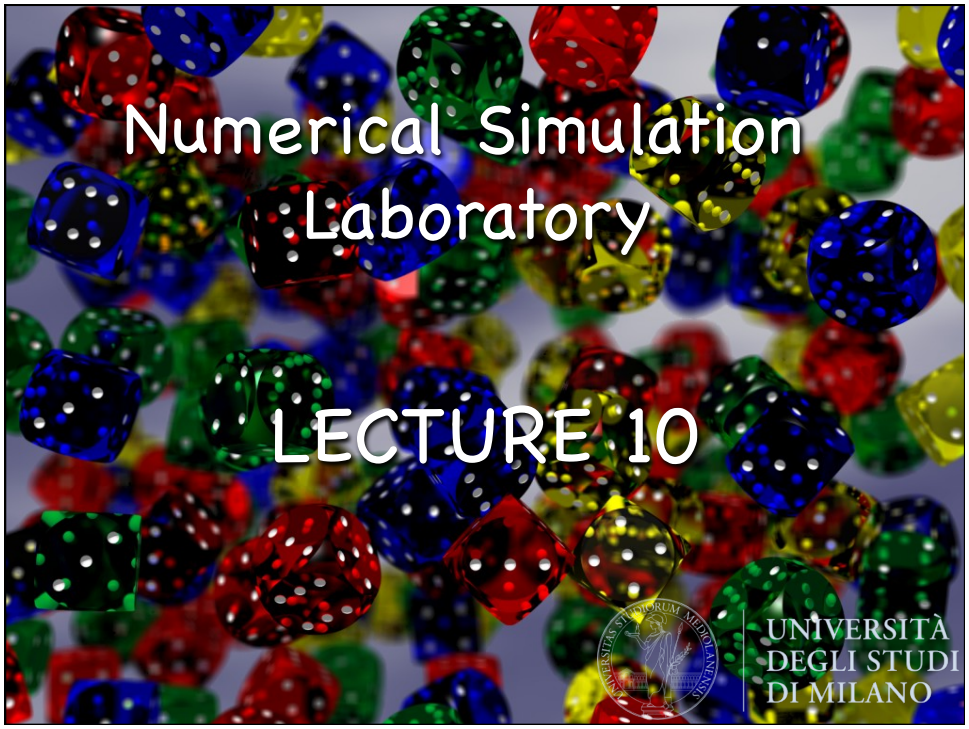


Outline

Introduction to parallel computation:

- Moore's law
- Parallel computing
- Parallel computers
- Single Program Multiple Data paradigm
- Measuring performances
- Amdahl's law
- Message Passing Interface

- Scope: Unity makes strength!

Computational complexity

- Not all computational problems are created equal. Some problems are so simple that even large instances may be solved rapidly. Other problems are intractable – once the problem is large enough there is essentially no hope of solving it exactly.
- Computer scientists have formalized this observation, and divided problems into computational complexity classes of varying difficulty. Knowing the complexity class of a problem won't solve it, but it will help you decide what kind of solution method is appropriate.
- If the problem is hard, finding an exact solution is apt to be costly or impractical, and you will probably have to resort to enumerative methods (which may be slow or useless for large cases) or settle for an approximate solution obtained with heuristic/stochastic methods
- How long does it take to find the maximum in a set of numbers? The answer should be a function of the size of the particular instance to be solved. The function tells how the algorithm run time grows as the input size increases. Suppose the numbers are A(i): i=1, …, n. The obvious algorithm finds the maximum in time proportional to n. We say the algorithm runs in linear time, or is O(n).

- If instead we wish to sort the numbers in ascending order, the fastest algorithms (e.g. quicksort) require time proportional to n log n. We say that sorting is done in O(n log n) time.
- The major distinction we make between algorithms is whether they take polynomial time or not. An algorithm requires polynomial time if for some k it has a time bound of $O(n^k)$
- In a decision problem, given an input, we are required to give a Boolean YES/NO (1/0) answer. That is, in a decision problem we are only asked to verify whether the input satisfies a certain property.
- We denote by **P** the class of decision problems that are solvable in polynomial time. A standard convention is to call an algorithm "feasible" if belongs to P, i.e. if there is some polynomial p such that the algorithm runs in time at most p(n) on inputs of length n.
- Most algorithms that are not polynomial are exponential, requiring more than c^{dn} time in the worst case, for some constants c>1 and d>0. Any exponential function will eventually exceed any polynomial function as n increases. In practice, exponential time algorithms are usually slower than polynomial time algorithms even for quite modest values of n.

NP complexity

- Let us consider the subset sum problem: Assume that we are given some integers, such as {-7, -3, -2, 5, 8}, and we wish to know whether some of these integers sum up to zero.
- In this example, the answer is "yes", since the subset of integers {-3, -2, 5} corresponds to the sum (-3) + (-2) + 5 = 0. The task of deciding whether such a subset with sum zero exists is called the subset sum problem.
- To answer if some of the integers add to zero we can create a combinatorial algorithm which obtains all the possible subsets. As the number of integers that we feed into the algorithm becomes larger, the number of subsets grows exponentially (2^n) and so does the computation time, in fact:

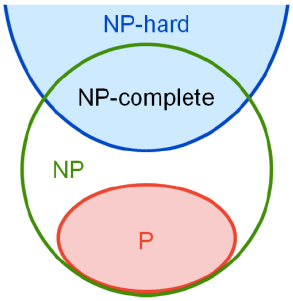
Binomial Theorem:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k \underset{x=y=1}{\Rightarrow} 2^n = \sum_{k=0}^{n} \binom{n}{k}$$

# of ways of choosing k elements from the set with n elements

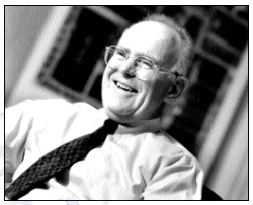
- However, notice that, if we are given a particular subset, we can easily verify whether the subset sum is zero, by just summing up the integers of the subset. So if the sum is indeed zero, that particular subset is the proof for the fact that the answer is "yes".

NP-complete complexity



- An algorithm that verifies whether a given subset has sum zero is called a verifier
- More generally, a problem is said to be **NP** (nondetermistic polynomial-time) if there exists a verifier V for the problem and the answer can be obtained in polynomial time
- The class of problems which are at least as hard as the hardest problems in NP is called **NP-hard** (nondetermistic polynomial-time hard problem)
- Finding the ground state (minimum energy state) of a generic Ising model is NP-hard (Barahona, 1982). So, maybe we can map hard problems on an Ising model …
- Problems that are NP-hard do not have to be necessarily elements of NP; indeed, they may not even be decidable. So let's take the intersection of these two sets: **NP-complete**
- In computational complexity theory, **NP-complete** problems are contained in NP; although any given solution to an NP-complete problem can be verified quickly (in polynomial time… in fact, it is in NP), there is no known efficient way to locate a possible solution

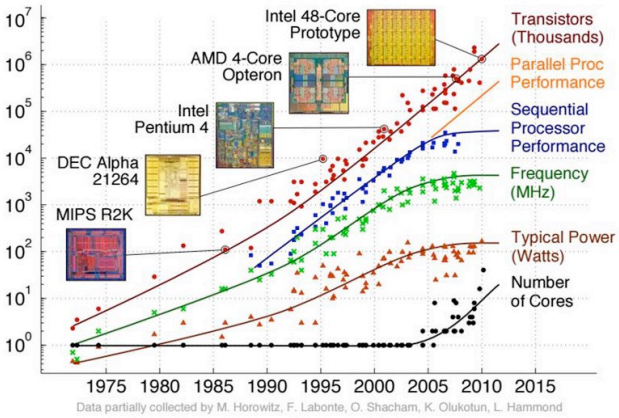
Moore's law



- Gordon E. Moore (co-founder of Intel) predicted (Electronics, Vol. 38, 1965) that transistor density of a semiconductor chip is going to double about every 2 years …

⇒ double performances about every 2 years

- It is not a law! It is a prediction on the progress of physical devices and engineering processes… but, it works … since 50 years!

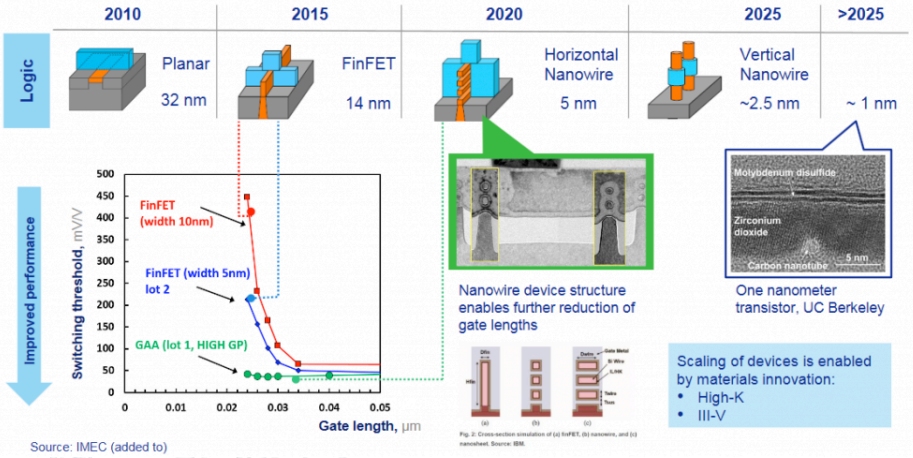


- But, how long? It seems up to 2025. Note, that recently we had to move towards multi-core architectures

Logic device and shrink roadmap

New devices for 5 nm and beyond are demonstrated to work

ASML



- Present: Parallel computing …
- Future: Natural computing (e.g. Quantum computation? But it seems we are still far from a *Universal Quantum computer*)

Natural Computation

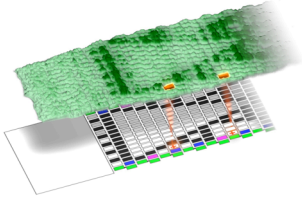
- Natural computing encompasses three classes of methods:

1. Methods based on the use of computers to synthesize natural phenomena (Computational Intelligence, Neural Networks, Genetic Algorithms, etc.)
2. Methods that employ natural materials (electrons, atoms, molecules, DNA) to compute.
3. Methods that take inspiration from nature for the development of novel computational architectures or problem-solving techniques

- A remarkable example of the second class, which is also a remarkable example of (this time expected) emergence/self-organization in a complex system is DNA computing.
- DNA computing is a form of parallel computing that uses DNA, biochemistry, and molecular biology hardware, instead of the traditional silicon-based computer technologies
- 4 key components coding: in principle 1 liter of fluid with 6 grams of DNA could memorize 3 Zbytes and perform up to 1 EFLOPS

(Zetta=10^{21}) (Exa=10^{18})

DNA thin slab self-assembly of a Sierpinski triangle as a Cellular Automata, *Nano Letters* (2007)

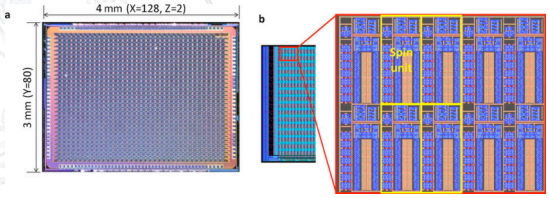


New computational architectures

- Along the third class of Natural computing methods, recently, several special hardwares or physical systems for solving Ising models have been proposed.

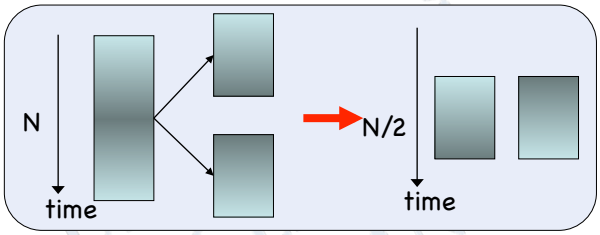
1. D-Wave machines, based on quantum annealing
   - Dwave One (2011, 128 qubits)
   - Dwave Two (2013, 512 qubits)
   - Dwave 2X (2015, 1152 qubits)

2. (classical) Ising chip by Hitachi (2015) 20480 spins based on CMOS (complementary metal-oxide semiconductor) annealing



3. Coherent (quantum) Ising Machine (Optical/laser networks, 2011/2015)

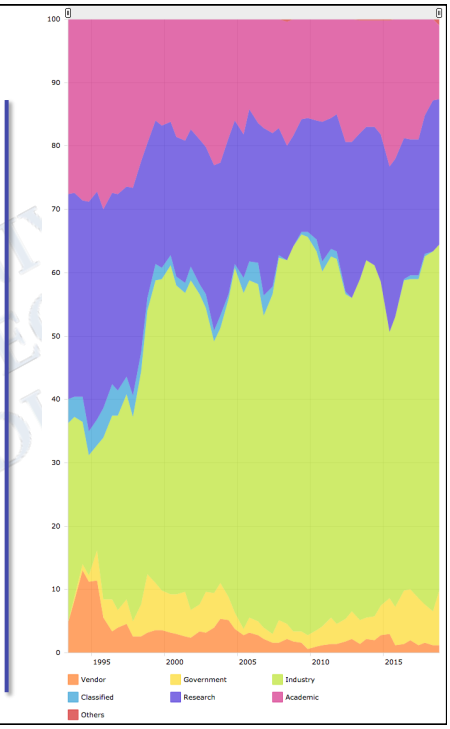
What is parallel computing?



- Split a single application code to run across multiple processors
- Computational time may be reduced:
  in principle if the number of CPU is p the time t_p involved to complete the computational task could be reduced to $t_p=t_1/p$ (being t_1 the time needed to complete the task serially)
- Memory requirements per processor may be reduced
- Rather then using one expensive processor (...maybe waiting some years to have it on the market), many less expensive processors can be used
- Programming is now more difficult, because dividing a problem to run efficiently across multiple processors is a difficult task
- The efficiency depends on the algorithms and on the characteristics of the parallel computer

Parallel computing examples

- Nature:
  - Our brain is made of about 10^{11} neurons
  - The living organism are made from cells
  - Leaves of trees
  - Etc.etc.
  - ... Reality ?
- Society:
  - Highway tollgates
  - Supermarket cash registers
  - Written exams
  - Etc.etc.



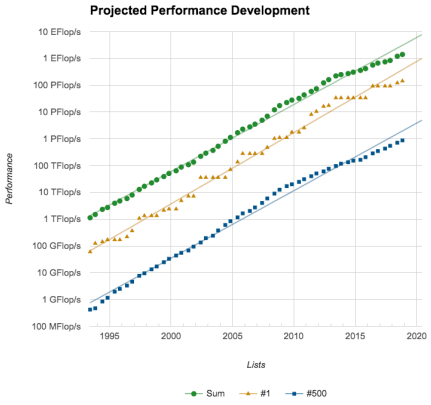
Units of High Performance Computing (HPC)

- Flop/s: floating points operation per second
- Bytes: dimension of data

Typical dimensions:

Mega 10^6	Mflop/s		Mbyte	(giurassic)
Giga	10^9	Gflop/s	Gbyte	(prehistory)
Tera	10^{12}	Tflop/s	Tbyte	(history)
Peta	10^{15}	Pflop/s	Pbyte	(present)
Exa	10^{18}	when ?		
Zetta 10^{21}	... ??			
Yotta 10^{24}	... ???			

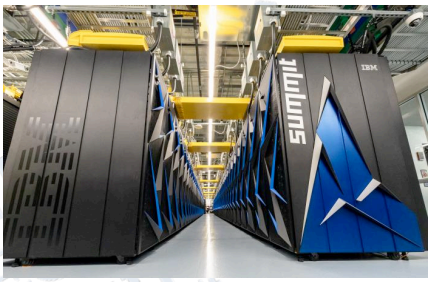
- Expected projected performance is changing: mainly due to electric power consumption!
  #1 : 18 MegaWatt !!!



High Performance Computing (HPC)

- Present Number 1 (Autumn 2018):

Summit - IBM Power System AC922
CPU cores: 2,397,824
Site: Oak Ridge National Laboratory
Peak performance: 143 Pflop/s



- First in Italy (N.19 in Autumn 2018)

Marconi - Lenovo Cluster
CPU cores: 348,000
Site: CINECA (Bo)
Peak performance: 18 Pflop/s

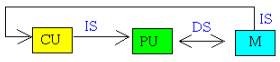
Parallel computers

- If we define a parallel computer as "a collection of computing elements able to communicate and to cooperate in order to solve quickly large computational tasks" … then we have to answer to the following questions:
- "…a collection of computing elements…"
  - How many?
  - What is the computational power of each one of them?
  - What kind of memory they have?
- "…able to communicate…"
  - How they are connected?
  - What kind of communication protocol they use?
- "…and to cooperate…"
  - Is there any synchronization mechanism?
  - What is their operating systems?
- "…in order to solve…"
  - What algorithms are parallelizable?
  - How can I parallelize an algorithm?
  - Automatic or manual parallelization?

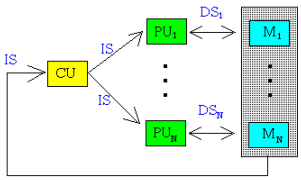
Parallel computers: Flynn's classification

	Data flux: single	Data flux: multiple
Instruction flux: single	SISD Serial computing	SIMD Vector computing
Instruction flux: multiple	MISD Evolute SISD	MIMD Multi-processors computing

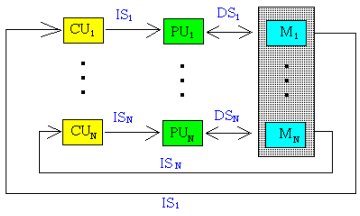
SISD:



SIMD:



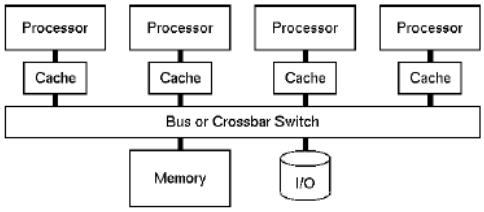
MIMD:



- CU: control unit
- PU: processing unit
- M: memory

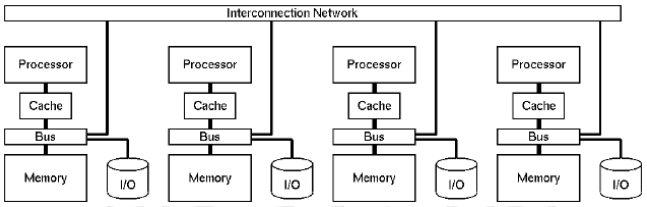
Parallel computers: memory classification

	Shared address space	Individual address space
Shared memory	SMP symmetric multiprocessor	N/A
Distributed memory	NUMA non uniform memory access	MPP massively parallel processor



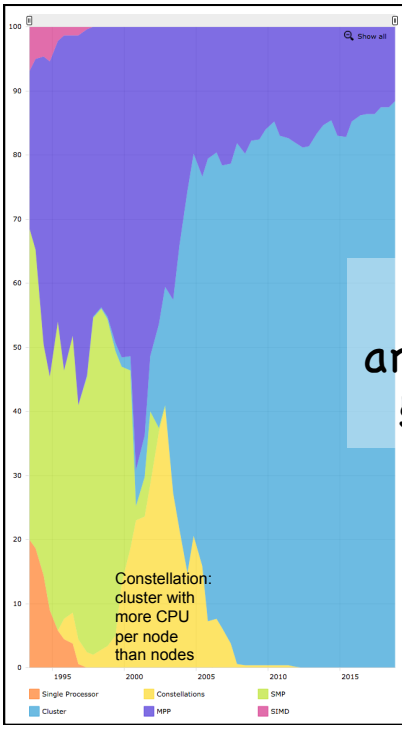
- SMP architecture uses shared system resources such as memory and I/O subsystem that can be accessed equally from all the processors. As shown in figure, each processor has its own cache which may have several levels.
- SMP machines have a mechanism to maintain coherency of data held in local caches.
- A single operating system controls the SMP machine and it schedules processes and threads on processors so that the load is balanced.

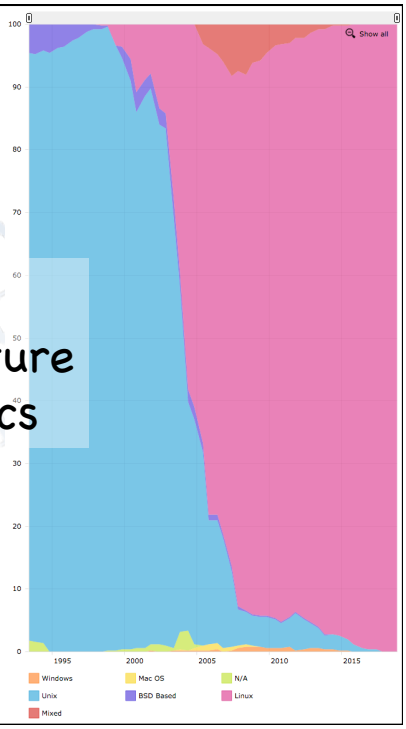
- MPP architecture consists of nodes connected by a network that is usually high-speed. Each node has its own processor, memory, and I/O subsystem (see the figure).



- The operating system is running on each node, so each node can be considered a workstation.
- Despite the term *massively*, the number of nodes is not necessarily large. In fact, there is no criteria. What makes the situation more complex is that each node can be an SMP node (typical situation!)
- NUMA architecture machines are built on a similar hardware model as MPP, but it typically provides a shared address space to applications using a hardware/software directory-based protocol that maintains cache coherency.
- As in an SMP machine, a single operating system controls the whole system. The memory latency varies according to whether you access local memory directly or remote memory through the interconnect. Thus the name *non-uniform memory access*.

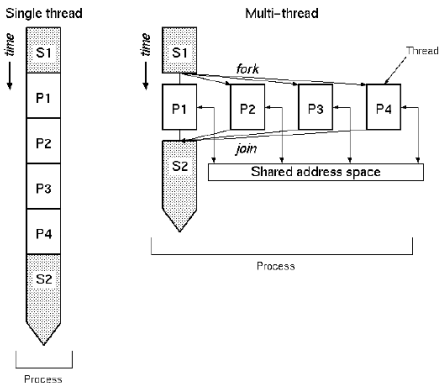
HPC architecture statistics





Symmetric Multiprocessor

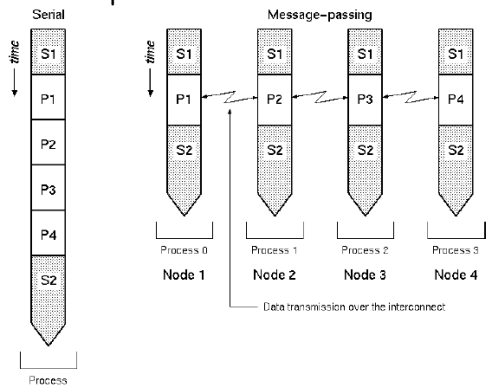
- The shared address space simplify coding
- Automatic parallelization via compiler possible: the compiler automatically parallelizes certain types of DO loops… rarely efficient!
- Generally, parallelization efficiency decreases by increasing the number of CPU: not scalable



- Multi-threaded programs are the best fit with SMP because threads that belong to a process share the available resources.
- In figure, single-thread program processes S1 through S2, where S1 and S2 are inherently sequential and P1 through P4 can be processed in parallel. The multi-thread program proceeds in the *fork-join model*. It first processes S1, and then the first thread forks (creates) three threads. The four threads process P1 through P4 in parallel, and when finished they are joined to the first thread.

MPP based on single-processor nodes

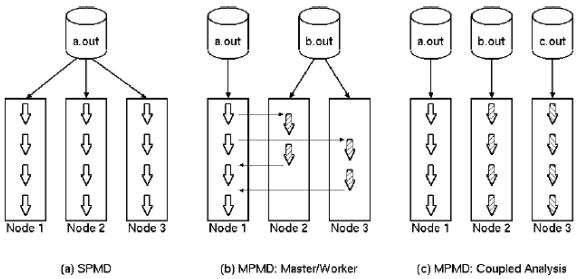
- If the address space is not shared among nodes, parallel processes have to transmit data over an interconnecting network in order to access data that other processes have updated.
- One can have many different interconnection geometries
- Scalability is algorithmic dependent but can be efficient
- Manual parallelization



- One process runs on each node and the processes communicate with each other during the execution of the parallelizable part, P1-P4. In general, each process communicates with all the other processes. Due to the communication overhead, work load unbalance, and synchronization, time spent for processing each of P1-P4 is generally longer in the message-passing program than in the serial program. All processes in the message-passing program are bound to S1 and S2.

Message passing: parallel programming models

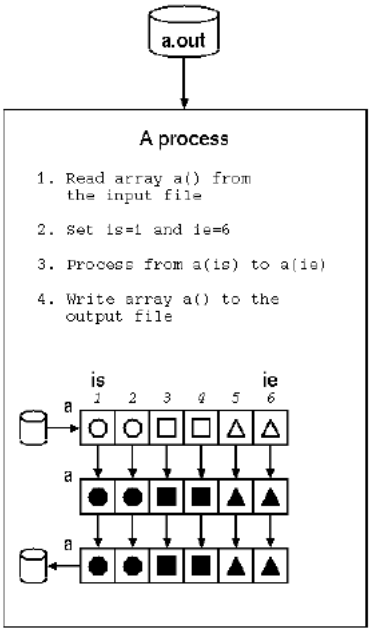
- When you run multiple processes with message-passing, there are further categorizations regarding how many different programs are cooperating in parallel execution.



- In the SPMD (Single Program Multiple Data) model, there is only one program and each process uses the same executable working on different sets of data.
- The MPMD (Multiple Programs Multiple Data) model uses different programs for different processes, but the processes collaborate to solve the same problem. Typical usage of the MPMD model can be found in the master/worker style of execution or in the coupled analysis.

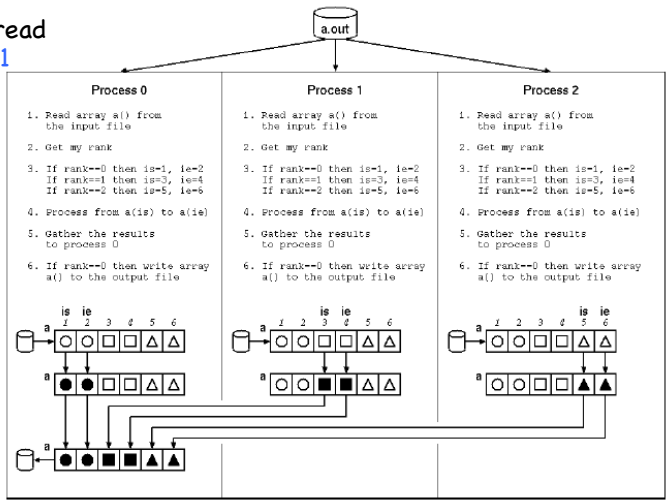
Single Program Multiple Data

- This figure shows a sequential program that reads data from a file, does some computation on the data, and writes the data to a file.
- White circles, squares, and triangles indicate the initial values of the elements, and black objects indicate the values after they are processed.
- Remember that in the SPMD model, all the processes execute the same program. To distinguish between processes, each process has a unique integer called rank. You can let processes behave differently by using the value of rank.



Single Program Multiple Data

- All the processes read the array in Step-1 and get their own rank in Step-2.
- In Steps 3 and 4, each process determines which part of the array it is in charge of, and processes that part.
- After all the processes have finished, none of the processes have all of the data, which is an undesirable side effect of parallelization. It is the role of message-passing to consolidate the processes separated by the parallelization. Step 5 gathers all the data to a process and that process writes the data to the output file.



Parallel programming: general notions

- We will now introduce some basic notions of parallel programming via a simple example: imagine you need to compute the sum

  $S = a_1 + a_2 + \dots + a_{16}$,

  a total of 15 additions, with a computer with p processors.
- We will call T_p (with p≥1) the time spent in computing S with a p-CPU computer via a parallel algorithm
- We will take the machine time to elaborate a single addition as standard unit; thus: $T_1=15$
- Consider now the case p=2: the best way to divide the task among the 2 CPU is to ask to CPU-1 to compute

  $S_1 = a_1 + a_2 + \dots + a_8$, and simultaneously to CPU-2 to compute

  $S_2 = a_9 + a_{10} + \dots + a_{16}$.

  Time spent to compute S_1 and S_2 is 7. At this point one of the two CPU can compute $S=S_1+S_2$ while the other CPU remains idle. Total time is therefore $T_2=8$. We discuss now the cases p=3, 4, 8.
- (p=3) Step 1: $S_1= a_1+\dots+a_5$ / $S_2= a_6+\dots+a_{10}$ / $S_3= a_{11}+\dots+a_{15}$ (time=4)
  Step 2: $S_a= S_1+S_2$ / $S_b= S_3+a_{16}$ (time=1 one CPU is idle)
  Step 3: $S = S_a+S_b$ (time=1 two CPUs are idle) Total time: $T_3=6$

Simple example … continue

- (p=4)

Step 1: $S_1= a_1+\dots+a_4$ / $S_2= a_5+\dots+a_8$ / $S_3= a_9+\dots+a_{12}$ / $S_4= a_{13}+\dots+a_{16}$ (time=3)
Step 2: $S_a= S_1+S_2$ / $S_b= S_3+S_4$ (time=1, two CPUs are idle)
Step 3: $S = S_a+S_b$ (time=1, three CPUs are idle)
Total time: $T_4=5$

- (p=8)

Step 1: $S_1= a_1+a_2$ / $S_2= a_3+a_4$ / … / $S_8= a_{15}+a_{16}$ (time=1)
Step 2: $S_a=S_1+S_2$ / $S_b=S_3+S_4$ / $S_c=S_5+S_6$ / $S_d=S_7+S_8$ (time=1, 4 CPUs are idle)
Step 3: $S_\mu= S_a+S_b$ / $S_\nu = S_c+S_d$ (time=1, 6 CPUs are idle)
Step 4: $S = S_\mu + S_\nu$ (time=1, 7 CPUs are idle)
Total time: $T_8=4$

- As you should note, by increasing the number of CPUs the calculation of S is performed more and more quickly, but the efficiency decreases because more and more CPUs remain idle once the majority of the calculations have been performed

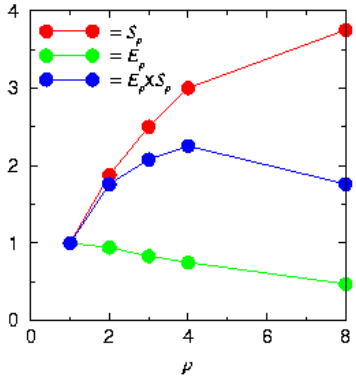
Parallel programming: some definitions

- Cost: $C_p = p \times T_p$
- Speed-up: $S_p = T_1/T_p$ ($S_p \le p$; if $S_p \div p$ linear speed-up)
- Efficency: $E_p = S_p/p = T_1/C_p$ ($E_p \le 1$)

- In the optimization of a parallel calculation, a good compromise is obtained by trying to maximize the product $[E_p \times S_p]$

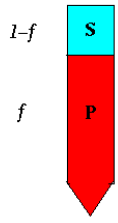
- Summary table:

p	T_p	C_p	S_p	E_p	$E_p \times S_p$
1	15	15	1	1	1
2	8	16	1.88	0.94	1.76
3	6	18	2.5	0.83	2.08
4	5	20	3	0.75	2.25
8	4	32	3.75	0.47	1.76



Amdahl's law (1967)

- In late '60 there was considerable skepticism regarding the viability of massive parallelism; the skepticism was centered around *Amdahl's law*, an argument put forth by Gene Amdahl [*AFIPS Conference Proceedings* vol. 30 (Atlantic City, N.J., 1967) pp. 483-485] that even when the fraction of serial work in a given problem is small, say *s*, the maximum speed-up obtainable from even an infinite number of parallel processors is only $1/s$

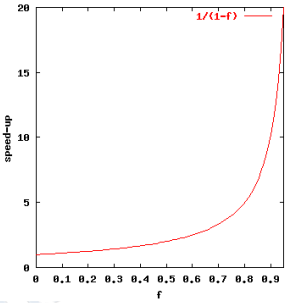


- Consider, in fact, a code which is composed by a fraction f that is completely parallelizable and by a fraction 1-f that is intrinsically serial and thus that cannot be parallelized (every code is in this situation). Imagine also to have a parallel computer with p CPUs. If T_1 is the time spent in executing the code (serially) on a serial computer, we will have:
- $f \times T_1$= time spent in executing the fraction of the code P serially
- $(1-f) \times T_1$= time spent in executing the fraction of the code S serially
- $(f \times T_1)/p$ = time spent in executing the fraction of code P on a parallel computer with p CPUs; thus

$$T_p = (1-f) \times T_1 + (f \times T_1)/p = T_1[f+(1-f)p]/p$$

Amdahl's law (1967)

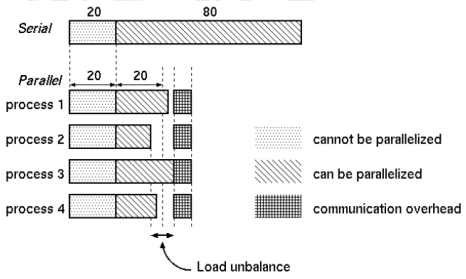
- It now possible to obtain the speed-up:
  $S_p=T_1/T_p=p/[f+(1-f)p]=1/[f/p+(1-f)]$
- By using an increasing number p of CPUs, one can reduce the ratio f/p to zero; It follows that $S_p \leq 1/(1-f)$. This is Amdahl's law. Is this the end of parallel computing?
- No! Usually the non-parallelizable fraction (1-f) tends to zero when computational complexity of the problem increase (Gustafson's law)



- Amdahl's Law approximately suggests: Suppose a car is travelling between two cities 120 Km apart and has already spent one hour travelling half the distance at 60 Km/h. No matter how fast you drive the last half, it is impossible to surpass 120 Km/h average (speed-up) before reaching the second city. Going infinitely fast you would only achieve 120 Km/h.
- Gustafson's Law approximately states: Suppose a car has already been travelling for some time at less than 120 Km/h. Given enough time and distance to travel, the car's average speed can always eventually reach and surpass 120 Km/h (speed-up), no matter how long or how slowly it has already travelled.

Real cases

- The situation just described is ideal, real cases are much more complicated
- Up to now we haven't considered that in real parallel computation there is a time spent in communicating (needed) partial results among CPUs
- Efficiency depends also on (communication overhead) how many data should travel across the network, on how many times data should travel on the network, on the network speed (bandwidth), on its latency



- Generally a good parallelization is obtained when:
  - The parallelizable fraction increases
  - Balance the workload of parallel processes
  - Communication is minimized as quantity of data and number of times
  - Design parallel algorithms from scratch (think parallel, write parallel!)

MPI: Message Passing Interface

- The Message Passing Interface (MPI) is a standard developed by the Message Passing Interface Forum (MPIF).
- MPIF, with the participation of more than 40 organizations (Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC, … but also USA Universities and National Laboratories), started working on the standard in 1992. The first draft (Version 1.0), which was published in 1994, was strongly influenced by the work at the IBM T. J. Watson Research Center.
- It specifies a portable interface for writing message-passing programs, and aims at practicality, efficiency, and flexibility at the same time.
- MPIF has further enhanced the first version to develop a second version (MPI-2) in 1997, a third version (MPI-3) in 2014/15 and a forthcoming version (MPI-4). For details about MPI and MPIF, visit http://www.mpi-forum.org
- It is a library for Fortran, C, C++, Python and other languages which implements the message passing model on parallel computers thus giving the possibility to processes running on different CPUs to synchronize information even if a common address space for memory (generally distributed) is lacking

MPI: Message Passing Interface

- Communication is obtained via explicit messages
- The MPI standard is small and large: it consists of functions for
  - Process identification
  - Point-to-point communication
  - Collective operations
  - Process groups
  - Synchronizations
  - Dynamical creation of processes (MPI-2; important for fault-tolerance)
  - I/O portable functions (MPI-2)
- But you can obtain everything via the following 6 functions:
  - `MPI_Init()` -> MPI initialization
  - `MPI_Comm_size()` -> to know how many processes
  - `MPI_Comm_rank()` -> to know who am I
  - `MPI_Send()` -> to send a message
  - `MPI_Recv()` -> to receive a message
  - `MPI_Finalize()` -> MPI finalization

A first example

- Note that the program is executed in the SPMD (Single Program Multiple Data) model. All the nodes that run the program, therefore, need to see the same executable file with the same path name, which is either shared among nodes by NFS or other network file systems, or is copied to each node's local disk.
- This program does not include any communication, thus it is banally parallelized in the SPMD model

Python

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

size = comm.Get_size()
rank = comm.Get_rank()
print('Sono il nodo ', rank, ' dei ', size, ' che hai
utilizzato!')
```

C++

```
#include "mpi.h"
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{

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

 cout<<" Sono il nodo "<<rank<<" dei "<<size<<" che hai
utilizzato!"<<endl;

 MPI_Finalize();
 return 0;
}
```


- Via the inclusion of `mpi.h` one defines MPI-related objects such as `MPI_COMM_WORLD`.
- `MPI_Init()` is called for initializing an MPI environment. `MPI_Init()` must be called once and only once before calling any other MPI calls. `MPI_Finalize()` terminates MPI processing and no other MPI call can be made afterwards. Attention: Including code before `MPI_Init()` or after `MPI_Finalize()` can produce non-portable programs
- The method `MPI_Comm_size()` returns the number of processes belonging to the `MPI_COMM_WORLD` object. A *communicator* is an identifier associated with a group of processes. `MPI_COMM_WORLD` defined in `mpi.h` represents the group consisting of all the processes participating in the parallel job.
- Each process in a communicator has its unique *rank*, which is in the range [0,size-1] where size is the number of processes in that communicator. A process can have different ranks in each communicator that the process belongs to. `MPI_Comm_rank()` returns the rank of the process within the communicator given as the first argument.

- Although each process executes the same program in the SPMD model, you can make the behaviour of each process different by using the value of the rank.
- This is where the parallel speed-up comes from; each process can operate on a different part of the data or the code concurrently.
- Let's see how it works: copy&paste into `example_1.cpp` , then …
  - Load MPI module (1 time): `module load mpi/mpich-3.2-x86_64`
  - Compile: `mpicxx example_1.cpp`
  - Run on 4 processors: `mpiexec -np 4 a.out`

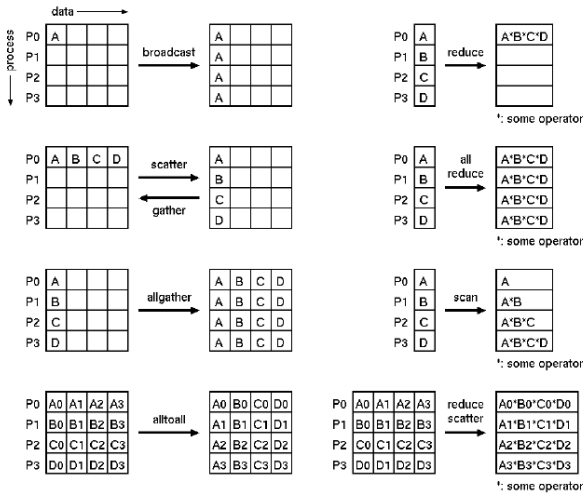
  (on Python: `mpiexec -np 4 python example_1.py`)

Typical result:

```
SHELL$ mpiexec -np 4 a.out
Sono il nodo 0 dei 4 che hai utilizzato!
Sono il nodo 1 dei 4 che hai utilizzato!
Sono il nodo 2 dei 4 che hai utilizzato!
Sono il nodo 3 dei 4 che hai utilizzato!
```


Collective communications

- Collective communication allows you to exchange data among a group of processes.
- The communicator argument in the collective communication subroutine calls specifies which processes are involved in the communication. In other words, all the processes belonging to that communicator must call the same collective communication subroutine with matching arguments.



- All of the MPI collective communication subroutines are blocking i.e. they ends only when all the involved processes have completed their task. There are several types of collective communications, as illustrated in the picture.

MPI_BCAST

- The subroutine `MPI_Bcast()` broadcasts the message from a specific process called *root* to all the other processes in the communicator given as an argument.

Python

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

my_values = np.zeros(3)

for i in range(3):
   if rank == 0:
      my_values[i] = i+1

print('Prima: ', my_values[0], ' ', my_values[1], ' ',
my_values[2], ' per il processo: ', rank)

comm.Bcast(my_values, root=0)

print('Dopo: ', my_values[0], ' ', my_values[1], ' ',
my_values[2], ' per il processo: ', rank)
```

C++

```
#include "mpi.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int my_values[3];
for(int i=0;i<3;i++)
 if(rank==0) my_values[i]=i+1;
 else my_values[i]=0;
cout<< "Prima: "<< my_values[0]<< " "<< my_values[1]<<
" "<< my_values[2]<< " per il processo "<< rank<< endl;
 MPI_Bcast(my_values,3,MPI_INTEGER,0, MPI_COMM_WORLD);
cout<< "Dopo: "<< my_values[0]<< " "<< my_values[1]<< "
"<< my_values[2]<< " per il processo "<< rank<< endl;

 MPI_Finalize();
 return 0;
}
```


MPI data types

- Descriptions of MPI data types and communication buffers follow.
- MPI subroutines recognize data types as specified in the MPI standard. The following is a description of MPI data types in the Fortran language bindings.
- Make sure that the amount of data transmitted matches between the sending process and the receiving processes.

MPI Data Types	Description (Fortran Bindings)
MPI_INTEGER1	1-byte integer
MPI_INTEGER2	2-byte integer
MPI_INTEGER4, MPI_INTEGER	4-byte integer
MPI_REAL4, MPI_REAL	4-byte floating point
MPI_REAL8, MPI_DOUBLE_PRECISION	8-byte floating point
MPI_REAL16	16-byte floating point
MPI_COMPLEX8, MPI_COMPLEX	4-byte float real, 4-byte float imaginary
MPI_COMPLEX16, MPI_DOUBLE_COMPLEX	8-byte float real, 8-byte float imaginary

MPI Data Types	Description (Fortran Bindings)
MPI_COMPLEX32	16-byte float real, 16-byte float imaginary
MPI_LOGICAL1	1-byte logical
MPI_LOGICAL2	2-byte logical
MPI_LOGICAL4, MPI_LOGICAL	4-byte logical
MPI_CHARACTER	1-byte character
MPI_BYTE, MPI_PACKED	N/A

MPI_GATHER

- The subroutine `MPI_Gather()` transmits data from all the processes in the communicator to a single receiving process. The length of the message sent from each process must be the same.
- The memory locations of the send buffer (isend) and the receive buffer (irecv) must not overlap.

Python

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if size > 3:
   exit('Hai scelto troppi processi')

irecv = np.zeros(3)
isend = np.zeros(1)
isend[0] = rank+1

comm.Gather(isend, irecv, root=0)

if rank == 0:
   print('irecv: ', irecv[0], ' ', irecv[1], ' ',
irecv[2])
```

C++

```
#include "mpi.h"
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(size>3){cout<<"Hai scelto troppi processi"<<endl;
          return 1;}
int irecv[3];
for(int i=0;i<3;i++) irecv[i]=0;
int isend = rank + 1;

MPI_Gather(&isend,1,MPI_INTEGER,irecv,1,MPI_INTEGER,0,
MPI_COMM_WORLD);

if(rank==0) cout<< "irecv: " <<irecv[0] <<" "
<<irecv[1] <<" " <<irecv[2] <<endl;

MPI_Finalize();
return 0;
}
```

MPI_REDUCE

- The subroutine `MPI_Reduce()` does reduction operations such as summation of data distributed over processes, and brings the result to the root process.

Python

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

Sum      = np.zeros(1)
Prod     = np.zeros(1)

isend = np.zeros(2)
for i in range(2):
   isend[i] = rank+i+1

comm.Reduce(isend[0], Sum,  op=MPI.SUM,  root=0)
comm.Reduce(isend[1], Prod, op=MPI.PROD, root=0)

if rank == 0:
   print('Sum: ', Sum, 'Product: ', Prod)
```

C++

```
#include "mpi.h"
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int isend[2],irecv[2];
for(int i=0;i<2;i++) isend[i]=rank+i+1;

MPI_Reduce(&isend[0],&irecv[0],1,MPI_INTEGER,
           MPI_SUM,0,MPI_COMM_WORLD);
MPI_Reduce(&isend[1],&irecv[1],1,MPI_INTEGER,
           MPI_PROD,0,MPI_COMM_WORLD);

if(rank==0)
       cout<<"irecv: "<<irecv[0]<<" "<<irecv[1]<<endl;

MPI_Finalize();
return 0;
}
```

MPI_REDUCE

- As is the case with `MPI_Gather()`, the send buffer and the receive buffer cannot overlap in memory.
- The fifth argument of `MPI_Reduce()`, `MPI_SUM`, specifies which reduction operation to use. The MPI provides several common operators by default, where `MPI_SUM` is one of them, which are defined in `mpif.h`

Operation	Data type
MPI_SUM (sum), MPI_PROD (product)	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX
MPI_MAX (maximum), MPI_MIN (minimum)	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION
MPI_MAXLOC (max value and location), MPI_MINLOC (min value and location)	MPI_2INTEGER, MPI_2REAL, MPI_2DOUBLE_PRECISION
MPI_LAND (logical AND), MPI_LOR (logical OR), MPI_LXOR (logical XOR)	MPI_LOGICAL
MPI_BAND (bitwise AND), MPI_BOR (bitwise OR), MPI_BXOR (bitwise XOR)	MPI_INTEGER, MPI_BYTE

Managing groups: MPI_COMM_SPLIT

- It is possible that a problem consists of several sub-problems and each problem can be solved concurrently. This type of application can be found in the category of coupled analysis. MPI allows you to create a new group as a subset of an existing group. Specifically, use `MPI_COMM_SPLIT` for this purpose.

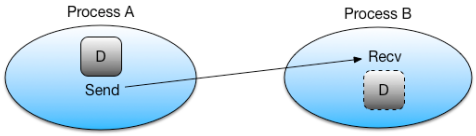
Python

```python
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
if size != 4:
   exit('Servono 4 processi, non ',size,' !!!')
if rank == 0:
   icolor = 1
   ikey   = 2
if rank == 1:
   icolor = 1
   ikey   = 1
if rank == 2:
   icolor = 2
   ikey   = 1
if rank == 3:
   icolor = 2
   ikey   = 2
newcomm = comm.Split(icolor,ikey)
newsize = newcomm.Get_size()
newrank = newcomm.Get_rank()
print('Ero: ', rank, ' di ', size, ' ... e adesso sono:
', newrank, ' di ', newsize)
```

C++

```cpp
#include "mpi.h"
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(size!=4){cout<<"Servono 4 processi, non
"<<size<<"!!"<<endl; return 1;}
int icolor, ikey;
if(rank==0){icolor=1;ikey=2;}
if(rank==1){icolor=1;ikey=1;}
if(rank==2){icolor=2;ikey=1;}
if(rank==3){icolor=2;ikey=2;}
MPI_Comm nuovocom;
MPI_Comm_split(MPI_COMM_WORLD,icolor,ikey,&nuovocom);
int newsize,newrank;
MPI_Comm_size(nuovocom, &newsize);
MPI_Comm_rank(nuovocom, &newrank);
cout<<"Ero: "<<rank<<" di "<<size<<" ... e adesso sono:
"<<newrank<<" di "<<newsize<<endl;
MPI_Finalize();
return 0;
}
```

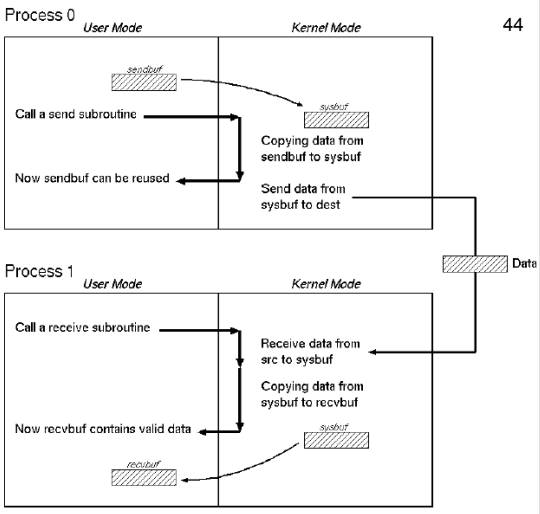
Point to point communications



43

- When you use point-to-point communication subroutines, you should know about the basic notions of blocking and non-blocking communication, as well as the issue of deadlocks.
- When a message is sent from process 0 to process 1, there are several steps involved in the communication. At the sending process, the following events occur one after another.
  1. The data is copied to the *user buffer* by the user.
  2. The user calls one of the MPI send subroutines.
  3. The system copies the data from the user buffer to the system buffer.
  4. The system sends the data from the system buffer to the destination process.
- The term *user buffer* means scalar variables or arrays used in the program. The following occurs during the receiving process:
  1. The user calls one of the MPI receive subroutines.
  2. The system receives the data from the source process and copies it to the system buffer.
  3. The system copies the data from the system buffer to the user buffer.
  4. The user uses the data in the user buffer.

- When you send data, you cannot or should not reuse your data from user buffer to the system buffer. Also when you receive data, the data is not ready until the system completes copying data from a system buffer to a user buffer.
- In MPI, there are two modes of communication: blocking and non-blocking. When you use blocking communication subroutines such as `MPI_SEND` and `MPI_RECV`, the program will not return from the subroutine call until the copy to/from the system buffer has finished. On the other hand, when you use non-blocking communication subroutines such as `MPI_ISEND` and `MPI_IRECV`, the program immediately returns from the subroutine call. That is, a call to a non-blocking subroutine only indicates that the copy to/from the system buffer is initiated and it is not assured hat the copy has completed.



44

- Therefore, you have to make sure of the completion of the copy by MPI_WAIT. If you use your buffer before the copy completes, incorrect data may be copied to the system buffer (in case of non-blocking send), or your buffer does not contain what you want (in case of non-blocking receive).
- Why do you use non-blocking communication despite its complexity? Because non-blocking communication is generally faster than its corresponding blocking communication. Some hardware may have separate co-processors that are dedicated to communication. On such hardware, you may be able to hide the latency by computation. In other words, you can do other computations while the system is copying data back and forth between user and system buffers.
- At the simplest level one can have unidirectional communications or bidirectional communications
- When two processes need to exchange data with each other, you have to be careful about deadlocks. When a deadlock occurs, processes involved in the deadlock will not proceed any further. Deadlocks can take place either due to the incorrect order of send and receive, or due to the limited size of the system buffer.

MPI_SEND / MPI_RECV

- When you send a unidirectional message, there are 4 combinations of MPI subroutines to choose from depending on whether you use a blocking or non-blocking subroutine for sending or receiving data (the example show usage of blocking subroutines).

Python

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

imesg = np.zeros(1)
imesg[0] = rank

if rank == 1:
   comm.Send(imesg, dest=0)
elif rank == 0:
   comm.Recv(imesg, source=1)

print('Io sono: ', rank,' Il mio messaggio: ', imesg)
```

C++

```cpp
#include "mpi.h"
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status stat;

int itag=1;
int imesg = rank;
if(rank==1)
  MPI_Send(&imesg,1,MPI_INTEGER,0,itag,MPI_COMM_WORLD);
else if(rank==0)
  MPI_Recv(&imesg,1,MPI_INTEGER,1,itag,MPI_COMM_WORLD,
           &stat);

cout<<"messaggio = "<<imesg<<endl;

MPI_Finalize();
return 0;
}
```

Bidirectional communications

- ***Case 1*** Both processes call send, and then receive (dangerous if blocking send)
- ***Case 2*** Both processes call receive, and then send (really dangerous!)
- ***Case 3*** One process calls send and receive subroutines in this order, and the other calls in the opposite order (safe)

Python

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

n = 100 # try to increase n
imesg  = np.zeros(n)
imesg2 = np.zeros(n)

for i in range(n):
   imesg[i]  = rank
   imesg2[i] = rank+1

if rank == 1:
   comm.Send(imesg, dest=0)
   comm.Recv(imesg2, source=0)
   print('Io sono: ', rank,' Il mio messaggio: ',
imesg2[0])
elif rank == 0:
   comm.Send(imesg2, dest=1)
   comm.Recv(imesg, source=1)
   print('Io sono: ', rank,' Il mio messaggio: ',
imesg[0])
```

C++

```cpp
#include "mpi.h"
#include <iostream>
using namespace std;
const int n = 100; // try to increase n
int main(int argc, char* argv[]){
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status stat1, stat2;
int* imesg = new int[n]; int* imesg2 = new int[n];
int itag=1; int itag2=2;
for(int i=0;i<n;i++){imesg[i]=rank; imesg2[i]=rank+1;}
if(rank==1){MPI_Send(&imesg[0],n,
            MPI_INTEGER,0,itag,MPI_COMM_WORLD);
            MPI_Recv(&imesg2[0],n,
            MPI_INTEGER,0,itag2, MPI_COMM_WORLD,&stat2);
            cout<<"messaggio = "<<imesg2[0]<<endl;}
else if(rank==0){MPI_Send(&imesg2[0],n,
                 MPI_INTEGER,1,itag2, MPI_COMM_WORLD);
                 MPI_Recv(&imesg[0],n,MPI_INTEGER,1,
                 itag, MPI_COMM_WORLD,&stat1);
                 cout<<"messaggio = "<<imesg[0]<<endl;}
MPI_Finalize();
return 0;}
```


- The following code is free from deadlock because the program immediately returns from MPI_ISEND and starts receiving data from the other process.

Python

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

n = 100 # try to increase n
imesg  = np.zeros(n)
imesg2 = np.zeros(n)
for i in range(n):
   imesg[i]  = rank
   imesg2[i] = rank+1

if rank == 1:
   req = comm.Isend(imesg, dest=0)
   comm.Recv(imesg2, source=0)
   req.Wait()
   print('Io sono: ', rank,' Il mio messaggio: ',
imesg2[0])
elif rank == 0:
   comm.Send(imesg2, dest=1)
   comm.Recv(imesg, source=1)
   print('Io sono: ', rank,' Il mio messaggio: ',
imesg[0])
```

C++

```cpp
#include "mpi.h"
#include <iostream>
using namespace std;
const int n = 100; // try to increase n
int main(int argc, char* argv[]){
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status stat1, stat2;
MPI_Request req;

int* imesg = new int[n]; int* imesg2 = new int[n];
int itag=1; int itag2=2;
for(int i=0;i<n;i++){imesg[i]=rank; imesg2[i]=rank+1;}
if(rank==1){MPI_Isend(&imesg[0],n,
            MPI_INTEGER,0,itag, MPI_COMM_WORLD,&req);
            MPI_Recv(&imesg2[0],n,MPI_INTEGER,0,itag2,
            MPI_COMM_WORLD,&stat2);
            cout<<"messaggio = "<<imesg2[0]<<endl;}
else if(rank==0){MPI_Send(&imesg2[0],n,
                 MPI_INTEGER,1,itag2,MPI_COMM_WORLD);
                 MPI_Recv(&imesg[0],n,MPI_INTEGER,
                 1,itag,MPI_COMM_WORLD, &stat1);
                 cout<<"messaggio = "<<imesg[0]<<endl;}
MPI_Finalize();
return 0;}
```

MPI_WTIME

- Measurement of time and thus of performances can be obtained as in the following example

Python

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

tstart = MPI.Wtime()

n = 10000
imesg  = np.zeros(n)
Sum = 0

for i in range(n):
   imesg[i]  = rank
   if rank == 1:
      comm.Send(imesg, dest=0)
   elif rank == 0:
      comm.Recv(imesg, source=1)
   Sum += imesg[i]

tend = MPI.Wtime()
print('Io sono:', rank, 'Ho impiegato:', tend-tstart,
'secondi')
```

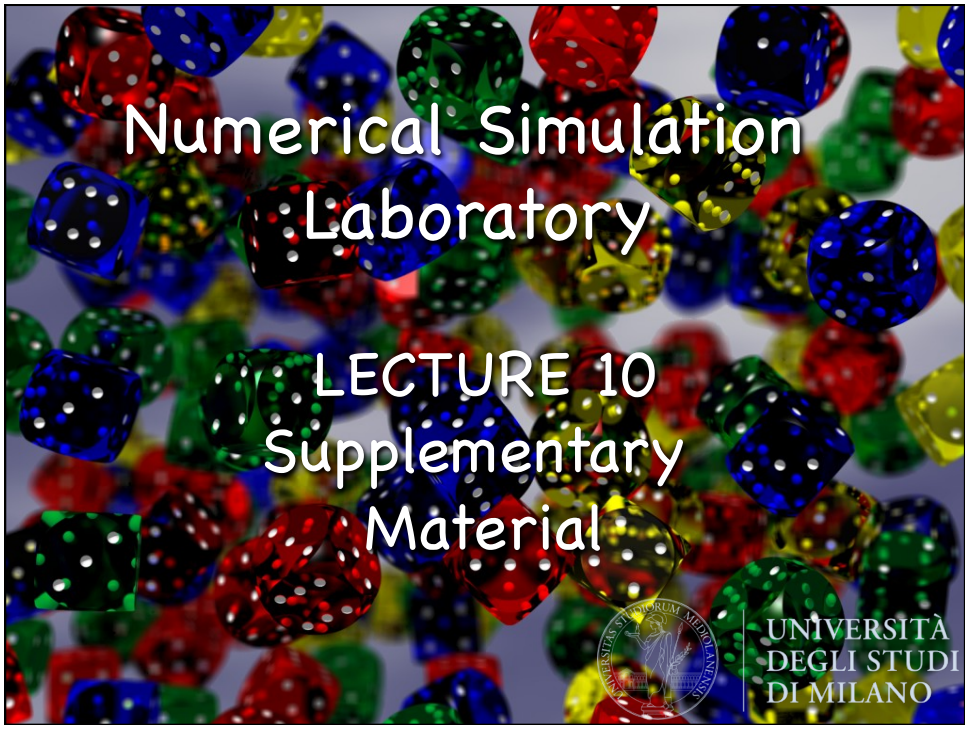
C++

```cpp
#include "mpi.h"
#include <iostream>
using namespace std;
int main(int argc, char* argv[]){
int size, rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status stat;

double tstart = MPI_Wtime();
int n = 100;
int* imesg = new int[n];
int sum=0;
for(int i=0;i<n;i++){
  imesg[i]=rank;
  if(rank==1) MPI_Send(&imesg[0],n,
              MPI::INTEGER,0,i,MPI_COMM_WORLD);
  else if(rank==0) MPI_Recv(&imesg[0],n,MPI_INTEGER,
                   1,i,MPI_COMM_WORLD,&stat);
  sum += imesg[i];}
double tend = MPI_Wtime();
double dt = tend - tstart;
cout<<"io sono "<<rank<<"; somma = "<<sum<<"; tempo =
"<<dt<<endl;
MPI_Finalize();
return 0;}
```


Lecture 10: Suggested books

- Gropp, Lusk, Skjellum, *Using MPI, Portable Parallel Programming with the Message-Passing Interface* – The MIT Press (2014)
- https://computing.llnl.gov/tutorials/mpi/



Parallelizing do-loops

- In almost all of the scientific and technical programs, the hot spots are likely to be found in DO loops. Thus parallelizing DO loops is one of the most important challenges when you parallelize your program.
- The basic technique of parallelizing DO loops is to distribute iterations among processes and to let each process do its portion in parallel.
- Usually, the computations within a DO loop involves arrays whose indices are associated with the loop variable. Therefore distributing iterations can often be regarded as dividing arrays and assigning chunks (and computations associated with them) to processes.
- In block distribution, the iterations are divided into *p* parts, where *p* is the number of processes to be executed in parallel. The iterations in each part is consecutive in terms of the loop variable.
- In cyclic distribution, the iterations are assigned to processes in a round-robin fashion. Note that, in general, the cyclic distribution incurs more cache misses than the block distribution because of non-unit stride access to matrices within a loop.
- When you use some redistribution, it is often sufficient for each process to have only a part of the arrays and to do the computation associated with that part. If a process needs data that another process holds, there has to be some transmission of the data between the processes.

Block distribution

- Suppose when you divide *n* by *p*, the quotient is *q* and the remainder is *r*.
- For example, in the case of *n* = 14 and *p* = 4, *q* is 3 and *r* is 2. One way to distribute iterations in such cases is as follows. Processes 0..r-1 are assigned *q*+1 iterations each. The other (p-r) processes are assigned *q* iterations.This distribution corresponds to expressing *n* as n=r(q+1)+(p-r)q

```
#include "mpi.h"
#include <iostream>
using namespace std;
const int n=1000000;
void blocchi(int,int,int,int,int&,int&);
int main(int argc, char* argv[])
{
        int size, rank;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        int* a = new int[n];
        int sum=0, ssum;
        int istart, iend;
        blocchi(0,n-1,size,rank,istart,iend);
        for(int i=istart;i<iend;i++){
                a[i]=i;
                sum += a[i];}
        MPI_Reduce(&sum,&ssum,1,MPI_INTEGER,MPI_SUM,
0,MPI_COMM_WORLD);
        if(rank==0) cout<<"somma = "<<ssum<<endl;
        MPI_Finalize();
        return 0;
}
```

```
void blocchi(int nstart, int nend, int nprocs, int
mype, int& istart, int& iend)
{
        int i1= nend - nstart +1;
        int i2 = i1/nprocs;
        int i3 = i1 % nprocs;
        int min = mype;
        if(i3<mype)
                min=i3;
        istart = mype*i2 + nstart + min;
        iend = istart + i2 -1;
        if(i3>mype)
                iend++;
}
```

In the example above:

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	3	3	3

Cyclic distribution

- To avoid load unbalance when workload is not uniform among iterations one can try cyclic distribution
- The following is an example of how to implement it
- In the previous example the distribution corresponds to:

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	1	2	3	0	1	2	3	0	1	2	3	0	1

```
#include "mpi.h"
#include <iostream>
using namespace std;

const int n = 10000;
int main(int argc, char* argv[])
{
 int size, rank;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        int* a = new int[n];
        int sum=0,ssum;
        for(int i=rank;i<n;i+=size)
        {
                a[i]=i+1;
                sum += i+1;
        }
        MPI_Reduce(&sum,&ssum,1,MPI_INTEGER,MPI_SUM,
0,MPI_COMM_WORLD);

        if(rank==0) cout<<"somma = "<<ssum<<endl;

        MPI_Finalize();
        return 0;
}
```