Parallellprogramering introduksjon

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

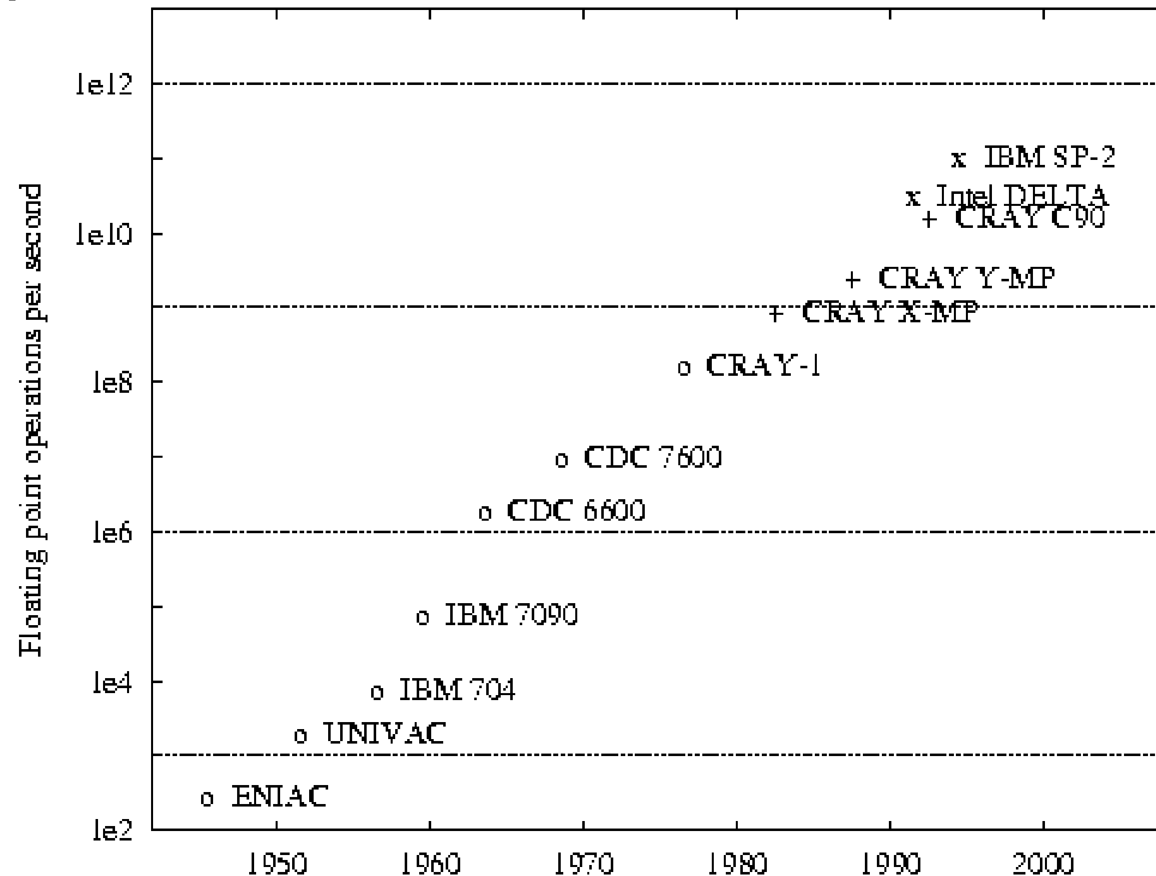
UNIVERSITETET I OSLO

Bakgrunn

Stadig raskere CPUer tidligere begrenset behovet for å programere i parallell.

Milepeler :

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



Bakgrunn

Like etter årtusenskiftet ble ikke CPUene raskere.

Mores lov så ikke ut til å holde følge med ytelse
Den handler om transistortetthet ikke ytelse.

Ytelsen kunne ikke økes ved å kjøpe / produsere raskere CPUer.

Parallellprogramering blir derfor tvingende nødvendig!

Bakgrunn

Implisitt parallellitet i dagens brikker

Dagens Xeon/Opteron har stort grad av innebygget parallelitet.

«Out of order excution»
«Prefetch»
«Branch prediction»

Alt dette bidrar til superlineær skalering, dvs at ytelsen øker mer enn klokkefrekvensen.

Bakgrunn

2006 quad core
    4 fullverdige CPUer på hver prosessorchip

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

Intel, SUN og IBM har også HyperThreading der de har to sett registere (men ikke to eksekveringsenheter, kun et sett).

Grafikkort har fra 320, 640 til 800 lettvekts CPUer.

Bakgrunn – dagens og fremtidens situasjon

I praksis har vi i dag overflod av prosessorkjerner

Vi har to brikker/sokler pr node (vanlige noder)
Vi har minst to kjerner på en laptop, selv netbooks!

I fremtiden blir det enda flere kjerner, men hver enkelt kjerne blir ikke raskere. Det blir bare flere.

Bakgrunn – dagens og fremtidens situasjon

I praksis har vi i dag overflod av prosessorkjerner

Men :

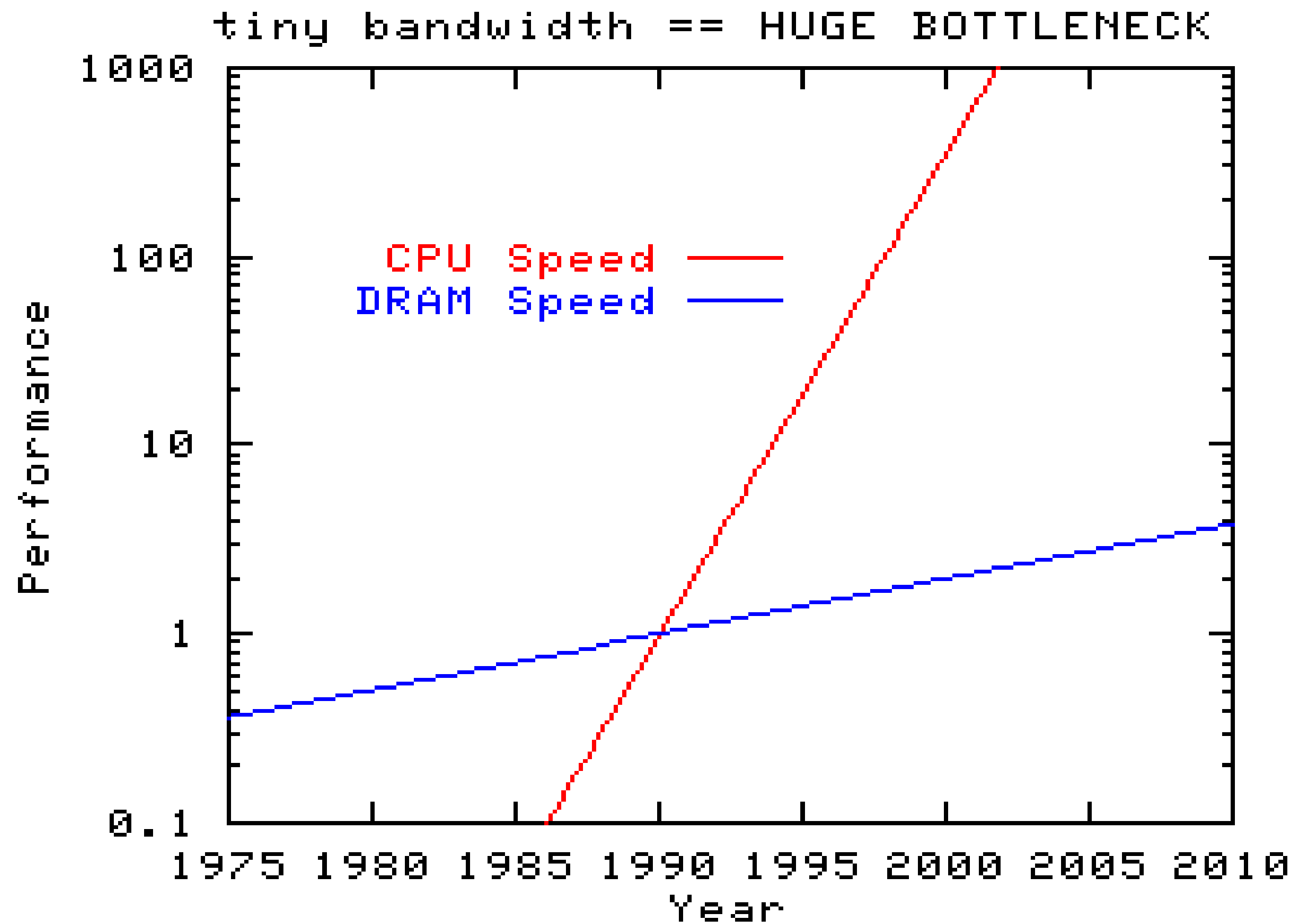
Minnebåndbredde har ikke økt på samme måte.

Minneaksesstiden (random aksess i minne) har heller ikke økt særlig mye.

Når mengden minne øker øker også krevene til begge overnevnte. Gapet mellom tilbud og etterspørsel øker.

Hva med IO, det har ikke øket noe særlig på 10 – 20 år.

Bakgrunn – Minnebåndbredde



Vi må programere i parallell !

Mange CPUer og et problem !

Dette er en utfordring

Jeg har ingen gode forslag!

Anyone ???

Tre vanlige metoder

1. Posix Threads
   Delt minne

2. OpenMP (Open Multi Processing)
   Delt minne

3. MPI (Message Passing Interface)
   Distribuert minne

4. Jobb og slavebaserte metoder, LINDA, etc. Metoder som er lite brukt. Gamle metoder som PVM (Parallel Virtual Machine)

Skalering – Amdahls lov

Amdahls lov fra 1967

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

P er andel av programmet som har en speedup på S

Hva betyr dette ?

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

P er andelen av program som paralleliseres
N er antall prosessorer

Skalering – Amdahls lov

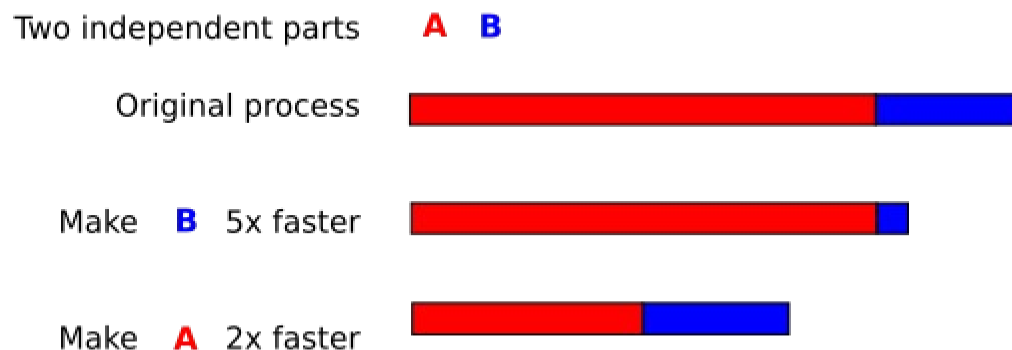
$$\frac{1}{((1-P)+\frac{P}{N})}$$

P er andelen av program som paralleliseres
N er antall prosessorer

Vi setter N veldig stor og får : $\frac{1}{(1-P)}$

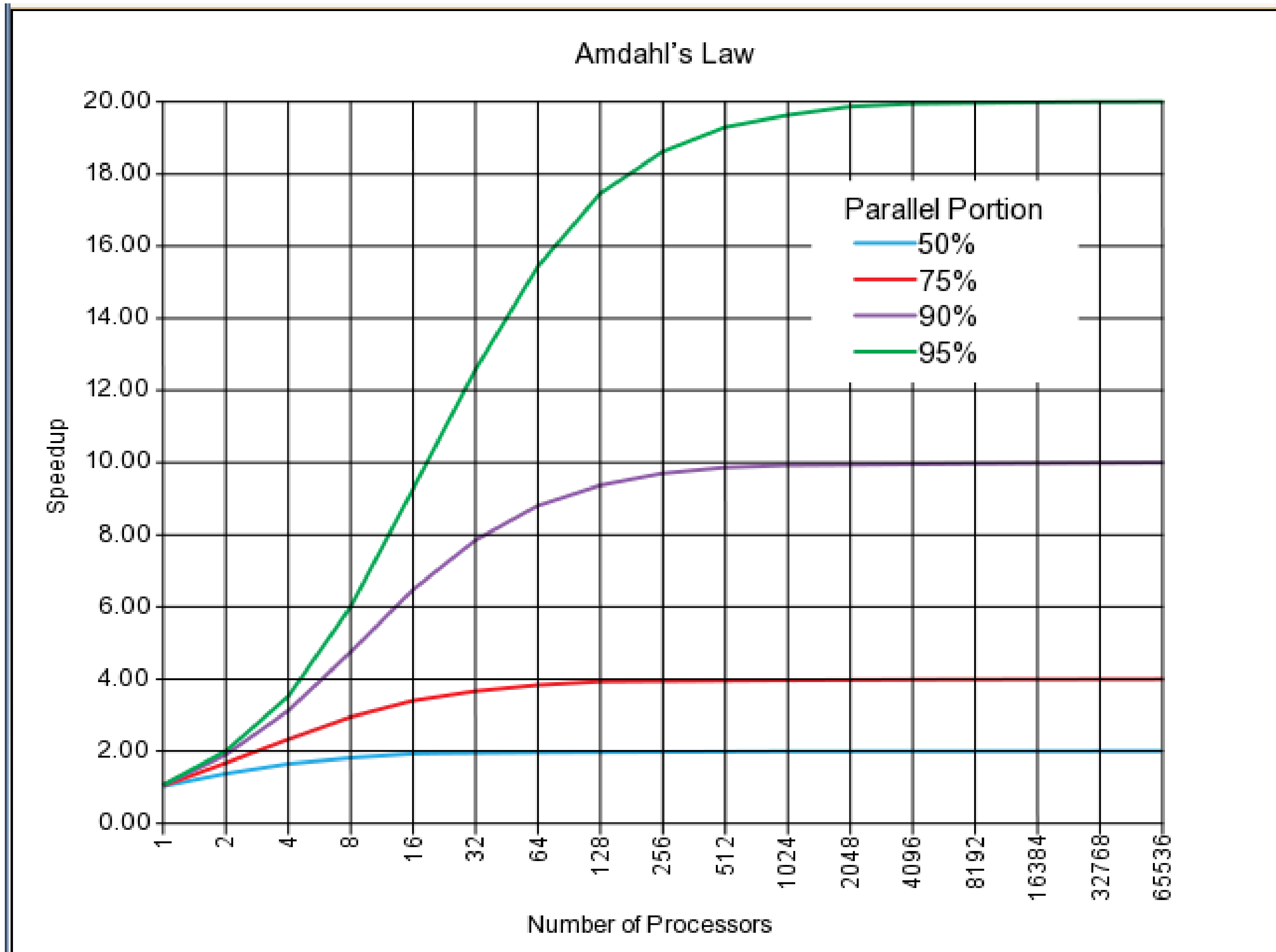
Uansett hvor mange prosessorer vi bruker går det ikke fortere enn den serielle delen!

Optimalisere og parallelisere hvor og hva ?



UNIVERSITETET I OSLO

Amdahls lov



Skalering – Gustafsons lov

Gustafsons lov fra 1988 :
any sufficiently large problem can be efficiently parallelized.

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

S er speedup, P antall prosessorer og α er andelen av kode som kan paralleliseres.

Store problemer kan derfor skalere godt ! Jo større maskin jo strørre problem må løses.

Analogi med biler :

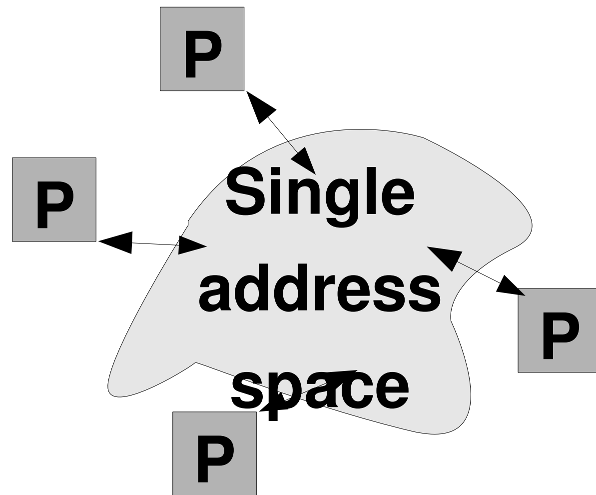
Amdahls Lov:

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.

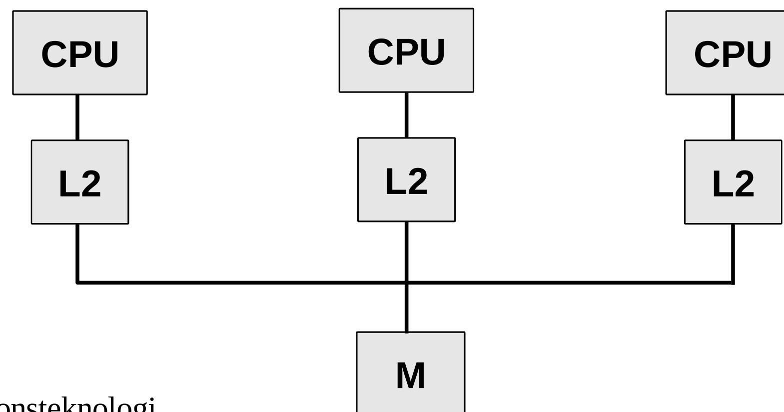
Gustafsons Lov:

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.

Delt minne



- Tråder interakterer via lesning og skriving av minne lokasjoner
- Just-in-time-ness av cache subsystem
  - delay incurred by every cache miss
  - latency hiding techniques
    - multitasking/threading
    - prefetching



Posix Threads

Krever et minneområde. Dette er «shared memory» programering.

En tråd er en lettvektsprosess som arver omgivelsene til morprosessen.

All trådadministrasjon, kommunikasjon, synkronisering må gjøres av brukeren i programmet

I praksis er det meget lite brukt for vitenskapelige C og Fortran programmer.

Men mye brukt i biblioteker og system type applikasjoner

OpenMP

Krever et delt minneområde. Dette er «shared memory» programering.

OpenMP er veldig mye brukt

Meget raskt å komme i gang

Kompilatordirektiver, krever kompilatorsupport

Kan skalere utmerket til **noen håndfuller** CPUer.

MPI

Kan bruke distribuert minne (dvs mange små noder)

MPI er veldig mye brukt

Litt tungt å komme i gang

Krever MPI bibliotek med kommunikasjonsfunksjoner

Kan skalere utmerket til ti-tusenvis av CPUer

Krever godt nettverk mellom noder (InfiniBand)

OpenMP

OpenMP er en standard for kompilatordirektiver

- OpenMP er ute i versjon 3 nå
- www.openmp.org

Versjoner for C og Fortran er i utstrakt bruk

Parallelisering på løkke og oppgave nivå

- Løkkeparallelisering av mest utbredt

OpenMP

Parallelle regioner

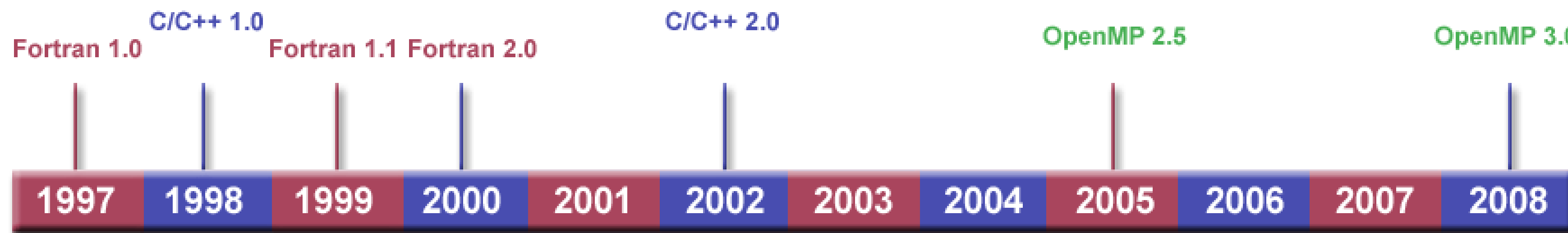
Et hovedprogram som spinner av og genererer Posix tråder

OpenMP administerer trådene

En CPU kjører hovedprogram
- Det er kun EN prosess med EN PID
- Trådene er subprosesser dvs tråder

UNIVERSITETET I OSLO

OpenMPI – Moden standard



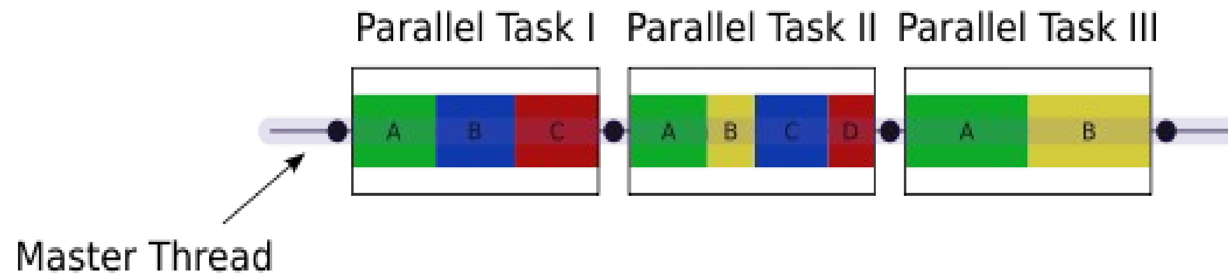
Ingen grunn til at OpenMP forsvinner med det første.

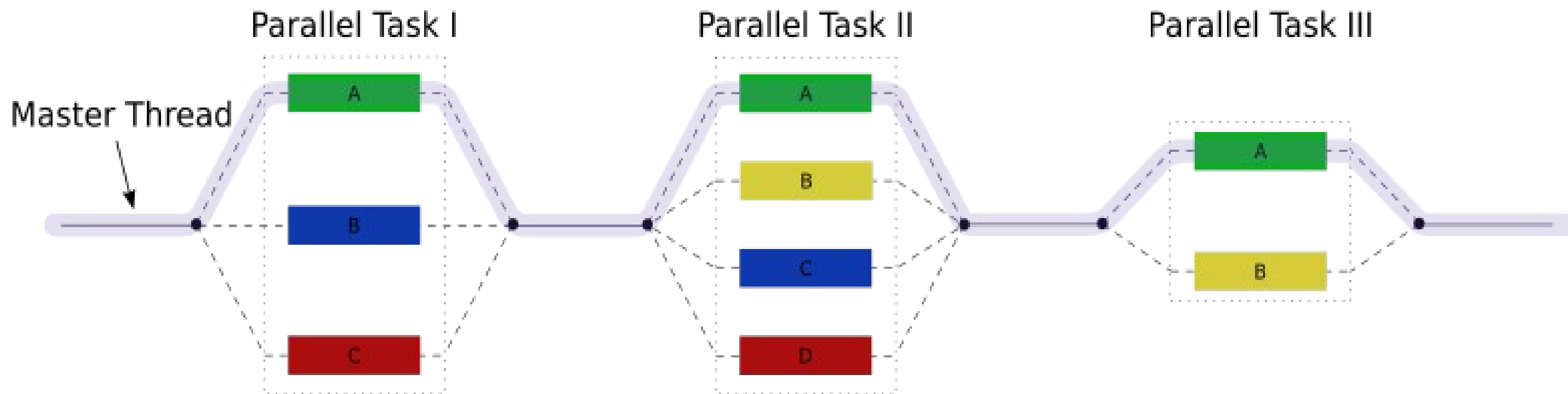
Trygt å velge OpenMP som plattform

Web adresse for OpenMP : www.openmp.org

UNIVERSITETET I OSLO

OpenMPI – multiple tråder





OpenMPI

Bruker må selv holde kontroll på globale og lokale variable

Typisk er tellevariable i en løkke som åpenbart må være lokale

Arrayer er typisk globale

Men oppdaterer man x(i+1) som kanskje tilhører en annen prosess ?

Alle data er i utgangepunktet delt.

OpenMPI direktiver

Parallel regions

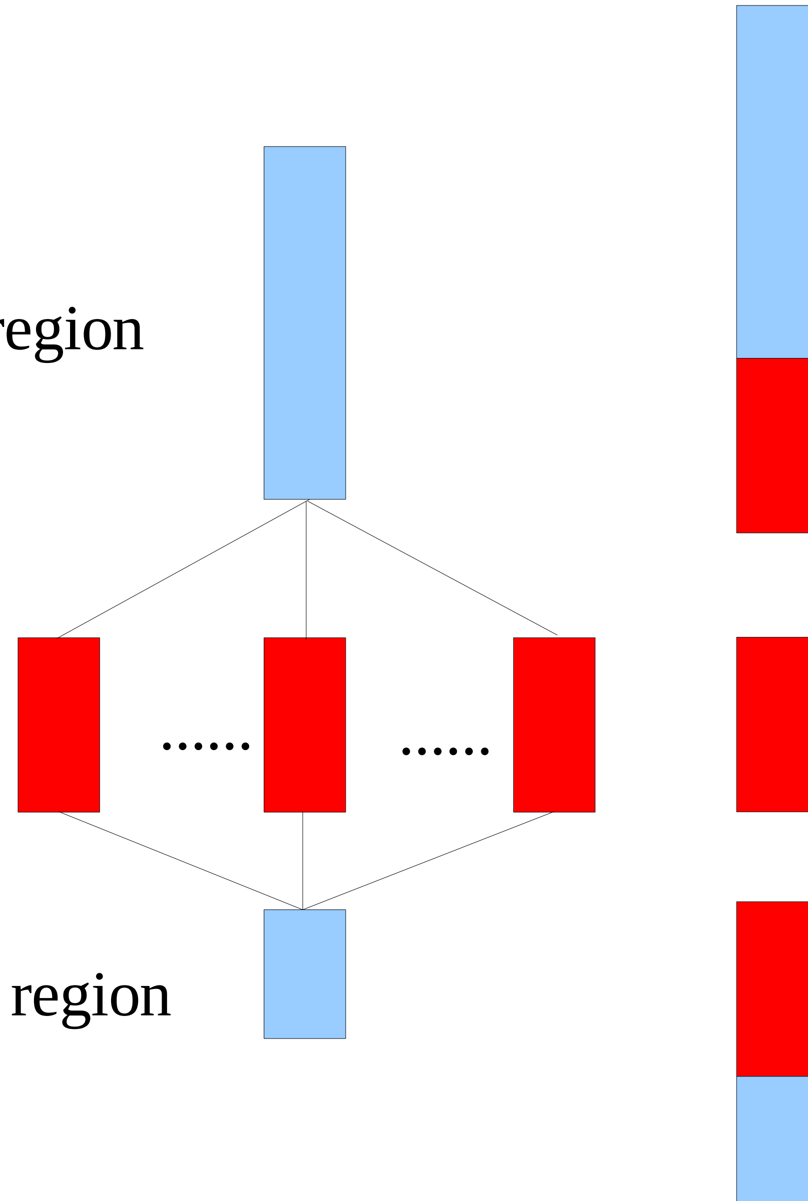
```
!$OMP PARALLEL
<Your parallel code here>
!$OMP END PARALLEL
```

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
```

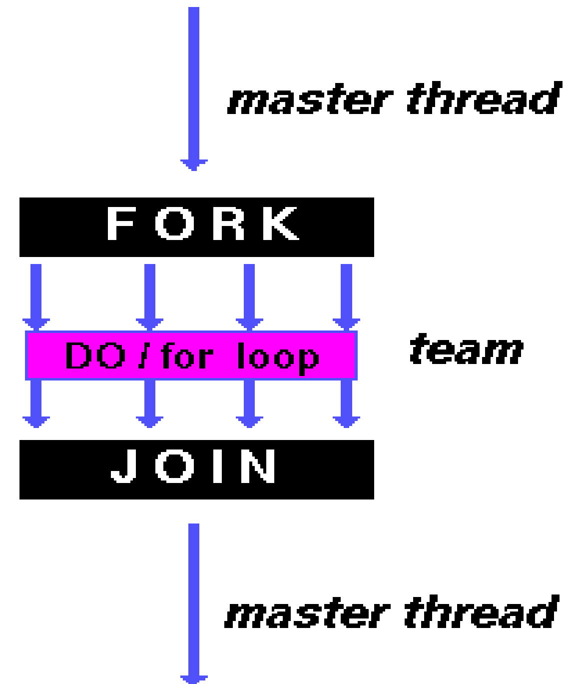


Work-Sharing Constructs

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

Dette er det mest vanlige for Fortran kode

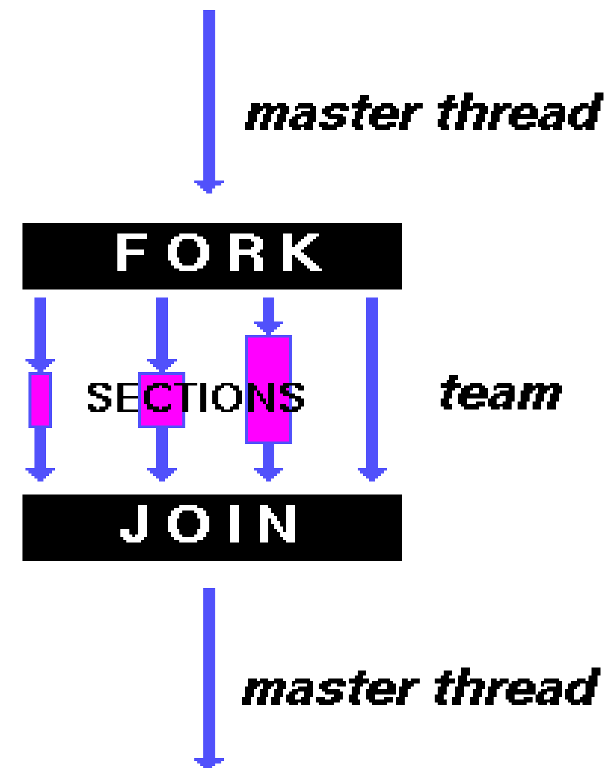
God fordeling av lasten med data parallell kode



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".

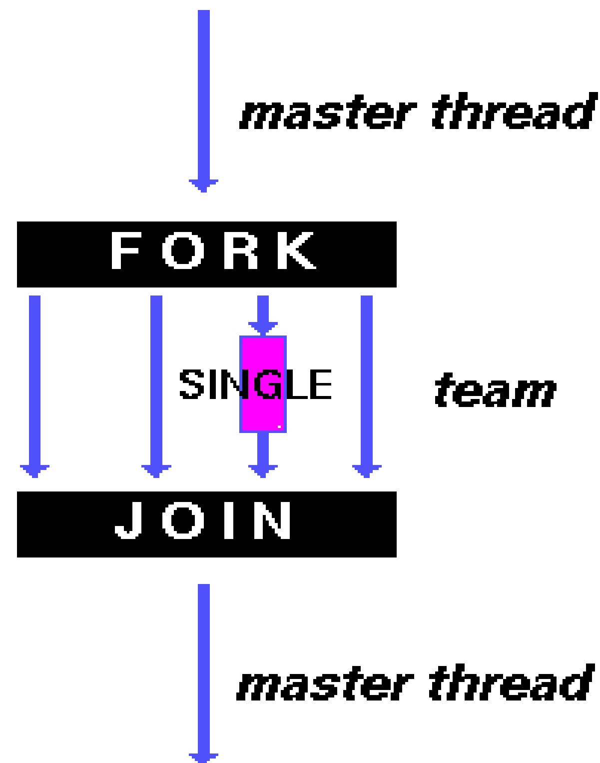
Ikke alltid lett å lastbalansere



Work-Sharing Constructs

SINGLE - serializes a section of code

Noen ganger er det umulig å parallelisere



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

Dette er kanskje den mest brukte i 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 - DO / for Directive

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team.

NO WAIT / nowait: If specified, then threads do not synchronize at the end of the parallel loop.

ORDERED: Specifies that the iterations of the loop must be executed as they would be in a serial program.

COLLAPSE: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

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 definerer en oppgave som kan tas nå eller siden av master tråden eller en annen tråd.

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

  block

!$OMP END TASK
```

Bruker må selv styre synkronseringen.

Fortran - Workshare

The WORKSHARE directive divides the execution of the enclosed structured block into separate units of work, each of which is executed only once.

```
!$OMP WORKSHARE

   structured block

!$OMP END WORKSHARE [ NOWAIT ]
```

Bruker må selv styre synkronseringen.

Fortran – Kritiske regioner

Noen ganger kan ikke operasjoner overlappes men må gjøres sekvensiellt.

```
!$OMP CRITICAL [ name ]
  block
!$OMP END CRITICAL
```

Kun en tråd av gangen kan utføre denne regionen.

Kritiske regioner - master

Noen ganger kan ikke operasjoner overlappes men må gjøres sekvensiellt.

Kun master kan utføre denne operasjonen

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

Datahåndtering

Alle data er synlige for alle og riktig håndtering må til !

Called Data-sharing Attribute Clauses

* PRIVATE
* FIRSTPRIVATE
* LASTPRIVATE
* SHARED
* DEFAULT
* REDUCTION
* COPYIN

Datahåndtering

The PRIVATE clause declares variables in its list to be private to each thread.

FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.

SHARED clause declares variables in its list to be shared among all threads in the team.

Datahåndtering

DEFAULT clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.

REDUCTION clause performs a reduction on the variables that appear in its list.

COPYIN clause provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team.

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		●	●	●		

Message Passing Interface - MPI

Dagens mest brukte standard for parallelle programmer!

Fortran (alle dialekter f77 ,f90, f95, f2003)
C (kan også kalle std. C rutiner fra C++)

De fleste andre språk kan kalle C rutiner og dermed også bruke MPI.

MPI implementasjoner er skrevet i C.

Kan skalere til hundretusenvis av prosessorer

Kan også tilby parallell IO

Message Passing Interface - MPI

MPI går i korte trekk ut på å sende datagrammer mellom uavhengige prosesser på uavhengige maskiner.

Hver MPI rank (hver prosess kalles rank) er totalt uavhengig av de andre og vet ikke om hverandre

De deler en kommunikator som de bruker som kommunikasjons merkelapp

Minne for hver rank er kun lokalt og ingen ting er delt, distribuert minne.

Message Passing Interface - MPI

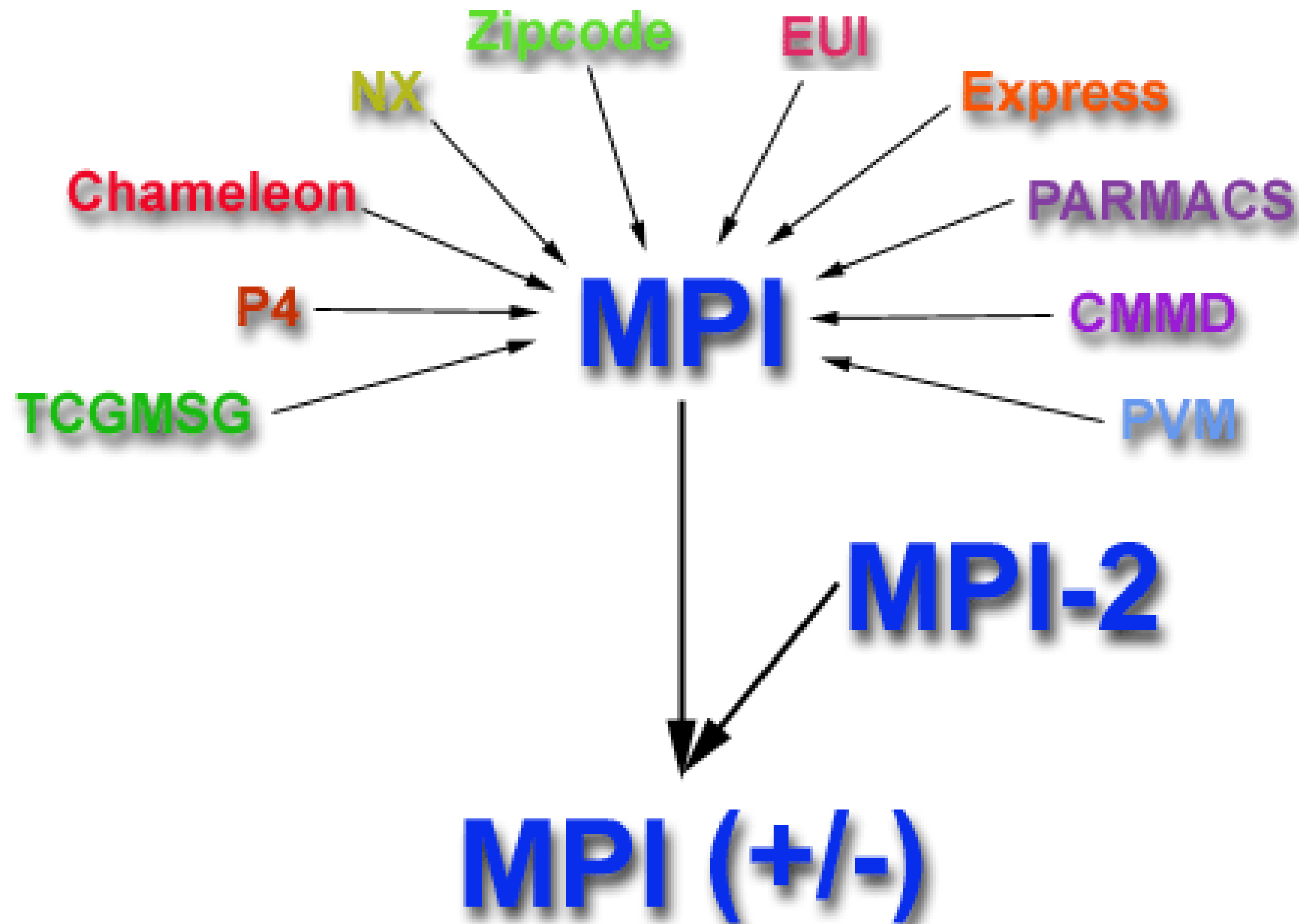
MPI er en standard som kom fra diverse forsøk med parallel programering som PVM (parallel virtual machine).

MPI standard 1 kom i 1994, MPI 2 kom i 1998

Mest vanlig er ennå MPI 1.2 standard, men MPI-IO (fra 2 std) er tatt med i 1.2 utgaver som et tillegg.

OpenMPI følger MPI 2, Scampi følger 1.2 std med IO som tilegg.

Message Passing Interface - MPI



Message Passing Interface - MPI

Det er 128 funksjoner i MPI 1 standarden!

Man trenger kun 6 for å gjøre noe :

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.

Resten er ekstras og hjelpefunksjoner

Message Passing Interface - MPI

MPI programmer startes opp av en liten applikasjon som starter MPI-applikasjonen der brukeren ønsker den skal starte.

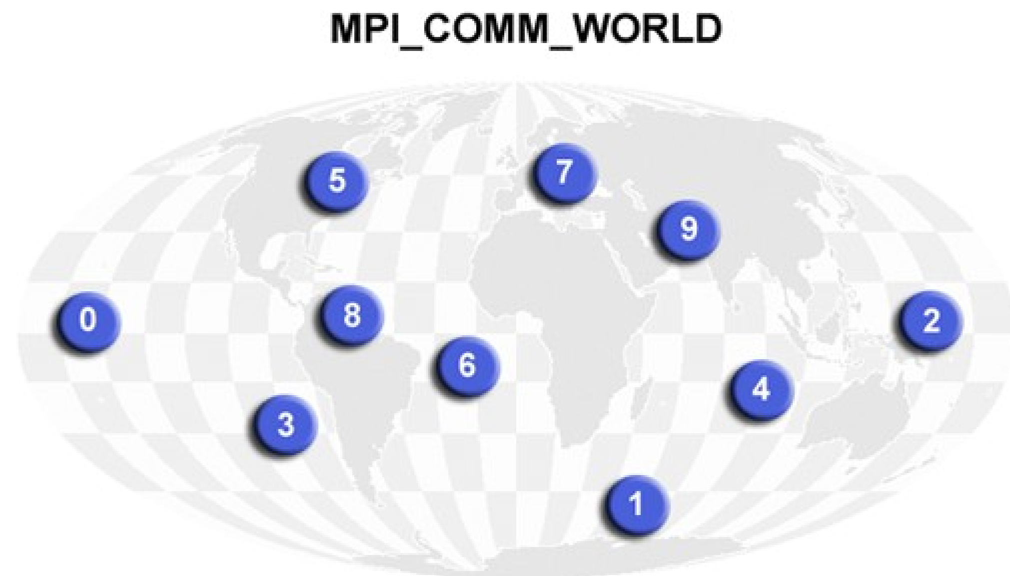
MPI start applikasjonen sørger også for at kommunikatoren blir satt opp riktig slik at alle MPI ranker kan kommunisere.

Hver MPI-applikasjon som startes startes som en helt selvstandig frittstående applikasjon, men starteren vet hva den har startet og hvilke prosesser som er med i kommunikatoren.

Message Passing Interface - MPI

MPI bruker en kommunikator slik at prosesser kan kommunisere med hverandre.
Standard kommunikator for alle MPIer er MPI_COMM_WORLD – denne finnes alltid.

Hver prosess har en MPI rank som er et heltall som identifiserer prosessen.
Hver prosess har en unik rank innen kommunikatoren.



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

Husk at alle MPI programmer kjører selvstendig !

Det er ingen ting som binder dem sammen.

Hver executable kjører til den blir stanset av et blokkerende MPI kall.

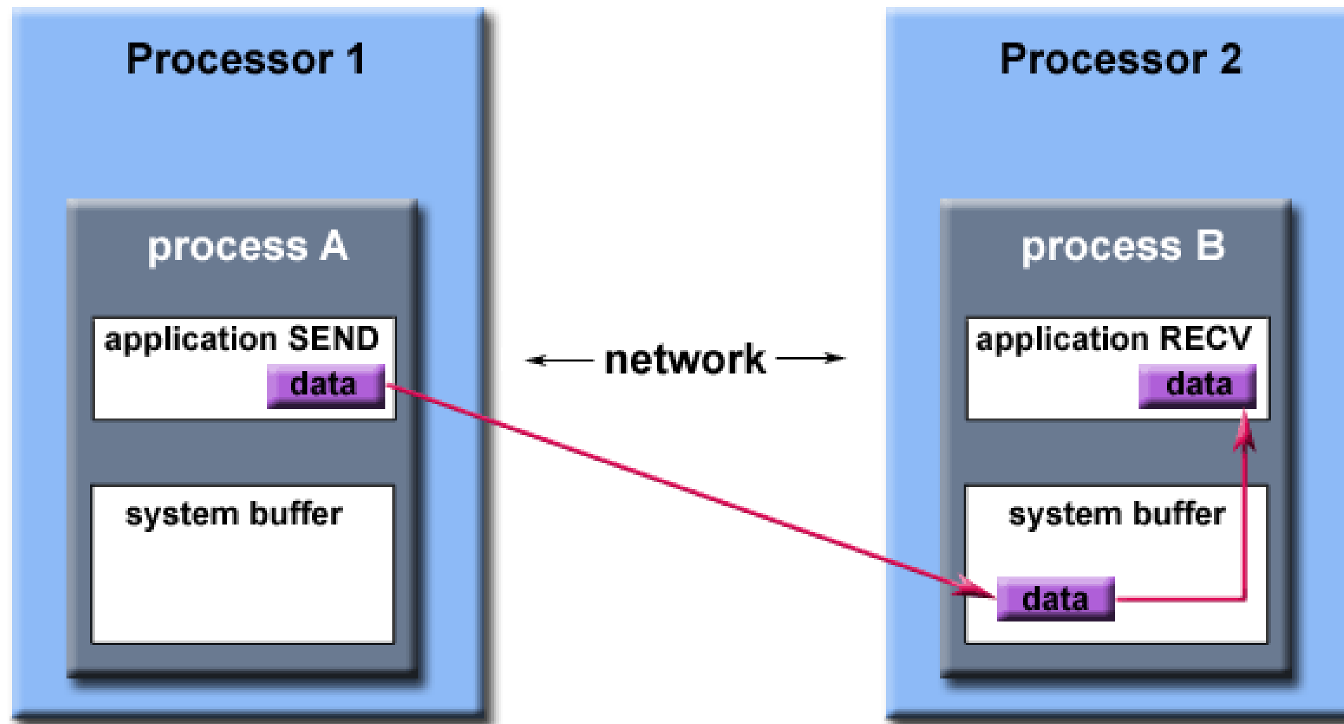
Message Passing Interface - MPI

Vi ser at det ikke er noen komminikasjon eller synkronisering og dette programmet kjører uten avbrudd til enden og terminerer på Finalize kallet, dette kan imidlertid blokkere.

```
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 – Punkt til punkt kommunikasjon

Det enkleste er å sende en melding mellom to ranker.



UNIVERSITETET I OSLO

Message Passing Interface – Data typer

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

Vi må bruke riktig data type på begge sider :

```
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

Blokkerende og ikke blokkerende kall

Send og Receive blokkerer

De henger og returner ikke før mottager eller sender er klar

Sender henger til mottager har mottatt meldingen

Mottager henger til sender har sendt meldingen

Det finnes ikke blokkerende kall som returner øyeblikkelig, dette er ikke alltid like trygt.

MPI – Kollektive operasjoner

Operasjoner som virker på alle rankene samtidig

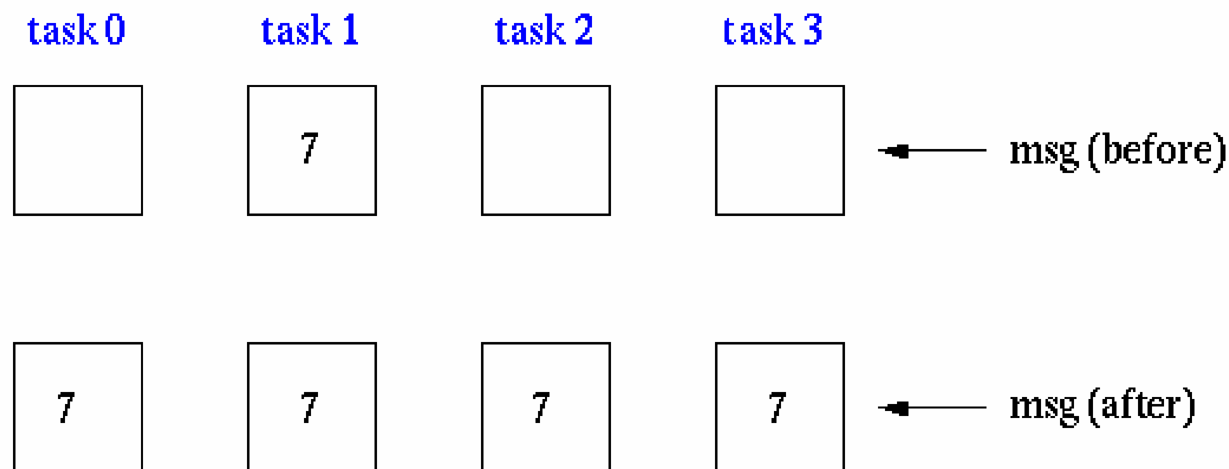
- Dette synkroniserer alle rankene til samme sted i koden

Enkleste er Barrier

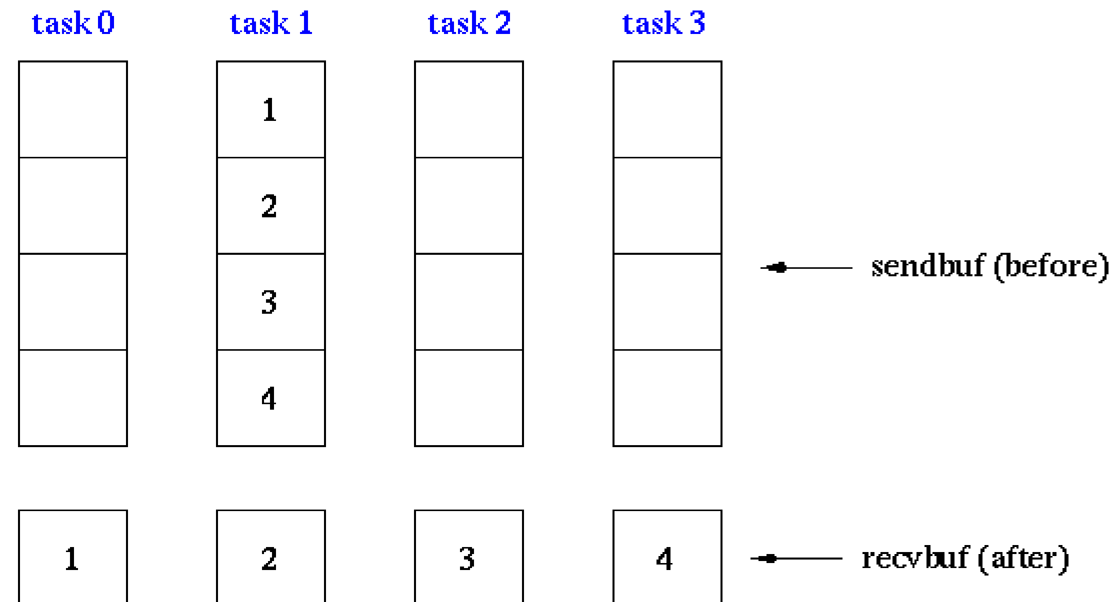
- Alle må stoppe ved barriæren og ingen kan fortsette før alle har utvekslet informasjon om at nå er alle i barrier funksjonen.

Mer kompliserte innebærer broadcast, scatter, gather, reduksjoner etc

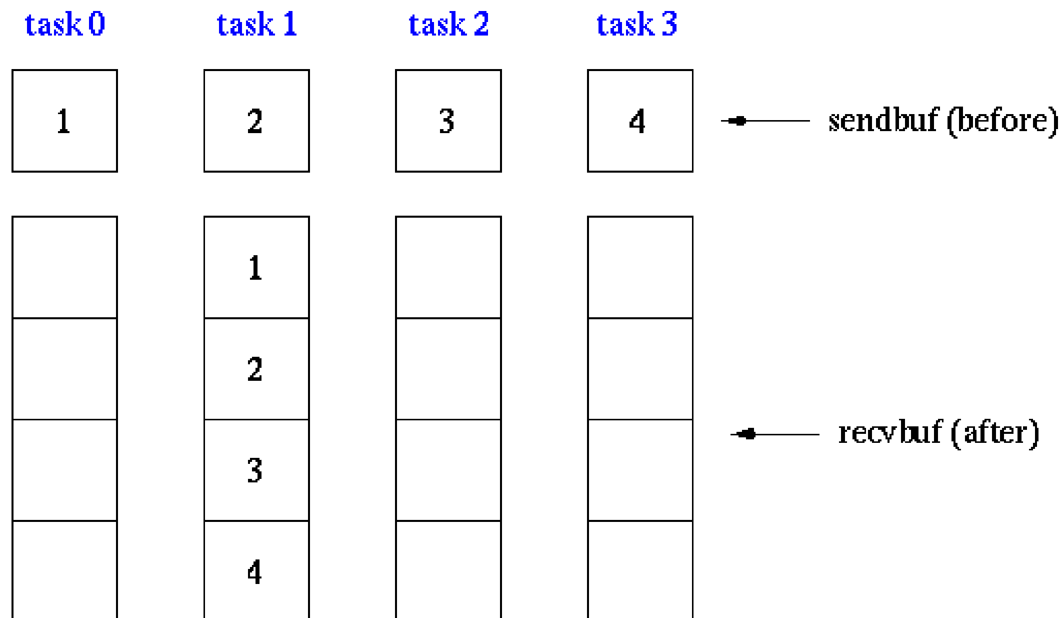
MPI – Kollektive operasjoner - Broadcast



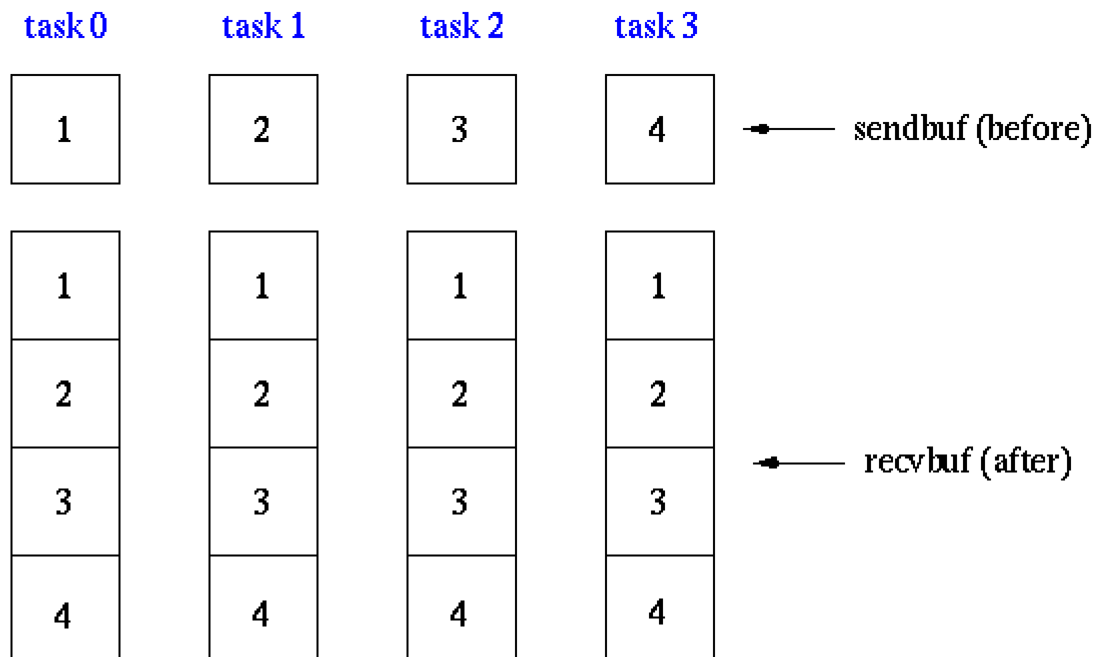
MPI – Kollektive operasjoner - Scatter



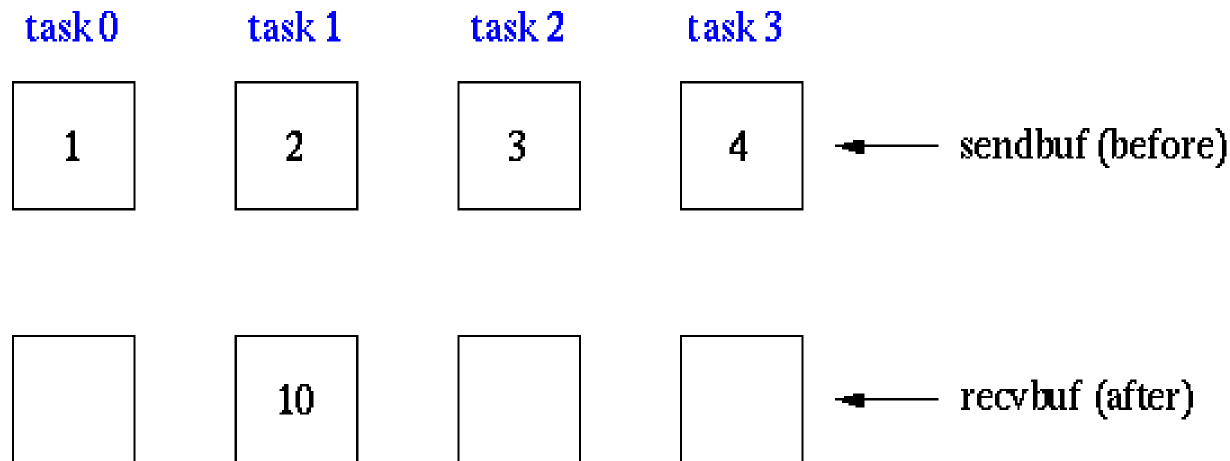
MPI – Kollektive operasjoner - Gather



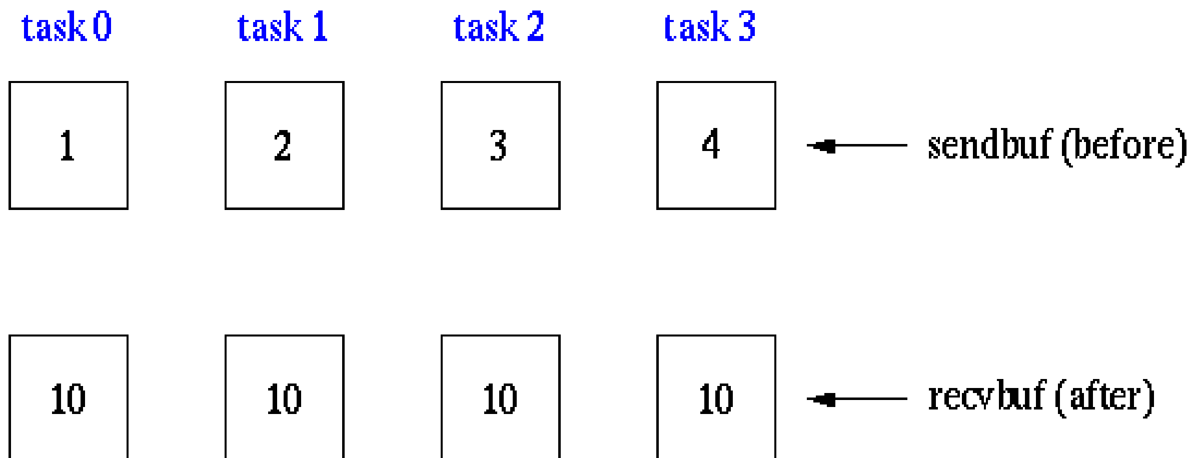
MPI – Kollektive operasjoner - Allgather



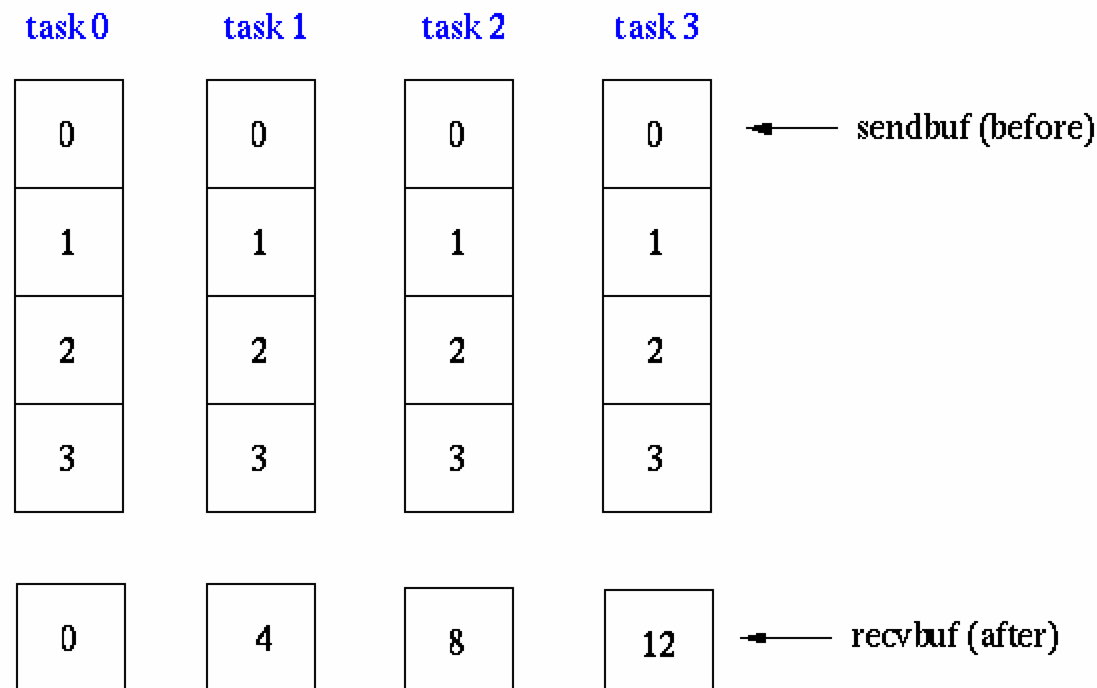
MPI – Kollektive operasjoner - Reduksjoner



MPI – Kollektive operasjoner - Reduksjoner



MPI – Kollektive operasjoner - Reduksjoner

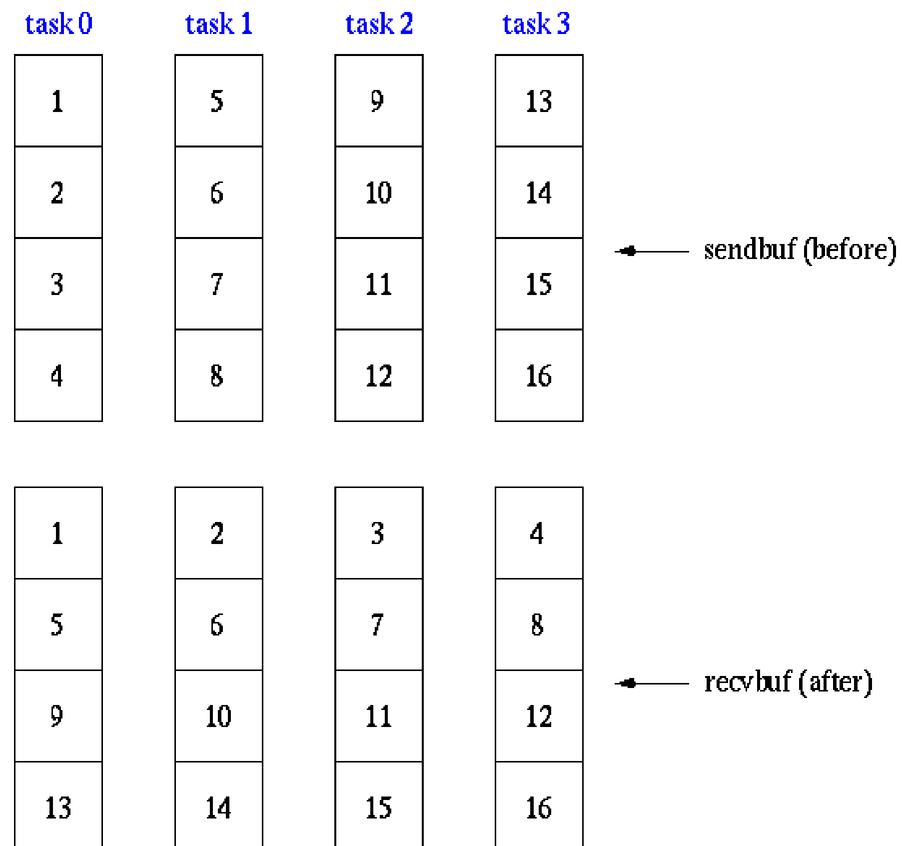


UNIVERSITETET
I OSLO

MPI – Kollektive operasjoner - Reduksjoner

MPI Reduction Operation	
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bit-wise AND
MPI_LOR	logical OR
MPI_BOR	bit-wise OR
MPI_LXOR	logical XOR
MPI_BXOR	bit-wise XOR
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MPI – Kollektive operasjoner – alle til alle



MPI – Kollektive operasjoner

```
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 – Kollektive operasjoner

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
```

Hver rank har nå fått sin vektor. Dette er en mye brukt metode for å dele ut dara til rankene når masterprosessen leser inn en fil som skal prosesseseres.

I praksis - 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
```

I praksis - OpenMP

Kompilere :

```
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 kompilatoren laget tråder av OpenMP direktivene. Betyr det noe for kjøretiden ?

I praksis - OpenMP

Kompilator support

Portland, pgcc og pgfortran : -mp

Gcc og gfortran : -openmp og -lgomp

Intel ifort og icc : -openmp

Pathscale pathcc og patghf90 : -mp

I praksis - OpenMP

Kjøring med 1 tråd :

```
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 $
```

Kjøring 4 tråder :

```
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 $
```

Vi observer en speedup på nesten 4 ved å bruke alle fire kjernene i prosessoren. Mer kompliserte løkker vil skalere opp til mye høyere antall CPUer.

I praksis - Autoparallelisering ??

Siden OpenMP parallelisering av løkker er så lett kan kompilatoren gjøre det automatisk ?

Intel fortran ifort kan ta opsjonen -parallel

enable the auto-parallelizer to generate multi-threaded code for loops that can be safely executed in parallel

Noen ganger virker det, andre ganger ikke. Det er nesten like lett å jobbe med OpenMP.

I praksis - MPI

Det finnes mange MPI biblioteker i bruk (på titan finnes flere).

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

Scali MPI - Platform produkt i dag, fleksibel. Tcp/ip, IB support, MPI 1.2 + IO

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

HP-MPI – Brukes embedded in noen applikasjoner

Intel MPI – Ikke installert pr. i dag.

I praksis - MPI

To tilnærminger

mpif90 / mpicc – mpi er bygget med en spesiell kompilator som ligger i kulissene

f77 -I/opt/scali/include64 -L/opt/scali/lib64 prog.f -lfmpi -mpi -- her har man full kontroll på bibliotekene og include filene

Den enkle varianten er den vanligste. MPI bibliotekene er ofte kopmpilator avhengige. Scali MPI er i praksis uavhengig av kompilator, openMPI er det ikke. Den må bygges for hver fortran kompilator.

I praksis - MPI

Kompilere og kjøre :

```
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
```

I praksis - MPI

Kompilere og kjøre :

module load scampi og module load intel/11.0 er nødvendig

```
ifort -I/opt/scali/include64 -L/opt/scali/lib64 -o mini-barrier.x  mini-barrier.f90 -lfmpi -lmpi

[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]$
```

I praksis - MPI

Kompilere og kjøre :

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]$
```