A short overview of parallel paradigms

Fabio Affinito, SCAI

Why parallel?

In principle, if you have more than one computing processing unit you can exploit that to:

-Decrease the time to solution

- Increase the size of the problem to be solved

The perfect parallelism is achieved when all the processes can run independently to obtain the final result.

Parallelism impacts on:

-The source structure

- the computer architecture

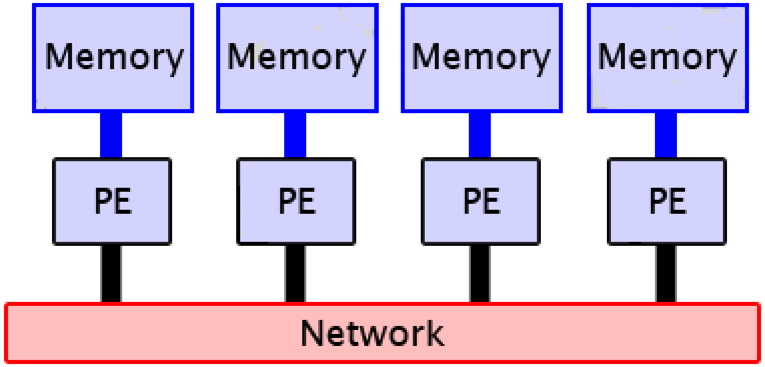
Parallel architectures

A first division, on the architectural side can be:

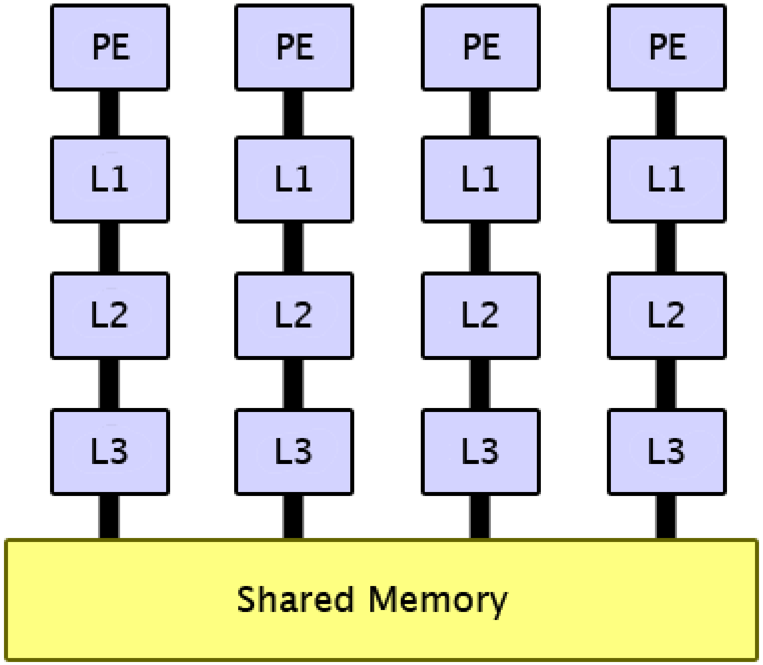
-**Distributed memory systems**: each computing unit has its own memory address space

- **Shared memory systems:** computing units (cores/processors) share the same address space. Knowledge of where data is stored is no concern of the programmer.

On new MPP this division is not particularly defined. Both the models can live together.



Distributed memory



Shared memory

Programming models

Distributed memory systems:

-Message-passing approach. A paradigm to send and receive data among processes and synchronize them

-Message-passing libraries:
- -MPI
- -PVM (out-of-date...)

Shared memory systems:

-Thread based programming approach
-Compiler directives (i.e. OpenMP)
-Can be used together with message-passing paradigm

Parallel approaches

There are several typical cases of problems that are suitable for a parallel approach:

-Linear algebra (or FT problems)

  -Problems where an inversion/multiplication/diagonalization of matrices can be partitioned on different processing units. Communications can be important and it be can be an important limit to the scalability.

-Domain (or functional) decomposition

  -Large size problems can be partitioned on different processors. Typical examples are CFD, geophysical problems, molecular dynamics. Communication is the limit if the domains are correlated by forces, interactions and so on.

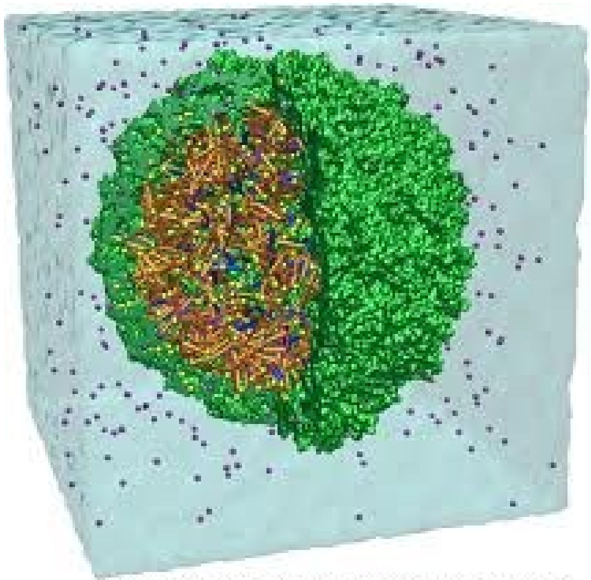
-Parametric studies (high-througput)

  -Sensitivity approaches,

  -Montecarlo calculations,

  -Ensemble techniques

    -Communication is close to zero, high scalability.



Theoretical and Computational Biophysics Group
Beckman Institute
University of Illinois at Urbana-Champaign

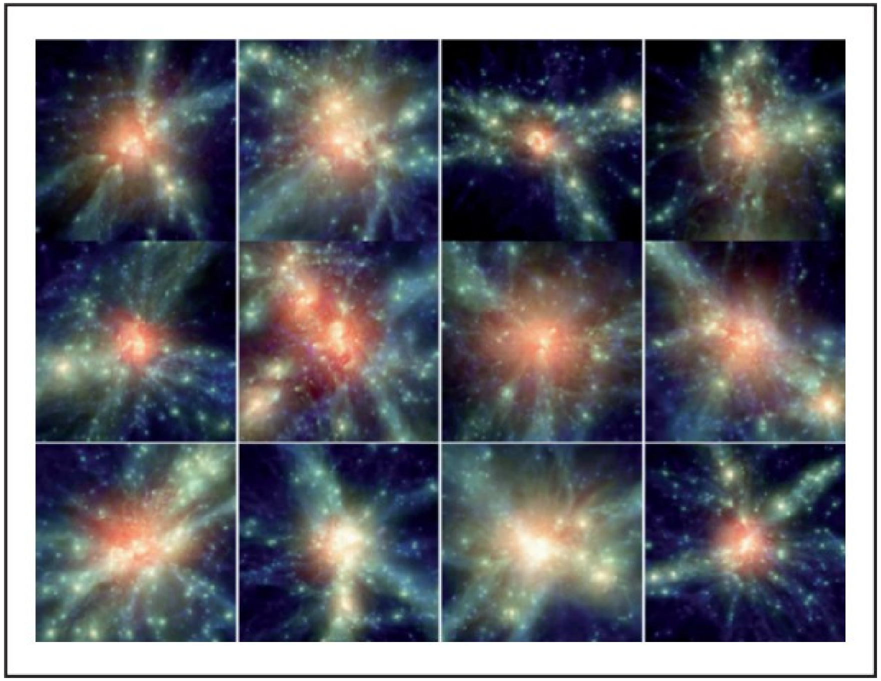
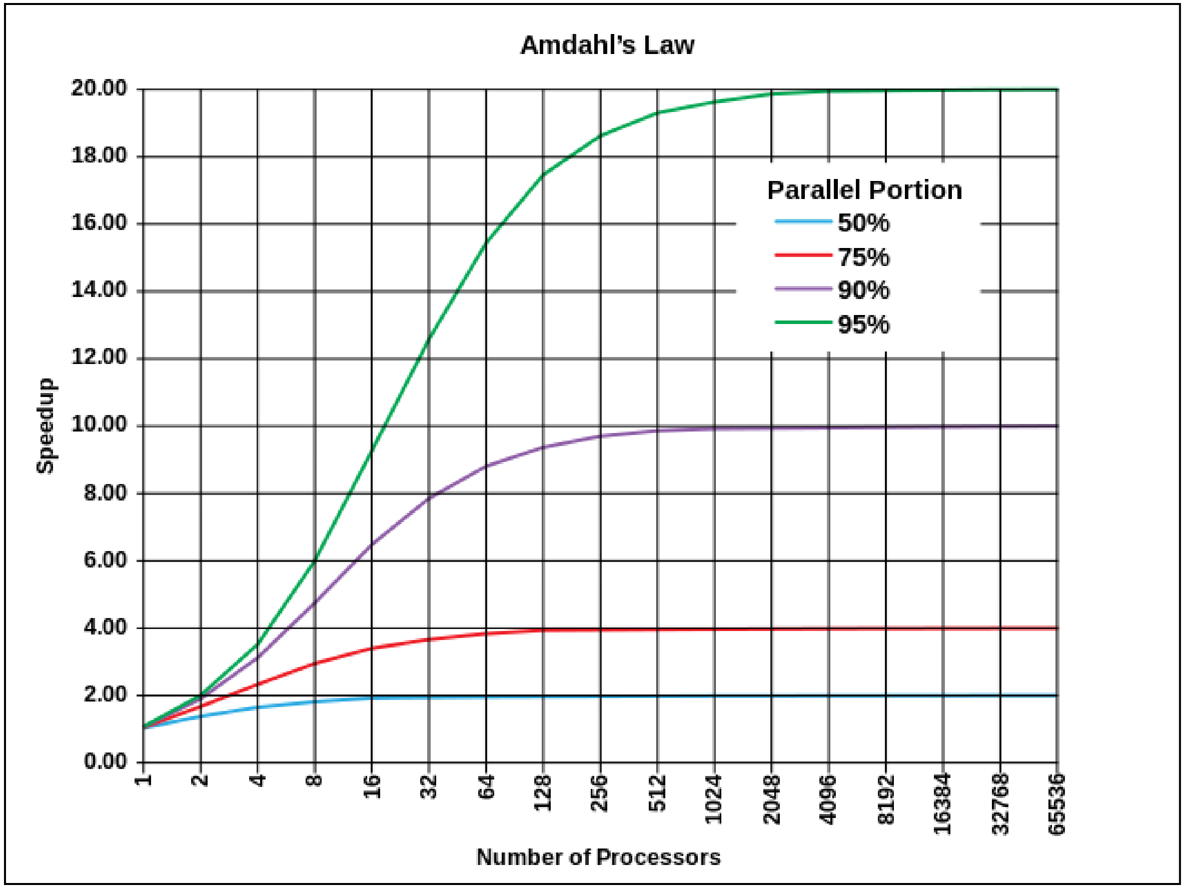


Fig. 1. – Ray-tracing images of a 15 Mpch−1 regions around the center of the individual clusters. Color coded is the temperature of the gas, while brighter regions correspond to higher-density gas.

Amdahl’s law

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



Message Passing Interface

-Each process has its own memory

-Processes communicate among themselves with “messages”

-The interface (API) for message passing is implemented in a library

-Code contains library calls

-The code must be deeply modified to implement message-passing algorithms

-If well implemented it can lead to high scalability

Message Passing Interface

-MPI is a message passing standard

- the 3.0 standard specification has been published very recently
- it enforces portability of the code to very different architectures
- it has an interface for Fortran/C/C++ languages
- (also a Python and Java interfaces are available)

Types of communications

Communications are the building blocks of MPI ☺

They can be distinguished in:

- Initialization, finalization and sync calls

- Point to point communications
  - deadlocks
  - blocking/non-blocking

- Collective calls (data movement, reduction operations, sync)
  - one to many
  - many to one
  - many to many

A summary of MPI

-Point-to-point communications

-Collective communications

communications

-One-sided communications

-Communicators

-User-defined datatypes

“structure”

-Virtual topologies

-MPI I/O

Hello world!

```
PROGRAM hello
IMPLICIT NONE
INCLUDE ‘mpif.h’
INTEGER:: myPE, totPEs, i, ierr

CALL MPI_INIT(ierr)

CALL MPI_COMM_RANK( MPI_COMM_WORLD, myPE, ierr )

CALL MPI_COMM_SIZE( MPI_COMM_WORLD, totPEs, ierr )

PRINT *, “myPE is “, myPE, “of total ”, totPEs, “ PEs”

CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

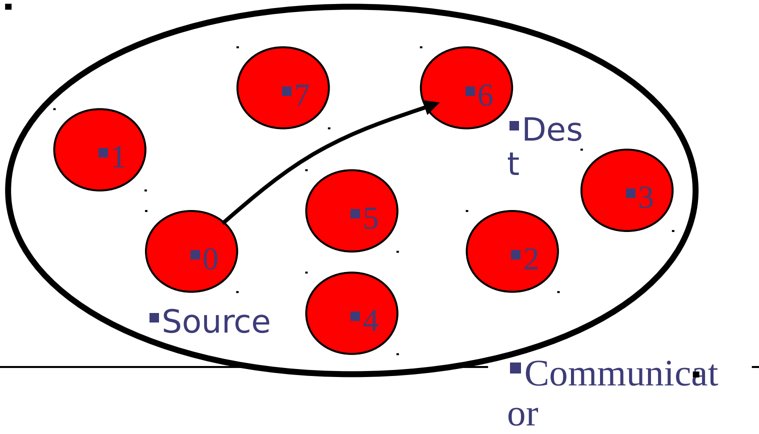
```
PROGRAM hello
IMPLICIT NONE
INCLUDE ‘mpif.h’              <- call the MPI library

CALL MPI_INIT                 <- let the play start

CALL MPI_COMM_RANK( )         <- who am I?

CALL MPI_COMM_SIZE( )         <- how many players are there?

                   DO SOME STUFF HERE


CALL MPI_FINALIZE(ierr)       <- let’s go back home

END PROGRAM hello
```

Point-to-point

- It is the basic communication method provided by MPI library.

Communication between 2 processes

- It is conceptually simple: source process A sends a message to destination process B, B receive the message from A.
- Communication take places within a communicator
- Source and Destination are identified by their rank in the communicator

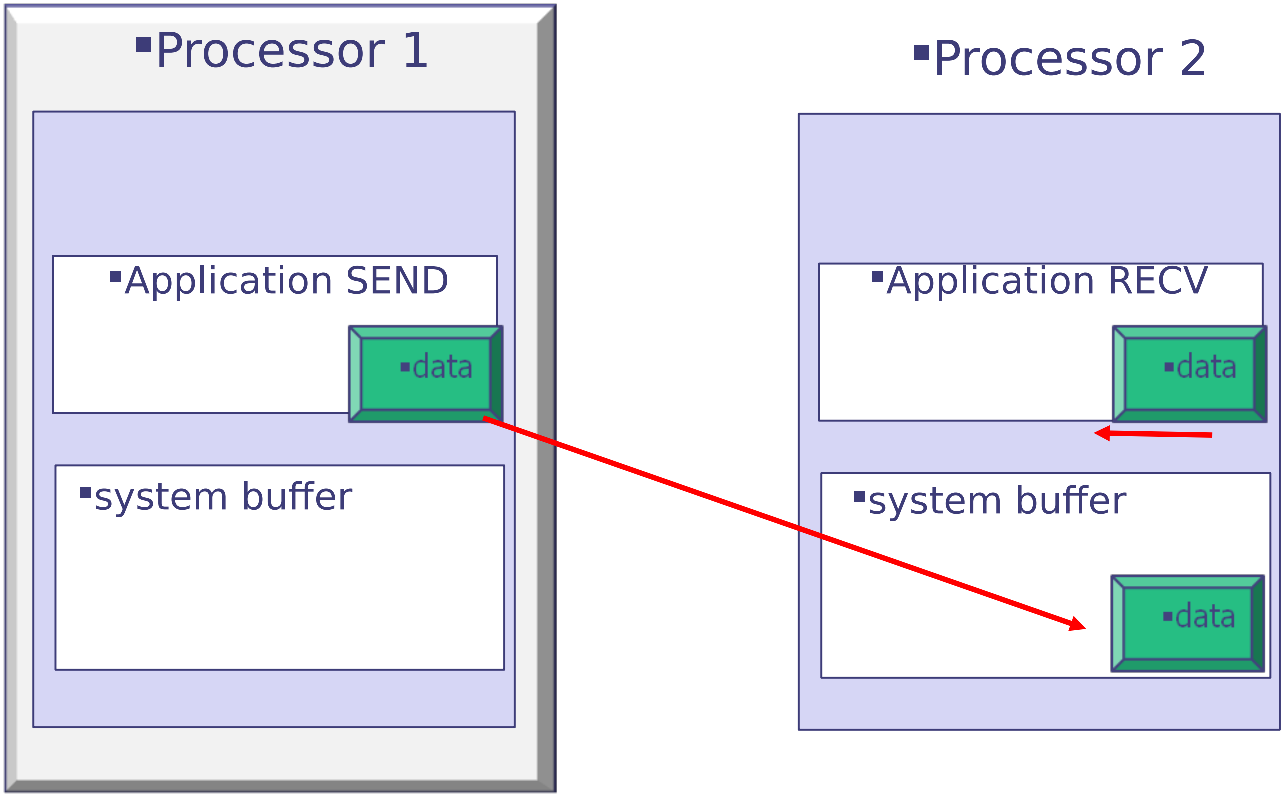


Blocking mode

Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

Blocking:

- A blocking send returns after it is safe to modify the application buffer (your send data) for reuse. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
- A blocking send can be synchronous
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- A blocking receive only "returns" after the data has arrived and is ready for use by the program.



Point-to-point flowchart

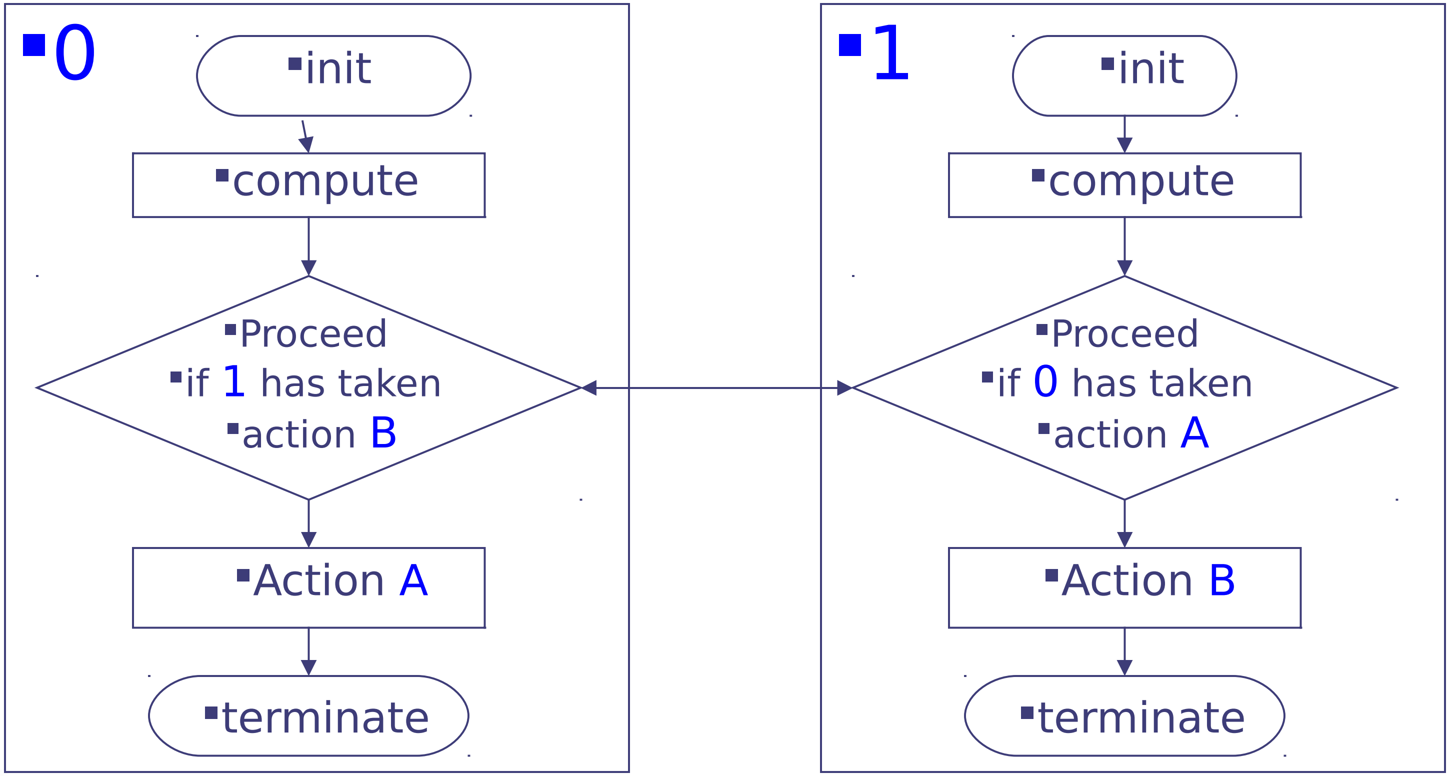
Non blocking

Non-blocking:

- Non-blocking send and receive routines will return almost immediately. They do not wait for any communication events to complete
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication.

Deadlock

Deadlock or a Race condition occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress.



```
PROGRAM deadlock
INCLUDE ‘mpif.h‘
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
END IF
WRITE(6,*) myid, ’: b(1)=’, b(1), ’ b(2)=’, b(2)
CALL MPI_FINALIZE(ierr)
END
```

```
PROGRAM avoid_lock
INCLUDE ‘mpif.h‘
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
END IF
WRITE(6,*) myid, ’: b(1)=’, b(1), ’ b(2)=’, b(2)
CALL MPI_FINALIZE(ierr)
END
```

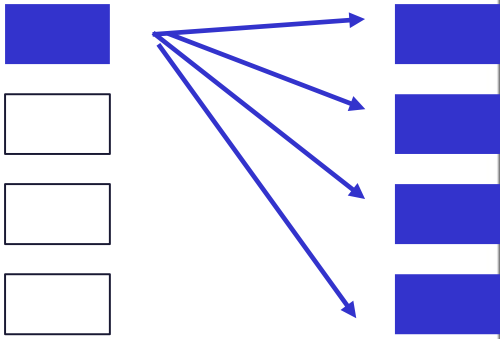
Collective communications

Communications involving a group of processes. They are called by all the ranks involved in a communicator (or a group)

- Barrier synchronization
- Broadcast
- Gather/scatter
- Reduction

- Collective communications will not interfere with point-to-point
- All processes (in a communicator) call the collective function
- All collective communications are blocking (in MPI 2.0)
- No tags are required
- Receive buffers must match in size (number of bytes)

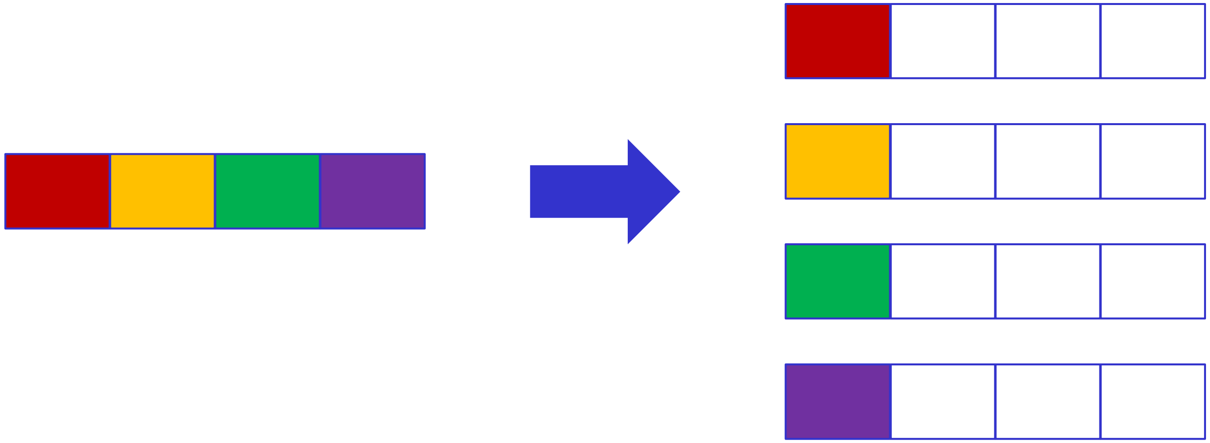
An example: Broadcast



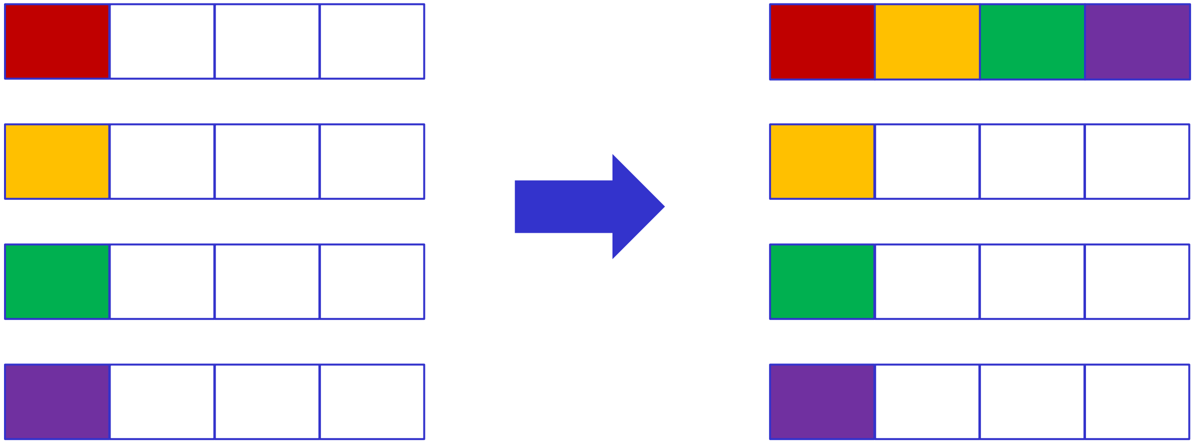
```
PROGRAM broad_cast
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
IF( myid .EQ. 0 ) THEN
             a(1) = 2.0
             a(2) = 4.0
END IF
CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
CALL MPI_FINALIZE(ierr)
```

Most used...

MPI_SCATTER



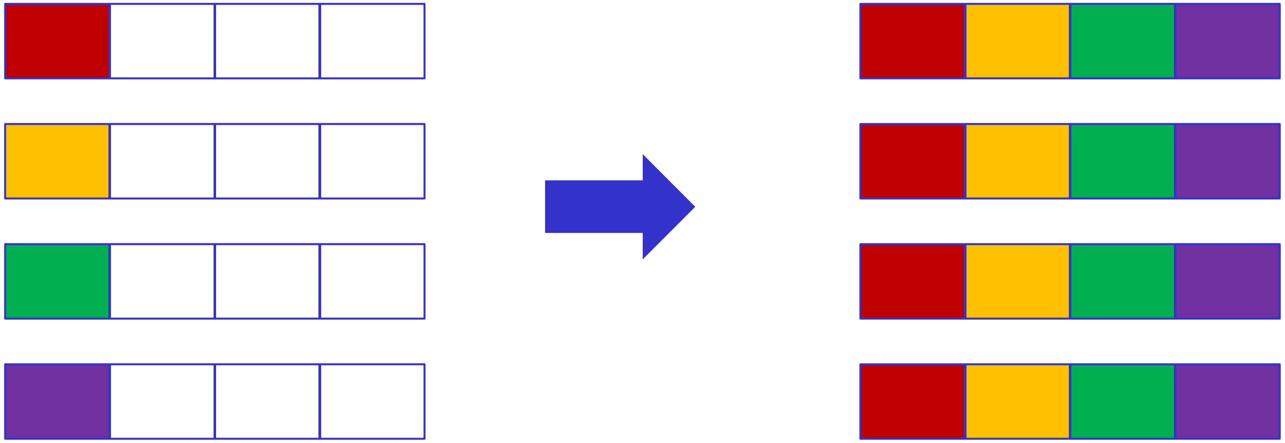
MPI_GATHER



There are possible combinations of collective functions.
For example,

MPI Allgather

It is a combination of a gather + a broadcast

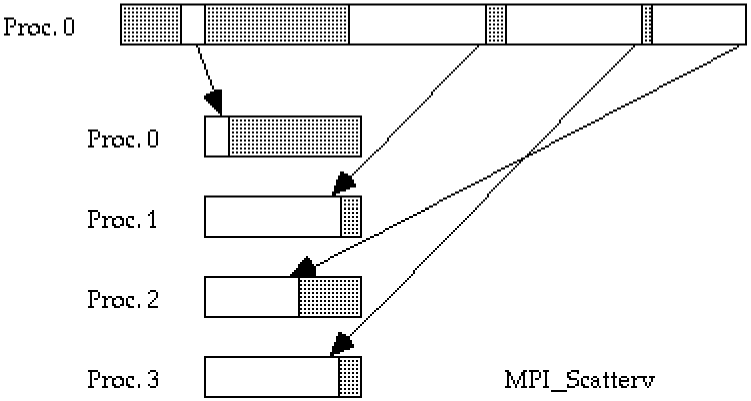


For many collective functions there are extended functionalities.

For example it’s possible to define the length of arrays to be scattered or gathered with

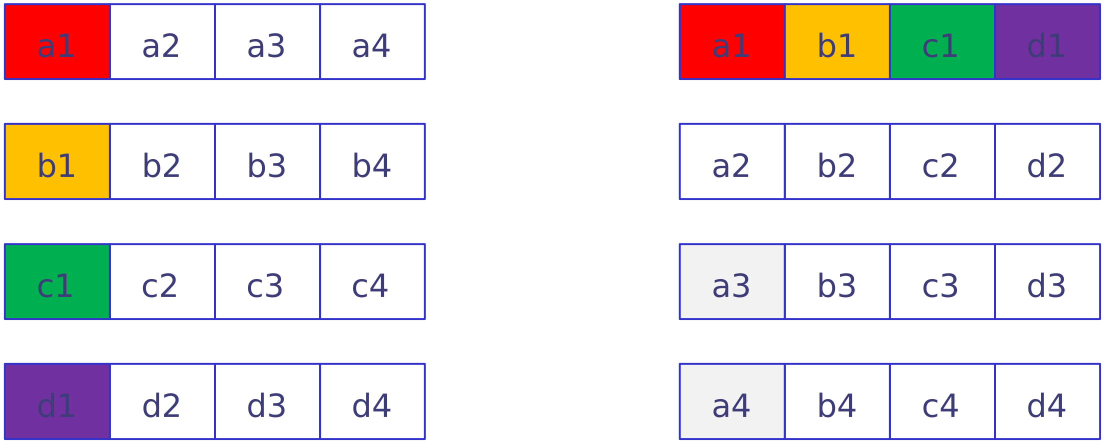
MPI_Scatterv

MPI Gatherv



MPI All to all

This function makes a redistribution of the content of each process in a way that each process know the buffer of all others. It is a way to implement the matrix data transposition.

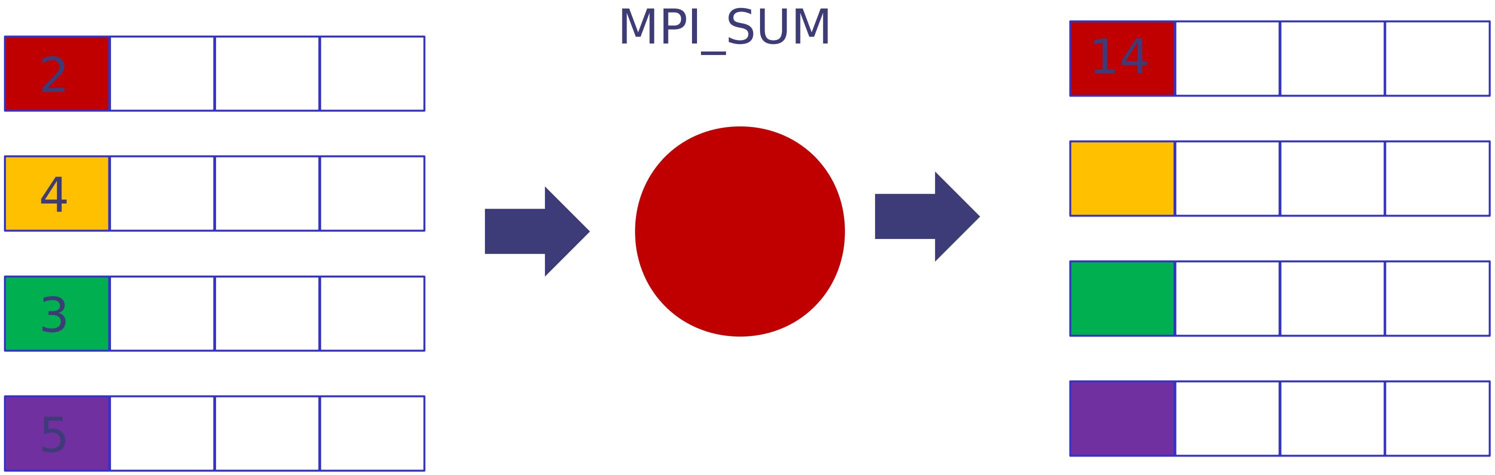


a1	a2	a3	a4
b1	b2	b3	b4
c1	c2	c3	c4
d1	d2	d3	d4

a1	b1	c1	d1
a2	b2	c2	d2
a3	b3	c3	d3
a4	b4	c4	d4

MPI Reduction

MPI_REDUCE



In addition to the default MPI_COMM_WORLD, MPI offers the possibility to create user-defined communicators to deal with the programmer’s needs.

Each communicator is defined on a group of MPI processes and it redefines the rank of each process within each communicator.

Collective functions take place only inside a defined communicator.

User defined communicators can be useful when one needs to manage several levels of parallelism inside the same code.

```
if(myid%2==0){
        color=1;
}else{
        color=2;
}
MPI_COMM_SPLIT(MPI_COMM_WORLD,color,myid,&subcomm);
MPI_COMM_RANK(subcomm,mynewid);
printf(“rank in MPICOMM_WORLD %d”,myid,”rank in Subcomm %d”,mynewid);
```

```
I am rank 2 in MPI_COMM_WORLD, but 1 in Comm 1.
I am rank 7 in MPI_COMM_WORLD, but 3 in Comm 2.
I am rank 0 in MPI_COMM_WORLD, but 0 in Comm 1.
I am rank 4 in MPI_COMM_WORLD, but 2 in Comm 1.
I am rank 6 in MPI_COMM_WORLD, but 3 in Comm 1.
I am rank 3 in MPI_COMM_WORLD, but 1 in Comm 2.
I am rank 5 in MPI_COMM_WORLD, but 2 in Comm 2.
I am rank 1 in MPI_COMM_WORLD, but 0 in Comm 2.
```

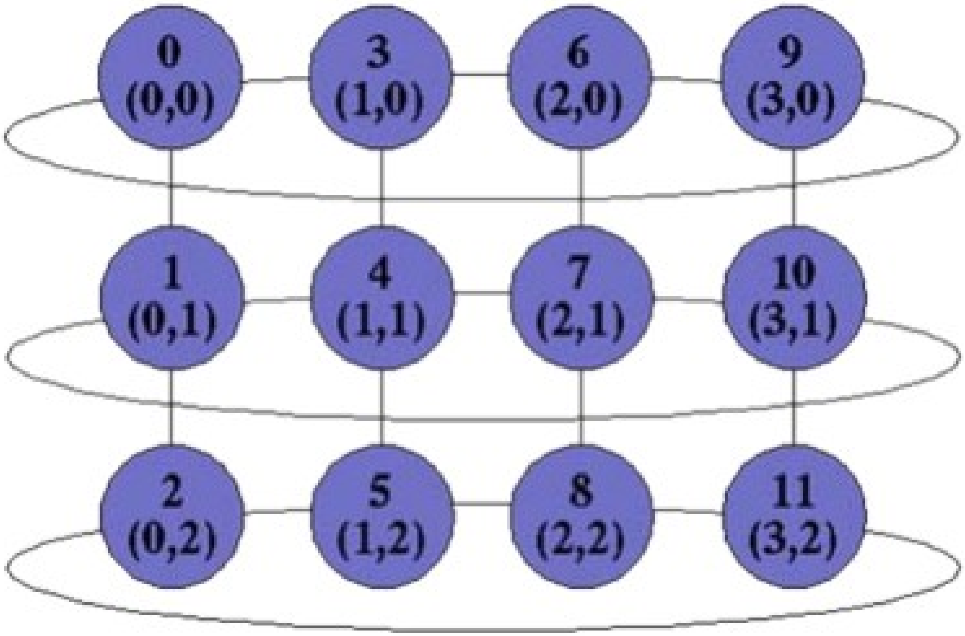
Virtual topologies are particular communicators that reflect a topology in the distribution of the processes.

A virtual topology can help the programmer to map a physical problem onto the MPI map of processes.

This semplifies the writing of the code and it permit to optimize communications.

MPI provides tools to manage virtual topologies with “mapping functions”.

Cartesian topology on a 2D torus



OpenMP

-It is an API extension to C/C++ and Fortran languages

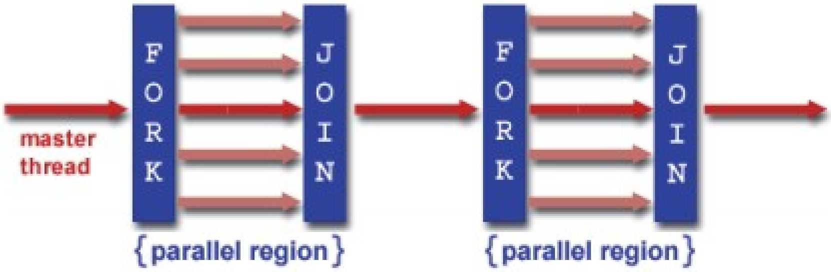
  -Most compilers support version 3.0

    -GNU, IBM, Intel, PGI, etc.

-Used for writing programs for shared memory architectures

OpenMP is based on a fork-join model

Master-worker threads



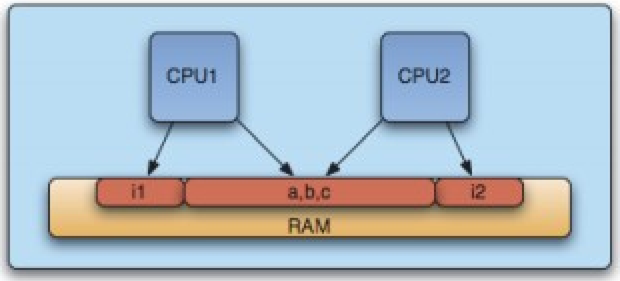
OpenMP is implemented through the use of pragmas directives within the source code

There’s no message passing: all the thread access the same memory address space: communication is implicit.

Programmers must take care to define:

- local data
- shared data

between threads.



```
PROGRAM hellomp
INTEGER numthreads, thread_id,omp_get_num_threads,omp_get_thread_num

!$OMP PARALLEL PRIVATE(NUM_THREADS, THREAD_ID)

NUM_THREADS=OMP_GET_NUM_THREADS()
THREAD_ID=OMP_GET_THREAD_NUM()

WRITE(*,*)’Hello world from thread num ”,THREAD_ID

!$OMP END PARALLEL

END
```

```
#include <iostream.h>
#include <omp.h>

using namespace std;

int main(int argc, char* argv[]){
        int thread, num_threads;
        #pragma omp parallel private(thread_id, num_threads)
        {
                #if defined(_OPENMP)
                        num_threads=omp_get_num_threads();
                        thread_id=omp_get_thread_num();
                #endif
        printf(“Hello world from thread num %d”,thread_id);
        }
}
```

API	description
int omp_get_num_threads()	returns the number of threads in the concurrent team
int omp_get_thread_num()	returns the id of the thread inside the parallel region
int omp_get_num_procs()	returns the number of processors in the machine
int omp_get_max_threads()	returns the max number of threads that will be used in the next parallel region
double omp_get_wtime()	returns the number of seconds since a time in the past

And more and more...

For each parallel region, the programmer should take care to define the data sharing attributes for each variable, through a number of clauses:

Clause	Description
default	It sets the default sharing attribute when no specified explicitly (caution!)
shared	Variable is common among threads
private	Variable inside the parallel construct is a new variable
firstprivate	Variable is new, but initialized to its original value
lastprivate	Variable's last value is copied outside the construct
reduction	Variable's value is reduced at the end among all threads

```
INTEGER X
X=1
!$ OMP PARALLEL SHARED(X) NUM_THREADS(2)
X=X+1
PRINT*, X
!$ OMP END PARALLEL
```

It will print EITHER 2 or 3

```
INTEGER X
X=1
!$ OMP PARALLEL PRIVATE(X) NUM_THREADS(2)
X=X+1
PRINT*, X
!$ OMP END PARALLEL
```

It will print ANYTHING

```
INTEGER X
X=1
!$ OMP PARALLEL FIRSTPRIVATE(X) NUM_THREADS(2)
X=X+1
PRINT*, X
!$ OMP END PARALLEL
```

It will print 2 TWICE

OpenMP provides several **synchronization mechanisms**:

-Barrier (synchronizes all threads inside the parallel region)

-Master (only the master thread will execute the block)

-Critical (only one thread at at time will execute)

-Atomic (same as critical but for one memory location)

-....

```
INTEGER X
X=1
!$OMP PARALLEL SHARED(X) NUM_THREADS(2)
X=X+1
!$OMP BARRIER
PRINT*,X
!$OMP END PARALLEL
```

```
3
3
```

```
INTEGER X
X=1
!$OMP PARALLEL SHARED(X) NUM_THREADS(2)
!$OMP MASTER
X=X+1
!$OMP END MASTER
PRINT*,X
!$OMP END PARALLEL
```

```
2
2
```

```
INTEGER X
X=1
!$OMP PARALLEL SHARED(X) NUM_THREADS(2)
!$OMP ATOMIC
X=X+1
PRINT*,X
!$OMP END PARALLEL
```

```
2
3
```

Worksharing constructs:

-Threads cooperate in doing some work

-Thread identifiers are not used in an explicit manner

-Most common use is in loop worksharing

-Worksharing constructs may not be nested

-DO/for directives are used in order to determine a parallel region

```
Int i,j;
#pragma omp parallel
#pragma omp for private(j)
for(i=0;i<N;i++)
{
  for(j=0;j<N,j++)
    m[i][j]=f(i,j);
}
```

-this loop is parallel on the i variable (private by default)

-j must be declared as private explicitly

-synchronization is implicitly obtained at the end of the loop

You may need that in some region a statement is executed by only one thread, no matter which one.

In this case you can use a **SINGLE** region.

```
....

!$omp parallel
...
!$omp single
read *,n
!$omp end single
...
!$omp end parallel
```

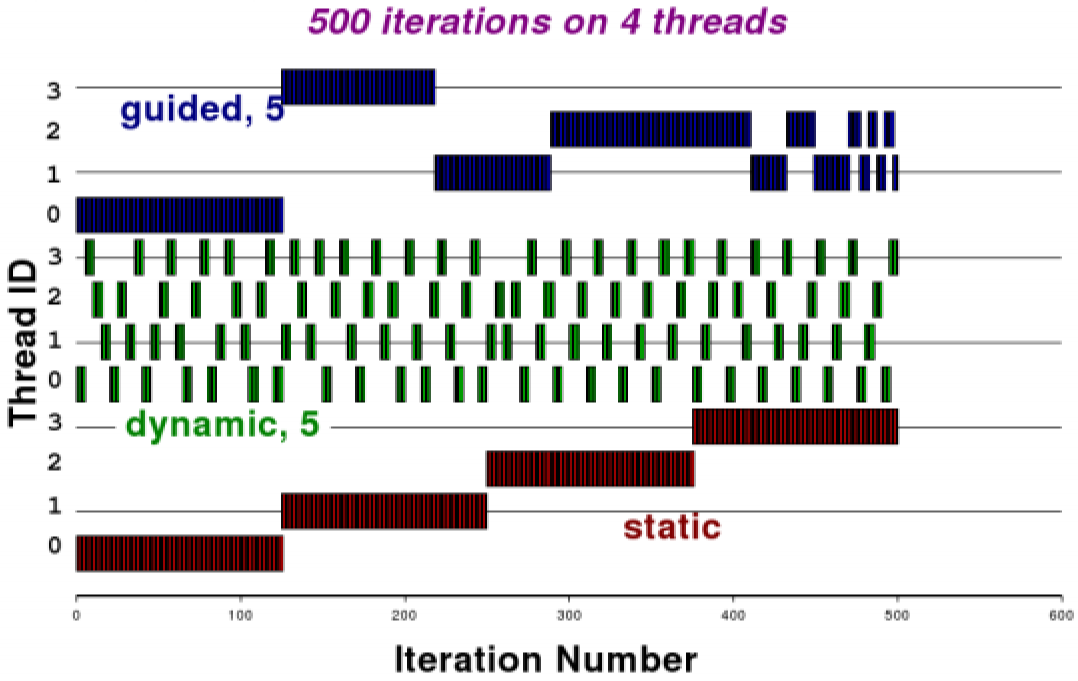
Using the **schedule** clause one can determine the distribution of computational work among threads:

-static, chunk: the loop is equally divided in pieces of size chunk whic are evenly distributed among threads in a “round-robin” way

-dynamic,chunk: the loop is equally divided among pieces chunk that are distributed for execution dynamically to threads. If no chunk is specified, then chunk=1

-guided: similar to dynamic with the variation that chunk size is reduced as threads grab iterations

This is configurable with environment variable OMP_SCHEDULE:
-i.e. setenv OMP_SCHEDULE “dynamic,4”


500 iterations on 4 threads
guided, 5
dynamic, 5
static
Thread ID
3
2
1
0
3
2
1
0
3
2
1
0
Iteration Number
0
100
200
300
400
500
600

The **reduction** clause can be used when a variable is accumulated at the end of a loop.

Using the reduction clause:

-a private copy per thread is created and initialized

-at the end of the region the compiler safely updates the shared variable

```
!$omp do reduction(+:x)
do i=1,n
  x=x+a(i)
End do
!$omp end do
```

False sharing

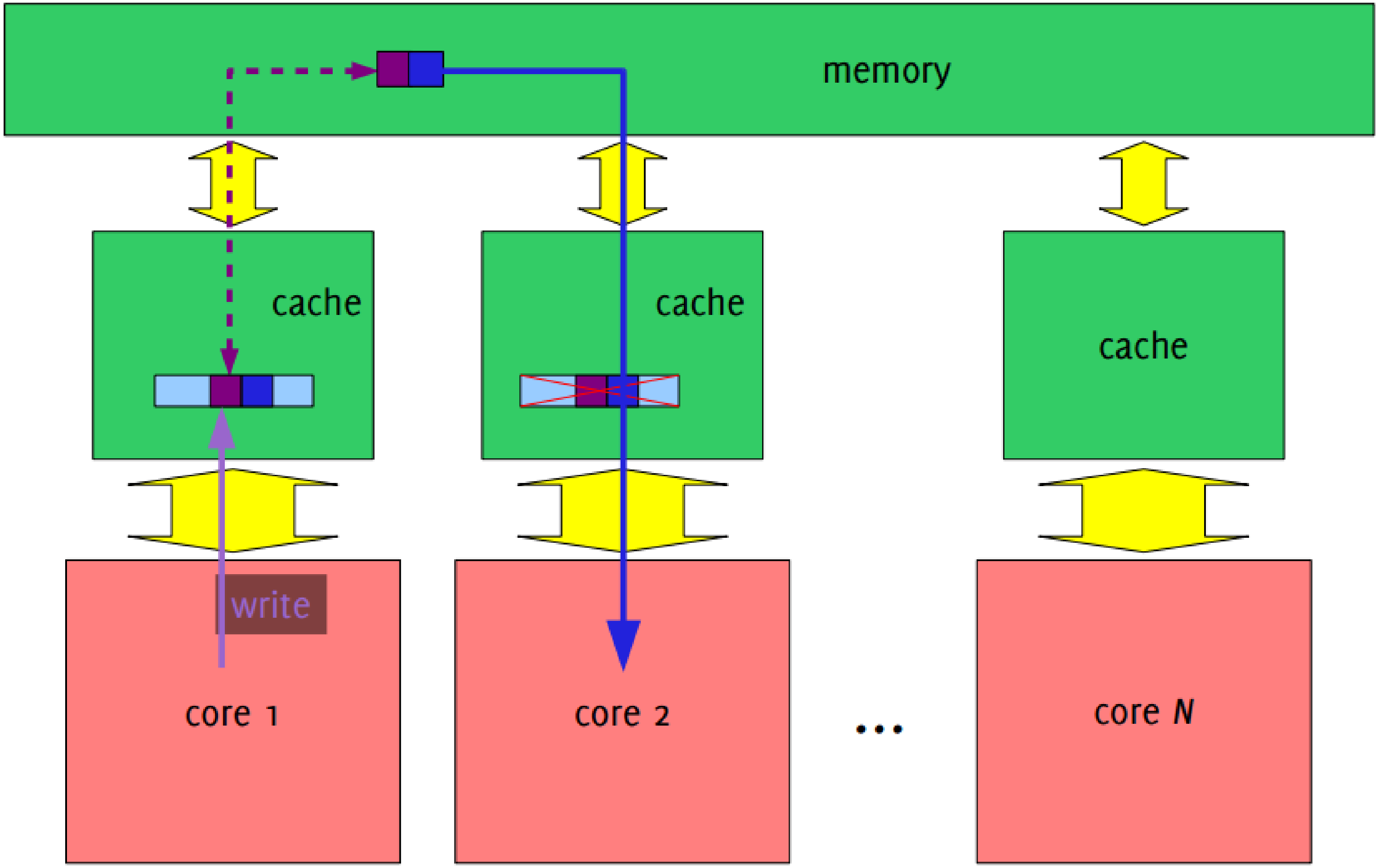
-Memory addresses are grouped into cache lines

-If one element of the cache line is changed, the whole line is invalidated

-If more than one thread is working on the same cache line, this cache line is continously invalidated and there’s a lot of traffic between cache and memory.

-False sharing is one of the reason of poor scalability in pure OpenMP approaches

```
float data[N], total=0;
int ii;
#pragma omp parallel num_threads(N)
{
  int n = omp_get_thread_num();
  data[n] = 0;
  while(moretodo(n))
    data[n] += calculate_something(n);
}
for (ii=0; ii<N; ii++)
  total += data[n];
```


memory
cache
cache
cache
write
core 1
core 2
...
core *N*

```
float data[N], total=0;
int ii;
#pragma omp parallel num_threads(N) private(data)
{
  int n = omp_get_thread_num();
  data[n] = 0;
  while(moretodo(n))
    data[n] += calculate_something(n);
}
for (ii=0; ii<N; ii++)
  total += data[n];
```

- use private attribute

- use padding

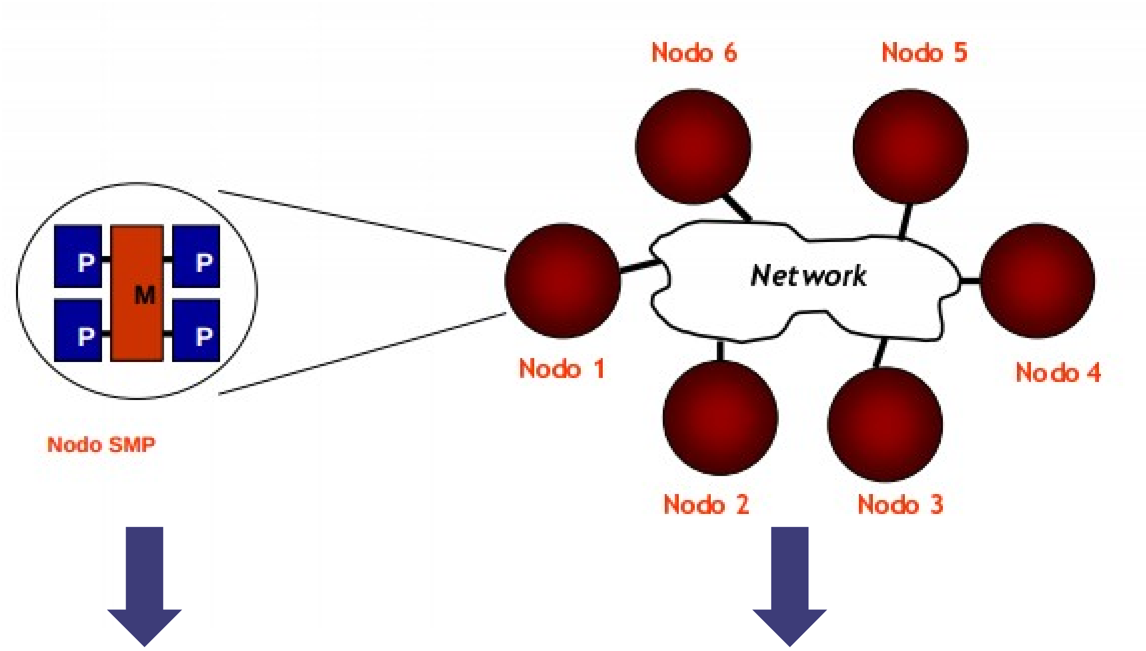
```
#pragma omp parallel for
   for (j=0; j<N; j++) {
      for (i=0; i<M; i++) {
         image[ i + j*M ] /= 2;
      }
   }
```

Each thread accesses a contiguous region in memory:
not much false sharing.

```
#pragma omp parallel for
   for (i=0; i<M; i++) {
      for (j=0; j<N; j++) {
         image[ i + j*M ] /= 2;
      }
   }
```

Each thread accesses alternating memory elements:
a lot of false sharing!

Using together MPI and OpenMP



OpenMP

MPI

Since MPI2, a support is provided for hybrid programming MPI+OpenMP:

MPI_INIT_THREAD(required, provided,ierr)

There are 4 levels supported:

-MPI_THREAD_SINGLE: no threads are allowed
-MPI_THREAD_FUNNELED: threads are allowed; Only the master thread can call MPI primitives
-MPI_THREAD_SERIAL: threads are allowed. All threads can call MPI primitives. Communications are scheduled in a serial manner.
-MPI_THREAD_MULTIPLE: threads are allowed. All threads can call MPI communication primitives in an arbitrary order

```
INCLUDE ‘mpif.h’
INTEGER :: rnk,sz,n,i,ierr,chunk
INTEGER,PARAMETER :: n=100
REAL*8  :: x(n),y(n),buff(n)

CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rnk,ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,sz,ierr)

chunk=n/sz

CALL MPI_SCATTER(buff(rnk*chunk),chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,   &
&               MPI_COMM_WORLD,ierr)
CALL MPI_SCATTER(buff(rnk*chunk),chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,   &
&               MPI_COMM_WORLD,ierr)

DO i=1,chunk
  x(i)=x(i)+y(i)
END DO

CALL MPI_GATHER(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD,ierr)

CALL MPI_FINALIZE(ierr)

END
```

```
...

INTEGER :: i
INTEGER, PARAMETER :: n=100
REAL*8  :: x(n),y(n),buff(n)



!$OMP PARALLEL DO PRIVATE(i) SHARED(x,y)
DO i=1,n
  x(i)=x(i)+y(i)
END DO
```

```
INCLUDE ‘mpif.h’
INTEGER :: rnk,sz,n,i,ierr,info,chunk
INTEGER,PARAMETER :: n=100
REAL*8  :: x(n),y(n),buff(n)

CALL MPI_INIT_THREAD(MPI_THREAD_FUNNELED,info,ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rnk,ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,sz,ierr)

chunk=n/sz

CALL MPI_SCATTER(buff(rnk*chunk),chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,   &
&               MPI_COMM_WORLD,ierr)
CALL MPI_SCATTER(buff(rnk*chunk),chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,   &
&               MPI_COMM_WORLD,ierr)

!$OMP PARALLEL DO
DO i=1,chunk
  x(i)=x(i)+y(i)
END DO
!$OMP END PARALLEL

CALL MPI_GATHER(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD,ierr)

CALL MPI_FINALIZE(ierr)

END
```

Task parallelism

Definition:

A task parallel computation is one in which parallelism is applied by performing distinct computations (i.e. tasks) at the same time. Since the number of tasks is fixed, this parallelism is not scalable

OpenMP 3.0 supports tasks parallelism

Tasks parallelism introduces a functional dependence between tasks.

Is similar to a sort of “pipelining” of functional units

OpenMP task construct

```
#pragma omp task [clauses]
{
      do something
}
```

Where clauses is: private, shared, default, if, untied, final, mergeable

-Immediately creates a new task but not a new thread
-This task is explicit (also threads can be thought as implicit tasks...)
-It will be executed by a thread in the current team
-It can be deferred until a thread is available to execute
-The data environment is built at creation time