Introduction to OpenMP

Sami Saarinen

Sami.Saarinen@ecmwf.int

ECMWF

Acknowledgements

- **Thanks to George Mozdzynski (ECMWF) for providing the original version of OpenMP training material from past few years**

- **Iain Miller & Peter Towers (ECMWF) for providing brand new data for IFS scaling results on Cray XC30**

- **Mikko Byckling (Intel) for an excellent set of OpenMP slides for reference**

Agenda

- OpenMP at a glance
- Matching with available hardware
- Processes vs. threads and core affinity
- Parallelization strategies with OpenMP
- Using OpenMP on ECMWF Cray system
- Performance & scalability of OpenMP
- Miscellaneous cool stuff

Agenda

- **OpenMP at a glance**
- Matching with available hardware
- Processes vs. threads and core affinity
- Parallelization strategies with OpenMP
- Using OpenMP on ECMWF Cray system
- Performance & scalability of OpenMP
- Miscellaneous cool stuff

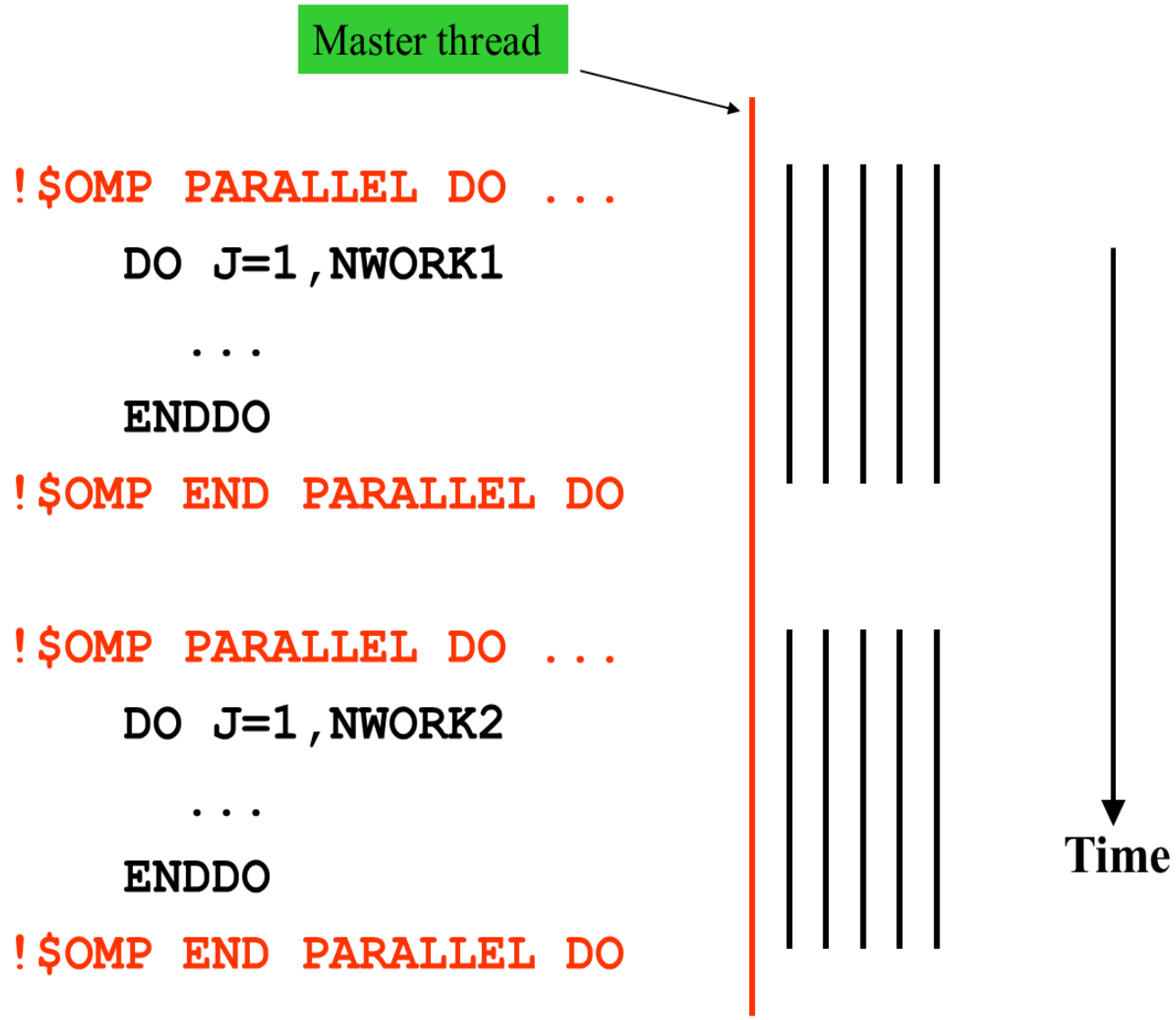
What is OpenMP ?

- **OpenMP = Open Multi-processing**
- **An application programming interface (API) that supports multi-platform *shared memory multiprocessing* programming**
- **Supported languages : Fortran (F77/F95/F2xxx), C & C++**
- ***Very portable* : supported on most computer platforms, processors architectures (NB: not on GPGPUs → OpenACC) & operating systems (Linux, AIX, Windows, Solaris, HP-UX,…)**
- **Parallelization is accomplished via specific *compiler directives*, calls to *library routines* and *environment variables***
- **Development of OpenMP standard is managed by a non-profit technology consortium – see more http://www.openmp.org**
- **OpenMP *can co-exist* with Message Passing Interface (MPI)**
  - **A hybrid (or mixed) parallel programming model**
  - **IFS performance & scalability relies on this mixed mode**

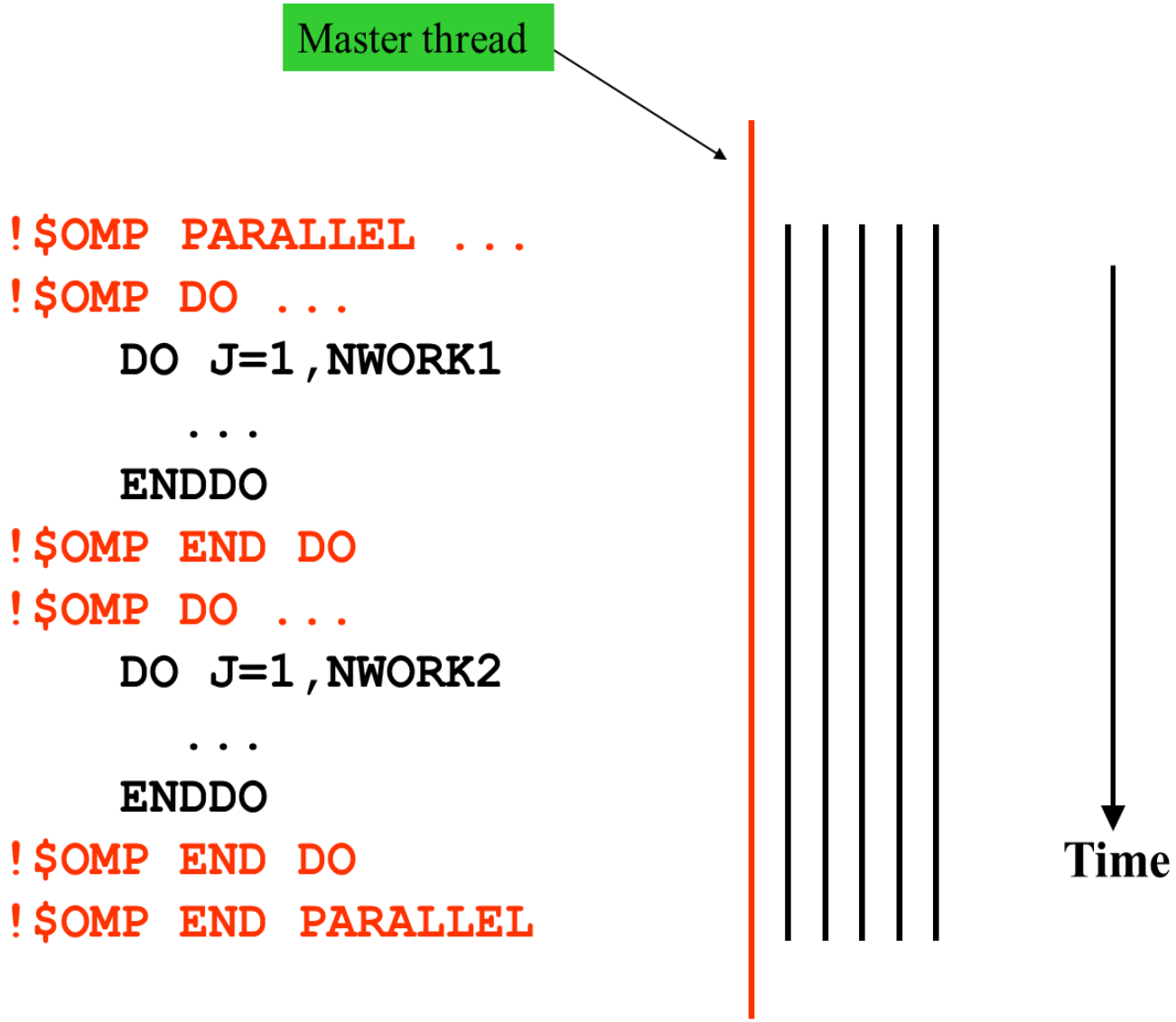
```
!$OMP PARALLEL PRIVATE(JKGLO,ICEND,IBL,IOFF,ZSLBUF1AUX,JFLD,JROF)
  IF (.NOT.ALLOCATED(ZSLBUF1AUX))ALLOCATE (ZSLBUF1AUX(NPROMA,NFLDSLB1))
!$OMP DO SCHEDULE(DYNAMIC,1)

  DO JKGLO=1,NGPTOT,NPROMA
    ICEND=MIN(NPROMA,NGPTOT-JKGLO+1)
    IBL=(JKGLO-1)/NPROMA+1
    IOFF=JKGLO
    ZSLBUF1AUX(:,:)=_ZERO_
    CALL CPG25(CDCONF(4:4) &
       &,ICEND,JKGLO,NGPBLKS,ZSLBUF1AUX,ZSLBUF2X(1,1,IBL) &
       &,RCORI(IOFF),GM(IOFF),RATATH(IOFF),RATATX(IOFF)&
           ...
       &,GT5(1,MSPT5M,IBL))
!        move data from blocked form to latitude (NASLB1) form
    DO JFLD=1,NFLDSLB1
      DO JROF=JKGLO,MIN(JKGLO-1+NPROMA,NGPTOT)
        ZSLBUF1(NSLCORE(JROF),JFLD)=ZSLBUF1AUX(JROF-JKGLO+1,JFLD)
      ENDDO
    ENDDO
  ENDDO

!$OMP END DO
  IF (ALLOCATED(ZSLBUF1AUX)) DEALLOCATE(ZSLBUF1AUX)
!$OMP END PARALLEL
```

ifs/control/gp_model_ad.F90

Many parallel regions → course grain parallelism (~avoid)

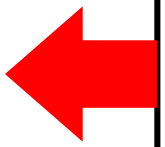


Single parallel region → fine grain parallelism (~better)



Two parallel regions with a single loop in each (BAD !)

```
PROGRAM MPYADD
INTEGER, PARAMETER :: N = 1000000
REAL(8) :: A(N), B(N), C
C = 3.14_8

!$OMP PARALLEL DO
DO J=1,N
  A(J) = 1
  B(J) = 2
ENDDO
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
DO J=1,N
  A(J) = A(J) + C * B(J)
ENDDO
!$OMP END PARALLEL DO

END PROGRAM MPYADD
```



Extra parallel region join & fork here reduces performance

One parallel region with two loop-nests (GOOD !)

```
PROGRAM MPYADD
INTEGER, PARAMETER :: N = 1000000
REAL(8) :: A(N), B(N), C
C = 3.14_8
!$OMP PARALLEL

!$OMP DO
DO J=1,N
  A(J) = 1
  B(J) = 2
ENDDO
!$OMP END DO

!$OMP DO
DO J=1,N
  A(J) = A(J) + C * B(J)
ENDDO
!$OMP END DO

!$OMP END PARALLEL
END PROGRAM MPYADD
```

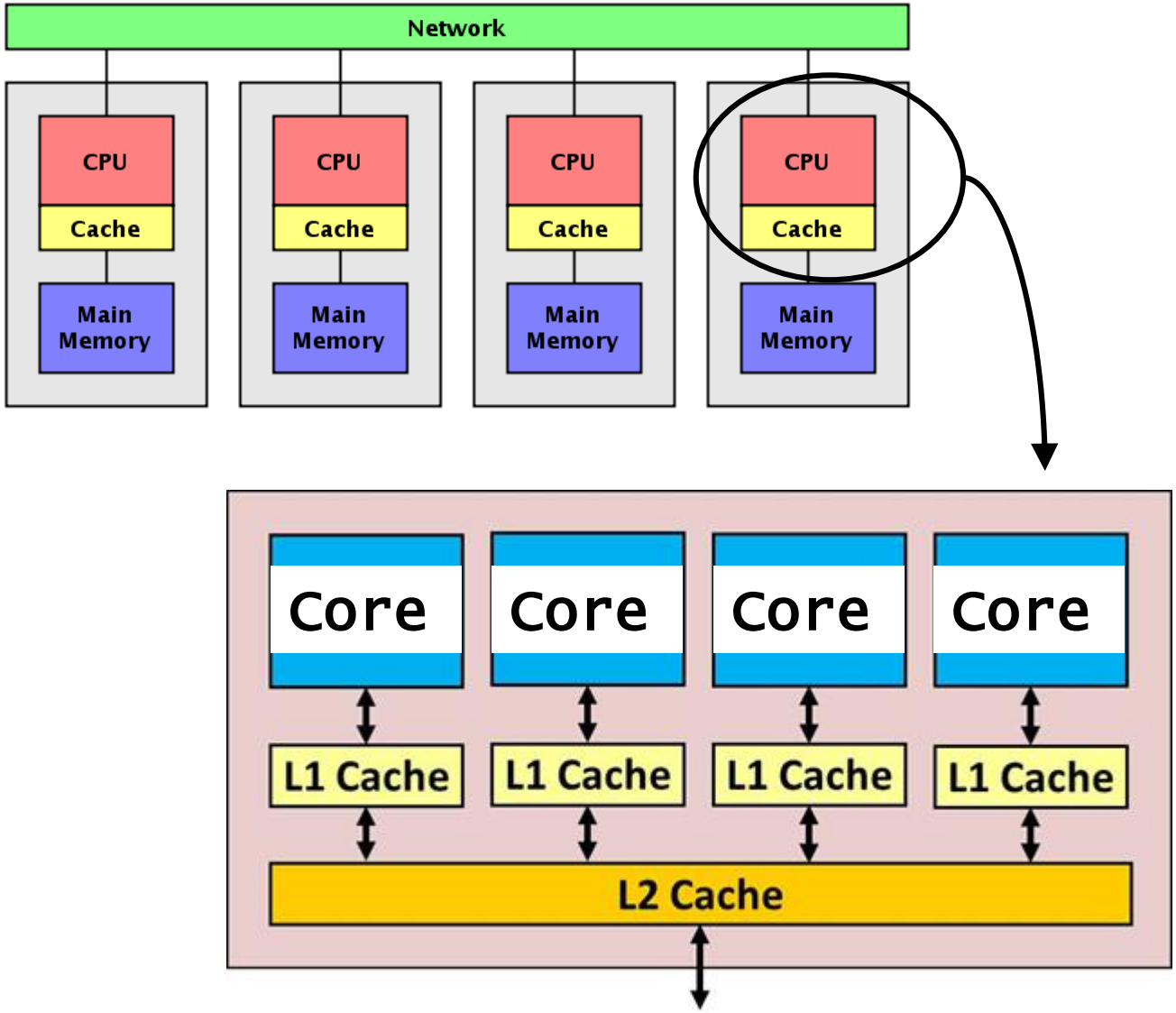
Agenda

- OpenMP at a glance
- **Matching with available hardware**
- Processes vs. threads and core affinity
- Parallelization strategies with OpenMP
- Using OpenMP on ECMWF Cray system
- Performance & scalability of OpenMP
- Miscellaneous cool stuff

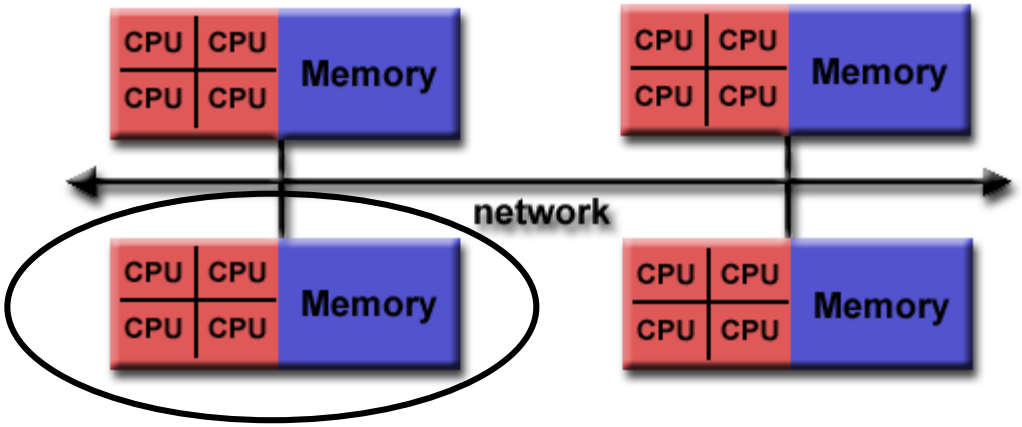
Typical hardware in scientific computing

- **State of the hardware in scientific computing**
  - **Clock speeds (GHz) not going up as in the past (not since ~ 2004)**
  - **Parallel programming skills are needed for achieving performance**
  - **Energy cost may push us to use accelerators**
    - **GPGPUs (e.g. NVIDIA Tesla)**
    - **Many integrated cores (e.g. Intel Xeon Phi "MIC")**
- **Robust programming models now**
  - **Use MPI (Message Passing Interface) or …**
  - **… OpenMP or …**
  - **… both together → hybrid computing**
- **Also good results can be achieved by use of OpenACC / CUDA**
  - **Less trivial to maintain single, portable code base**
  - **GPGPUs out of scope for this training**

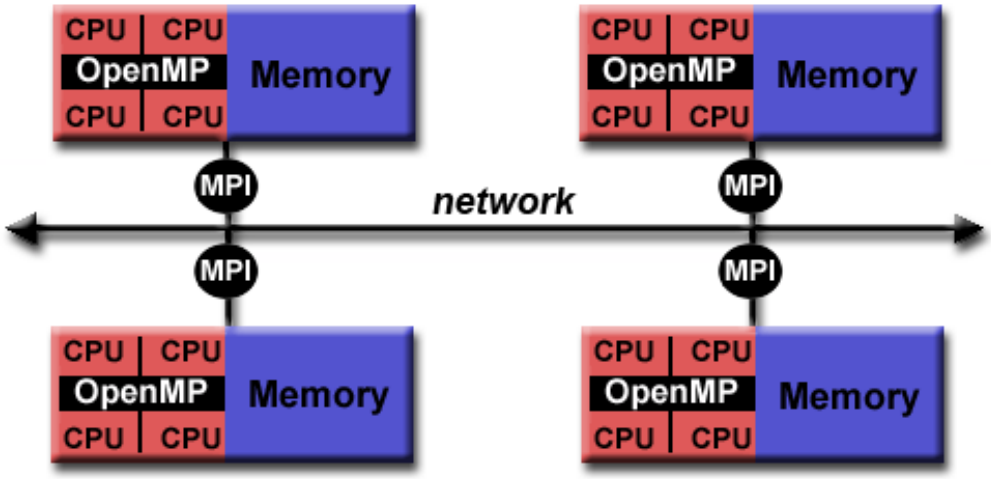
Typical hardware in scientific computing



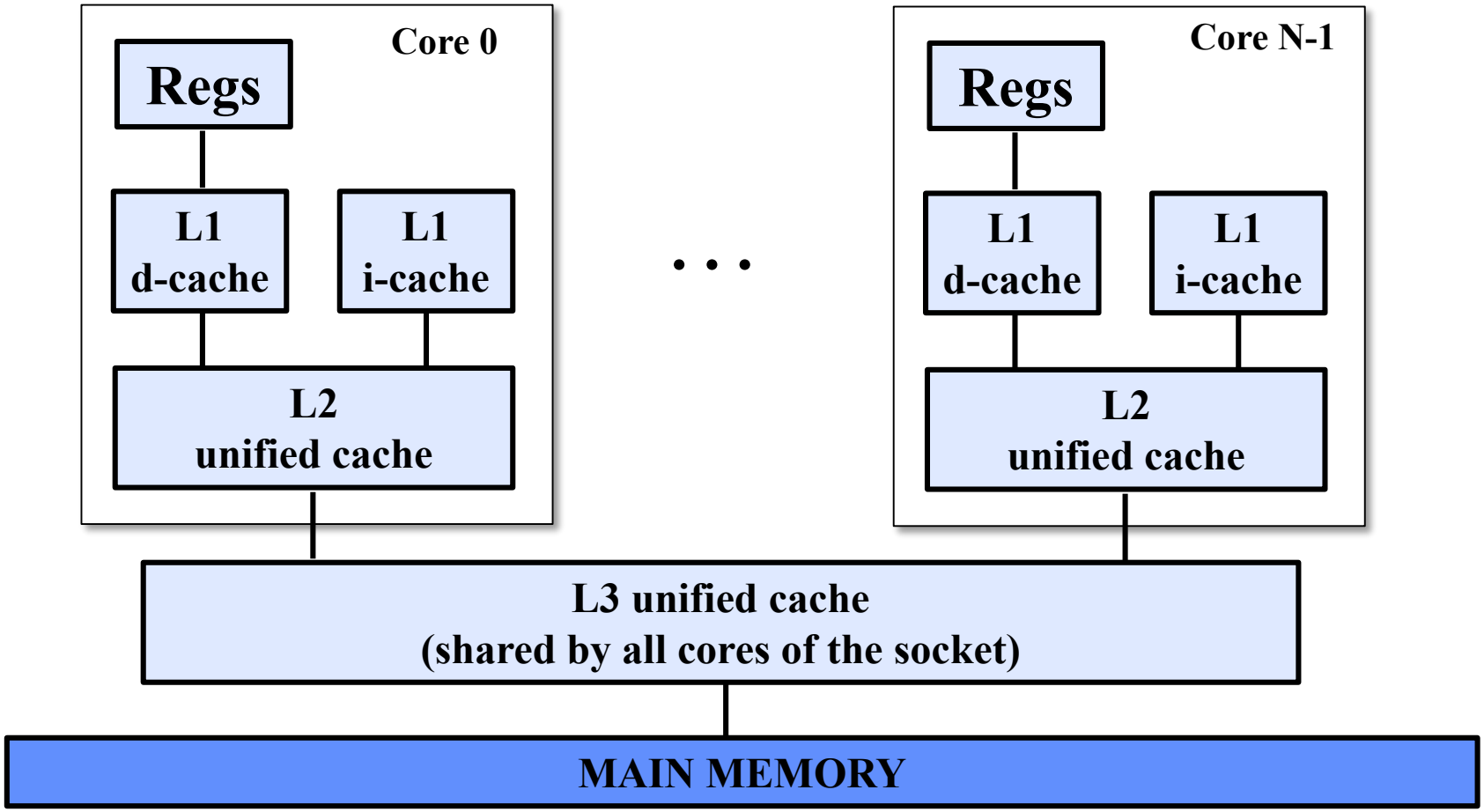
Distributed, shared & hybrid programming



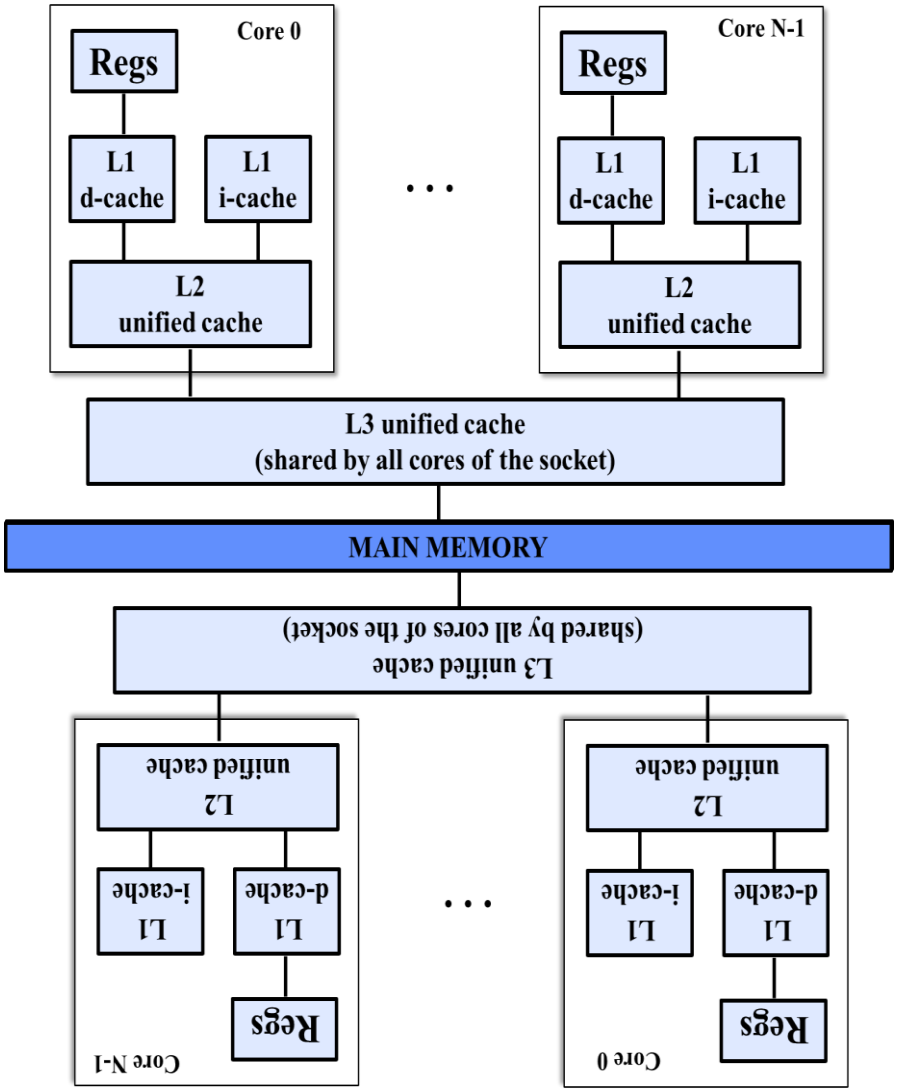
A compute node



Intel Core i7 socket (SnB, IvB, HsW, BdW)



ECMWF Cray systems (phases I & II)



I. **Cray XC30 node : N = 12**

- Ivy Bridge @ 2.7GHz
- 12 cores x 2 sockets
  - 24 cores / node
  - 64GB / node
- ~ 3400 nodes x 2

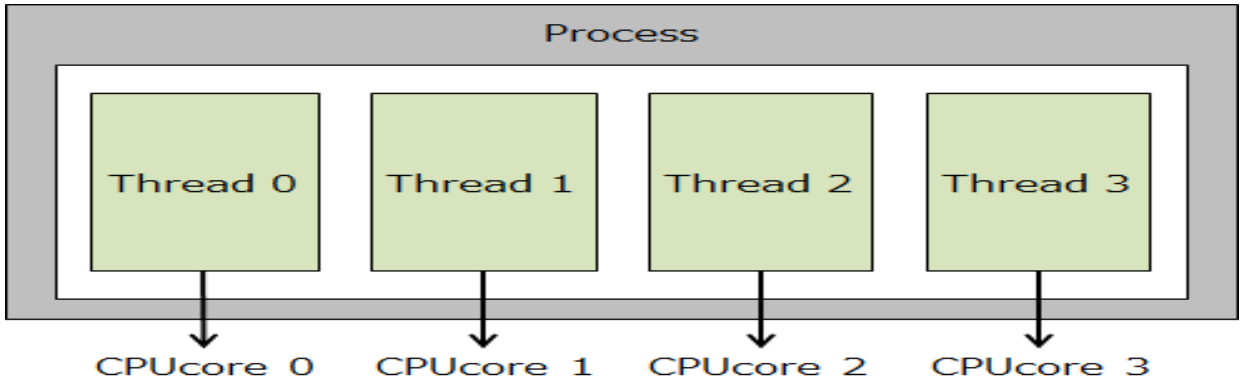
II. **Cray XC40 node : N = 18**

- Broadwell @ 2.1GHz
- 18 cores x 2 sockets
  - 36 cores node
  - 128GB / node
- ~ 3500 nodes x 2

Agenda

- OpenMP at a glance
- Matching with available hardware
- **Processes vs. threads and core affinity**
- Parallelization strategies with OpenMP
- Using OpenMP on ECMWF Cray system
- Performance & scalability of OpenMP
- Miscellaneous cool stuff

Processes vs. threads



- **Process (e.g. an MPI-task)**
  - **Independent execution unit**
  - **Own state & address space**
  - **Created upon start of program**
  - **Communication between processes usually via MPI**
  - **Not all processes have to reside on the same physical compute node**
- **Thread (as with OpenMP)**
  - **A single process can have multiple threads**
  - **All threads of a process share the same state & address space**
  - **Can be created & destroyed dynamically (as needed)**
  - **Communicate directly through the shared memory**

Core affinity

- **Core affinity or thread-to-core binding**
  - **Pins individual threads to cores upon start up of a program**
- **Often paramount for good performance & scaling**
  - **Prevents runtime migration of threads to another cores**
  - **Better memory locality and reduction of cache misses**
- **Usually set outside the program, e.g.**
  - **During aprun/mpirun invocation**
  - `export OMP_PROC_BIND=true`

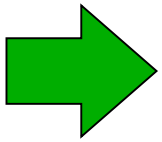
Agenda

- OpenMP at a glance
- Matching with available hardware
- Processes vs. threads and core affinity
- **Parallelization strategies with OpenMP**
- Using OpenMP on ECMWF Cray system
- Performance & scalability of OpenMP
- Miscellaneous cool stuff

Hello World – with OpenMP

- ***One of the main objectives:*** **The same code should run correctly with and without OpenMP directives**
  - **The !$ sentinel is treated as a comment in non-OpenMP runs**
  - **Use only in a very exceptional cases `#ifdef _OPENMP` -blocks**

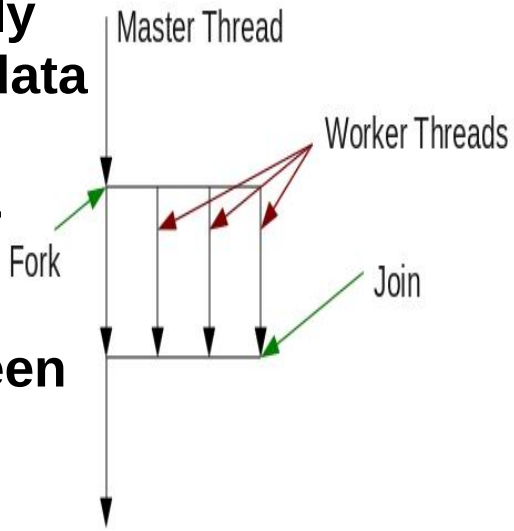
```
PROGRAM HELLO_WORLD
#ifdef _OPENMP
use omp_lib
#endif
INTEGER :: tid
!$OMP PARALLEL PRIVATE(tid)
#ifdef _OPENMP
   tid = omp_get_thread_num( )
#else
   tid = 0
#endif
print *,'Hello World! tid#',tid
!$OMP END PARALLEL
END PROGRAM HELLO_WORLD
```



```
PROGRAM HELLO_WORLD
!$ use omp_lib
INTEGER :: tid
!$OMP PARALLEL PRIVATE(tid)
   tid = 0
!$ tid = omp_get_thread_num( )
!$OMP CRITICAL
print *,'Hello World! tid#',tid
!$OMP END CRITICAL
!$OMP END PARALLEL
END PROGRAM HELLO_WORLD
```

OpenMP parallel regions and work sharing

- **The `!$OMP PARALLEL -- !$OMP END PARALLEL` defines a parallel region, where one or more threads (master + slaves) execute the same code usually independently working with their own copy of data (otherwise data race condition)**
- **Before and after parallel region only the master thread executes the code**
- **Within a parallel region work can be split between threads by**
  - **Loop nests (`!$OMP DO`)**
  - **Work sharing with F90 array syntax (`!$OMP WORKSHARE`)**
  - **Code sections (`!$OMP SECTION`)**
  - **Single and master constructs (`!$OMP SINGLE | MASTER`)**
  - **Using OpenMP tasks (`!$OMP TASK`)**



Key directives – Parallel Region

```
!$OMP PARALLEL [clause,[clause…]]
   code block
!$OMP END PARALLEL
```

Where *clause* can be

- PRIVATE(*list*)
- etc.,

Key directives – Work-sharing constructs/1

```
!$OMP DO [clause,[clause…]]
  do_loop
!$OMP END DO [nowait]
```

Where *clause* can be

- **PRIVATE(*list*)**
- **SCHEDULE(*type*[,*chunk*])**
- **REDUCTION(operator:variable)**
- **etc.,**

Key directives – combined parallel work-sharing/1

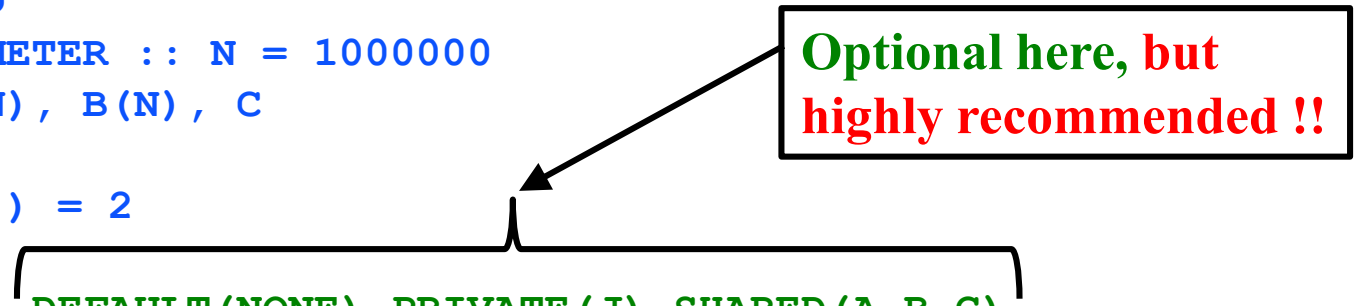
```
!$OMP PARALLEL DO [clause,[clause…]]
  do_loop
!$OMP END PARALLEL DO [nowait]
```

Where *clause* can be

- **PRIVATE(*list*)**
- **SCHEDULE(*type*[,*chunk*])**
- **etc.,**

For example vector multiply & add

```
PROGRAM MPYADD
INTEGER, PARAMETER :: N = 1000000
REAL(8) :: A(N), B(N), C
C = 3.14_8
A(:) = 1 ; B(:) = 2

!$OMP PARALLEL DEFAULT(NONE) PRIVATE(J) SHARED(A,B,C)

!$OMP DO
DO J=1,N
  A(J) = A(J) + C * B(J)
ENDDO
!$OMP END DO

!$OMP END PARALLEL

PRINT *,'SUM(A)=',SUM(A)
END PROGRAM MPYADD
```



Reduction loop : Dot product of two vectors

Due to floating point arithmetic the result is NOT reproducible !!

```
PROGRAM DAXPY
INTEGER, PARAMETER :: N = 1000000
REAL(8) :: A(N), B(N), S

A(:) = 1 ; B(:) = 2

S = 0
!$OMP PARALLEL REDUCTION(+:S)
!$OMP DO
DO J=1,N
  S = S + A(J)* B(J)
ENDDO
!$OMP END DO
!$OMP END PARALLEL

PRINT *,'S = ',S
END PROGRAM DAXPY
```

Key directives – Work-sharing constructs/2

```
!$OMP WORKSHARE
  code block with Fortran array syntax
!$OMP END WORKSHARE
```

No PRIVATE or SCHEDULE options

A good example for *code block* would be Fortran array assignment statements (i.e. no DO-loops involved)

Key directives – combined parallel work-sharing/2

```
!$OMP PARALLEL WORKSHARE [clause,[clause…]]
  code block with Fortran array syntax
!$OMP END PARALLEL WORKSHARE
```

Where *clause* can be

- PRIVATE(*list*)
- etc.,

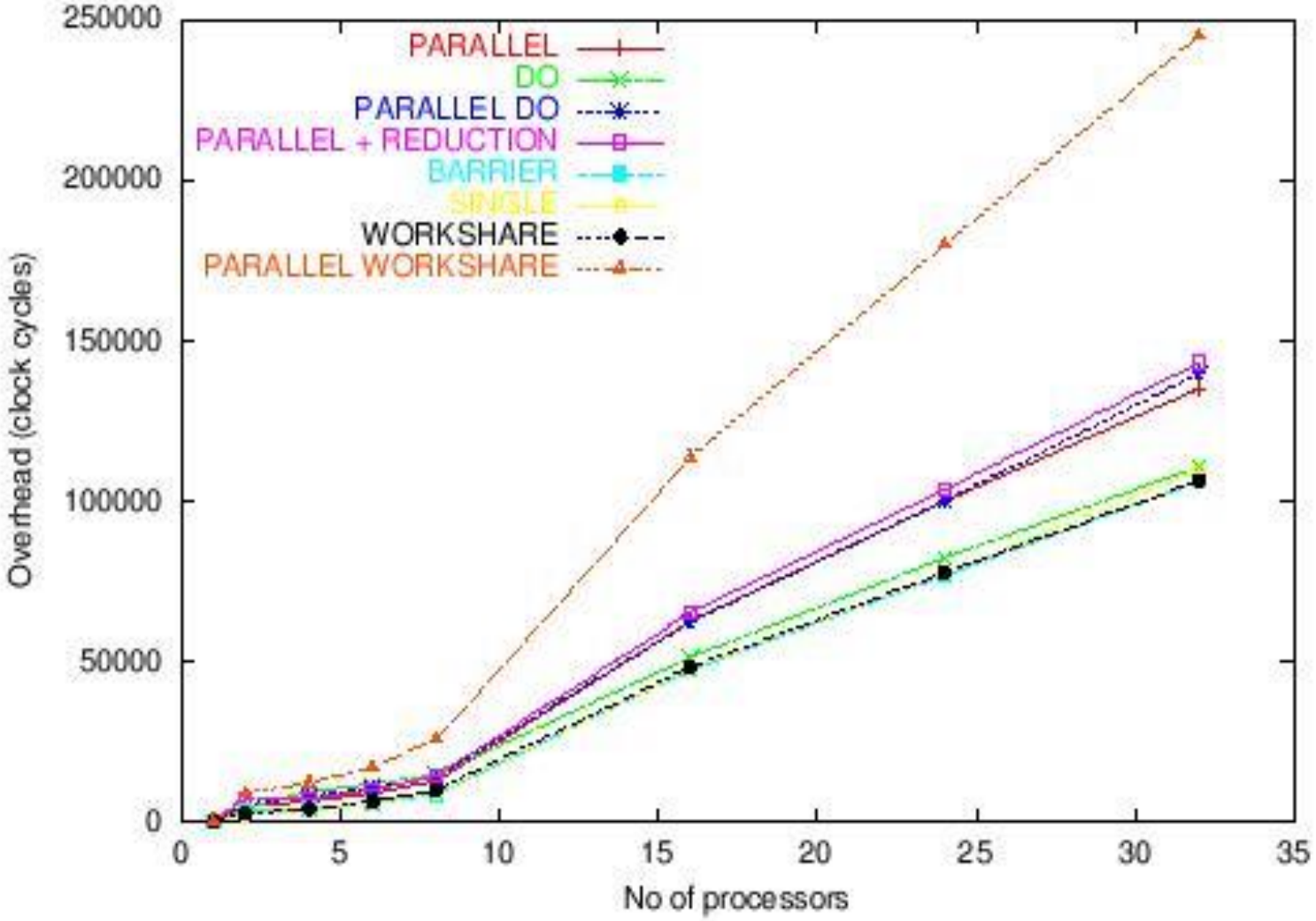
Using WORKSHARE with Fortran array syntax

```
PROGRAM WSHARE
INTEGER, PARAMETER :: N = 1000000
REAL(8) :: A(N), B(N), C(N)

A(1:N/2) = 0 ; A(1/N+1) = 1

!$OMP PARALLEL

!$OMP WORKSHARE
WHERE (A == 0)
   B = 1 ; C = 2
ELSEWHERE
   B = 0 ; C = 1
END WHERE
A = A + B * C
!$OMP END WORKSHARE

!$OMP END PARALLEL
END PROGRAM WSHARE
```

We can also have general code section parallelism

```
PROGRAM CODESEC
INTEGER, PARAMETER :: N = 1000000
REAL(8) :: A(N), B(N), C(N), D(N/2), E(N/2)

!$OMP PARALLEL

!$OMP SECTIONS

!$OMP SECTION
  CALL AFUNC (A,N)
!$OMP SECTION
  CALL BFUNC (B, N)
!$OMP SECTION
  CALL CFUNC (C, N)
!$OMP SECTION
  CALL EDCALC (D, E, N/2)

!$OMP END SECTIONS

!$OMP END PARALLEL
END PROGRAM CODESEC
```

Synchronisation overheads on IBM Power690+ (Power4)



Thread synchronization

- **Synchronization (implicit/explicit) is essential to ensure correctness of parallel execution (no data race conditions)**
  - **Prevents updates to the same memory location by many threads**
- **Synchronization occurs automatically (implicitly) at the end of !$OMP PARALLEL region**
  - **Also at the end of do loops i.e. !$OMP END DO unless additional NOWAIT clause is supplied**
- **Explicit synchronization (all threads participate)**
  - **!$OMP BARRIER**
  - **!$OMP ATOMIC**
  - **!$OMP CRITICAL – !$OMP END CRITICAL**
- **Explicit synchronization usually by a pair of threads**
  - **Calls to OpenMP lock routines (e.g. OMP_SET_LOCK)**
  - **Out of scope for this training**

Three ways to implement reduction e.g. summation

- **Critical sections are code sections that are run by a single thread at a time – e.g. PRINT'ing from a thread to logfile**
  - **Reduction is one typical example**
  - **When applied to scalars also !$OMP ATOMIC can be used**
  - **The way fastest (and simplest) is still use the REDUCTION –clause**

```
PR = 1
!$OMP PARALLEL DO &
!$OMP& REDUCTION(*:PR)
DO J=1,N
  PR = PR * A(J)
ENDDO
!$OMP END DO PARALLEL
```

```
PR = 1
!$OMP PARALLEL DO &
!$OMP& SHARED(PR)
DO J=1,N
!$OMP CRITICAL
  PR = PR * A(J)
!$OMP END CRITICAL
ENDDO
!$OMP END DO PARALLEL
```

```
PR = 1
!$OMP PARALLEL &
!$OMP& SHARED(PR) &
!$OMP& PRIVATE(PLCL)
PLCL = 1
!$OMP DO
DO J=1,N
  PLCL = PLCL * A(J)
ENDDO
!$OMP END DO
!$OMP ATOMIC
PR = PR * PLCL
!$OMP END PARALLEL
```

NOWAIT – now, be careful with it !!

- **NOWAIT provides way to avoid implicit synchronization between successive DO-loops in a PARALLEL region**
- **Supply NOWAIT at the !$OMP END DO directive, but make sure the DO-loops are not overwriting each others arrays (race condition)**
- **Occasionally !$OMP BARRIER needed to ensure correctness**

```
!- CORRECT CODE --
!$OMP PARALLEL
!$OMP DO
DO J=1,N
  A(J) = B(J) + C(J)
ENDDO
!$OMP END DO
!$OMP DO
DO J=1,N
  D(J) = B(J)
ENDDO
!$OMP END DO
!$OMP DO
DO J=1,N
  C(J) = A(J)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

```
!- WRONG at 3rd LOOP
!$OMP PARALLEL
!$OMP DO
DO J=1,N
  A(J) = B(J) + C(J)
ENDDO
!$OMP END DO NOWAIT
!$OMP DO
DO J=1,N
  D(J) = B(J)
ENDDO
!$OMP END DO NOWAIT
!$OMP DO
DO J=1,N
  C(J) = A(J)
ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

```
!- CORRECT AGAIN !!
!$OMP PARALLEL
!$OMP DO
DO J=1,N
  A(J) = B(J) + C(J)
ENDDO
!$OMP END DO NOWAIT
!$OMP DO
DO J=1,N
  D(J) = B(J)
ENDDO
!$OMP END DO NOWAIT
!$OMP BARRIER
!$OMP DO
DO J=1,N
  C(J) = A(J)
ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Summary of OpenMP parallelization strategies

- **Start with a correct serial execution of the application**
- **Apply OpenMP directives to time-consuming DO-loops one at a time and TEST – TEST – TEST !!**
- **Use high level approach where possible**
- **Use 'thread checker' (Intel Inspector) to perform a correctness check [not covered in this training]**
- **Results may change slightly, but the ultimate goal is that**
  - **Results are bit reproducible for varying number of threads**
- **Avoid reductions for REAL -numbers (except: max, min)**
  - **as they cause different results for different #'s of threads**
- **Fortran array syntax parallelization through WORKSHARE**

Agenda

- OpenMP at a glance
- Matching with available hardware
- Processes vs. threads and core affinity
- Parallelization strategies with OpenMP
- **Using OpenMP on ECMWF Cray system**
- Performance & scalability of OpenMP
- Miscellaneous cool stuff

OpenMP on ECMWF Cray system

- **Multiple choice of compiler vendors:**

  Cray/CCE (the default & our target in this training)

  Intel (ifort) : `module swap PrgEnv-cray PrgEnv-intel`

  GNU (gfortran): `module swap PrgEnv-cray PrgEnv-gnu`

- **All use the same `ftn` wrapper (in Fortran, `cc` for C-programs)**

  Cray/CCE: `ftn -homp f.F90 -o prog.x`

  Intel: `ftn -qopenmp -qopenmp-threadprivate compat f.F90`

  GNU: `ftn -fopenmp f.F90 -o prog.x`

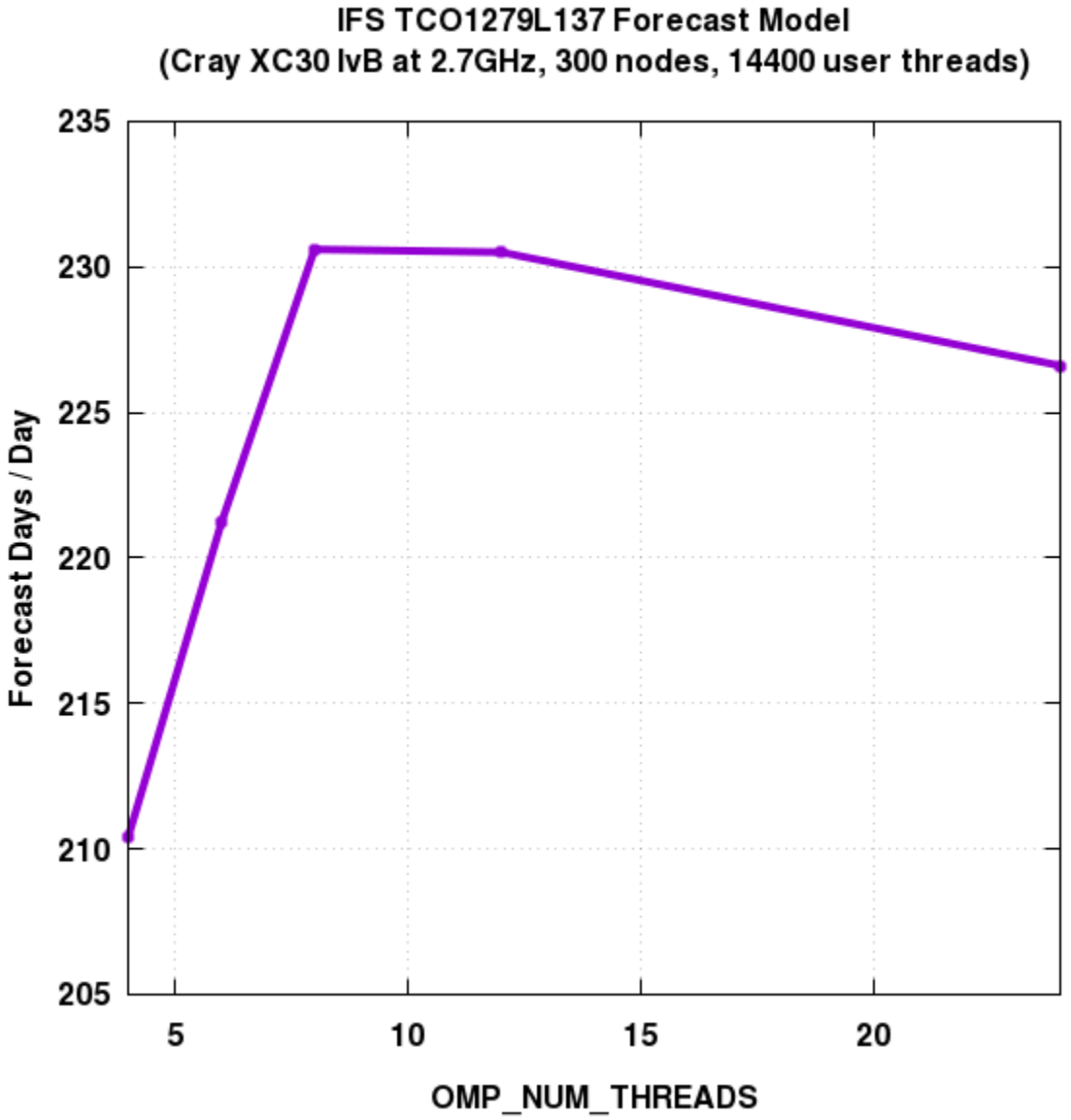
- **Run-script must contain at least (here using 6 threads):**

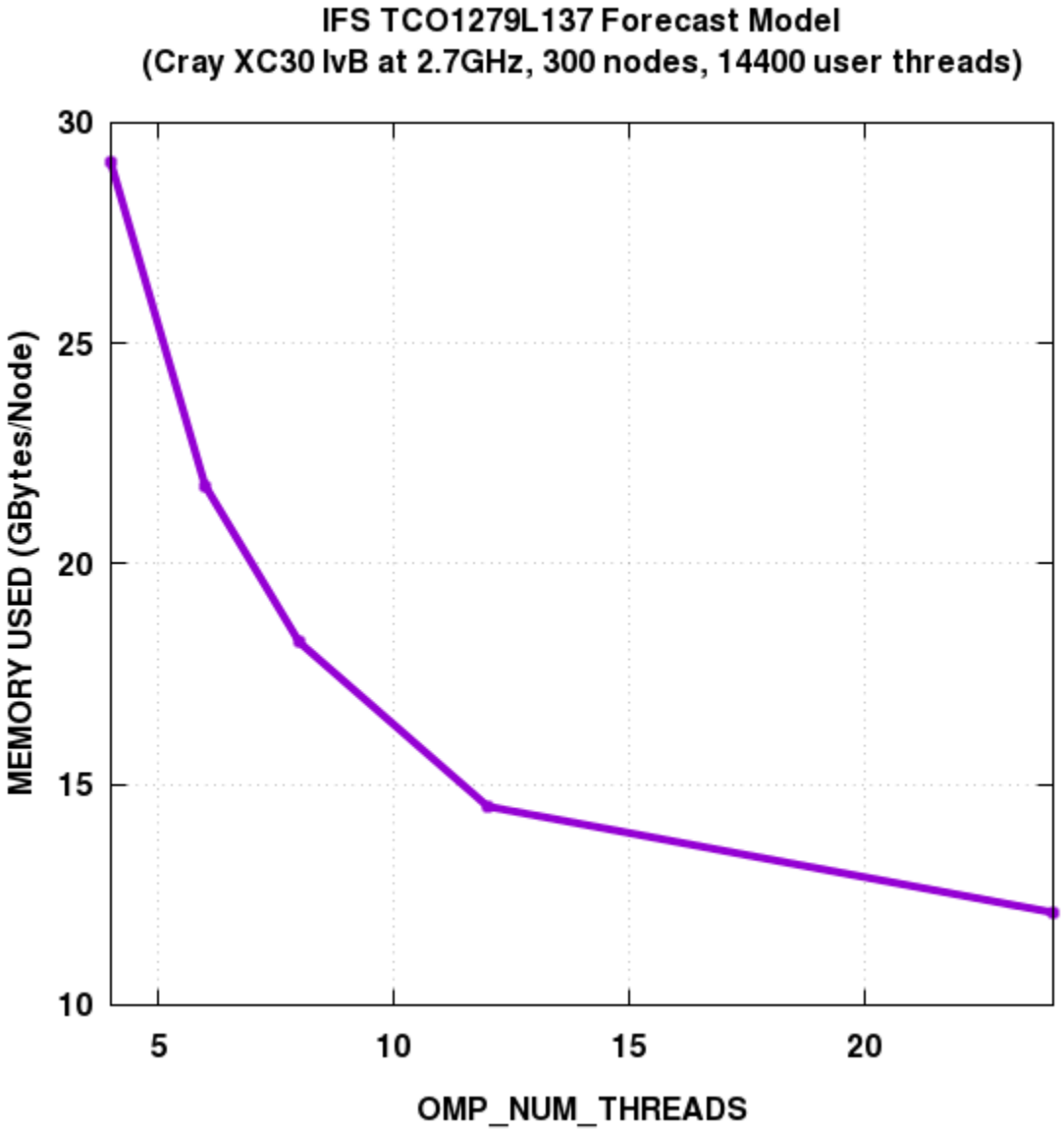
```
export OMP_NUM_THREADS=6
aprun -d $OMP_NUM_THREADS prog.x
```

```
#!/bin/ksh
#PBS -q np
#PBS -j oe
#PBS -N OMP1
#PBS -o omptest.out
#PBS -l EC_nodes=1
#PBS -l EC_total_tasks=1
#PBS -l EC_tasks_per_node=1
#PBS -l EC_threads_per_task=24
#PBS -l EC_hyperthreads=1
#PBS -l walltime=00:01:00

cd $PBS_O_WORKDIR

ftn -ra -homp -o omptest omptest1.F90

for omp in 1 2 3 6 12 24
do
  echo Using $omp threads
  export OMP_NUM_THREADS=$omp
  aprun -d $OMP_NUM_THREADS omptest
done
```

Agenda

- OpenMP at a glance
- Matching with available hardware
- Processes vs. threads and core affinity
- Parallelization strategies with OpenMP
- Using OpenMP on ECMWF Cray system
- **Performance & scalability of OpenMP**
- Miscellaneous cool stuff


IFS TCO1279L137 Forecast Model
(Cray XC30 IvB at 2.7GHz, 300 nodes, 14400 user threads)
Forecast Days / Day
235
230
225
220
215
210
205
5
10
15
20
OMP_NUM_THREADS


IFS TCO1279L137 Forecast Model
(Cray XC30 IvB at 2.7GHz, 300 nodes, 14400 user threads)
MEMORY USED (GBytes/Node)
30
25
20
15
10
5
10
15
20
OMP_NUM_THREADS

Scalability of memory copying – a key to performance

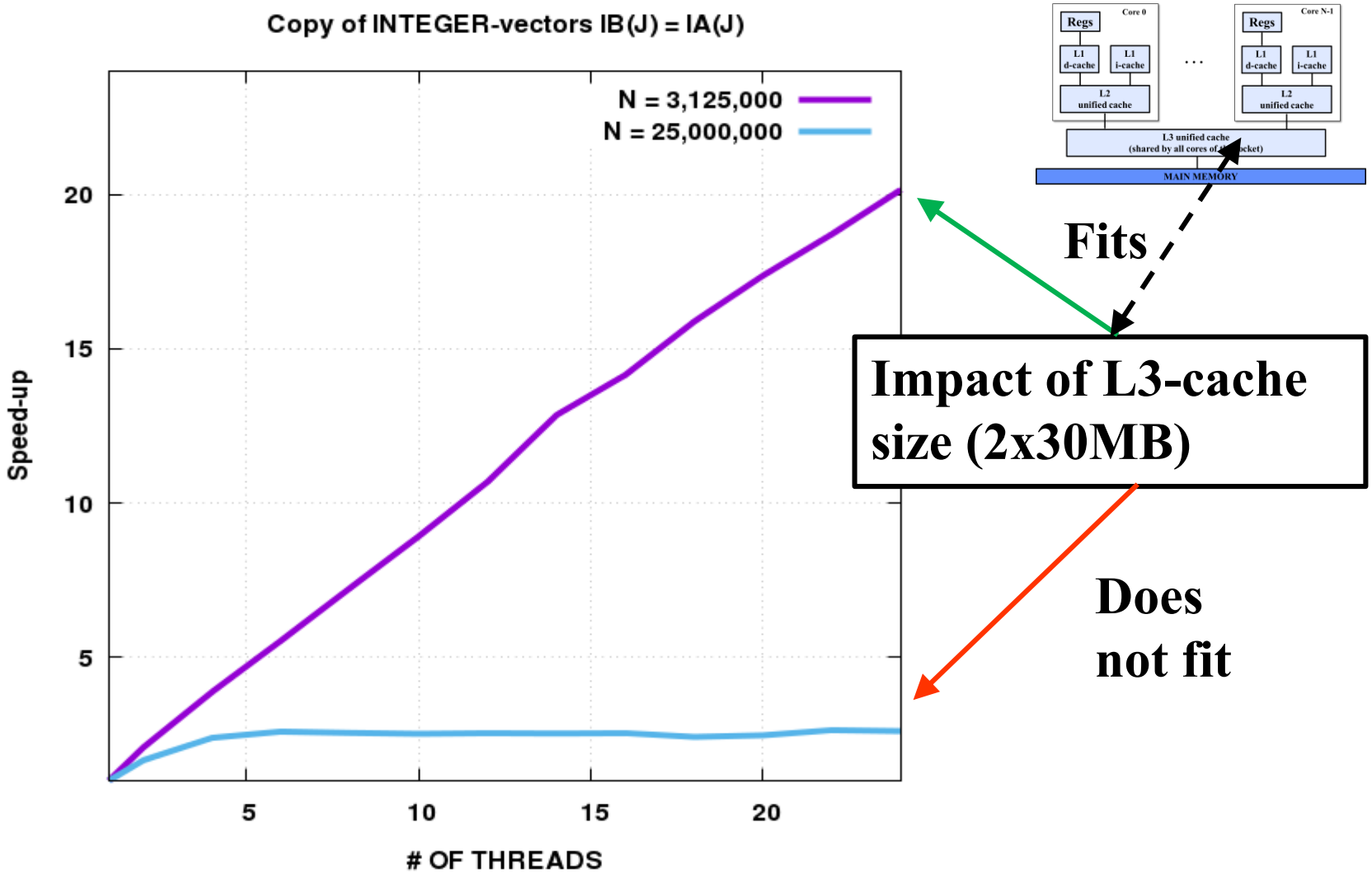
```
PROGRAM COPY
INTEGER, PARAMETER :: N = 1000000 ! Working set size
INTEGER :: J, IA(N), IB(N)
IA(:) = 1
!$OMP PARALLEL DEFAULT(NONE) PRIVATE(J) SHARED(IA,IB)
!$OMP DO
DO J=1,N
  IB(J) = IA(J)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
PRINT *,'SUM(IB)=',SUM(IB)
END PROGRAM COPY
```

```
PROGRAM COPY
INTEGER, PARAMETER :: N = 1000000
INTEGER :: J, IA(N), IB(N)
IA(:) = 1
!$OMP PARALLEL DO
DO J=1,N
  IB(J) = IA(J)
ENDDO
!$OMP END PARALLEL DO
PRINT *,'SUM(IB)=',SUM(IB)
END PROGRAM COPY
```

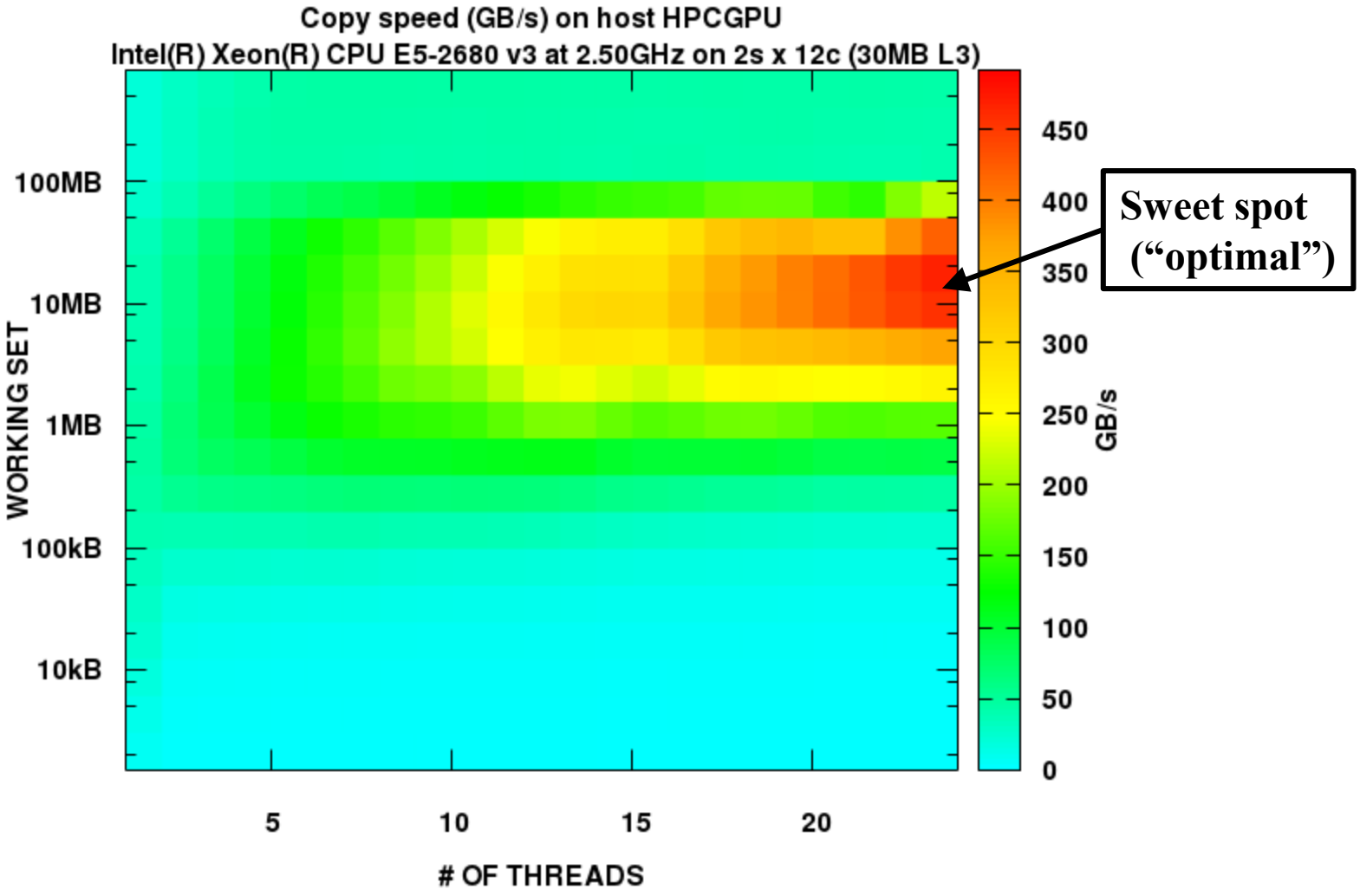
Timings (µsec) of INTEGER-copy : IB(j) = IA(j)

	N = 3,125,000		N = 25,000,000	
Threads	Speedup	Time(us)	Speedup	Time(us)
1	1.00	987	1.00	12672
2	2.08	475	1.66	7615
4	3.88	254	2.39	5307
6	5.54	178	2.59	4899
8	7.24	136	2.55	4963
10	8.92	111	2.52	5030
12	10.69	92	2.54	4986
14	12.85	77	2.53	5014
16	14.15	70	2.54	4998
18	15.88	62	2.42	5238
20	17.37	57	2.47	5128
22	18.71	53	2.64	4793
24	20.15	49	2.61	4859

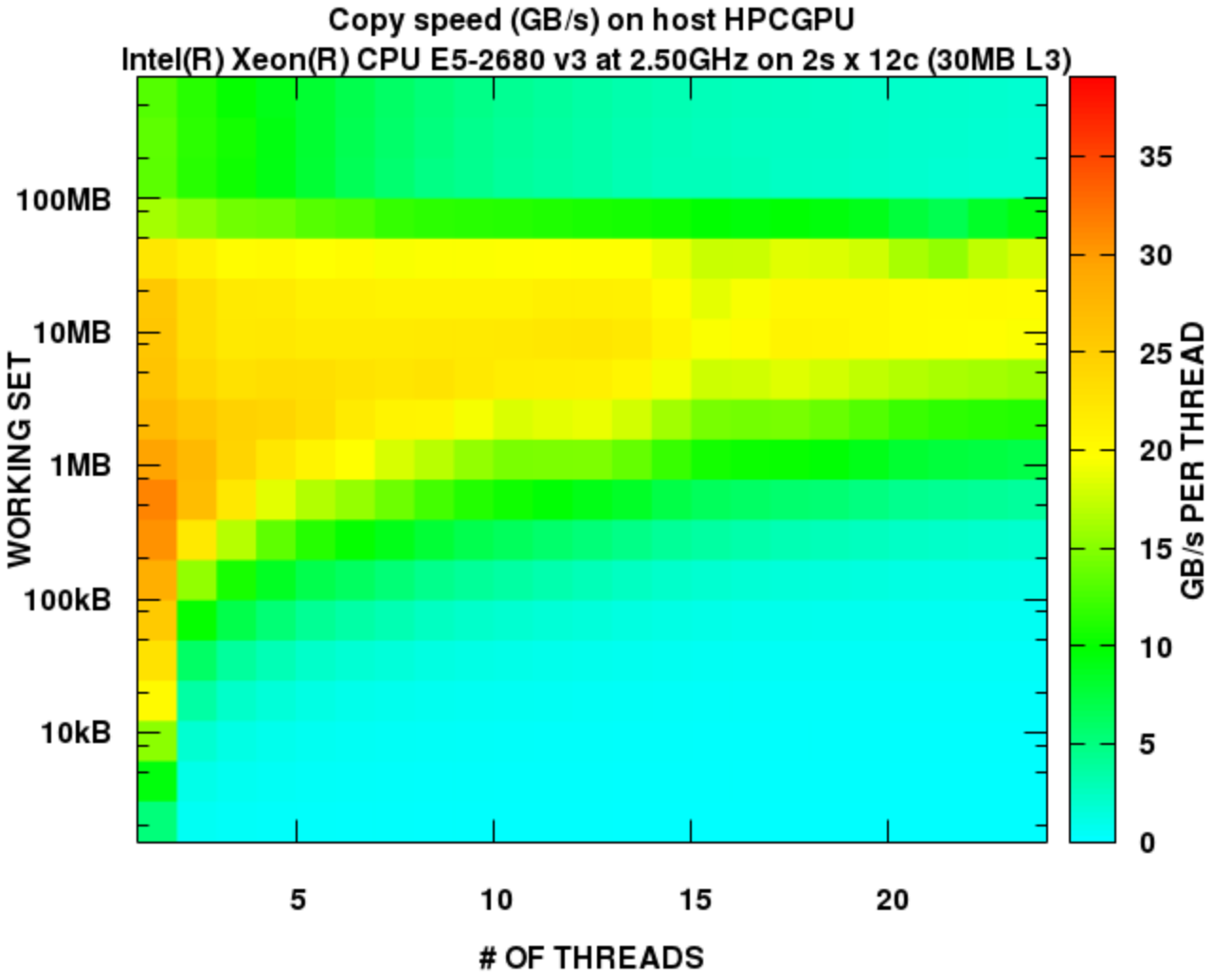
Scaling (speed-up) of INTEGER-copy : IB(j) = IA(j)



Array copying speed (GB/s) as F(working set size, #threads)

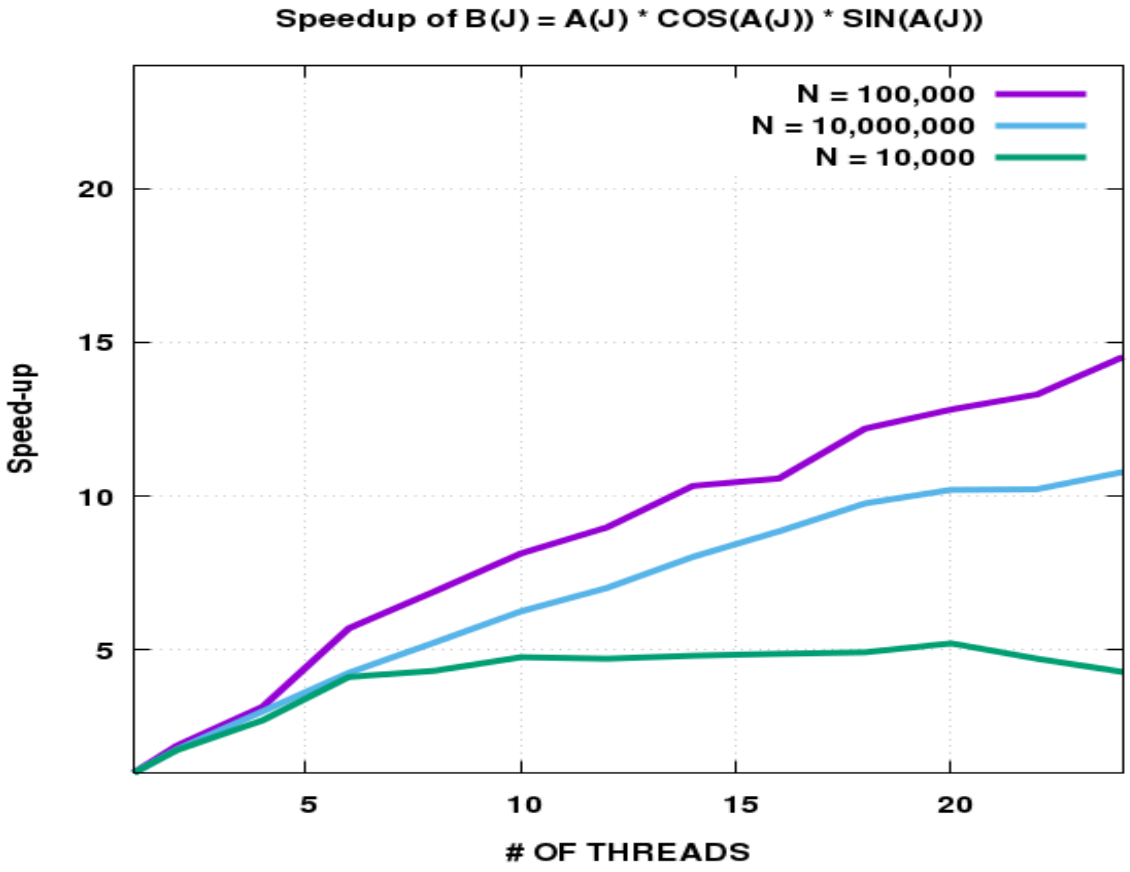


Copying speed in GB/s – per thread (24cores/node)



Scaling (speed-up) of B(j) = A(j) * cos(A(j)) * sin(A(j))

- **Varying vector lengths : N = 10k, 100k, 10M**
  - **Threads from 1 to 24**
  - **REAL(8) array A initialized with random numbers before PARALLEL DO**



The SCHEDULE-clause

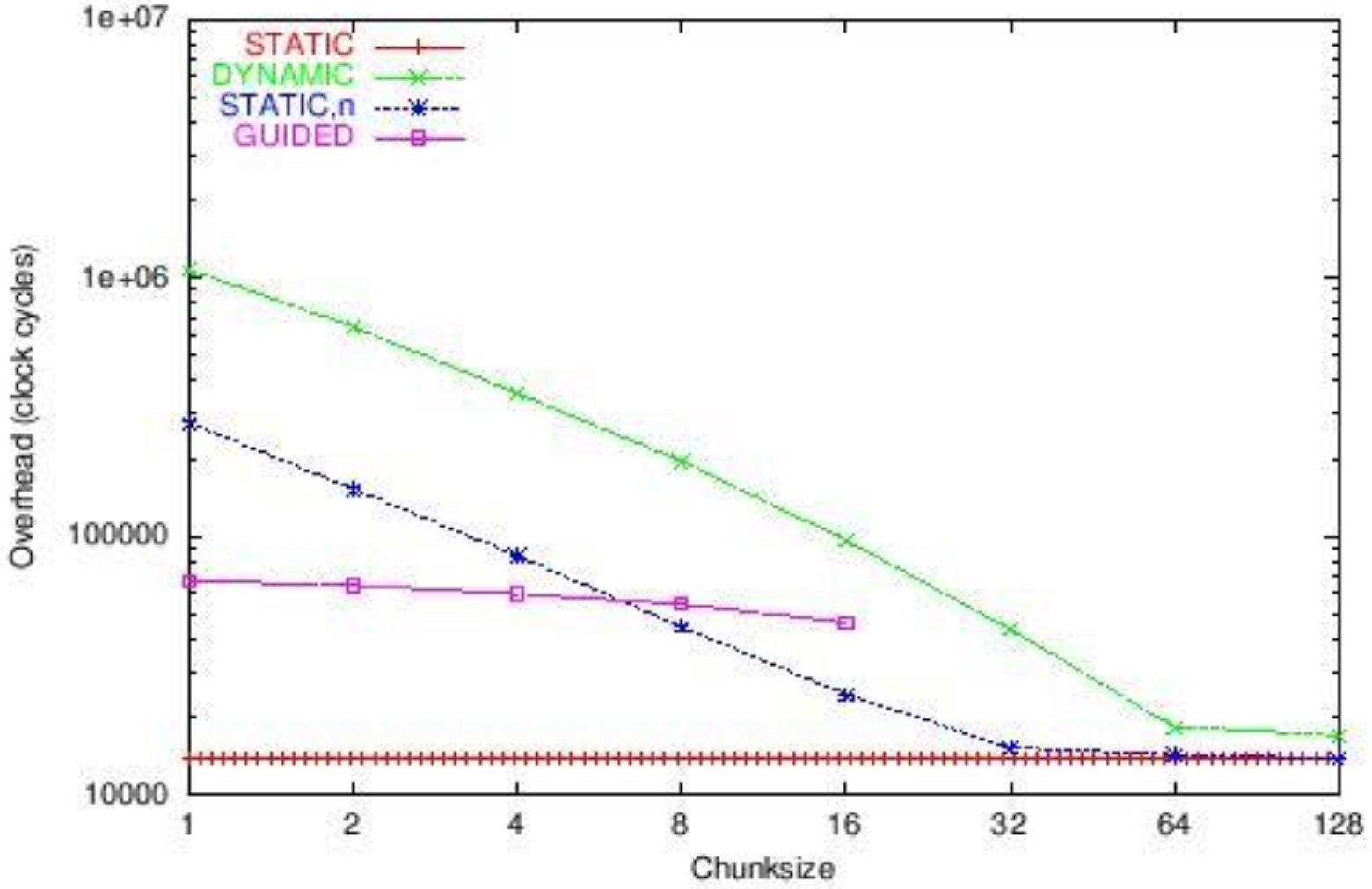
- We can control the chunk size i.e. how many consecutive DO-loop indices belong to a particular thread at once
- Scheduling scenarios can be STATIC , DYNAMIC or GUIDED
- The default is STATIC,NCHUNK where NCHUNK is `(N+numth-1)/numth`
- A special schedule RUNTIME can be used to test performance of various scheduling scenarios without changing the code
  - Just change the environment variable OMP_SCHEDULE

```
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC)
DO J=1,N
  A(J) = A(J) + C * B(J)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

```
!$OMP PARALLEL
!$OMP DO SCHEDULE(DYNAMIC,1)
DO J=1,N
  A(J) = A(J) + C * B(J)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

```
!$OMP PARALLEL
!$OMP DO SCHEDULE(RUNTIME)
DO J=1,N
  A(J) = A(J) + C * B(J)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

SCHEDULE overheads on IBM Power690+ (8 cores)



```
!$OMP PARALLEL PRIVATE(JKGLO,ICEND,IBL,IOFF)
!$OMP DO SCHEDULE(DYNAMIC,1)
    DO JKGLO=1,NGPTOT,NPROMA
      ICEND=MIN(NPROMA,NGPTOT-JKGLO+1)
      IBL=(JKGLO-1)/NPROMA+1
      IOFF=JKGLO

      CALL EC_PHYS(NCURRENT_ITER,LFULLIMP,LNHDYN,CDCONF(4:4)&
                  &,IBL,IGL1,IGL2,ICEND,JKGLO,ICEND &
                  &,GPP(1,1,IBL),GEMU(IOFF)&
                  &,GELAM(IOFF),GESLO(IOFF),GECLO(IOFF),GM(IOFF)&
                  &,OROG(IOFF),GNORDL(IOFF),GNORDM(IOFF)&
                  &,GSQM2(IOFF),RCOLON(IOFF),RSILON(IOFF)&
                  &,RINDX(IOFF),RINDY(IOFF),GAW(IOFF)&

                          ...

                  &,GPBTP9(1,1,IBL),GPBQP9(1,1,IBL)&
                  &,GPFORCEU(1,1,IBL,0),GPFORCEV(1,1,IBL,0)&
                  &,GPFORCET(1,1,IBL,0),GPFORCEQ(1,1,IBL,0))

    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

ifs/control/gp_model.F90

- More about scheduling during the exercises …

Agenda

- OpenMP at a glance
- Matching with available hardware
- Processes vs. threads and core affinity
- Parallelization strategies with OpenMP
- Using OpenMP on ECMWF Cray system
- Performance & scalability of OpenMP
- **Miscellaneous cool stuff**

OpenMP task construct

- **Sometimes data structures are not linear, e.g. unstructured meshes, linked lists**
- **To capture available parallelism, use !$OMP TASK construct**

```
TYPE(OBJECT) :: DATA(N)
REAL :: R
...
!$OMP PARALLEL SHARED(DATA) PRIVATE(J,R)
!$OMP SINGLE  !- runs on some thread
 DO J=1,N
   CALL RANDOM_NUMBER(R) ! Values = (0.0 .. 1.0)
   IF (R > 0.5) THEN
     !$OMP TASK !- runs on next available thread
     !- note: DATA() is still SHARED
     !-  but  J is now FIRSTPRIVATE
     CALL UPDATE(DATA(J))
     !$OMP END TASK
   ENDIF
ENDDO
!$OMP END SINGLE
!$OMP END PARALLEL
```

Some notable environment variables in OpenMP

- **OMP_NUM_THREADS**
  - **Set max number of threads**
- **OMP_SCHEDULE**
  - **Apply schedule for DO SCHEDULE(RUNTIME)**
- **OMP_PROC_BIND**
  - **Set to true to enable core affinity**
- **OMP_NESTED**
  - **Set to false to disable nested PARALLEL regions**
- **OMP_WAIT_POLICY**
  - **Set to active to keep threads running between PARALLEL regions**
- **OMP_STACKSIZE**
  - **See next page**

Stack issues with OpenMP

- **Master thread stack inherits its process' stack**
- **Non-master thread stacks**
  - **Default on CRAY 128 Mbytes**
  - **OMP_STACKSIZE=256M to increase to 256MBytes**
- **Large arrays (>1 Mbyte?) should use the heap**

```
REAL,ALLOCATABLE :: BIGGY(:)

ALLOCATE(BIGGY(100000000))

DEALLOCATE(BIGGY)
```

- **But in general : use of STACK instead of HEAP in PARALLEL regions improves performance of your application !**

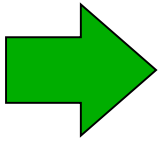
```
> ulimit -a
address space limit (Kibytes)   (-M)   52857040
core file size (blocks)         (-c)   unlimited
cpu time (seconds)              (-t)   unlimited
data size (Kibytes)             (-d)   unlimited
file size (blocks)              (-f)   unlimited
locks                           (-x)   unlimited
locked address space (Kibytes)  (-l)   unlimited
message queue size (Kibytes)    (-q)   800
nice                            (-e)   0
nofile                          (-n)   16000
nproc                           (-u)   516081
pipe buffer size (bytes)        (-p)   4096
max memory size (Kibytes)       (-m)   63221760
rtprio                          (-r)   0
socket buffer size (bytes)      (-b)   4096
sigpend                         (-i)   516081
stack size (Kibytes)            (-s)   unlimited
swap size (Kibytes)             (-w)   not supported
threads                         (-T)   not supported
process size (Kibytes)          (-v)   52857040
```

available memory per node ~ 54 Gbytes i.e. 54,000,000,000 bytes

First touch

- **Always try to allocate & especially initialize memory arrays as close as possible to the core on which the thread (or task) that the memory is going to be used**
  - **This is also known as memory affinity**
- **This initialization or first touch makes your application often to run faster since memory has been cached to the nearest core – the following code snippet shows to do it :**

```
REAL :: A(N)
A(:) = 0
...
!$OMP PARALLEL DO
  DO J=1,N
    CALL DOWORK(A, J)
  ENDDO
!$OMP END PARALLEL DO
```

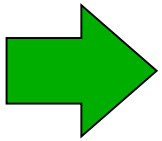


```
REAL :: A(N)
!$OMP PARALLEL DO
  DO J=1,N
    A(J) = 0
  ENDDO
!$OMP END PARALLEL DO
...
!$OMP PARALLEL DO
  DO J=1,N
    CALL WORK(A, B, J)
  ENDDO
!$OMP END PARALLEL DO
```

Collapsing loops : DO COLLAPSE(x) clause

- **Sometimes due to array dimensioning the loop we are parallelizing does not have enough parallelism for OpenMP**
- **In case of multidimensional array, we may get away with this by collapsing two or more *perfectly nested loops* into one much longer loop**

```
REAL :: A(100,100,3), S
...
!$OMP PARALLEL DO &
!$OMP& REDUCTION(+:S)
  DO K=1,3
    DO J=1,100
      DO I=1,100
        S = S + A(I,J,K)
      ENDDO
    ENDDO
  ENDDO
!$OMP END PARALLEL DO
```

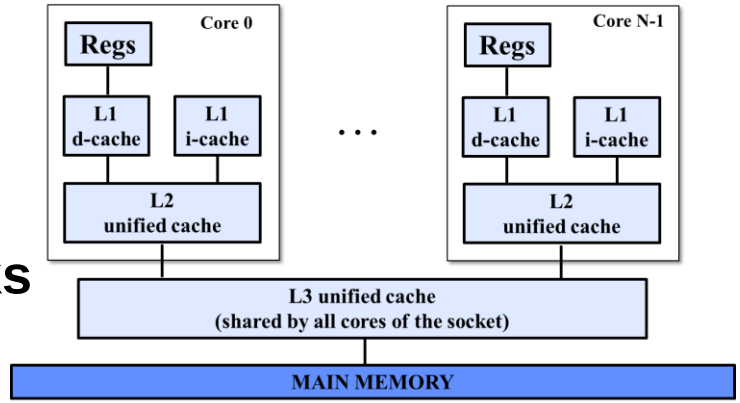


```
REAL :: A(100,100,3), S
...
!$OMP PARALLEL DO COLLAPSE(3)
!$OMP& REDUCTION(+:S)
  DO K=1,3
    DO J=1,100
      DO I=1,100
        S = S + A(I,J,K)
      ENDDO
    ENDDO
  ENDDO
!$OMP END PARALLEL DO
```

Load imbalance (LI)

- **Not always all participating threads end up doing the same amount of work**
  - **The total time is dictated by the slowest running thread**
- **Often programmer can only alleviate the problems as LI maybe due to the nature of the problem, e.g.**
  - **Low observation coverage in certain areas**
  - **Scattered cloud cover**
- **To lessen the impact of LI one could try for example**
  - **Try different SCHEDULE options (e.g. GUIDED, DYNAMIC)**
  - **Limits the max number of participating threads (NUM_THREADS)**
  - **Skip parallelization unless problems size of big e.g. IF (N > 1000)**
  - **Use !$OMP TASK**

False sharing

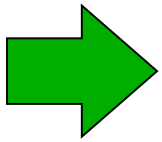


- **Modern CPUs have multilevel caches**
- **Data is accessed from caches in chunks**
  - **Usually 64 bytes cache lines**
  - **And even if you need just one byte from memory**
- **The same data item maybe needed by more than one thread**
  - **Data consistency requires cache coherence logic through H/W**
- **When different threads *modify* successive memory locations [falling potentially to the same cache line], then the cache coherence logic forces these updates to be performed to the all cache copies in all participating cores**
  - **A huge performance penalty due to high rate of cache misses**
  - **Fix : avoid writes to the successive memory locations by different threads by using array padding or even over-dimensioning, or by use of REDUCTION -clause**

False sharing example & fix

- **In histogram calculations we extract bin-index from data**
- **Since bins may be adjacent – potentially falling into the same cache line – then cache coherency protocol ensures that every core (and therefore thread) has a consistent view of histogram counts at any bin-index**

```
REAL :: DATA(N)
INTEGER :: IDX,H(0:NB)
H(:) = 0 ! Histogram
!$OMP PARALLEL PRIVATE(IDX)
!$OMP DO
 DO J=1,N
   CALL GETBIN(DATA(J),IDX)
   H(IDX) = H(IDX) + 1
 ENDDO
!$OMP END DO
!$OMP END PARALLEL
```



```
REAL :: DATA(N)
INTEGER :: IDX,H(0:NB)
H(:) = 0 ! Histogram
!$OMP PARALLEL PRIVATE(IDX)
!$OMP DO REDUCTION(+:H)
 DO J=1,N
   CALL GETBIN(DATA(J),IDX)
   H(IDX) = H(IDX) + 1
 ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

(Data) race condition

- **Data race condition occurs when multiple threads attempt to write the same memory location at once**
- **This is an error and results will be unpredictable**
- **To avoid race conditions, use DEFAULT(NONE) clause in your !$OMP PARALLEL statement**
  - **Forces you to think every single variable whether this is PRIVATE or SHARED**
  - **All SUBROUTINE/FUNCTION local variables are by default PRIVATE, unless they have been declared to SAVE-variables**
- **Another way to avoid race conditions is to use CRITICAL sections, ATOMIC constructs, OpenMP locks, !$OMP SINGLE , or !$OMP MASTER + BARRIER constructs**
  - **But be aware that serialization hurts your parallel performance !!**

Ensuring safe execution of parallel OpenMP regions

- **It is *unsafe* to store to the same location (scalar or array) from multiple threads of a parallel region**
- **A safe & recommended strategy is to**
  - ***only read shared variables***
  - ***only write to non-shared variables***
    - **parallel region PRIVATE variables**
    - **subroutine local variables within parallel region**
    - **threads writing to different memory locations in a module**

Storing data: Unsafe & Safe code snippets

Unsafe & Wrong

```
!$OMP PARALLEL DO
  DO J=1,N
    CALL WORK(J)
  ENDDO
!$OMP END PARALLEL DO
  ...
SUBROUTINE WORK(K)
USE MYMOD, ONLY : X, A
INTEGER K
X=A(K)  ! X has read/write conflict
CALL SUB(X)
```

Safe to write

```
!$OMP PARALLEL DO
  DO J=1,N
    CALL WORK(J)
  ENDDO
!$OMP END PARALLEL DO
  ...
SUBROUTINE WORK(K)
USE MYMOD, ONLY : X, A
INTEGER K
A(K)=X  ! Writing to different A(K) locations
CALL SUB(X)
```

Reduction example – all safe : Slow/Better/Fast

Slow

```
!$OMP PARALLEL DO
  DO J=1,N
    CALL WORK(J)
  ENDDO
!$OMP END PARALLEL DO
...
SUBROUTINE WORK(K)
USE MYMOD, ONLY : M
IVAL = <calculation here>
!$OMP CRITICAL
M=M+IVAL
!$OMP END CRITICAL
```

Better

```
USE MYMOD, ONLY : M
IVAL=0
!$OMP PARALLEL DO &
!$OMP& REDUCTION(+:IVAL)
  DO J=1,N
    CALL WORK(J,IVAL)
  ENDDO
!$OMP END PARALLEL DO
M=M+IVAL
...
SUBROUTINE WORK(K,KVAL)
IVAL = <calculation here>
KVAL = KVAL + IVAL
```

Fast

```
USE MYMOD, ONLY : M
INTEGER IVAL(N)
  ...
!$OMP PARALLEL DO
  DO J=1,N
    CALL WORK(J,IVAL(J))
  ENDDO
!$OMP END PARALLEL DO
M=M+SUM(IVAL)
...
SUBROUTINE WORK(K,KVAL)
KVAL = <calculation here>
```

Critical Region: Unsafe / Safe and Coolest !

Coolest: $OMP SINGLE

```
!$OMP PARALLEL
...
!$OMP SINGLE
<executed only once>
!$OMP END SINGLE
!- Implicit BARRIER here --
...
!$OMP END PARALLEL
```

Safe, but …

```
L_DO_ONCE = .TRUE.
!$OMP PARALLEL
...
!$OMP CRITICAL (REGION)
IF( L_DO_ONCE )THEN
  <executed only once>
  L_DO_ONCE=.FALSE.
ENDIF
!$OMP END CRITICAL (REGION)
...
!$OMP END PARALLEL
```

Unsafe & Wrong

```
L_DO_ONCE = .TRUE.
!$OMP PARALLEL
...
IF( L_DO_ONCE )THEN
!$OMP CRITICAL (REGION)
  <executed only once>
  L_DO_ONCE=.FALSE.
!$OMP END CRITICAL (REGION)
ENDIF
...
!$OMP END PARALLEL
```

Why OpenMP with MPI saves memory over pure MPI?

- **Consider a large 2D-grid with say 102400 x 102400 points**
  - **3rd dimension is fixed to say 137 levels**
- **Does not fit into memory of one compute node**
  - **Need MPI & 2D domain decomposition with halo-areas (width = 4)**
- **Imagine a computer with 6,400 nodes with 16 cores/node**
  - **Solving the problem with 102,400 MPI tasks, one for each core**
  - **Each sub-grid reduces to a region of just 320 x 320 x L137**
  - **Due to halo-areas *EACH* of many 3D-arrays is > 5% larger**
  - **MPI internal data structures (& overhead) would also be excessive**
- **Using 16-way OpenMP we only need 6,400 MPI-tasks**
  - **Decomposition for each MPI-task is now 1,280 x 1,280 x L137**
  - **Less MPI overheads – and memory overhead alone ~ 1%**
  - **But : OpenMP parallelism MUST now be almost perfect, too**

QUESTIONS ?