NTNU – Trondheim
Norwegian University of
Science and Technology

# Introduction to OpenMP

NTNU-IT HPC group/Vitenskapelig databehandling
John Floan (john.floan@ntnu.no)

| | |
|---|---|
| Notur: | www.sigma2.no |
| NTNU HPC | www.hpc.ntnu.no/ |

Slides: www.hpc.ntnu.no/display/hpc/Course+materials

# Plan for the day

-CPU history

-Introduction to OpenMP and parallel programming

-Tutorial 1. Parallel region. Thread creation.

- (Exercise Helloworld)

-Tutorial 2. Parallel for/do loop and Data Sharing

- (Exercise forsin and Matrix multiplication)

-Tutorial 3. Synchronization:  Critical and Atomic directives

- (Exercise Pi)

-Tutorial 4. Reduction. (Pi)

-False Sharing

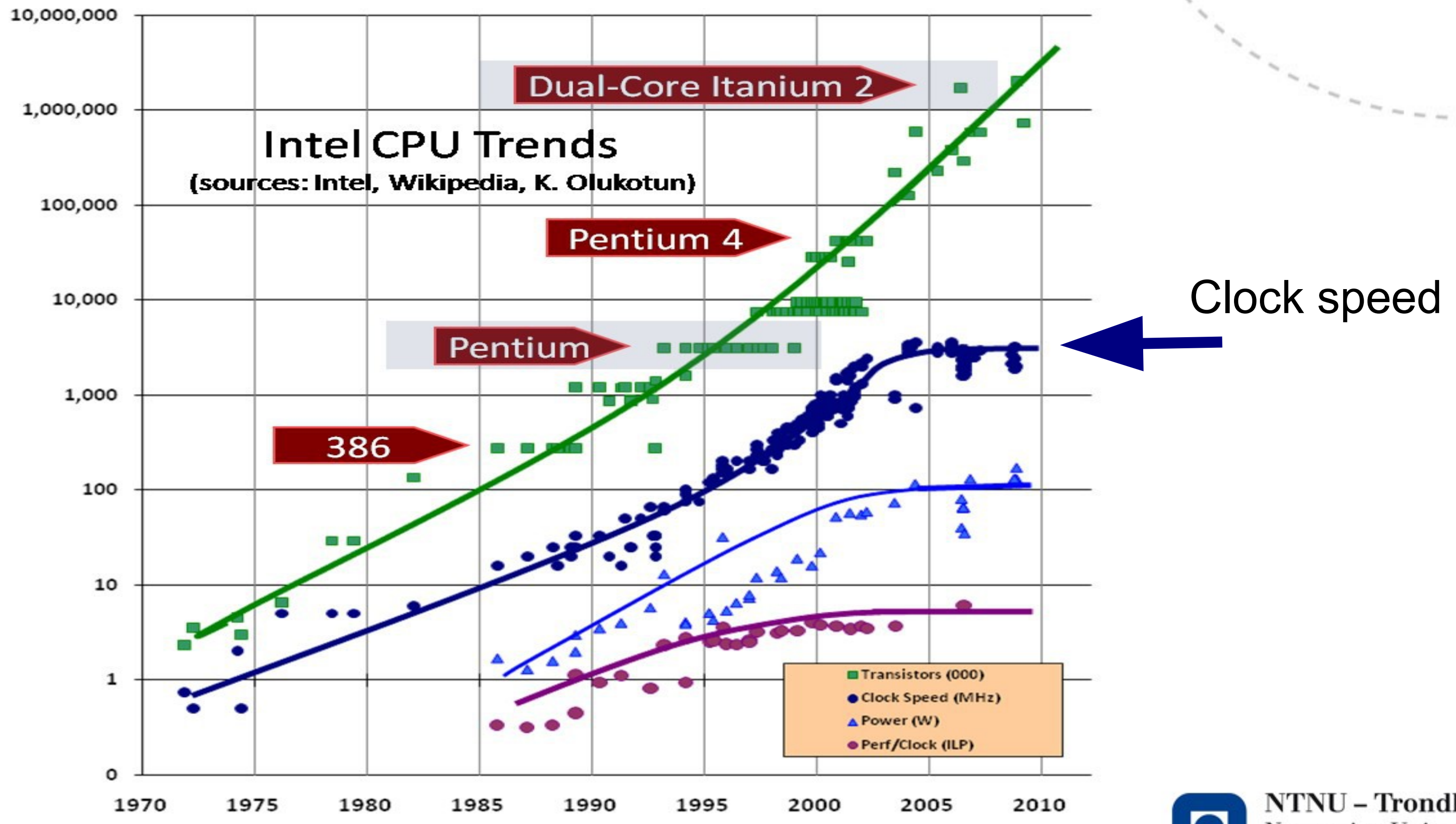
-Data Sharing

-Memory allocation

-How to optimize my sequential code with OpenMP?

## CPU HISTORY

- Moore's law (1965): Number of transistores doubles every two years.
- The clock frequency has flat out since 2005.



NTNU – Trondheim
Norwegian University of Science and Technology

## Performance

Higher frequency : 2 GHz gives 2 timer faster code than 1GHz (ideally)
More cores : 2 cores gives 2 time faster code than 1 core (ideally)

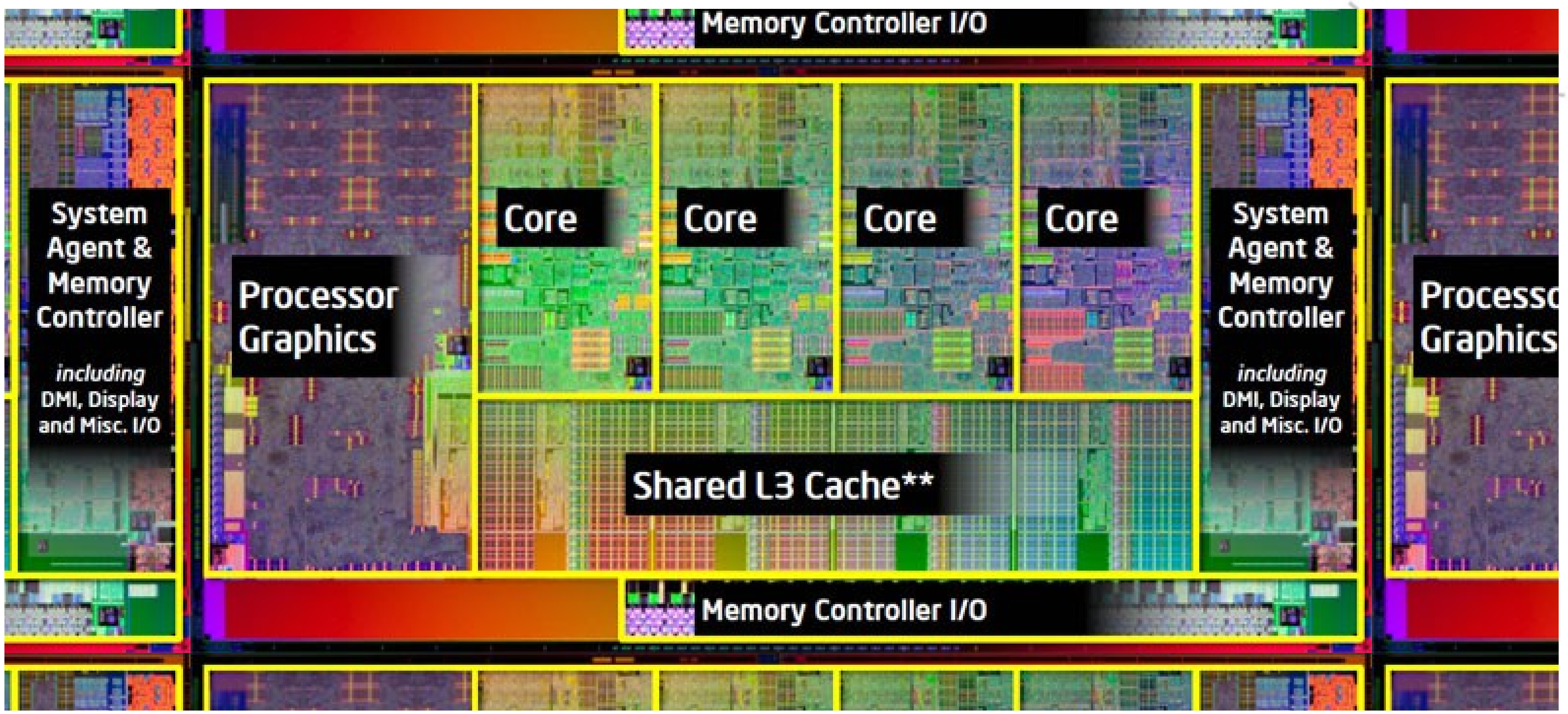

Memory Controller I/O
System Agent & Memory Controller including DMI, Display and Misc. I/O
Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller including DMI, Display and Misc. I/O
Processo Graphics
Shared L3 Cache**
Memory Controller I/O

Fig. Laptop/pc CPU

## What to do with your sequential code:

- Parallelizing your code (OpenMP, MPI etc)
- Use libraries that support multicore CPUs (as Lapack, MKL etc)
- Use 4th generation programming languages as Matlab, Scipy, R etc which have buildin libraries supporting multicore CPUs.

## OpenMP (Open Multi-processing).

OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran.

OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications e.g. laptops and supercomputers.

OpenMP is implemented in several Fortran and C/C++ compilers as GNU, IBM, Intel, Portland, Cray, HP, Microsoft etc.

The OpenMP is a SPMD – Single Program Multiple Data.
Each thread redundantly execute the same code.

This course will have focus on OpenMP 3.x

See http://openmp.org

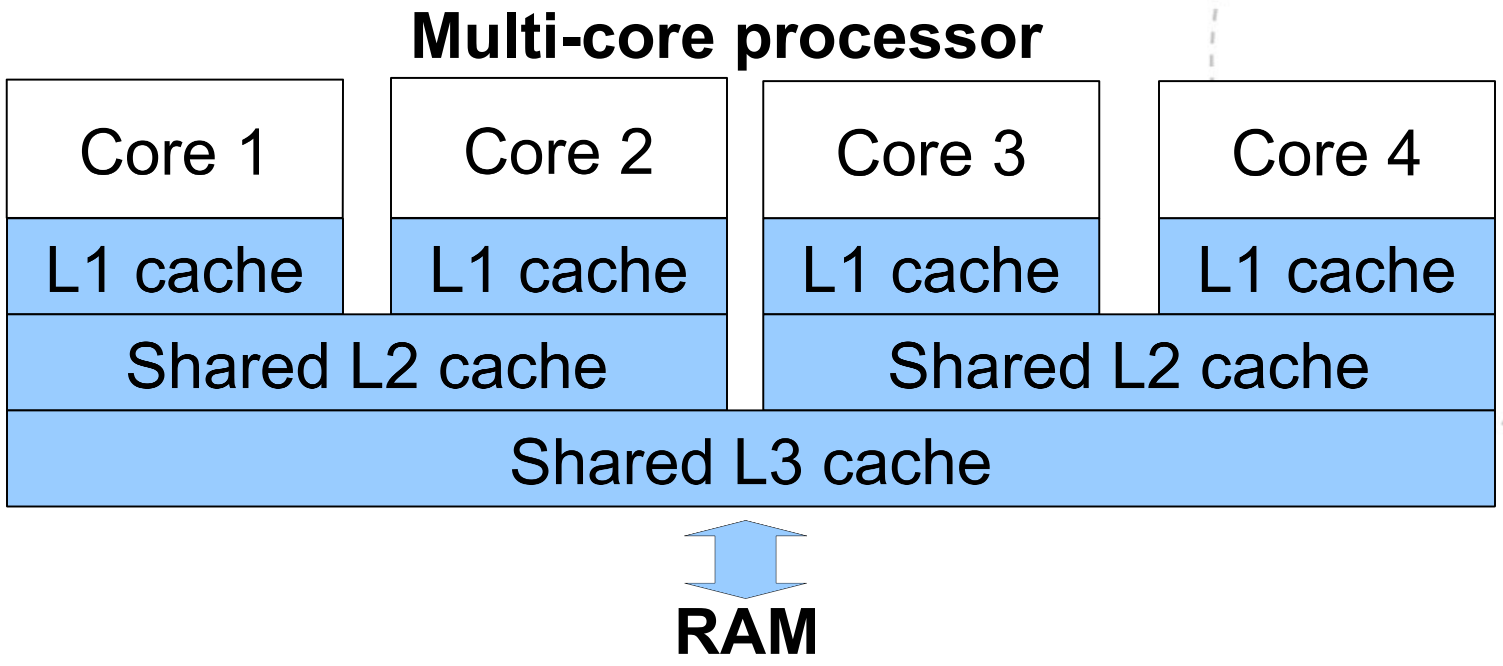


Figure 1. Intel i7 Sandy Bridge, 4 core processor with cache memory.
( L1: 64kB ~4cycles, L2: 256kB ~10cycles, L3: up to 20MB ~40cycles, RAM 32GB~120cycles)

- Each core run there own program block (thread), and simultaneously with the other cores.
- All cores share all the memory, and with fast memory access.
- All communication between the threads are via variables (shard memory).

NTNU – Trondheim
Norwegian University of Science and Technology

Supercomputers, clusters and PC/laptops today have processors with several cores, and with shared memory.

2, 4, 6 cores on PC processors are common today.

OpenMP support all this processors:
Intel Brodwell Server processor have up to 28 cores.
AMD Server processors have up to 64 cores.
Intel MIC processor have around 60 cores.
(MIC: Many Integrated Cores)
Nvidia/AMD GPUs have more than 3000 streaming cores.

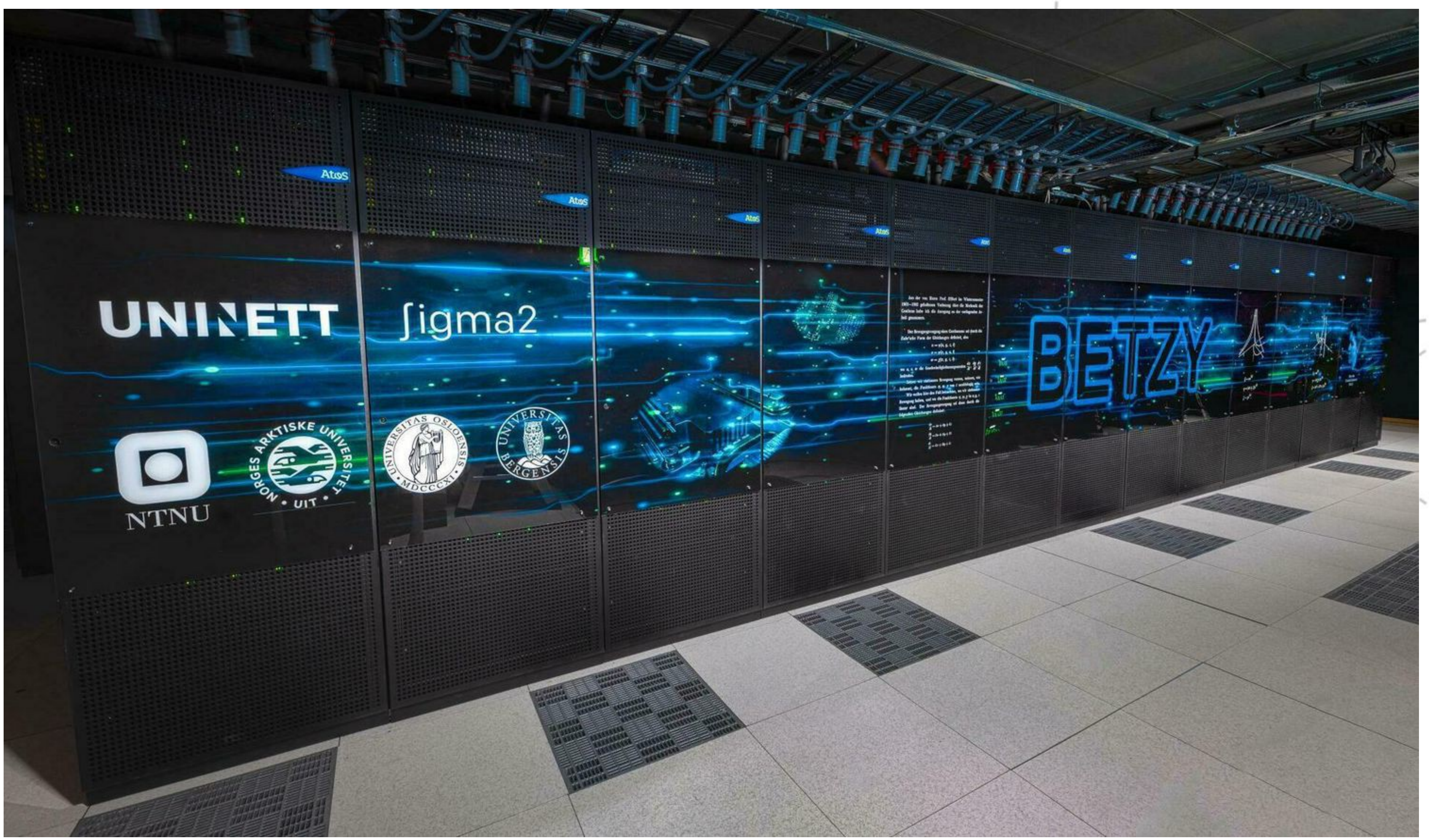
## National HPC systems:Betzy(Atos 2020)/Fram(Lenovo 2017)

| | Each Node | Total |
|---|---|---|
| Cores | 128 / 32 | 172032 / 32256 |
| Nodes | - | 1344 / 1006 |
| Memory | 256GB / 64GB | 336TB / 78TB |
| Storage | - | 2,5PB / 2.5PB |
| Flops | | 5.9Pflops / 1.1Pflops |

## Idun (Dell). Local NTNU

| | Each Node | Total |
|---|---|---|
| Cores | 20-48 | ~2000 |
| Nodes | - | ~80 |
| Memory | 128-768GB | |

NTNU – Trondheim
Norwegian University of
Science and Technology



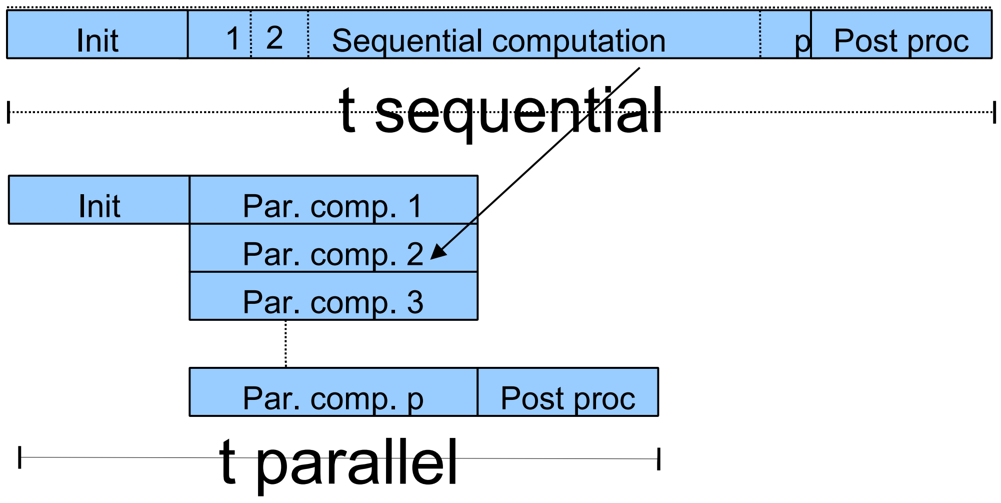
Betzy HPC computer.

NTNU – Trondheim
Norwegian University of
Science and Technology

## Parallel computation

A program can be split up to run on several processors that runs in parallel.

Sequential program:



Speedup

S = t sequential / t parallel

(t-sequential: Execution time for a single core/processor progam

t-parallel:Execution time for the multicore/multiprocessor program)

Speedup for p processors or cores:

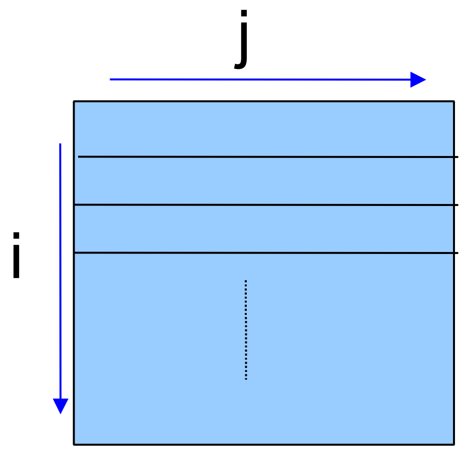
$S \leq p.$

## Example: Matrix calculation.

B = c * A, where A and B is mxn matrices and c is a constant

<u>Sequential computation:</u>

All computation is carry out on only one processor or core.

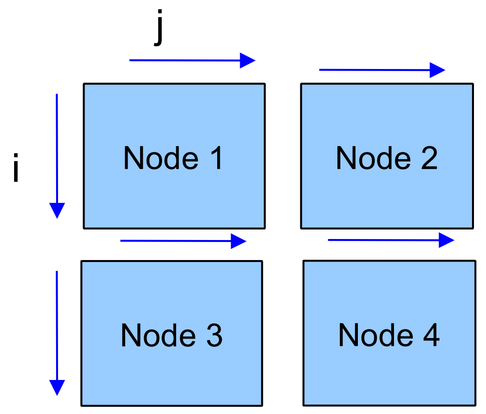


```
Program
Init the matrix A
for i = 1 to m
    for j = 1 to n
        B(i,j) = c * A(i,j)
```

Benefits: OK for small computation, fast memory access and none memory conflicts.
Drawback: Limited memory space (GB) and sequential computations.

## Parallel computation with MPI and cluster.

The matrix is split up and scattered to several computers/nodes which are interconnected to each other via IP, infinity band or other high performance serial link.



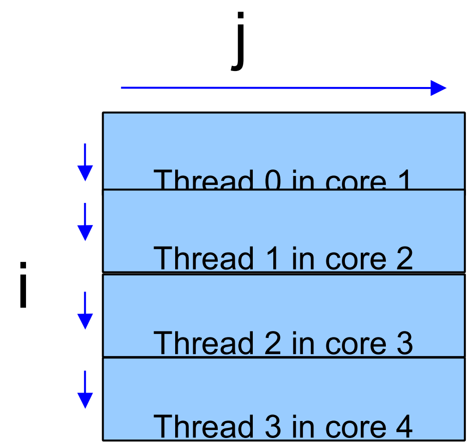
Program:
Master: Initialize the matrix.
Master split up and spread the matrix to all nodes.

```
For i = 1 to m_localnode
      for j = 1 to n_localnode
            localB(i,j) = c * localA(i,j)
```

Benefits: More memory space (TB) and parallel computation on each node.
Drawback: Communication latency between the nodes.

## Parallel computation with OpenMP and shared memory.

The matrix remains in the memory and each core/thread in the processor compute its part of the matrix in parallel.



Program

Set compiler directive for a parallel region.

```
 Parallel for i = 1 to m
      for j = 1 to n
            B(i,j) = c * A(i,j)
```

Benefits: Parallel computation and low communication latency between the cores.

Drawback: Small memory space (GB) and memory conflicts.

NTNU – Trondheim
Norwegian University of Science and Technology

## Tutorials

**Job scripts.**

The job scheduler distribute the job to the compute nodes.

The job script is a description to the scheduler and must contain number of nodes and cores, queue, account etc.

(Ex. helloworld_c.job is for c programs and _f.job for fortran prog.)

## Job schedulers (Slurm and PBS Pro)

Idun, Betzy, Saga and Fram have Slurm.
Example: Job scripts for running on 2 compute nodes

Slurm (Betzy/Fram/Saga/Idun)

```
#!/bin/bash
#SBATCH --job-name=myjob
#SBATCH --time=0:30:0
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH -c 32
#SBATCH --account=myaccount
module purge
module load OpenMPI/1.10.2-GCC-
4.9.3-2.25
srun ./my_program
```

Commands:
sbatch (submitt), squeue (status)

NTNU – Trondheim
Norwegian University of
Science and Technology

See more on www.hpc.ntnu.no and www.sigma2.no

## Editors

1. vi , vim or gvim (commands)

   ```
   $vi mytextfile.txt
   ```

   Write text: esc, insert button or i

   Save: esc, :w

   Quit: esc, :q

   Save and quit: esc, :wq

2. emacs (for window users)

   ```
   $emacs  filename
   ```

   You get a window

   Problem with fonts:

   ```
   emacs -fn 8x16
   ```

   Note! You have to log in as (-X):

   ```
   ssh -X vilje.hpc.ntnu.no
   ```

NTNU – Trondheim
Norwegian University of Science and Technology

**Tutorials Idun.** Some informations:

Login

```
ssh -X user@training.hpc.ntnu.no
```

Programs

Copy all files from /cluster/home/floan/tutorials/ to your home folder.

Commands (mkdir:make directory, cp:copy, cd:change directory):

On your home folder:

```
cp -r /cluster/home/floan/tutorials tutorials
cd tutorials
cd OpenMP_part1
```

(To copy a folder : cp -r myfolder1 myfolder2)

Slides:www.hpc.ntnu.no/display/hpc/Course+materials

Compile your program

**module load intel/2020b (only once)**

**make helloworld** ( or make forsin, make mult, make pi)

Run a job.

***Note! Do not start a job interactively*** (nice ./myprogram)

**sbatch helloworld_c.job**

**a**nd you get a job id.

(Note! There is a job script for each tutorials)

Check the queue status:

**squeue** or **squeue -u myusername**

Cancel the job

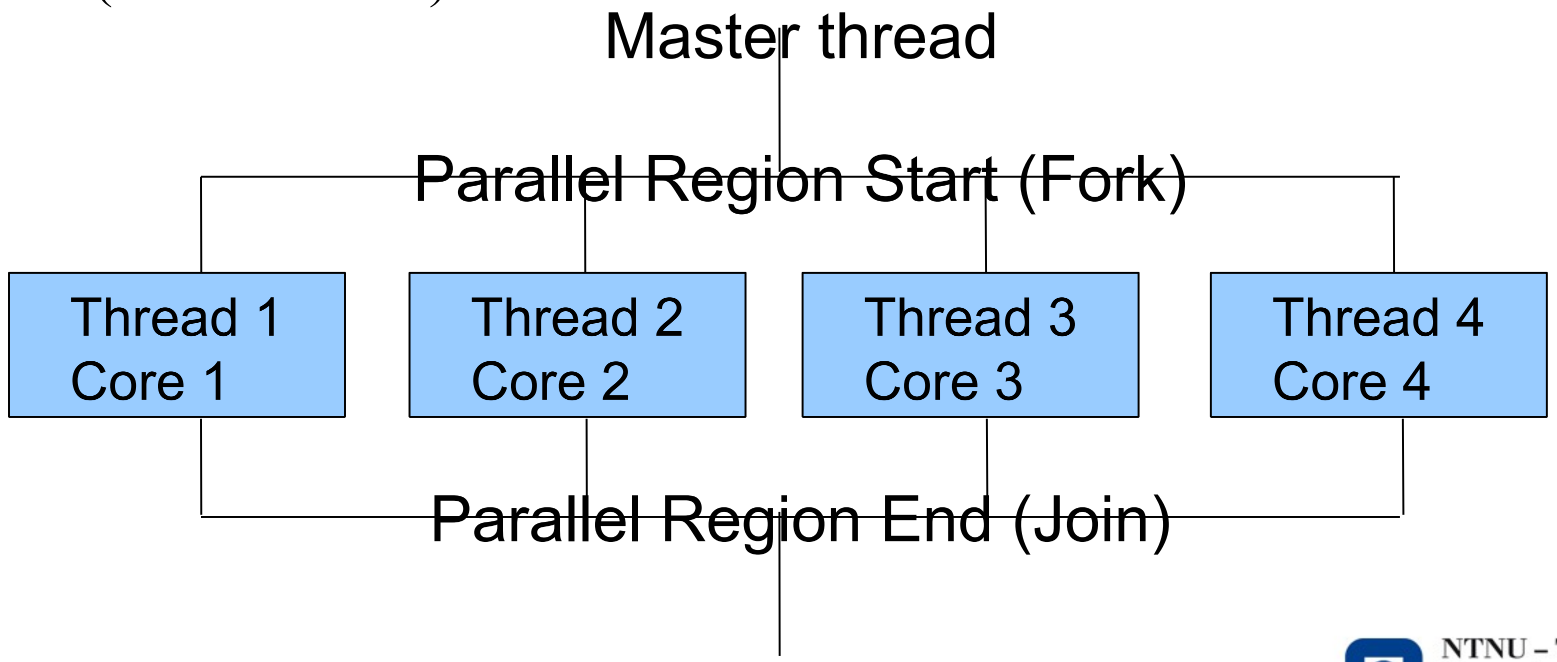
scancel jobid

NTNU – Trondheim
Norwegian University of Science and Technology

(Mac PC: If error when compiling, write: export LC_ALL=C (terminal setting(local))

## Tutorial 1. Parallel region. Thread creation.

A parallel region is the part of the program where program is spread in to several threads and core. Before and after a parallel region the program run on 1 thread (master thread). It is called fork when the program go from 1 thread to parallel region and join when the program go back to 1 thread (master thread).



All variables declared outside a parallel region are as default **shared**.

Example

C

```
int x;
//1 thread (Master thread)
 //Fork to several threads in parallel
x=0;
#pragma omp parallel
{
      // Variabel x is shared
      // between all threads.
       do_something_in_parallel(x);

}
//Join to 1 thread
              ....
```

Fortran

```
integer::x


x=0
!$OMP PARALLEL



do_something_in_parallel(x)

!$OMP END PARALLEL
```

## OpenMP

**Runtime library routines and environment variable.**

Important environment variable

```
OMP_NUM_THREADS
(export OMP_NUM_THREADS=8; ./myprogram)
```

This environment variable set the number of threads.

The default is number of cores.

OpenMP Runtime Library Routines

Some routines for testing and debugging.

```
omp_set_num_threads(n)   // Set n number of threads before
                         // a parallel region
omp_get_num_threads()// Get the number of OpenMP threads
                     // inside parallel region. Return Integer
omp_get_thread_num()     // Get the current thread number.
                         // Return integer
omp_get_wtime()        // Get wall clock time in seconds.
                       // Return double/real(8)
```

## Exercise a. Hello world.

Modify the sequential “Hello world” program to print out “Hello world from thread 1” , “.... thread 2”, “... thread 3” ..

C

```
int main()
{
    printf(“Hello world \n”);
}
```

Fortran

```
program helloworld

write(6,*) 'Hello world'
end program helloworld
```

-Compile your program: make helloworld

-Execute the batch job:

sbatch helloworld_c.sh (C) or sbatch helloworld_f.sh (Fortran)

-Open the output file helloworld_xxxxxxxx.sh.oxxxx

NTNU – Trondheim
Norwegian University of Science and Technology

## Synchronization: Barrier.

| C | Fortran |
|---|---|
| #pragma omp barrier | !$OMP BARRIER |

Each thread waits until all threads arrive.

**Master construction.**
The master construct specifies a structured block that is executed by a master thread of a team. There are no implemented barrier either on entry to, or exit from, the master construction.

```
        #pragma omp master            !$OMP MASTER
```

**Single construction**
The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread). A <u>barrier</u> is implemented at the end of the single block.

```
    #pragma omp single                !$OMP SINGLE
    (Example ex_barrier.c)
```

## Example Barrier and Master

C

```
#pragma omp parallel
{
    do_many_things_in_parallel();
    //All threads wait here until all arrives.
    #pragma omp barrier
    #pragma omp master
    {   // Only the master thread
        // will call this function:
        post_processing ( );
    }
    //All threads wait here until all arrives.
    #pragma omp barrier
    do_many_other_things_in_parallel();
}
```

Fortran

```
!$OMP PARALLEL

do_many.....()


!$OMP BARRIER
!$OMP MASTER

post_processing ( )

!$OMP END MASTER

!$OMP BARRIER
do_many....()
!$OMP END PARALLEL
```

## Example Barrier and Single

C

```
#pragma omp parallel
{
    do_many_things_in_parallel();
    //All threads wait here until all arrives.
    #pragma omp barrier

    #pragma omp single
    {   // Only one thread
        // will call this function:
        post_processing ( );
    }

    do_many_other_things_in_parallel();
}
```

Fortran

```
!$OMP PARALLEL

do_many.....()

!$OMP BARRIER

!$OMP SINGLE

post_processing()

!$OMP END SINGLE

do_many....()
!$OMP END PARALLEL
```

## Exercise B. Use OpenMP runtime library routines and Synchronization.

Modify your program Hello World to print out number of threads and thread no like this:.

"Number of threads: 16" (Always first)

"Hello world from thread 1"

## Tutorial 2. Parallel for/do loop and data sharing.

OpenMP automatically split up the for-loop to several threads and send a copy of the block to each core. This construction is called worksharing, and shall be initialize as this:

```
    C                                  Fortran
#pragma omp parallel for                  !$omp parallel do
    for (i=0 ; i<n ; i++)                 do i=0 , n
      do_someting();                          do_something()
                                          end do
                                          !$omp end parallel do
```

It is also allowed to initialize the for/do loop as:

```
    #pragma omp parallel                  !$omp parallel
    {
    #pragma omp for                       !$omp do
        for (i=0;.....                    do i=0, n
              ........                    .....
```

NTNU – Trondheim
Norwegian University of
Science and Technology

**Example: 4 threads and n=40.**

OpenMP divide the for/do loop into chunks
and the chunk size is 10 in this case.

```
    #pragma omp parallel for              !$omp paralle do
    for( i=1 ; i<=n ; i++ )               do i=1,n
```

| Thread 1 | Thread 2 | Thread 3 | Thread 4 |
|---|---|---|---|
| for i=1 to 10 | for i=11 to 20 | for i=21 to 30 | for i=31 to 40 |
| … | … | … | … |

Note! It is important that the parallel for/do loop is iterational independent.

That means; one iteration is independent of the iteration before.

Parallel loop iterations are not in sequential order.

```
#pragma omp parallel for
 for ( i=1 ; i<n ; i++)
     X[i]=X[i-1] + X[i+1];   //This will give wrong result
```

NTNU – Trondheim
Norwegian University of
Science and Technology

## Exercise parallel for loop  forsin

Measure the execution time for the sequential code.
Modify the program “forsin” with parallel for/do-loop
Measure the execution time for the parallel program
and calculate the speedup. (Note! Try larger n)

Compile: make forsin
Run: sbatch forsin_c.job    or   sbatch forsin_f.job

## Data sharing: Shared, private and firstprivate clause.

All variables declared outside a parallel region is shared inside the parallel region as default.

Note! The for/do iterator (e.g. “i”) is set to private/local inside the parallel region.

### Shared

Variables are shared inside a parallel region.

### Private

Variables are private inside the parallel region, but the variable has no value.

## Example 1. Private.

```
   int i;
   int n=1000;
   double tmp=0;
   ......
#pragma omp parallel for private (tmp)
   for(i=0;i<n;i++)
  {   //Tmp is local
      tmp = check(A[i]);
      if (tmp > 0)
          A[i] = tmp;

   }
   .......
```

```
integer::i,n
real::tmp
n=1000
tmp=0
!$omp parallel do private(tmp)
do i=1 , n

       tmp = check(A(i))
       if (tmp > 0)
              A(i) = tmp
       end if
end do
!$omp end parallel do
```

....

NTNU – Trondheim
Norwegian University of
Science and Technology

Firstprivate

Public variables can be set to be private inside the parallel region and initialize its value with the corresponding value from the master thread.

Private and firstprivate/private for arrays

Note that using arrays as firstprivate/private will copy the whole arrays to cache multiply with 20 (Idun:one each core) and may cause segmentation fault if the array is to big.

#pragma omp parallel for fistprivate (A,B)

## Example 2. Firstprivate.

C

```
int main ()
{
        int a=0, b=1;
        int i;

                .....
#pragma omp parallel for firstprivate(a,b)

        for (i=0;i<16;i++)
        {
                A[i] = func(a) + func(b)
                a++;
                b++;
        }

}//End main
```

Fortran

```
program tut3ex2

integer::a,b,i
a = 1
b = 0

.....
!$omp parallel do firstprivate(a,b)
        do i=1,16
                A(i) = func(a) + func(b)
                a = a +1
                b = b+1
        end do
!$end omp parallel do
end program tut3ex1
```

## Shared arrays

Shared arrays will be automatically load balanced to each core.
Exampel: 4 cores and n=40000

```
double A[n],B[n];
#pragma omp parallel for private (i,j)
for (i=0;i<n;i++)
   B[i] = c * A[i]
```

| | | | |
|---|---|---|---|
| Core 1<br>A [00000 .. 09999]<br>B [00000 .. 09999] | Core 2<br>A [10000 .. 19999]<br>B [10000 .. 19999] | Core 3<br>A [20000 .. 29999]<br>B [20000 .. 29999] | Core 4<br>A [30000 .. 39999]<br>B [30000 .. 39999] |

**Exercise b. Matrix multiplication. C=AB,**

Measure the execution time for the sequential code.
Modify the program “**mult**” with parallel for/do-loop.
Measure the execution time again and calculate the speedup.
Compile: make mult (Try with size=1000,2000 and 3000)

## Tutorial 3. Synchronization: Critical and Atomic directives.

The OpenMP do not protect a variable or a region as default. If several threads shall update same variable in same time, the result can be that one thread do not update the variable and cause wrong results of the calculation.

Critical:

Critical provides mutual exclusion: Only one thread at time can enter a critical region. Example:

| C | Fortran |
|---|---|
| `#pragma omp critical` | `!$omp critical` |
| `    calculate(B,n);` | `calculate(B,n)` |

Atomic:

Atomic provides mutual exclusion but only applies to the update of a memory location. Example:

| C | Fortran |
|---|---|
| `#pragma omp atomic` | `!$omp atomic` |
| `x += tmp;` | `x = x + tmp` |

## Exercise.

Calculation of Π (3.14159265358979...).
To calculate pi we can use this formula

$$\int_0^1 \frac{4.0}{1+x^2}\,dx = \Pi$$

Create a parallel version of the pi.c or pi.f90.
-make pi

Calculate the speedup S (Measure execution time before and after including OpenMP).
Change the value of nsteps and number of threads.

## Tutorial 4. Reduction.

The OpenMP reduction clause:
Reduction (op:list)

A local copy of each list variable is made and initialized depending on the operator "op" (ex "+").

Compiler finds standard reduction expressions containing "op" and uses them to update the local copy

Local copy are reduced into a single value and combined with the original global value.

## Example Average

C

```
double ave=0;
double A[n];
int i;

put_something_in (A);
#pragma omp parallel for reduction (+:ave)
for (i=0 ; i < n ; i++)
        ave += A[i];
ave = ave / n;
```

Fortran

```
real :: ave
real, dimension (n)::A
integer :: i
ave = 0
put_something_in (A)

!$omp parallel do reduction (+:ave)
do i = 1, n
    ave = ave + A(i)
end do
!$omp end parallel do
ave = ave / n
```

## Different reduction operators:

| C/C++ | Fortran |
|---|---|
| + | + |
| * | * |
| - | - |
| & | / |
| \| | .AND. |
| ^ | .OR. |
| && | .EQV. |
| \|\| | .NEQV. |
| max | iand |
| min | ior , ieor |
|  | max and min |

## Exercise.

Modify your pi program with reduction.

Calculate the speedup. (Measure execution time before and after including OpenMp)

**Data sharing**.

- You can change storage attributes for constructs using following clauses as
  - **shared**, **private** and **firstprivate**

This clauses can also be used for

parallel region, section, tasks, single

constructs.

Ex.

```
double Array[n];
double x=0,y=0;
double tmp;
#pragma omp parallel shared (Array) private(tmp)
  firstprivate(x,y)
{ … }
```

NTNU – Trondheim
Norwegian University of
Science and Technology

## **Data sharing (continue):**

Clause: Lastprivate.

The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop.

Note that the value of sum is the value for the last iteration.

Ex. (4 threads)

```
int sum=0;
#pragma omp parallel for firstprivate (sum) lastprivate(sum)
for (i=0;i<8;i++)
        sum++;  // sum=sum+1

printf("sum %d\n",sum);
```

The sum outside parallel region is 2.

With private and fistprivate; the sum is 0.

NTNU – Trondheim
Norwegian University of Science and Technology

## Data sharing(Continue):

<u>The default attribute.</u>

The default attribute can be overridden with

**Default ( private | shared | none)**

Note that default (private) is for fortran only.

-default(none) means that you have to set all variables shared,  private or first private.

-Parallel region is shared as default

-Parallel for/do loop is shared as default (except the iterator)

-Task is firstprivate as default.

## Data sharing (continue):

### Default attribute example:

This two examples are internal equivalent:

1) #pragma omp parallel { ... }

```
#pragma omp parallel default (shared) { ... }
```

2) int n=100;

```
int x,each;
 #pragma omp parallel private(x,each)
{ each = x / n; }
#pragma omp parallel default(none) shared (n) private(x,each)
{ each = x / n; }
```

Only for Fortran:

```
!$omp parallel default (private) shared (n)
   each = x / n
!$omp end parallel
```

NTNU – Trondheim
Norwegian University of
Science and Technology

## Loop worksharing constructs

Schedule clause:

**#pragma omp paralle for schedule (static | dynamic | guided, chunk_size)**

The schedule clause affects how loop iteration are mapped onto threads:

- **schedule (static, [chunk])**

Deal out blocks of iteration of size "chunk" to each thread

Example (4 threads):

```
#pragma omp parallel for schedule (static,3)
for (i=0;i<10;i++) ....
```

| Iteration i: | | |
|---|---|---|
| | Thread 0: | 0 , 1 , 2 |
| | Thread 1: | 3 , 4 , 5 |
| | Thread 2: | 6 , 7 , 8 |
| | Thread 3: | 9 |

The interation follow the thread order

NTNU – Trondheim
Norwegian University of
Science and Technology

## schedule (dynamic,[chunk])

Each thread grabs “chunk” iterations off a queue until iteration have been handled.

Example (4 threads):

```
#pragma omp parallel for schedule (dynamic,3)
for (i=0;i<10;i++) ....
```

| | | |
|---|---|---|
| Iteration i: | Thread 0: | 3 , 4 , 5 |
| | Thread 1: | 0 , 1 , 2 |
| | Thread 2: | 6 , 7 , 8 |
| | Thread 3: | 9 |

The interation DO NOT follow the thread order

## schedule(guided[,chunk])

Threads dynamically grab blocks of iterations.

The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.

Example (4 threads):

```
#pragma omp parallel for schedule (guided,1)
for (i=0;i<10;i++) ....
```

| | | |
|---|---|---|
| Iteration i: | Thread 0: | 0 , 1 , 2 , 3 |
| | Thread 2: | 4 , 5 , 6 |
| | Thread 1: | 7 , 8 |
| | Thread 3: | 9 |

The interation DO NOT follow the thread order

## schedule(runtime)

Schedule and chunk size taken from the OMP_SCHEDULE environment variable

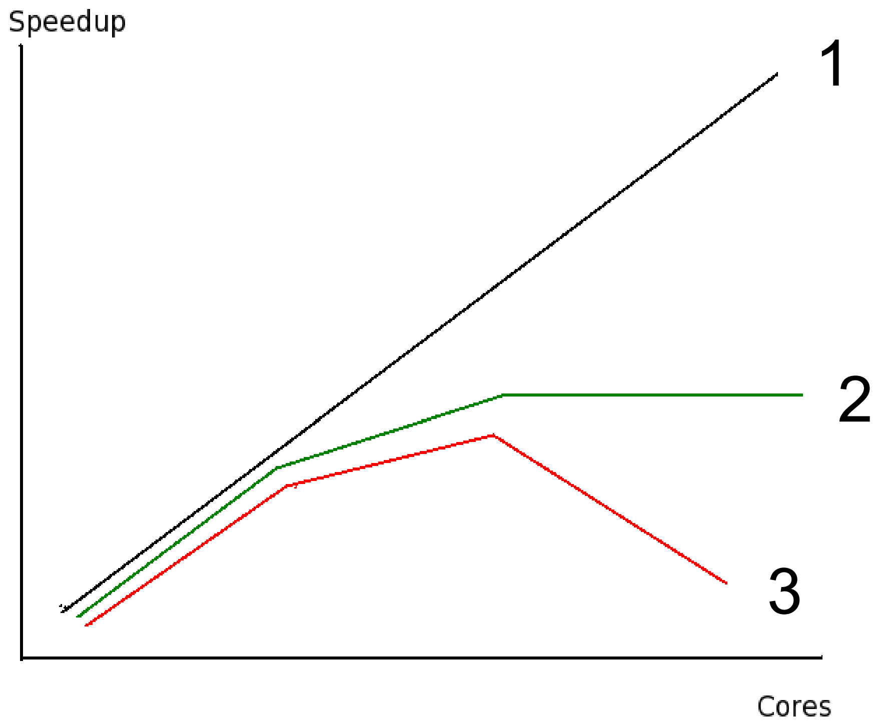
NTNU – Trondheim
Norwegian University of
Science and Technology

## Collapse Clause

The exampel, the i and j loops are collapsed into one loop with larger iteration space. Collapse clause set the iterators to private.

Exampel

```
#pragma omp parallel for collapse(2) private(i,j)
for (i=0;i<n;i++)
    for (j=0;j<m;j++)
       C[ i ][ j ] = A[ i ][ j ] * B[ i ][ j ];
```



Typical speedup performance

1. The program scale
2. Part of the program can not be/or is not parallelized
3. Typical memory conflict (eg. use of atomic)

NTNU – Trondheim
Norwegian University of
Science and Technology