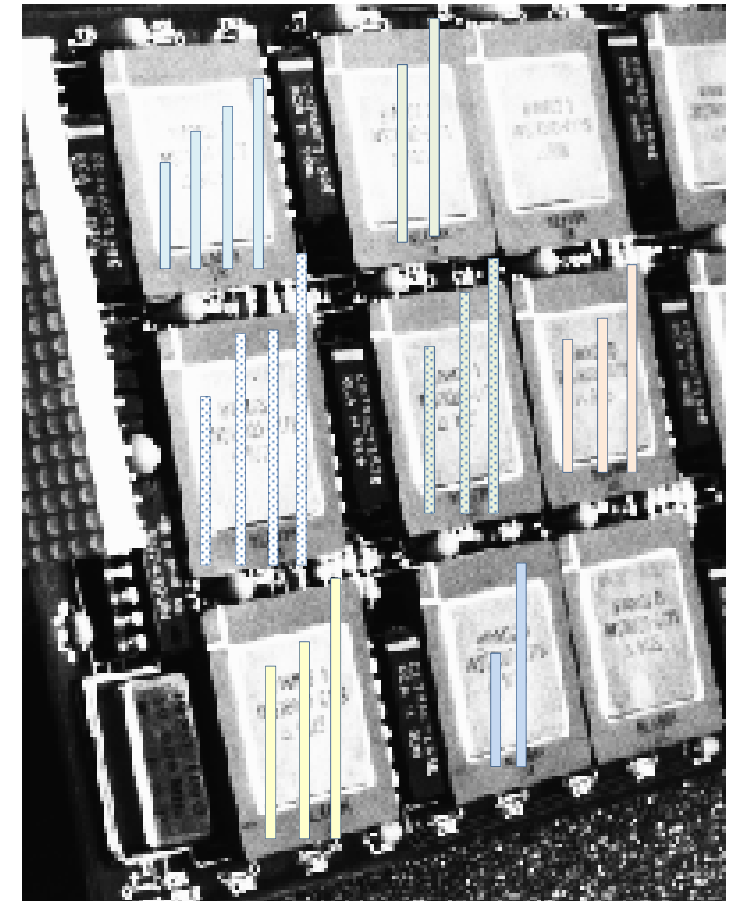
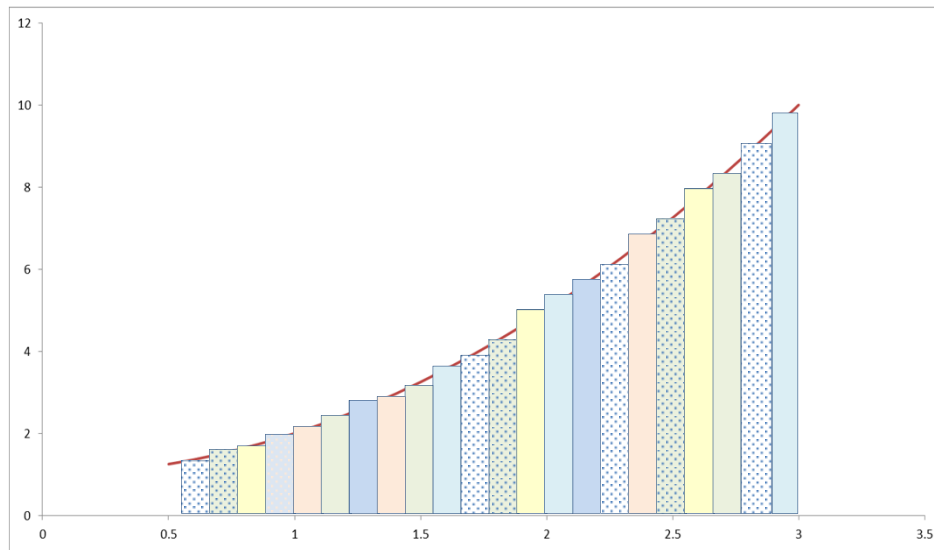
- Approximate integral is sum of areas under line
- Each area approximated by a rectangle
- Can we calculate these in parallel?

Hands on

- Quad on Archer CPU
 - (rather than KNL we saw earlier)
- Show serial
- OMP times for 1,2,4, ...
 - (login v batch)
 - (ifort v ftn)
- **An OpenMP thread runs on a processor core**



```
do i=1, numberQuads
  x = a + (i-1)*width
  y = x + width
  meanHeight = 0.5*(func(x)+func(y))
  integrand = integrand + meanHeight*width
end do
```



```

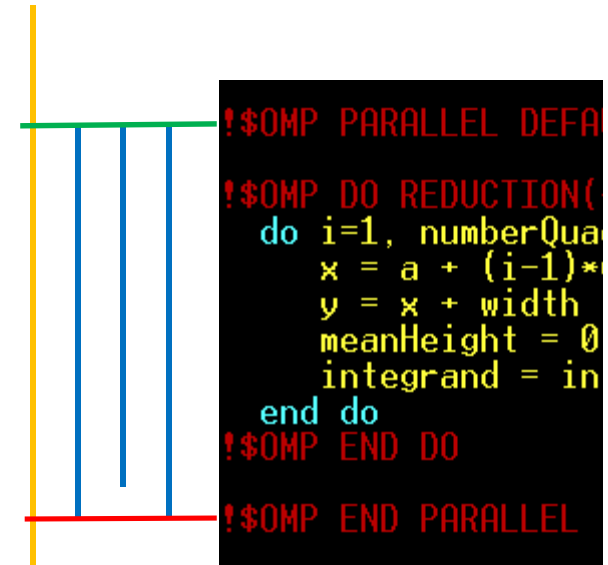
!$OMP PARALLEL DEFAULT(NONE) SHARED(width) PRIV
!$OMP DO REDUCTION(+:integrand)
  do i=1, numberQuads
    x = a + (i-1)*width
    y = x + width
    meanHeight = 0.5*(func(x)+func(y))
    integrand = integrand + meanHeight*width
  end do
!$OMP END DO
!$OMP END PARALLEL

```


OPENMP MAIN PRINCIPLES

Recap: fork-join, parallel regions, team of threads

- Master thread – lives forever
- Worker threads – short lived, in a parallel region
- Fork at start of a parallel region
 - Creation of new threads
 - Possible creation of new memory locations
- Join at end of a parallel region
- Can have many parallel regions (with differing number of threads in the team)



Threads & Regions

- More than 1 par region
 - Could have differing number of threads in each par region
- Also need to think efficiency
 - Coarse grained usually best
 - Set-up and close/sync costs
- Threads:
 - Master #0
 - Workers: 1, 2, ..., $\{OMP_NUM_THREADS\}-1$

This region has 7 threads.
`omp_get_num_threads()`
will return 7 from inside
par region

0 1 2 3 4 5 6

OPENMP SYNTAX

OpenMP

- We are **not** going to attempt to cover all the syntax, nor the edge cases, nor all the gotchyas...
- Syntax of each & everything, with interpretation
<http://openmp.org/wp/openmp-specifications/>
- We *will* cover the key concepts (with examples), focussed on FORTRAN

OpenMP: the three ingredients

- Directives
 - Tell the compiler to create assembler for thread creation (etc) and concerning work placement within the team of threads
- Run time functions
 - Allow running code to make run time decisions. Typically how to distribute the work load based on input data size and number of threads available
- Environment variables
 - Control how a compiled code will run, eg set the maximum number of threads

OpenMP Directives

`!$OMP <directive> <clauses>`

- `<directive>` is ~command
- `<clauses>` are ~how to implement command
 - Typically on scope of data and control of thread & work placement
- Comment to non-OpenMP compliant compilers
- Interpreted by OpenMP compliant compilers

Syntax for Directives

`!$OMP <directive> <clauses>`

- No whitespace within "`!$OMP`"
- At least one space after the *sentinel*
- First string on a line
- Cannot mix OMP directives and other FORTRAN
- FORTRAN, so case insensitive
- Continuation
 - Line to be continued: end with `&`
 - Continuation line: start with either `!$OMP` or `!$OMP&`

Setting up a Parallel Region

- `!$OMP PARALLEL <clauses>`
... code
`!$OMP END PARALLEL`
- Code will be replicated
 - Single source code but may be different paths and different "evaluations"

```
!$OMP PARALLEL
```

```
    start = workSize * omp_get_thread_num()
```

```
    do j = start, workSize
```

```
        y(j) = y(j) + func(x(j))
```

```
    end do
```

```
!$OMP END PARALLEL
```

! At this point we will have updated

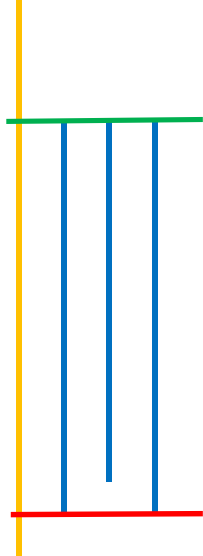
! elements $0, 1, \dots, Z$ of Y

! where Z depends on the number of threads

Sharing the Work

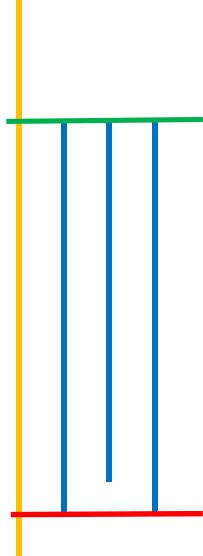
- Once we have a parallel region
 - Threads created
- Constructs to share work
 - Divide the work over the threads
 - Exist within a parallel region
- Common
 - !\$OMP DO Most common! (as per examples)
 - !\$OMP SECTIONS
 - !\$OMP WORKSHARE

OMP DO

A diagram on the left side of the slide shows a vertical yellow line representing a master thread. To its right, three vertical blue lines represent worker threads. A green horizontal line at the top and a red horizontal line at the bottom connect the master thread to the worker threads, indicating the start and end of the parallel region.

```
!$OMP PARALLEL DEFAULT(NONE) SHARED(width) PRIVATE(integrand)
!$OMP DO REDUCTION(+:integrand)
  do i=1, numberQuads
    x = a + (i-1)*width
    y = x + width
    meanHeight = 0.5*(func(x)+func(y))
    integrand = integrand + meanHeight*width
  end do
!$OMP END DO
!$OMP END PARALLEL
```

- distributes the iterations of the following DO loop to threads
 - doesn't create a loop
 - no need to write an extra FORTRAN loop re threads
- Remember that the code worked before adding OpenMP do no need for extra loops – we are just describing *who* does the work



```
!$OMP PARALLEL DEFAULT(NONE) SHARED(width) PRIV
!$OMP DO REDUCTION(+:integrand)
  do i=1, numberQuads
    x = a + (i-1)*width
    y = x + width
    meanHeight = 0.5*(func(x)+func(y))
    integrand = integrand + meanHeight*width
  end do
!$OMP END DO
!$OMP END PARALLEL
```

- The "REDUCTION" is a data clause – see later
- We can optionally use NOWAIT clause at end:
!\$OMP END DO NOWAIT
which **removes a synchronisation** and lets threads
continue with next statement without waiting for
every other statement (**can be dangerous**)

How much parallelism

- We can control the number of threads:

- Env var

```
export OMP_NUM_THREADS=12
```

- Directives

```
!$OMP PARALLEL num_threads (N)
```

- Run time library call

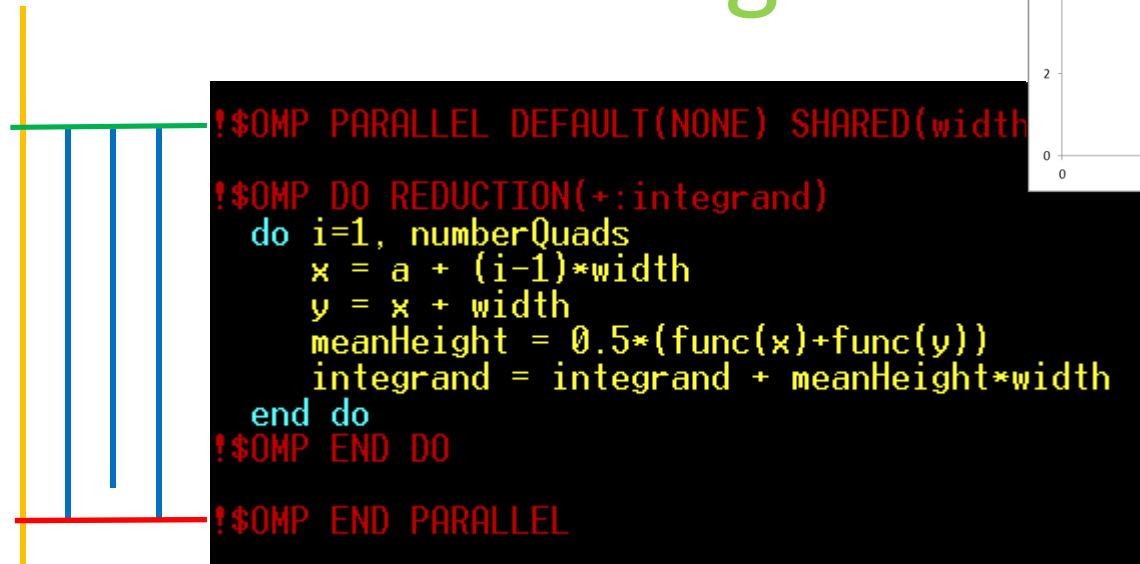
(effect varies where is called)

```
call omp_set_num_threads (N)
```

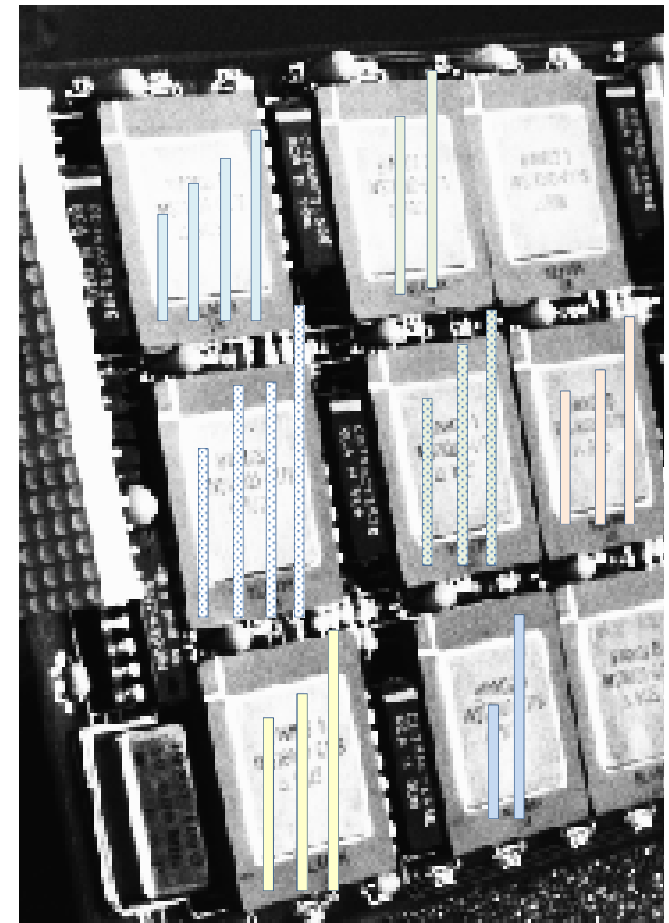
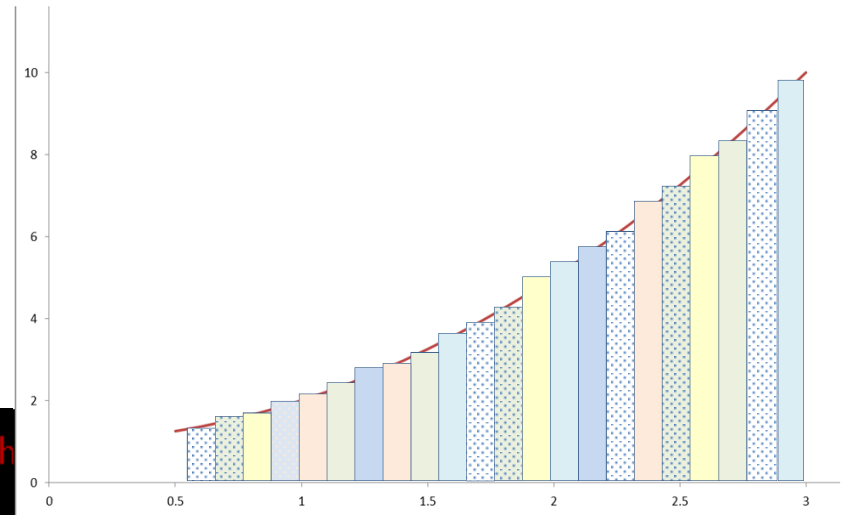
- Use N threads for the parallel region

(best to have $N \leq \text{\$OMP_NUM_THREADS}$)

Replication .v. Work Sharing

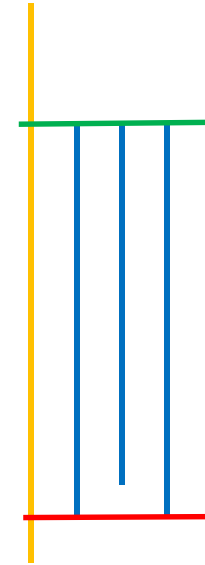


- OMP PARALLEL: creates threads
- OMP DO: divides up the work
- So on N threads that part (should) go N times quicker



Replication .v. Work Sharing

- OMP PARALLEL creates threads
- Everything in the parallel region is done by every thread
- So if no extra directive*
- to split up the work
- Then all N threads do everything so might only go 1 times quicker
- * Or alternative (programmable) method



UNLIMITED PARALLELISM

The sky is the limit?

```
init()  
for timesteps {  
    update_my_cell()  
    update_globals()  
    output_current_state()  
}  
fini()
```

	1 core	2 cores	5 cores	500 cores
Time for SEQ /seconds	120			
Time for PAR / seconds	500	250	100	1

	1 core	2 cores	5 cores	500 cores
Total /seconds	620	370	220	121

	1 core	2 cores	5 cores	500 cores
Speed-up	1	1.67	2.81	5.12

Speed up: How much faster on p cores than on 1 core: $S_p = T_1/T_p$

Efficiency: How close to ideal speed-up on p cores: $E_p = S_p/p$

Amdahl's Law

- Alpha : serial proportion of original code
- $T_p = \alpha * T_1 + (1-\alpha) * T_1 / p$
- $S_p = T_1 / T_p$
- Thus $S_p = 1 / (\alpha + (1-\alpha)/p)$
- Speed-up (and max speed-up) only dependent on the proportion of code that is serial
- Max speed-up ($p \rightarrow \infty$): is $1/\alpha$

EXERCISE

- Determine alpha for the above example
 - And thus max speed-up

	1 core
Time for SEQ /seconds	120
Time for PAR /seconds	500

- Time seq: 120 seconds
- Time par: 500 seconds
- $\text{Alpha} = 120/620 = 19\%$
- Max speed up= 5.17

So why 100K machines?

- A1: consider a code that spends just 1% of its time doing non-parallelisable work.
 - What is the maximum speed-up? $1/0.01 = 100$
 - How many cores do we need to achieve 99% of this maximum? $99*(1-0.01) / .01 = 9801$
- A2: we can live with a few seconds or minutes of serial but want to take what took days for the parallel part to be done in seconds or minutes

Flavours of Scaling

- Reducing the time for a *given* problem by increasing the number of cores
 - strong scaling
 - Amdahl's Law
- BUT we also interested in using more cores so we can run *bigger problems* but *in the same time*
 - weak scaling
 - Gustafson's Law: $S'(p) = \alpha + (1-\alpha)*p$

What is Missing?

(other than inherently serial logic part of code...)

The cost (overheads) of implementing the parallelism

- fork (creating, particularly PRIVATE vars)
- join (synchronisation)
- locks
- comms
- load imbalance

Time to Try!

- Two exercises
- Exercise002 "func"
 - To learn about replicated and work sharing
- Exercise003 "amdahl"
 - To learn about effect on scaling of serial code

But of "clause"..

- ... it's not quite so simple!
- Race conditions
- Data "sharing"

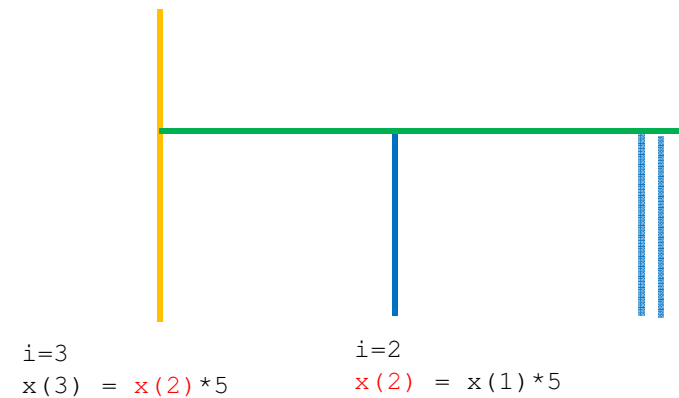
```
!$OMP PARALLEL DO
    DO I=2, N    ! WHERE N IS >2
        TMP = F(I)
        X(IND(I)) = X(IND(I-1)) * F(I)
    END DO
!$OMP END PARALLEL DO
```

Dependencies

WARNING

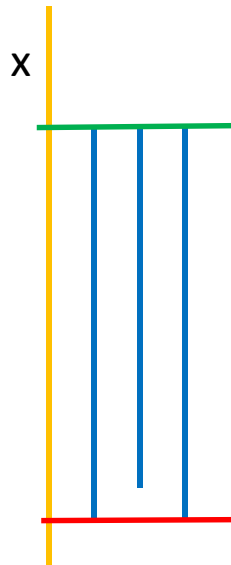
- OpenMP directives will do what you tell them to
- Even if it's wrong!

```
!$OMP  PARALLEL DO
      DO I=2, 10
          X(I) = X(I-1) * 5
      END DO
!$OMP  PARALLEL DO
```



Is this the "old" value of $x(2)$
or the value updated on
another thread?

SCOPE OF VARIABLES

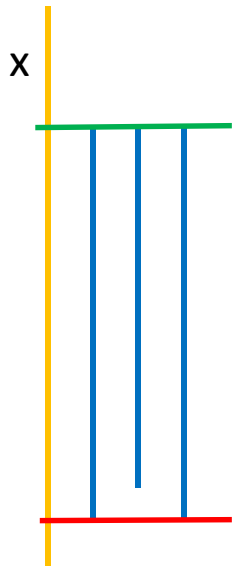


X could be a scalar
or an array
(only scalar shown for
ease)



SHARED (global)

- Everybody reads the value
- Nobody updates the variable
- Only need 1 physical memory local

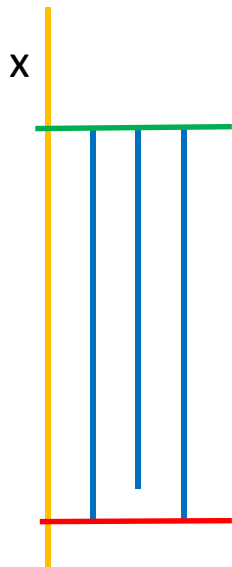


SHARED (x)

– "global" memory



PRIVATE (local)



- Each thread wants to update the variable but only for its own use
- Need a physical memory location for each thread
 - Set up at the entry to the parallel region

PRIVATE (**x**)

- "local" memory
- Does not carry value of x in to par reg
- Nor of local values back to master thread at end of parallel region



Group Exercise: sort data clauses

```
! Example that needs data clauses
INTEGER:: X(100), Y(100)
READ(*,*) N, Y
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP&          SHARED(what does here?), PRIVATE(what goes here?)
TMP = 1.5
!$OMP DO
DO I=2, N
    TMP2 = X(1) + Y(I)
    Y(I) = 10.0 * Y(I)
    NEWX(I) = TMP2*X(I) + TMP1*Y(I)
END DO
!$OMP DO
DO I=2, N
    X(I) = NEWX(I)
END DO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

TMP:
TMP2:
I:
N:

X
Y
NEWX

```

! Example that needs data clauses
INTEGER:: X(100), Y(100)
READ(*,*) N, Y
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP&          SHARED(TMP, N, X, Y, NEWX), PRIVATE(TMP2, I)
TMP = 1.5
!$OMP DO
DO I=2, N
    TMP2 = X(1) + Y(I)
    Y(I) = 10.0 * Y(I)
    NEWX(I) = TMP2*X(I) + TMP1*Y(I)
END DO
!$OMP DO
DO I=2, N
    X(I) = NEWX(I)
END DO
!$OMP END DO NOWAIT
!$OMP END PARALLEL

```


Patterns

- Reduction
 - Every thread finds global max (or min) of local data
 - Every thread wants global sum (or multiple) of local data
 - **QUESTION:** why not global division?
Or global subtraction?
- REDUCTION operator
 - Which we have already seen...

Example

- Global sum of local

```
!$OMP PARALLEL SHARED(X) PRIVATE(MYSUM)
MYSUM = 0.0
!$OMP DO
DO I=1, N
    MYSUM = MYSUM + X(I)
END DO
! *BUT MYSUM IS LOCAL SO HOW SHARE?*
```

- But we've seen how better to do this

Example

- Global sum of local

```
!$OMP PARALLEL SHARED(X, MYSUM)
```

```
THREAD = OMP_GET_NUM_THREAD()
```

```
MYSUM(THREAD) = 0.0
```

```
!$OMP DO
```

```
DO I=1, N
```

```
    MYSUM(THREAD) = MYSUM(THREAD) + X(I)
```

```
END DO
```

```
!* MYSUM ARRAY IS SHARED BUT WRITE PATTERNS MAY BE BAD
```

- But we've seen how better to do this

```

width = (b-a)/float(numberQuads)
integrand = 0.0

!$OMP PARALLEL DEFAULT(NONE) SHARED(width) PRIVATE(x,y,meanHeight) SHARED(integrand)
!$OMP DO REDUCTION(+:integrand)
  do i=1, numberQuads
    x = a + (i-1)*width
    y = x + width
    meanHeight = 0.5*(func(x)+func(y))
    integrand = integrand + meanHeight*width
  end do
!$OMP END DO

!$OMP END PARALLEL

  write(*,*) 'Approx integrand:', integrand
!  write(*,*) 'Exact integrand:', sol(b)-sol(a)

```

- OMP REDUCTION CLAUSE
 - REDUCTION(*oper* : *varList*)
 - Variable/s in *varList* do not need to be defined as SHARED or PRIVATE, just in REDUCTION
 - OMP/OS takes care of the rest

Do I care how I do Reduction?

- (Yes: order of summation may matter)

TIPS

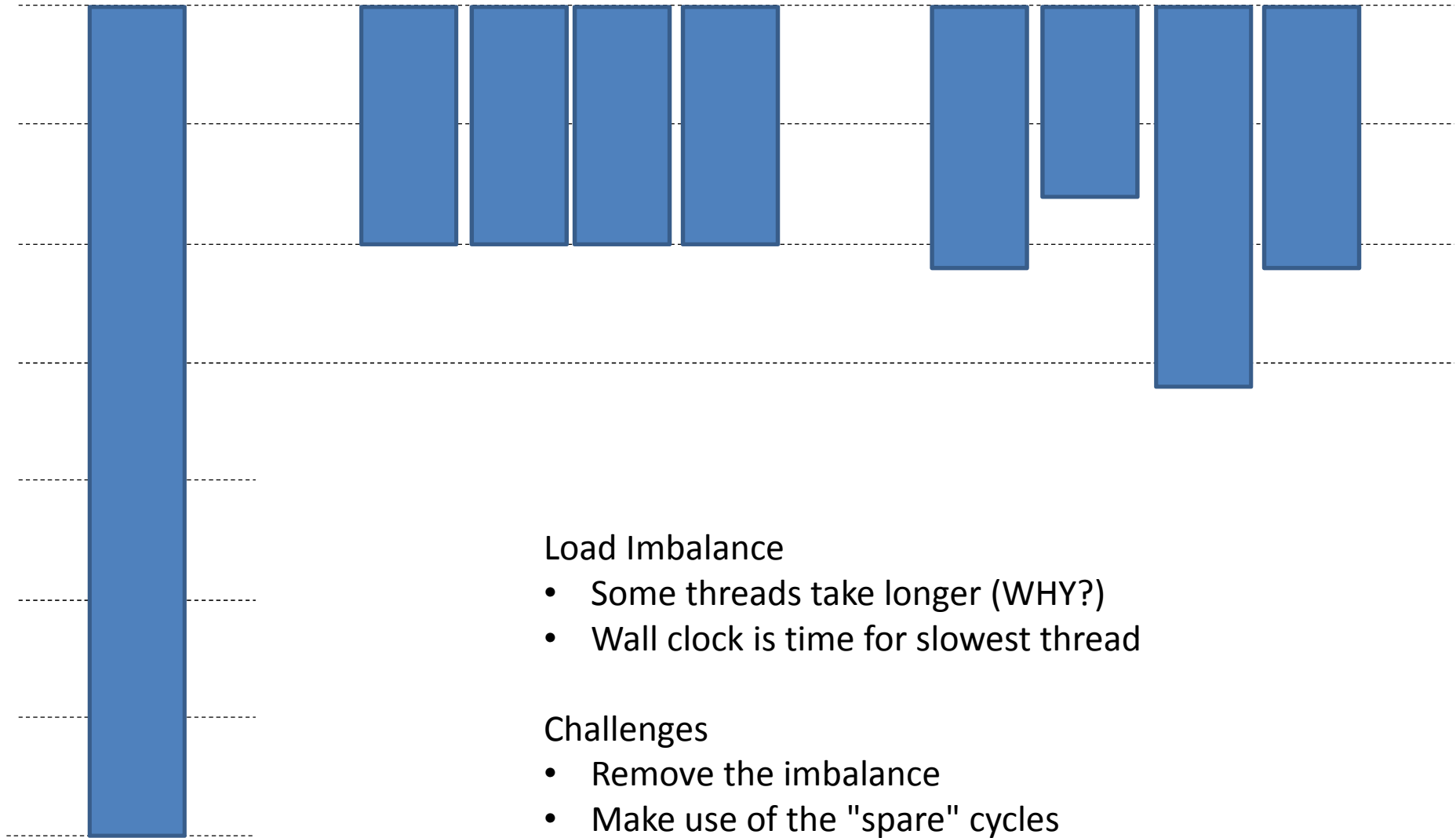
- Think very carefully whether variable is being updated by more than 1 thread
- Write it out on paper: unroll parallelised loops to thread timelines
- **Threads are not lock step nor can you presume which iteration goes where or the order they occur**
- Unit test on varying number of threads, including 1 thread, odd numbers
 - Check results! Look at timings – perhaps profile too

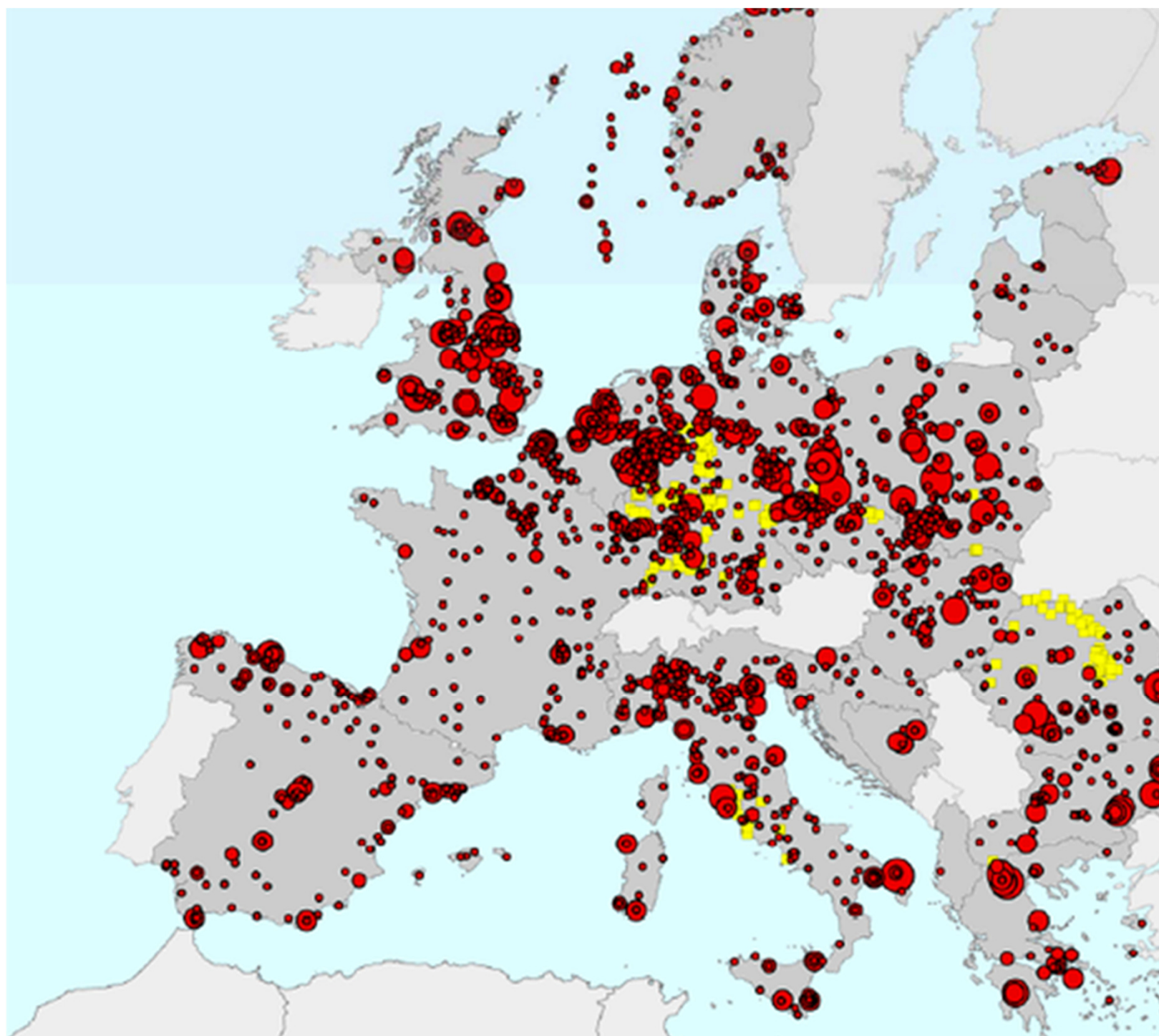
Time to Try!

- A more substantial exercise
- Exercise004 "advection"
 - To take a serial code and determine how to use OpenMP directives yourself to parallelise
 - (explain – see practicals.pptx)

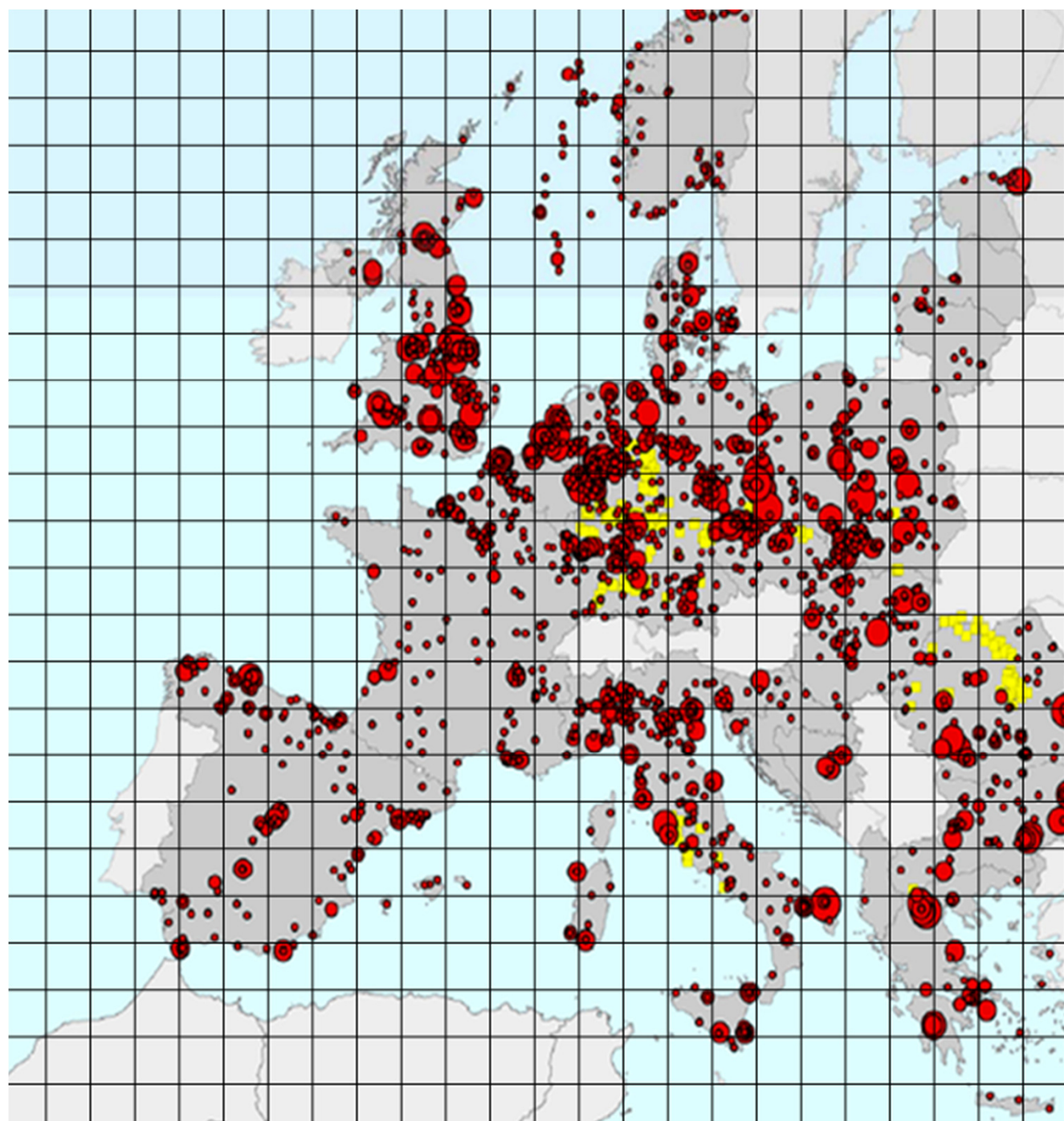
TUNING OPENMP

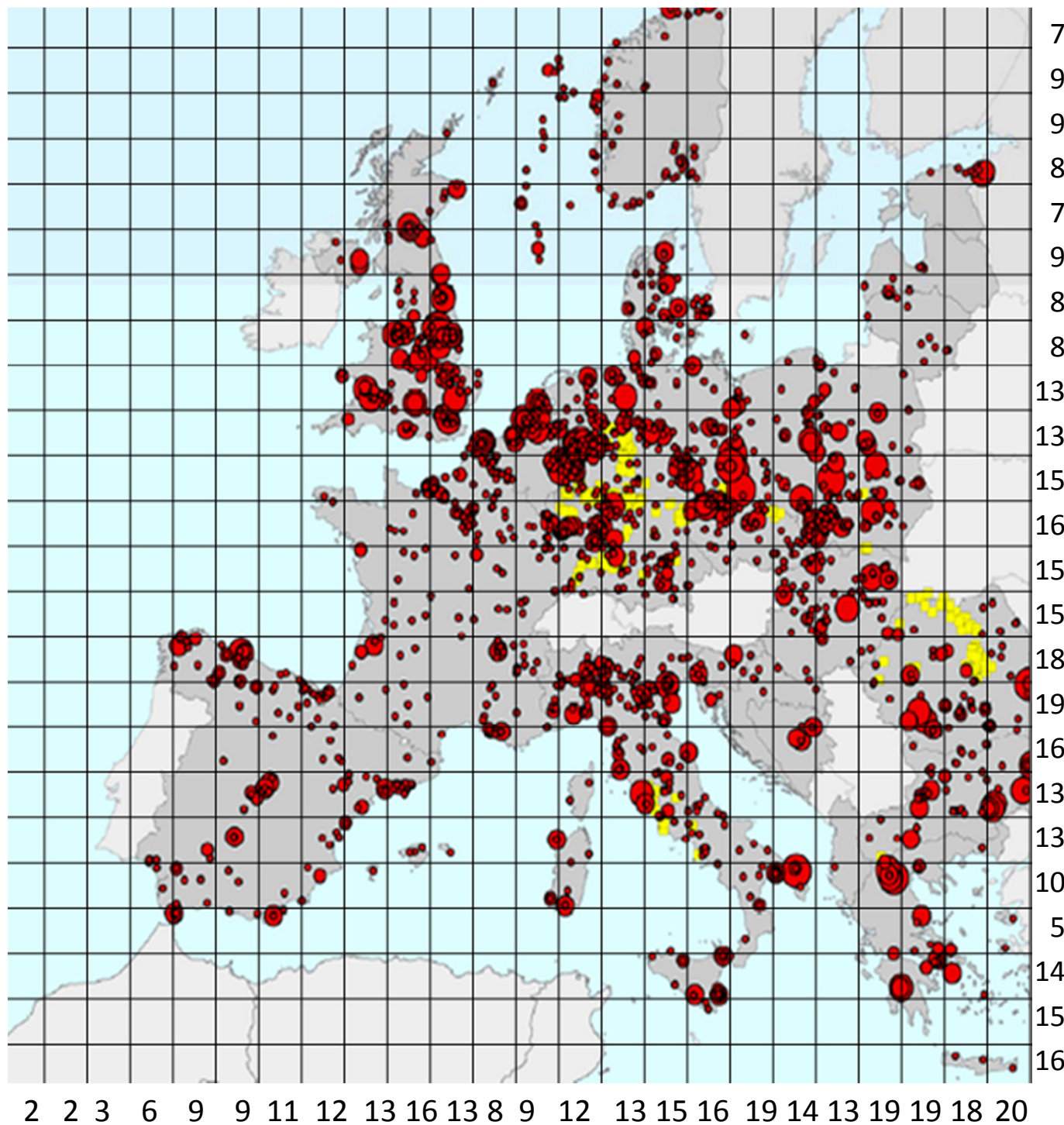
Imperfect Parallelism











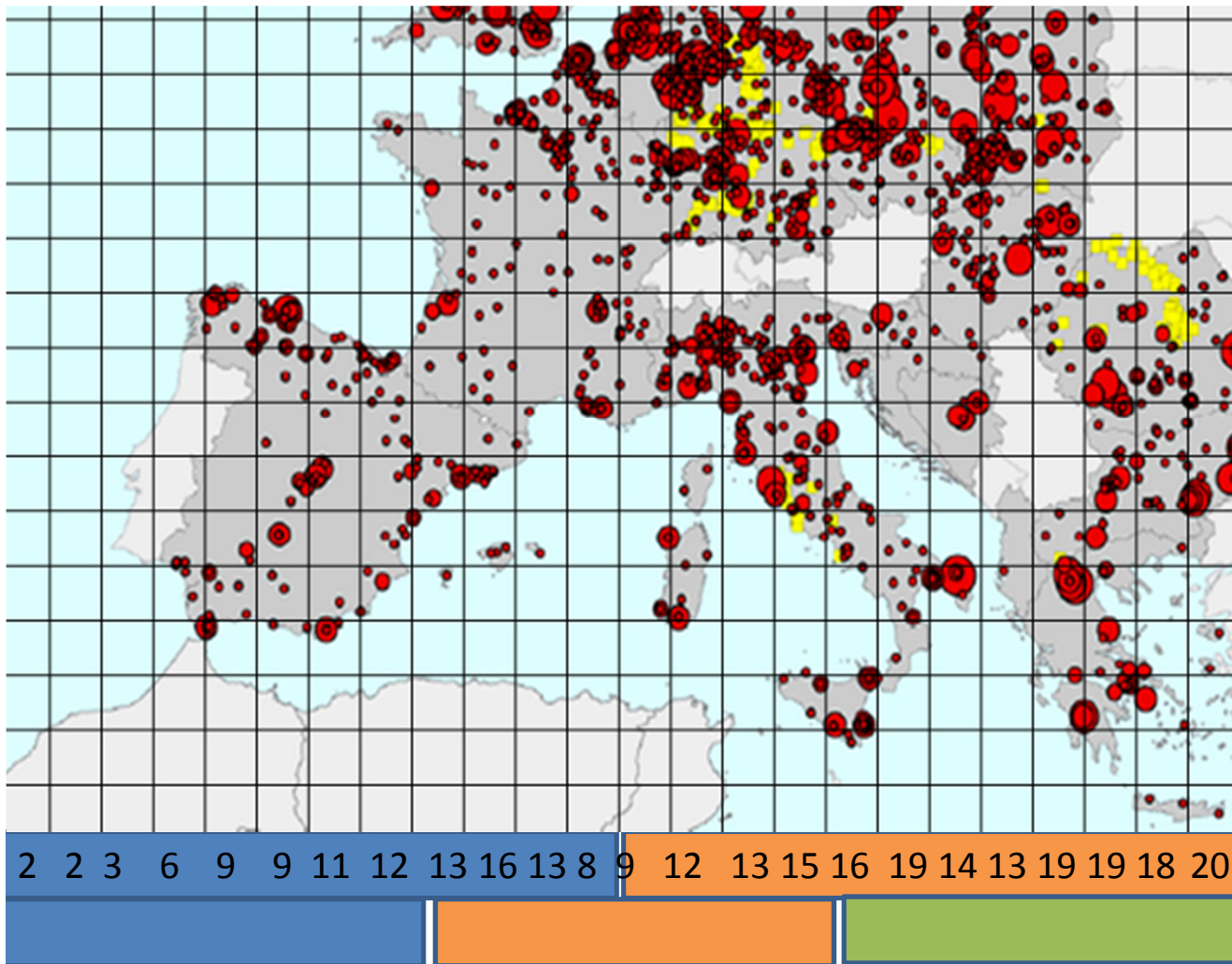
How would you
share out this
work equally?

Re-balancing

- How to control how the iterations are distributed to threads?
- SCHEDULE clause
 - Controls placement of DO iterations on to threads

SCHEDULE(type, chunk)

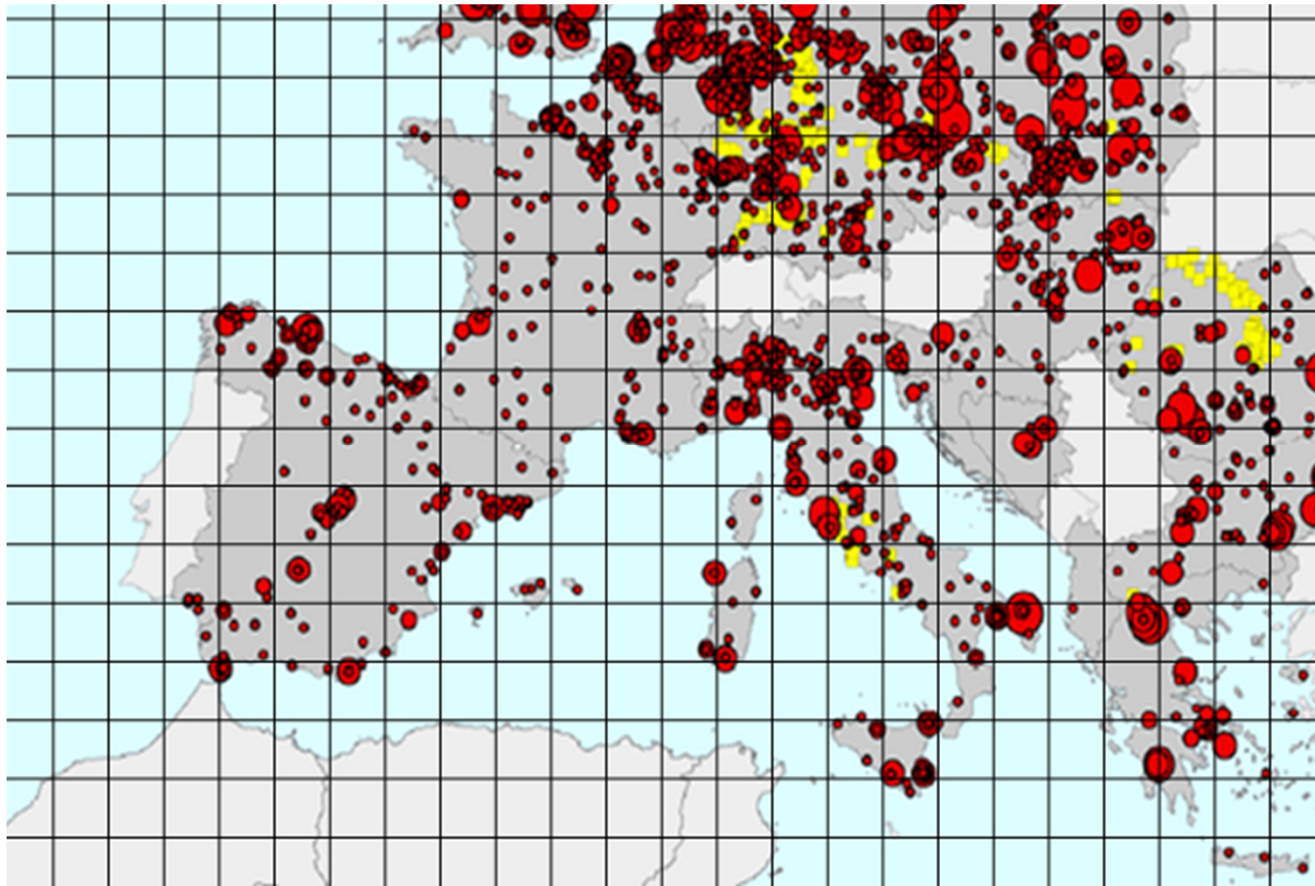
- Default typically: SCHEDULE(static)
- Options: SCHEDULE(static, n)
SCHEDULE(dynamic) SCHEDULE(guided)
SCHEDULE(dynamic,n) SCHEDULE(guided,n)
SCHEDULE(runtime)



SCHEDULE(static)

104, 187 2 threads

54, 99, 138 3 threads



2 2 3 6 9 9 11 12 13 16 13 8 9 12 13 15 16 19 14 13 19 19 18 20



140, 151 SCHEDULE(static,1)

104, 187 SCHEDULE(static)

OPENMP UNCOVERED

Beyond scope

- Synchronisation directives
 - BARRIER
 - ATOMIC & CRITICAL
 - SINGLE & MASTER
 - LASTPRIVATE & FIRST PRIVATE
- Common Block clauses re data scope
- COLLAPSE clause: takes nested DO loops, collapses to larger iteration space
- Dynamic mode for threads per region