# Parallel programming introduction

Ole W. Saastad, Dr.Scient
USIT / SUF / VD

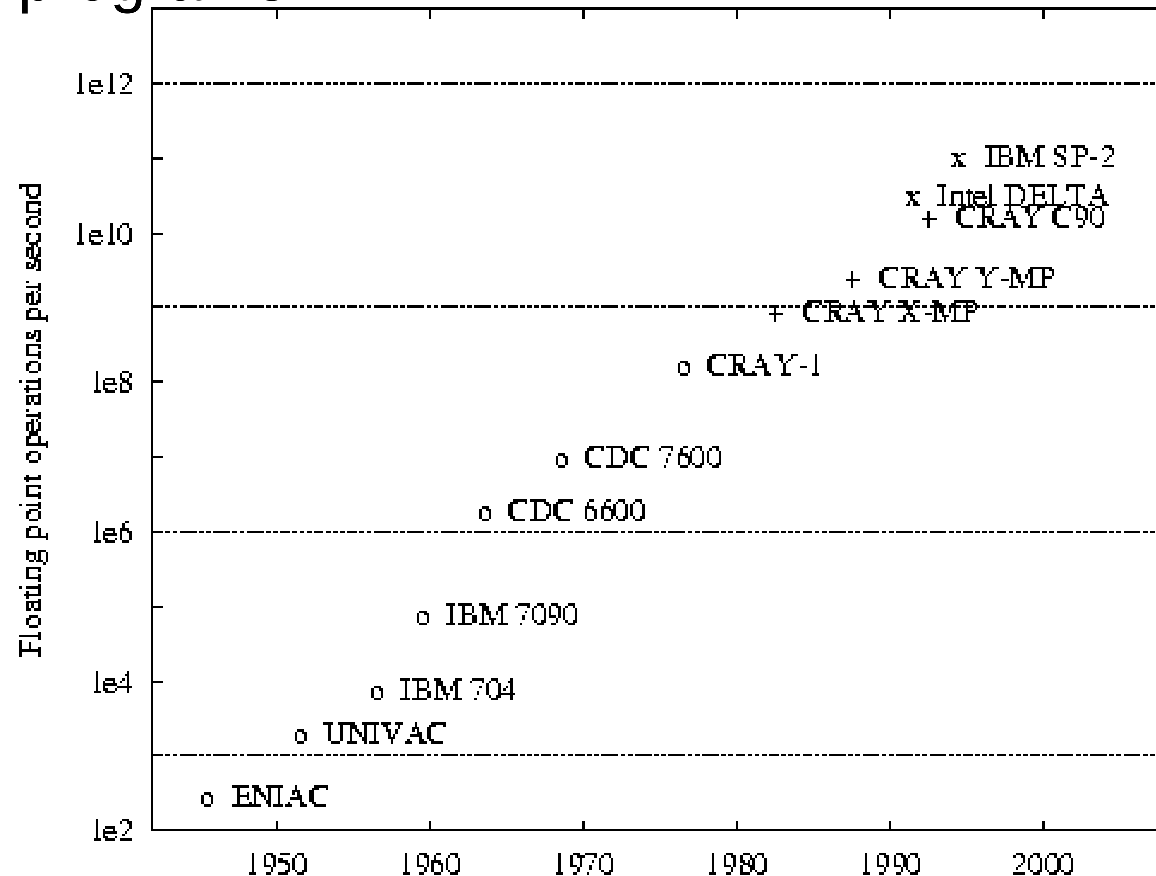
UNIVERSITETET I OSLO

# Background

Before the devlopment of faster and faster single cpus masked the need for multicpu programs.

Milestones (flops):

CDC6600 (1964) 3 M
Star 100 (1971) : 100 M
Cray-1 (1976) : 80(140) M
Cray-YMP (1988) : 333 M
Cray-T90 (1995) : 1.8  G
NEX SX-6 (2001) : 8 G
Pentum 3.8 (2006) : 7 G
Interlagos (2011) : 220 G



# Background

Just after 2000 each single CPU did not become any faster.

Mores law did not keep up with performance.
It is about transistor density, not performance.

Performance could not be bought by buying faster CPUs.

Parallel programming became a necessity.

# Background

2006 quad core
- 4 full blown cores on each prosessor chip

2009 hexacore
- 6 fullverdige CPUer på hver prosessorchip.
- 4 slike i en enkel 4 veis server gir 24 cores.

2012 MIC
- very many cores 64-80 simple x86-64 cores

Graphic processors have 320, 640 or 800 small cores
- Many more the future....

UNIVERSITETET
I OSLO

# Background – today and tomorrow

Abundance of processing capacity !
Computing if generally for free!

Two cores even on the mobile, four cores on laptops.

In the future even more cores, 100+ or even a million cores for large systems.

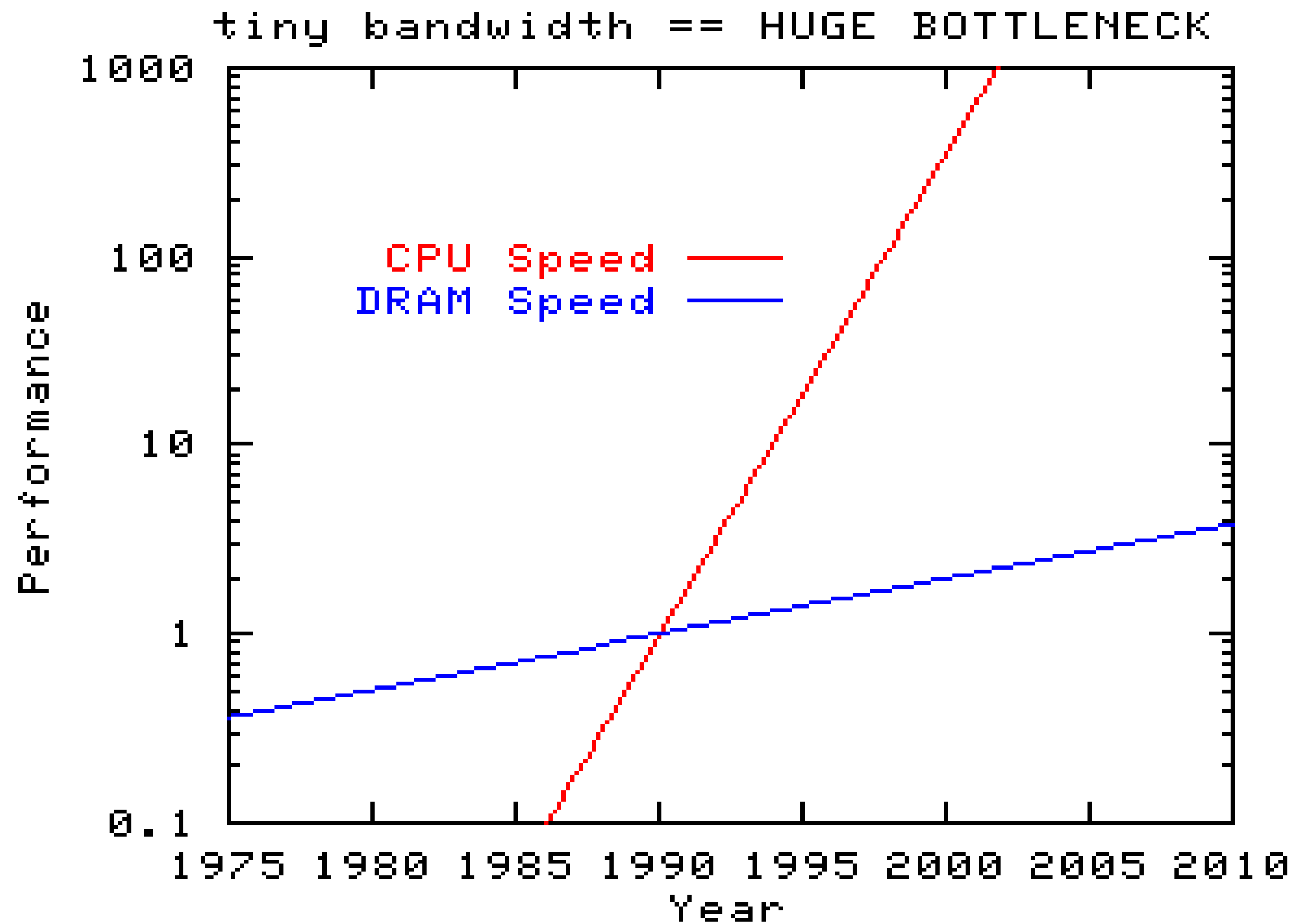
Computing is not the problem !

# Background – today and tomorrow

Abundance of processing capacity

- Memory bandwidth has not grown with the same rate
- Memory access time about constant
- IO subsystem performance have not increased to keep up with processing capacity

# Background – memory bandwidth



# We are forced to program in parallel !

# We have no choice.

To utilize many cores is a challenge!

Suggestions ?

Strategies ?

# 3 common methods

1. Posix Threads

Shared memory

2. OpenMP (Open Multi Processing)

Shared memory

3. MPI (Message Passing Interface)

Distributed memory

4. Master slave methods like LINDA and PVM (Parallel Virtual Machine)

# Scaling – Amdahl's law

Amdahls law, formulated in 1967

$$\frac{1}{((1-P)+\frac{P}{S})}$$

P is the fraction of the program that show a speedup S

What does it mean ?

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

P parallel fraction
N number of cores

# Scaling – Amdahl's law

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

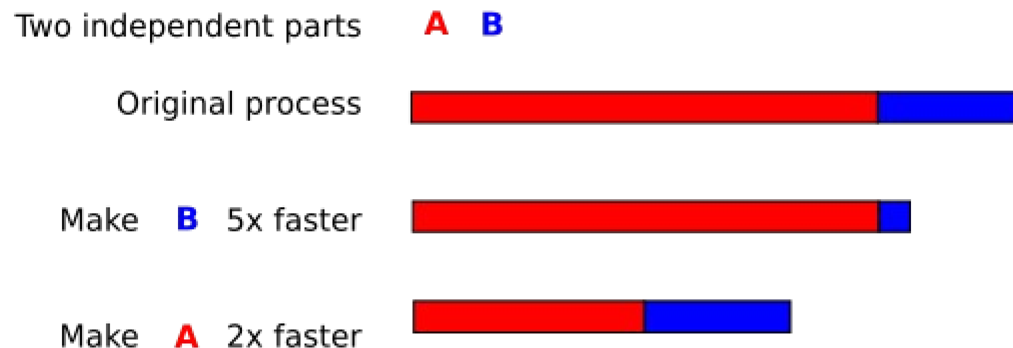
P fraction of program that is parallel
N number of processors/cores/cpus

Setting N very large.

$$\frac{1}{(1-P)}$$

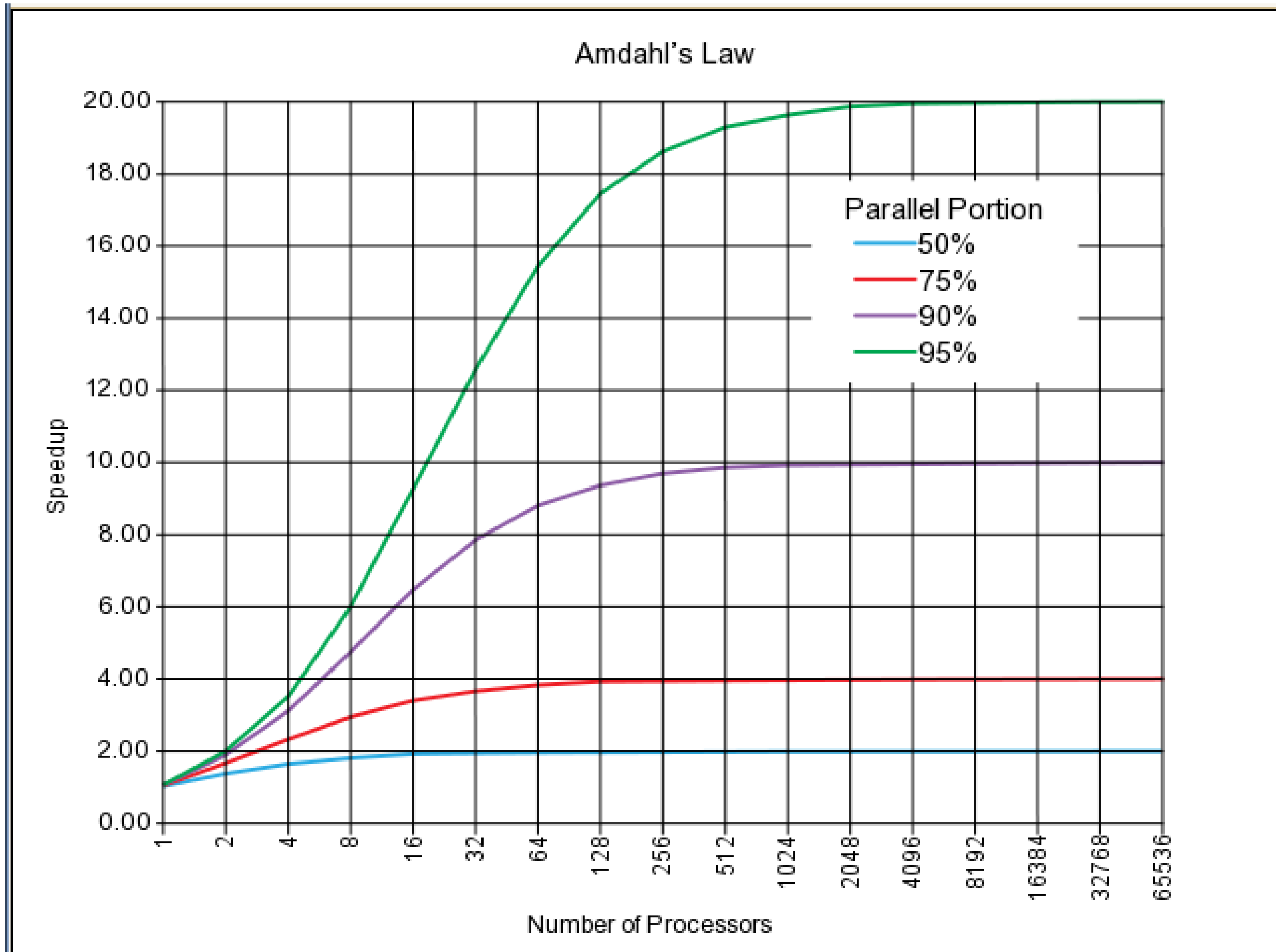
No matter how many processors we use the run time cannot be shorter than the serial part.

Which part to work on?



UNIVERSITETET I OSLO

# Amdahl's law



# Scaling – Gustafson's law

Gustafsons law from 1988 :
*any sufficiently large problem can be efficiently parallelized.*

$$S(P) = P - \alpha * (P - 1)$$

S is speedup, P number of cores og $\alpha$ fraction of parallel code

Big problems might scale well, large machine for large problems !

# Analogy using cars :

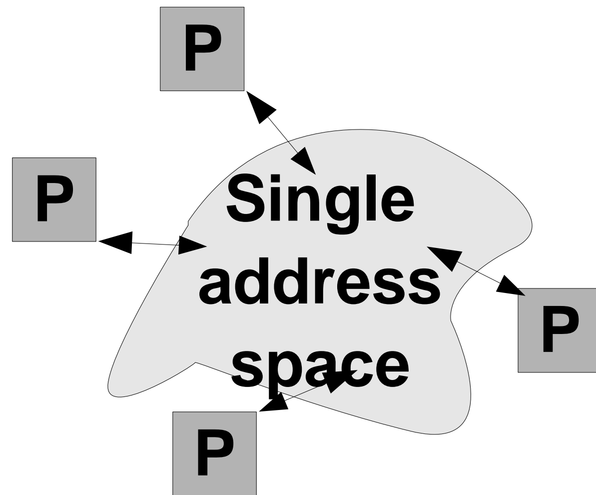
Amdahl's Law:

Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph.

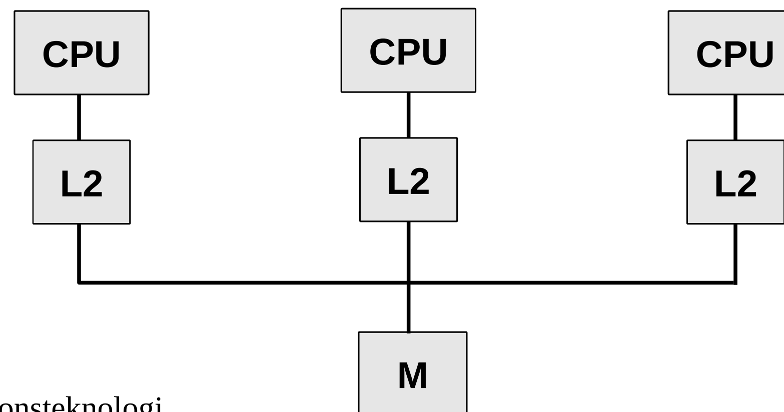
Gustafson's Law:

Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, if the car spent one hour at 30 mph, it could achieve this by driving at 120 mph for two additional hours, or at 150 mph for an hour, and so on.

# Shared memory




- Tthreads communicate by memory
- Just-in-time-ness av cache subsystem
  - delay incurred by every cache miss
  - latency hiding techniques
    - multitasking/threading
    - prefetching



# Posix Threads

Single memory image, «shared memory» programming.

A thread is a small process.

You need to do it all youself, syncing and semaphores etc.

Litle usage for scientific programming using C & Fortran.

Much used in libraries.

# OpenMP

«shared memory» programming.

OpenMP very common

Easy to start using.

Compiler directives, inserted as comment

Scale to a few handfulls or more cores.

# MPI

Distributed memory

MPI very common

A high first step to start using.

Need dedicated library

Can scale to a very very large core count if done well.

Demanding a high performance interconnect to perform well

# OpenMP

OpenMP standard with most compilers

- OpenMP version 3
- www.openmp.org

Versions for C & Fortran widely used

Loop or region parallel

UNIVERSITETET
I OSLO

# OpenMP

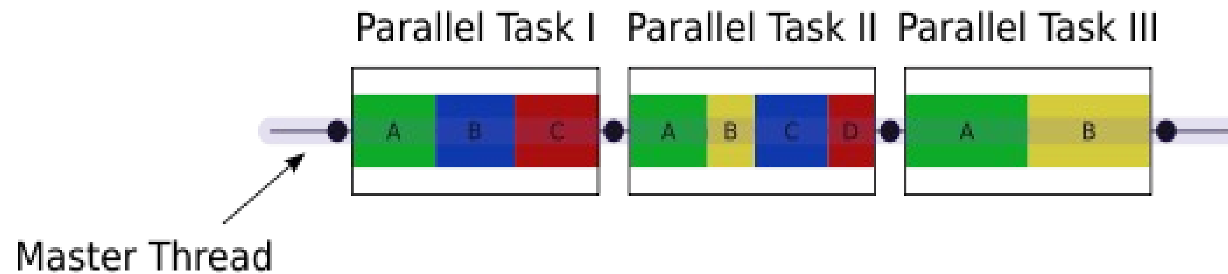
Parallel regions

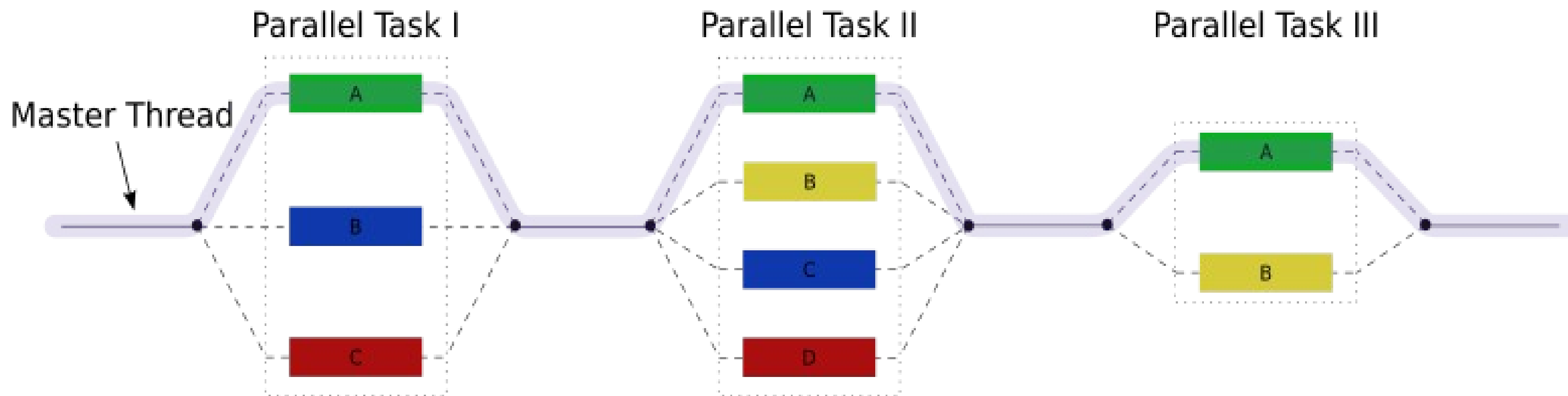
A main function generating Posix threads

OpenMP admin the the threads

OpenMP takes out all the hassle with POSIX threads

# OpenMP – multiple threads



# OpenMP directives

Parallel regions

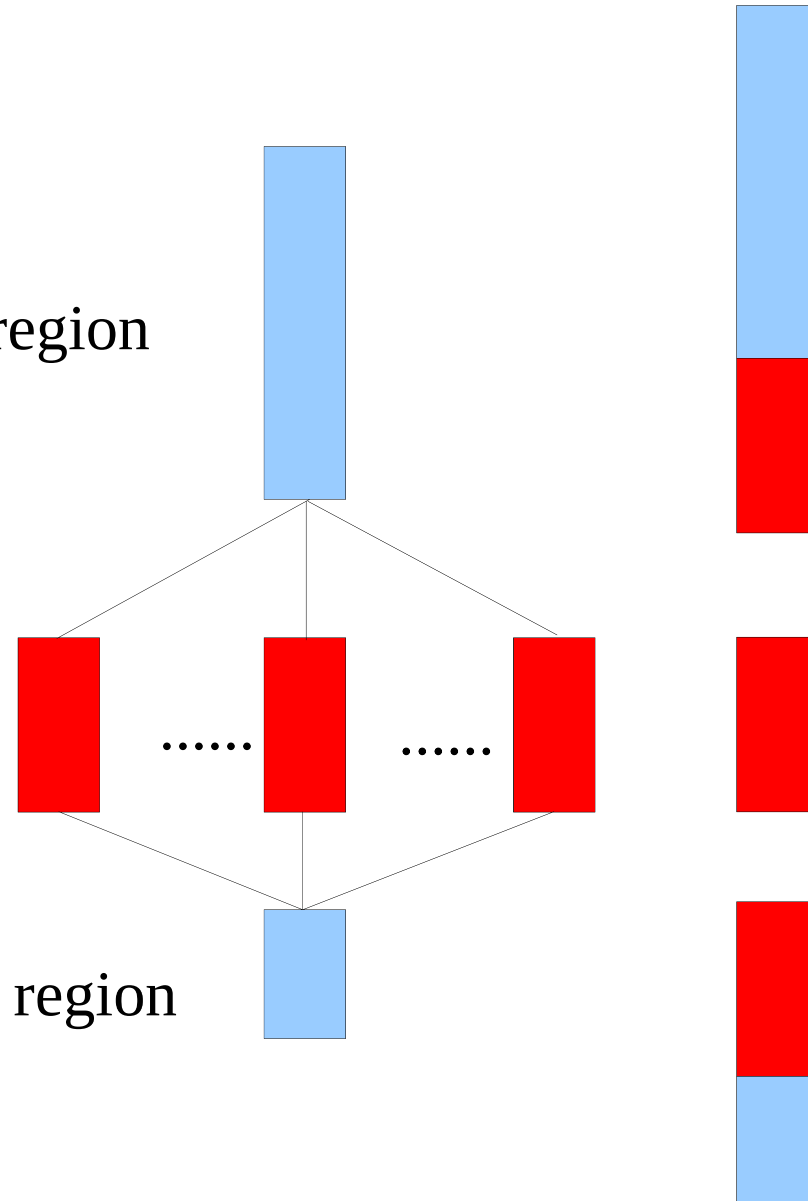
```
!$OMP PARALLEL
<Your parallel code here>
!$OMP END PARALLEL
```

Parallel loops

```
!$OMP PARALLEL DO
<Your parallel do loop here>
!$OMP END PARALLEL DO
```

# OpenMPI eksempel

```
program calc_pi program calc_pi
implicit none
integer, parameter :: wp = selected_real_kind (12)
real (wp) :: f, w, sum, pi, a, x
integer :: i, n

f(a) = 4.0_wp / (1.0_wp + a * a)

read (5,*) n
w = 1.0_wp / real(n, wp)
sum = 0.0_wp

!$OMP PARALLEL PRIVATE(x,i), SHARED(w,n) &
!$OMP REDUCTION(+:sum)
!$OMP DO
do i = 1, n
   x = w * (real(i, wp) - 0.5_wp)
   sum = sum + f(x)
end do
!$OMP END DO
!$OMP END PARALLEL

pi = w * sum
write (6,*) 'pi =', pi

stop
```

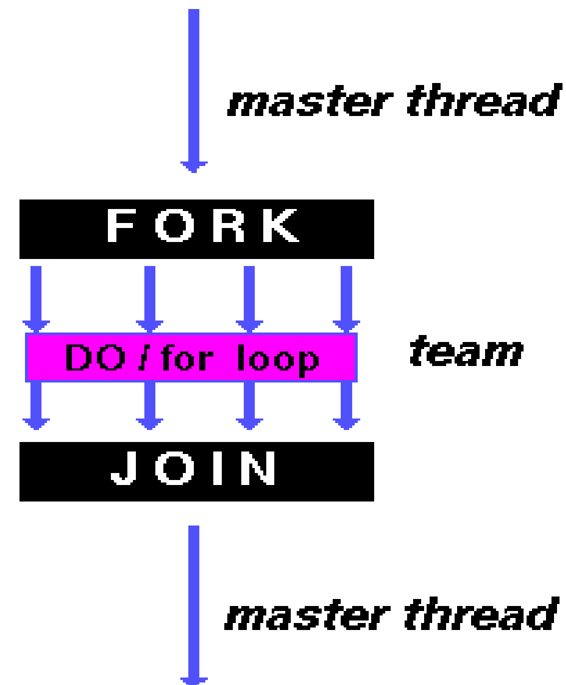


# Work-Sharing Constructs

DO / for - shares iterations of a loop across the team
Represents a type of "data parallelism".

Most common for Fortran kode

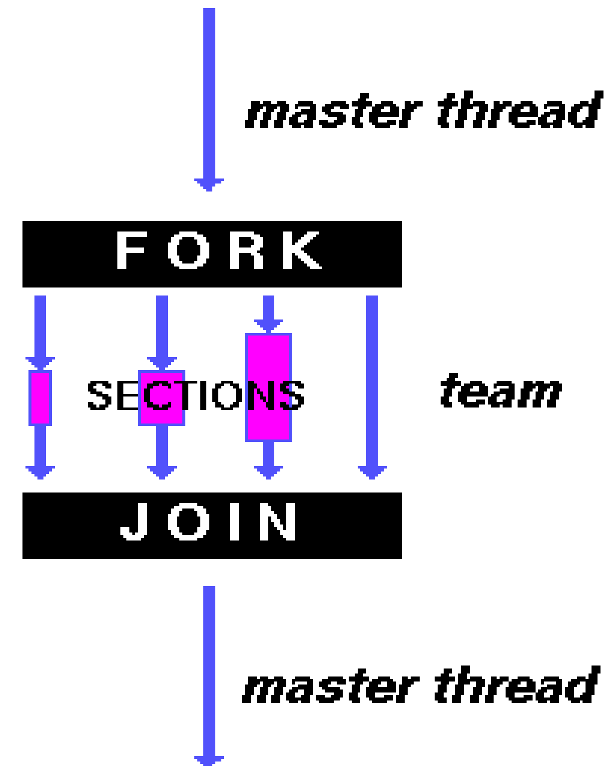
You can good workshare using
Data parallel construction



# Work-Sharing Constructs

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

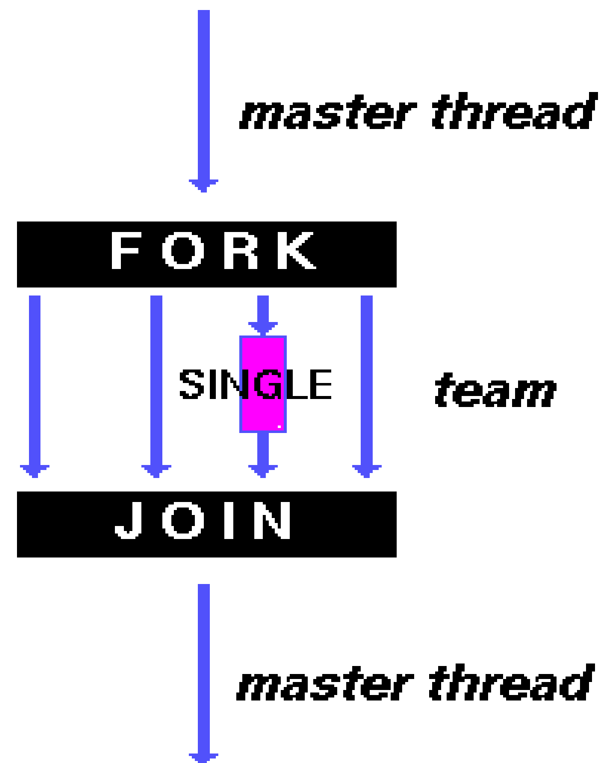
Not always easy to load balance



# Work-Sharing Constructs

SINGLE - serializes a section of code

Some times impossible to it
in parallel



# Fortran - Parallel Construct

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

```
!$OMP PARALLEL [clause ...]
            IF (scalar_logical_expression)
            PRIVATE (list)
            SHARED (list)
            DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
            REDUCTION (operator: list)

  block of parallel code

!$OMP END PARALLEL
```

# Fortran - Parallel Region Example

```
      PROGRAM HELLO

       INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
      +   OMP_GET_THREAD_NUM

C     Fork a team of threads with each thread having a private TID variable
!$OMP PARALLEL PRIVATE(TID)

C     Obtain and print thread id
       TID = OMP_GET_THREAD_NUM()
       PRINT *, 'Hello World from thread = ', TID

C     Only master thread does this
       IF (TID .EQ. 0) THEN
         NTHREADS = OMP_GET_NUM_THREADS()
         PRINT *, 'Number of threads = ', NTHREADS
       END IF

C     All threads join master thread and disband
!$OMP END PARALLEL

       END
```

# Fortran - Parallel Region Example

```
      PROGRAM VEC_ADD_DO

       INTEGER N, CHUNKSIZE, CHUNK, I
       PARAMETER (N=1000)
       PARAMETER (CHUNKSIZE=100)
       REAL A(N), B(N), C(N)

!      Some initializations
       DO I = 1, N
         A(I) = I * 1.0
         B(I) = A(I)
       ENDDO
       CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)

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

!$OMP END PARALLEL

       END
```

# Fortran - DO / for Directive

Mostly used construct Fortran

```
!$OMP DO [clause ...]
      SCHEDULE (type [,chunk])
      ORDERED
      PRIVATE (list)
      FIRSTPRIVATE (list)
      LASTPRIVATE (list)
      SHARED (list)
      REDUCTION (operator | intrinsic : list)
      COLLAPSE (n)

  do_loop

!$OMP END DO  [ NOWAIT ]
```

# Fortran - Parallel do loop

```
PARAMETER (CHUNKSIZE=100)
    REAL A(N), B(N), C(N)

!    Some initializations
     DO I = 1, N
       A(I) = I * 1.0
       B(I) = A(I)
     ENDDO
     CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
     DO I = 1, N
        C(I) = A(I) + B(I)
     ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

     END
```

# Fortran - TASK Construct

Task is a small piece of work that can be done indepent

```
!$OMP TASK [clause ...]
          IF (scalar expression)
          UNTIED
          DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
          PRIVATE (list)
          FIRSTPRIVATE (list)
          SHARED (list)

  block

!$OMP END TASK
```

You need to keep track of syncronising of variables.

# Critical regions - master

Only master can do this operation.

```
!$OMP MASTER
   block
!$OMP END MASTER
```

# Fortran - REDUCTION Clause Example

```
      PROGRAM DOT_PRODUCT
       INTEGER N, CHUNKSIZE, CHUNK, I
       PARAMETER (N=100)
       PARAMETER (CHUNKSIZE=10)
       REAL A(N), B(N), RESULT
!      Some initializations
       DO I = 1, N
         A(I) = I * 1.0
         B(I) = I * 2.0
       ENDDO
       RESULT= 0.0
       CHUNK = CHUNKSIZE
!$OMP  PARALLEL DO
!$OMP& DEFAULT(SHARED) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)
!$OMP& REDUCTION(+:RESULT)

       DO I = 1, N
         RESULT = RESULT + (A(I) * B(I))
       ENDDO

!$OMP  END PARALLEL DO NOWAIT
       PRINT *, 'Final Result= ', RESULT
       END
```

| Clause | Directive | | | | | |
|---|---|---|---|---|---|---|
| | PARALLEL | DO/for | SECTIONS | SINGLE | PARALLEL DO/for | PARALLEL SECTIONS |
| IF | ● | | | | ● | ● |
| PRIVATE | ● | ● | ● | ● | ● | ● |
| SHARED | ● | ● | | | ● | ● |
| DEFAULT | ● | | | | ● | ● |
| FIRSTPRIVATE | ● | ● | ● | ● | ● | ● |
| LASTPRIVATE | | ● | ● | | ● | ● |
| REDUCTION | ● | ● | ● | | ● | ● |
| COPYIN | ● | | | | ● | ● |
| COPYPRIVATE | | | | ● | | |
| SCHEDULE | | ● | | | ● | |
| ORDERED | | ● | | | ● | |
| NOWAIT | | ● | ● | ● | | |

UNIVERSITETET
I OSLO

# Message Passing Interface - MPI

Mostly used  standard for parallel programs

Fortran (f77 ,f90, f95, f2003)
C (and C++)

Most languages can call C functions and hence also use MPI.

MPI implementations written in C.

Scale to millions of cores

Also parallel IO

UNIVERSITETET I OSLO

# Message Passing Interface - MPI

MPI  send datagramms between independent prosesses on independent nodes.

Each MPI rank (prosess is called a rank) is totally independent of all the others.

All MPI processes share a communicator which is just a label.

Memoy for each rank is only local and nothing is shared.

# Message Passing Interface - MPI

MPI  standard that grew out from work like PVM (parallel virtual machine).

MPI standard 1 in 1994, MPI 2 in 1998

Most common MPI 1.2 standard, MPI-IO is backported to 1.2 implementations

OpenMPI is  MPI 2 compliant

# Message Passing Interface - MPI

128 functions in the MPI 1 standard.

You need only 6 to do work :

MPI_Init - Initialize MPI processes
MPI_Comm_size - Find out about the number of the processes in the MPI communicator
MPI_Comm_rank - What is my rank number within the pool of MPI communicator processes?
MPI_Send - Send a message.
MPI_Recv - Receive a message.
MPI_Finalize - Close down MPI processes and prepare for exit.

The rest is utilities and collectives.

UNIVERSITETET
I OSLO

# Message Passing Interface - MPI

MPI programs is started by a small application, mpirun

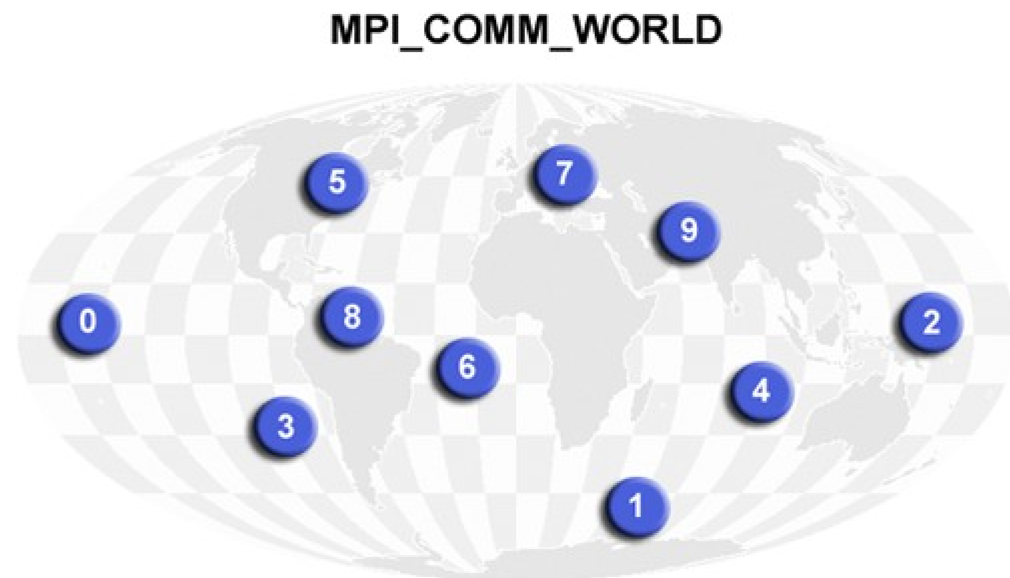
Mpirun sets up an environment and the communicator for all the mpi ranks

Each MPI-executable / rank is like a separate executable application and known nothing about the other, except a common communicator, myrank and total ranks.

# Message Passing Interface - MPI

MPI comminicator
Standard communicator is
MPI_COMM_WORLD – always present.

Each has it's own unique
rank number



# Message Passing Interface - MPI

```
program simple
  include 'mpif.h'

  integer numtasks, rank, ierr, rc

  call MPI_INIT(ierr)
  if (ierr .ne. MPI_SUCCESS) then
    print *,'Error starting MPI program. Terminating.'
    call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
  end if

  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
  print *, 'Number of tasks=',numtasks,' My rank=',rank

C ****** do some work ******

  call MPI_FINALIZE(ierr)

  end
```

# Message Passing Interface - MPI

MPI programms runs idependently !

Nothing binds them together, no master ring to bind them!

Each executable run untill stopped by a blocking MPI call.

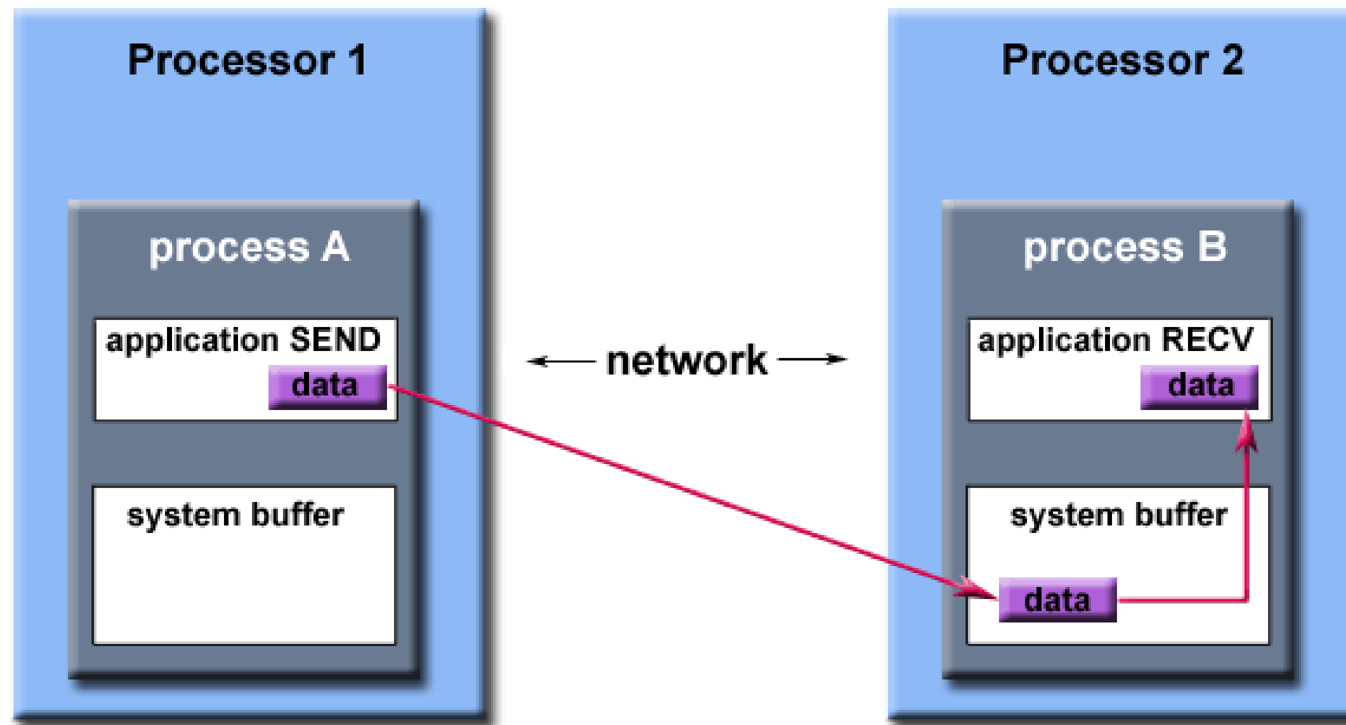
# Message Passing Interface - MPI

Simplest possible example, this will run to Finalize before blocking and syncronise.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Starting on rank %d %s\n", rank, name);
  fflush(stdout);


MPI_Finalize();
return 0;
```

# MPI – Point to point communication

Simplest possible message is from one rank to another :




Path of a message buffered at the receiving process

# Message Passing Interface – Data types

| Fortran Data Types | |
|---|---|
| MPI_CHARACTER | character(1) |
| | |
| MPI_INTEGER | integer |
| | |
| | |
| | |
| | |
| | |
| MPI_REAL | real |
| MPI_DOUBLE_PRECISION | double precision |
| | |
| MPI_COMPLEX | complex |
| MPI_DOUBLE_COMPLEX | double complex |
| MPI_LOGICAL | logical |
| MPI_BYTE | 8 binary digits |
| MPI_PACKED | data packed or unpacked with MPI_Pack()/ MPI_Unpack |

# Message Passing Interface - MPI

You need to have correct data types at each end !

```
if (rank .eq. 0) then
    dest = 1
    source = 1
    call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
&         MPI_COMM_WORLD, ierr)
    call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
&         MPI_COMM_WORLD, stat, ierr)

  else if (rank .eq. 1) then
    dest = 0
    source = 0
    call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
&      MPI_COMM_WORLD, stat, err)
    call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
&      MPI_COMM_WORLD, err)
  endif
```

# Message Passing Interface - MPI

Blocking and none blocking functions

Send and Receive blocking

They block until receiver has received and sender has sent.

There are none blocking versions of these calls, handle with care.

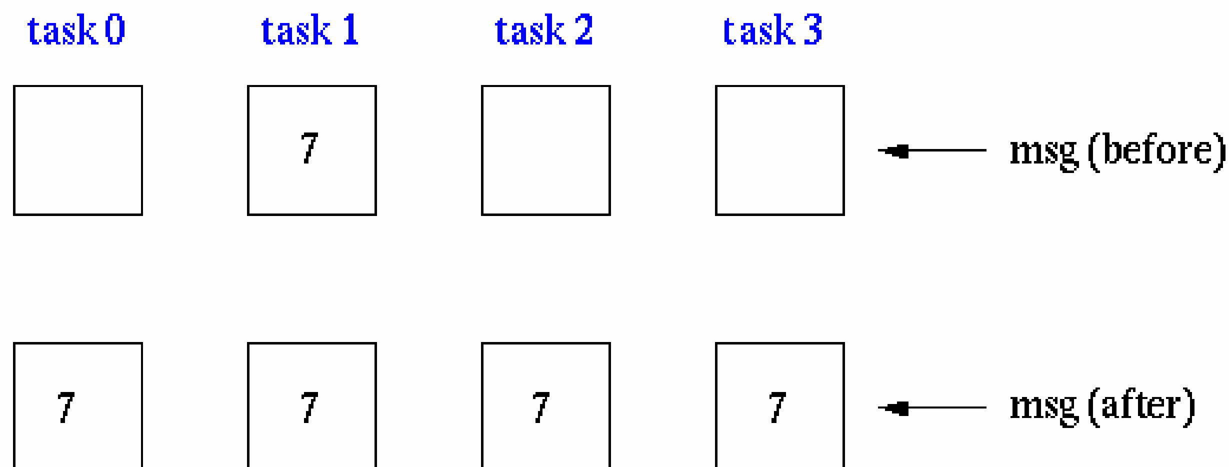
# MPI – Collective operations

Collective Operations operate on all ranks simultanously
  Syncronising all ranks to same line in source.

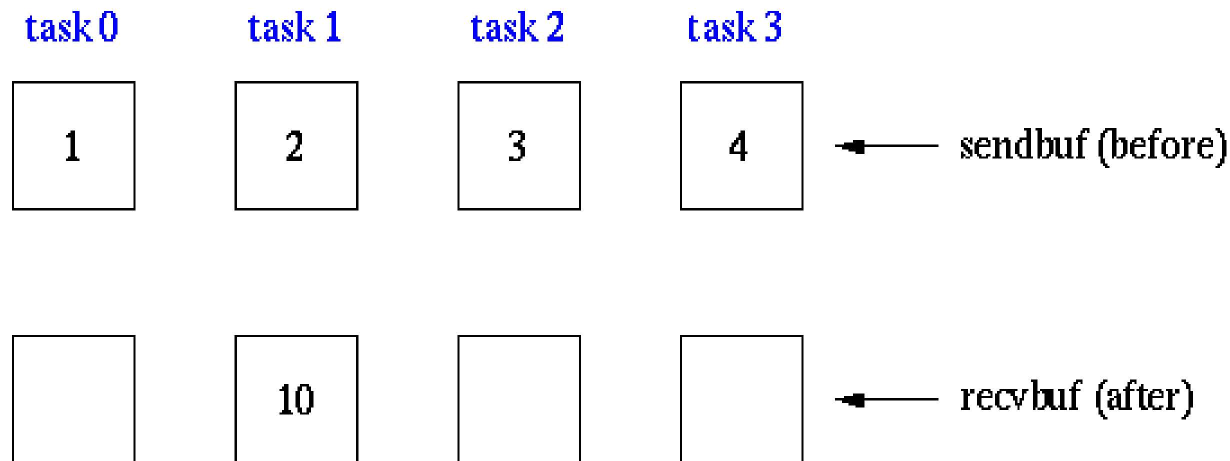
Simplest is Barrier
  All need to wait at the barrier until all ranks has executed the barrier call.

There are many complicated collective operations.

# MPI – Colectice operations - Broadcast



# MPI – Kollektive operasjoner - Reduksjoner



# MPI – Collective operations

```
C  Fortran stores this array in column major order, so the
C  scatter will actually scatter columns, not rows.
   data sendbuf /1.0, 2.0, 3.0, 4.0,
 &        5.0, 6.0, 7.0, 8.0,
 &        9.0, 10.0, 11.0, 12.0,
 &        13.0, 14.0, 15.0, 16.0 /

   call MPI_INIT(ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

   if (numtasks .eq. SIZE) then
     source = 1
     sendcount = SIZE
     recvcount = SIZE
     call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf,
 &   recvcount, MPI_REAL, source, MPI_COMM_WORLD, ierr)
     print *, 'rank= ',rank,' Results: ',recvbuf
   else
     print *, 'Must specify',SIZE,' processors.  Terminating.'
   endif
```

# MPI – Collective operations

Output after MPI scatter call :

```
rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
```

Each rank has their vector. Common way to give all ranks data for work.

# In practice - OpenMP

```
subroutine smooth( a, b, w0, w1, w2, n, m, niters )
  real, dimension(n,m) :: a, b
  real :: w0, w1, w2
  integer :: n, m, niters
  integer :: i, j, iter

  do iter = 1,niters
!$OMP PARALLEL DO private (i, j)
     do i = 2,n-1
        do j = 2,m-1
           a(i,j) = w0 * b(i,j) + &
               w1*(b(i-1,j)+b(i,j-1)+b(i+1,j)+b(i,j+1)) + &
                w2*(b(i-1,j-1)+b(i-1,j+1)+b(i+1,j-1)+b(i+1,j+1))
        enddo
     enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO private (i, j)
     do i = 2,n-1
        do j = 2,m-1
           b(i,j) = a(i,j)
        enddo
     enddo
!$OMP END PARALLEL DO
  enddo

end subroutine smooth
```

# In practice - OpenMP

Compile :

```
olews@styren ~/work/PGI $ pgfortran -o smooth.x -O3 -mp  -Minfo  smooth.f90
smooth:
     77, Invariant assignments hoisted out of loop
     78, Parallel region activated
     79, Parallel loop activated with static block schedule
     85, Parallel region terminated
     87, Parallel region activated
     88, Parallel loop activated with static block schedule
     92, Parallel region terminated
olews@styren ~/work/PGI $
```

Portland compiler produced threads from the OpenMP direktivene.

Does it compilete faster ?

# In practice - OpenMP

All common compilers has OpenMP support today

Portland, pgcc og pgfortran : -mp

Gcc og gfortran : -openmp og -lgomp

Intel ifort og icc : -openmp

Pathscale pathcc og patghf90 : -mp

UNIVERSITETET I OSLO

# In practice - OpenMP

One thread :

```
olews@styren ~/work/PGI $ ./smooth.x 1000 10
 n =          1000  iter           10
Problem size [MB] :   8.
 Num threads            1
 start sum(x) sum(y)    499971.6        499807.2
 end sum(x) sum(y)    3510.541        3487.164
 Calc time :     0.7957640
olews@styren ~/work/PGI $
```

4 threads

```
olews@styren ~/work/PGI$
OMP_NUM_THREADS=4 ./smooth.x 1000 10
 n =          1000  iter           10
Problem size [MB] :   8.
 Num threads            4
 start sum(x) sum(y)    499971.6        499807.2
 end sum(x) sum(y)    3510.541        3487.164
 Calc time :     0.2051560
olews@styren ~/work/PGI $
```

We observe a speedup of about 4 by usign all 4 cores.

# In use - MPI

Many MPI libraries

OpenMPI – Open source (gamle LAM MPI) IB support, MPI-2

MPICH(2) - OpenSource Tcp/ip også i versjon 2 utgave

Intel MPI – Ikke installert pr. i dag.

# In use - MPI

Use a wrapper :

mpif90 / mpicc – mpi is built using a combination of C and Fortran compiler.

The compiler can be changed by environment variables.

# In practice - MPI

Compile and run :

```
program minibarrier
  implicit none
  include "mpif.h"
  integer ierr,rank,nproc
  call MPI_Init(ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
  write(*,*) rank, "begun the code"
  call MPI_Barrier(MPI_COMM_WORLD,ierr)
  write(*,*) rank, "crossed the barrier"
  call MPI_Finalize(ierr)
end program minibarrier
```

# In practice - MPI

Compile and run :

module load openmpi/1.4.3.intel og module load intel/11.1u8 er nødvendig

```
Mpif90 -o mini-barrier.x  mini-barrier.f90
```

```
[olews@compute-9-8 mpi-tests]$ mpimon ./mini-barrier.x -- localhost 4
        0 begun the code
        1 begun the code
        2 begun the code
        3 begun the code
        0 crossed the barrier
        3 crossed the barrier
        1 crossed the barrier
        2 crossed the barrier
[olews@compute-9-8 mpi-tests]$
```

# In practice - MPI

Compile and run :

module load openmpi/1.2.8.intel og module load intel/11.0

```
mpif90 -o mini-barrier.x mini-barrier.f90

[olews@compute-9-8 mpi-tests]$ mpirun -np 4 ./mini-barrier.x
        0 begun the code
        3 begun the code
        2 begun the code
        1 begun the code
        3 crossed the barrier
        2 crossed the barrier
        0 crossed the barrier
        1 crossed the barrier
[olews@compute-9-8 mpi-tests]$
```