Foundations of HPC

book suggested:
Introduction to High-Performance Scientific Computing, by Victor Eijkhout

email prof:
cozzini@sissa.it

ssh

Ulysses

```
$ ssh gcarbone@frontend2.hpc.sissa.it
```

```
$ ssh igirotto@frontend2.hpc.sissa.it
qsub -I -l walltime=10:00 -q reserved3 //entro nel nodo per 10 minuti
```

Exact lab/cosilt

```
$ ssh exactlab // ssh cosilt // ssh gcarbone@hpc.c3e.exact-lab.it
$ qsub -q jrc -I -l nodes=1:ppn=20,walltime=00:30:00
```

Per settare gli alias per gli ssh:

```
~/.ssh/config
```

02/10/17

Computer cluster

A *computer cluster* consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system. Computer clusters have each node set to perform the same task, controlled and scheduled by software.
The *login* node is the node you get when you first log into HPC. The login nodes are meant for simple tasks such as submitting jobs, checking on job status, editing *(emacs, vi)* and performing simple tasks.
The *interactive nodes* are used for when you need to compile, test your code, and run a few 1-2 interactive sessions.

Connecting via SSH (server) to Ulysses cluster
D1_hands_on

Copy my files into the cluster

```
$ scp \path\to\your\file.txt user@cluster_address:\path\in\cluster
```

I have to type this from local

Now I am copying pi.c and mpi_pi.c into the server

```
$ scp -r /home/ginevracoal/Dropbox/DSSC/foundations_hpc/P1.2_seed/D1-hands-on/code gcarbone@frontend2.hpc.sissa.it:/home/gcarbone/DSSC/foundations_hpc
```

Or I can use SSHFS

```
sshfs Ulysses:/home/gcarbone/ /home/ginevracoal/Dropbox/DSSC/mount_point
```

Now I can switch to the cluster:

```
$ ssh gcarbone@frontend2.hpc.sissa.it
```

per non inserire ogni volta la password uso l'SSH
quando accedo mi basta inserire

```
ssh-add
```

Environment *modules* provide a way to selectively activate and deactivate modifications to the user environment which allow particular packages and versions to be found.

```
$ module avail //list the available modules
$ module load openmpi/1.8.3/gnu/4.9.2 //load the module
$ cd ~/
```

Measure time with /usr/bin/time
The command returns three different timings:

user + system = elapsed

where

- user is the CPU time of the instructions
- system is the time spent in kernel mode
- elapsed is the real, total wall time

***** Weak/strong scalability - D1**
report example
pi serial
pi paralell
plots

Scalability: I want to know the speedup depending on the number of procs and trials
Strong scalability: I only want to change the number of procs, not of trials (N)
Weak scalability: I want to increase the number of trials depending on the number of procs

```
nproc
```

number of procs

```
$ gcc pi.c -o serial_pi
```

where:
gcc is the c compiler
pi.c is the file to execute
-o <file> creates an executable <file> containg the output

```
$ ./serial_pi 10000000
```

runs serial_pi with input 10000000 and stores the result into a .dat file

qsub is the command used for job submission to the cluster

```
$ qsub -l nodes=1:ppn=20,walltime=02:00:00 -I //pay attention to the spac
es!
```

where:
-l nodes=1 :ppn=20 defines the list of resources required by the job and established a limit to the amount of resources that can be consumed
walltime=02:00:00 indicates the available time for the execution on the cluster
-I indicates the interactive job execution

```
$ module avail
$ module load gcc/4.8.2
$ gcc pi.c -o pi.x
$ /usr/bin/time ./pi.x 1000000 2>>time.txt //no procs specified, so it ru
ns on all the procs
```

2>>time.txt stores the time of execution inside a text file and considers the standard error

&>> considera sia standard error che standard output
1>> è il default standard output

```
$ mpicc mpi_pi.c -o mpi_pi.x
```

Ho scritto lo script *scalability.sh* per testare la scalability e poi ho esportato i risultati nei file di testo, dai quali voglio estrarre i risultati

```
cat strong_scalability.txt | grep elap
```

1.17user 0.05system 0:01.54elapsed 78%CPU (0avgtext+0avgdata 103072maxresident)k
2.31user 0.09system 0:01.55elapsed 154%CPU (0avgtext+0avgdata 103296maxresident)k
4.58user 0.18system 0:01.55elapsed 306%CPU (0avgtext+0avgdata 103424maxresident)k
9.22user 0.34system 0:01.62elapsed 588%CPU (0avgtext+0avgdata 103696maxresident)k
19.28user 0.73system 0:01.79elapsed 1117%CPU (0avgtext+0avgdata 104288maxresident)k
24.14user 0.92system 0:01.83elapsed 1365%CPU (0avgtext+0avgdata 141616maxresident)k

```
ginevracoal@ginevracoal:~/Dropbox/DSSC/foundations_hpc/my_exercises/HPC_g
raphs$ cat strong_scalability.txt | grep elap | awk '{print $3}'
```

0:01.54elapsed
0:01.55elapsed
0:01.55elapsed
0:01.62elapsed
0:01.79elapsed

0:01.83elapsed

per eliminare la parola *elapsed* considero *elapsed* come un separatore ed estraggo la prima colonna:

```
cat strong_scalability.txt | grep elap | awk '{print $3}' | awk -F 'elaps
ed' '{print $1}'
```

0:01.54
0:01.55
0:01.55
0:01.62
0:01.79
0:01.83

***** pi scalability - D1**

Redirect output to a file

```
command > output.txt
```

The standard output stream will be redirected to the file only, it will not be visible in the terminal. If the file already exists, it gets overwritten.

```
command >> output.txt
```

The standard output stream will be redirected to the file only, it will not be visible in the terminal. If the file already exists, the new data will get appended to the end of the file.

```
command 2> output.txt
```

The standard error stream will be redirected to the file only, it will not be visible in the terminal. If the file already exists, it gets overwritten.

```
command 2>> output.txt
```

The standard error stream will be redirected to the file only, it will not be visible in the terminal. If the file already exists, the new data will get appended to the end of the file.

```
command &> output.txt
```

Both the standard output and standard error stream will be redirected to the file only, nothing will be visible in the terminal. If the file already exists, it gets overwritten.

```
command &>> output.txt
```

Both the standard output and standard error stream will be redirected to the file only, nothing will be visible in the terminal. If the file already exists, the new data will get appended to the end of the file.

Loop optimization

argc è il numero di argomenti digitati
argv è un array di stringhe contenente ogni argomento

argv[0] è il nome del programma

```
argv[1]=argv + 1
```

```
atoi()
```

converte una stringa in un intero

```
*(argv + 1)
```

è la stringa contenuta in *argv[1]*

```
Np = atoi( *(argv + 1) )
```

inizializza Np con l'intero che ho dato in input

c'è un errore: dovrebbe essere Np * 3

calloc(numero di elementi, dimensione di ogni elemento) restituisce un puntatore a void, quindi faccio il cast a double

(double)* operatore di casting

```
x = (double*)calloc(Np * 3, sizeof(double))
```

x è il puntatore calloc ad un array di double

puntatore x - Np celle dell'array - puntatore y - etc

enum è una keyword descrittiva
typedef altra keyword

a register is a hint to the compiler that the variable will be heavily used and that you recommend it be kept in a processor register if possible.

***** loop optimization - D3**

Register keyword

The `register` keyword in C (rarely ever seen anymore) is only a hint to the compiler that it may be useful to keep a variable in a register for faster access.
The compiler is free to ignore the hint, and optimize as it sees best.
Since modern compilers are much better than humans at understanding usage and speed, the `register` keyword is usually ignored by modern compilers, and in some cases, may actually slow down execution speed.

06/10/17

Moore law

The number of transistors in a dense integrated circuit doubles approximately every two years.
The problem is that power consumption is directly proportional to clock speed^2, and this means that clock speed has hit a plateau. So the idea is to increase the number of cores in each CPU: this is the passage from the GHz era to the **multicore era**, which also refers to multithreading.

ISV = independent software vendent
Multi processor = collection of multicore cpus
Socket = place where you put the cpu

example: Ulysses has 2 sockets, 2 CPUs, 20 cores (10 per socket)

generate RSA key:

```
$ ssh-keygen -o -a 100
$ cat .ssh/id_rsa.pub
```

Access to exact-lab

alternative machine with 24 cores:

```
$ eval $(ssh-agent)
$ ssh-add
$ ssh gcarbone@hpc.c3e.exact-lab.it
```

find my rsa public key

```
$ cat ~/.ssh/id_rsa.pub
```

09/10/17

Optimization techniques

lecture one: introduction to optimization techniques (L.T.)
lecture two: more on optimization techniques (L.T.)
lecture three: code timing (L.T)

malloc
La funzione `malloc` è una di quelle appartenenti allo standard C per l'allocazione della memoria. Il suo prototipo è

```
void *malloc(size_t size);
```

Che alloca `size` byte di memoria. Se l'operazione ha successo, viene restituito un puntatore al blocco di memoria, mentre in caso contrario verrà restituito un puntatore null.

La memoria allocata tramite `malloc` è persistente: ciò significa che continuerà ad esistere fino alla fine del programma o fino a quando non sarà esplicitamente deallocata dal programmatore.

`malloc` restituisce un blocco di memoria allocato dal programmatore per essere utilizzato, ma non è inizializzato. Questa operazione è spesso effettuata a mano, se necessario, tramite la funzione **`memset`**. Un'alternativa è di usare la funzione **`calloc`**, che alloca la memoria e la inizializza.

13/10/17

GDB debugging

Debugging simple codes with the help of a debugger
Remember:

- compile with -g
- debugging infos are stored into the *.o file

go in the folder Basic_debugging

```
make  //compile all the examples in the folder
makefile  //execute the chosen file
```

ex01.f / ex01.f90 / ex01.c. - Task: run the program under the control of a debugger, set/delete break points, watches, inspect data.

Interactive debugging

setto i break point e analizzo le variabili

```
$ gdb ex01-f90
```

to begin debugging the program

Now we are in (gdb):

```
list              //to see all the lines in the file
break 11          //create a new break in line 11
info break        //gives you informations about the breaks you created
c                 //continue, go on till you find the break
n                 //next, entra nella funzione successiva o esegue l'istruzione successiva (=s)
s                 //step, esegue la riga successiva
```

quit oppure CTRL+D to exit from gbd

Post mortem debugging

A core file is an image of a process that has crashed It contains all process information pertinent to debugging: contents of hardware registers, process status, and process data. Gdb will allow you use this file to determine where your program crashed.

./ex02-f90 gives a segmentation fault

The program must have crashed and left a core file. It should tell you if it has left a core file with the message "core dumped".

ulimit -a mi fa vedere i limiti imposti dal sistema

ulimit -c unlimited rende la dimensione del core unlimited

gdb ex02-f90 core gdb stars looking at the core file produced by the program. Gdb will then load the program's debugging information and examine the core file to determine the cause of the crash. Gdb prints the last statement that was attempted which likely caused the crash.

bt backtrace serve a ricostruire la gerarchia delle funzioni nel punto corrente

Now we have to solve the bug, so we go back to the interactive mode

gdb ex02-f90 no core this time
run
bt
print i

invio esegue il comando precedente

```
break 12 if j=1000
p i                 print i
p j
s                step
```

Debugging distribute_v0

```
$ gcc -g distribute_v0.c -lm
```

-lm è una libreria matematica

-pg : Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

***** gprof**
profile distribute_v0 with gprof

```
x/16db (char*) x-16
```

interpret x like a pointer to byte and go back by 16 bytes

the memory needed to store the header is machine dependent

malloc doesn't guarantee you to have contiguous memory, while calloc does it.

I have a stream of data and an if condition inside a loop (which should always be avoided). How to optimize it:
If I sort the data the execution time drops because the pattern of branch is more predictable.
Try to perform the same operation avoiding the if condition.

perf is a system utility which gives informations on performance

the order in which I choose the instructions inside if-else depends on the statistics. The content of the else always requires less time, because it is a straight operation (no jump). So we always have to write the most frequent instruction inside the else condition.

**** nodeperf
*** latency MPI
*** HPL with MKL
*** HPCG

16/10/17

AM

slides 5

PM

cd /sys/devices/system/cpu
cat id, level, type, size,...

*** mountain exercise

```
make mountain
./mountain
./mountain > mountain.dat
gnuplot
load "ploumountain.gp"
```

20/10/17

Some tools for profiling

Perf
The *perf* command in Linux gives you access to various tools integrated into the Linux kernel. These tools can help you collect and analyze the performance data about your program or system.

```
perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
```

perf is not interacting with the core but with the kernel
perf does not introduce any ???, while valgrind does.

The `cache-misses` event represents the number of memory access that could not be served by any of the cache.

Valgrind
valdring -tool=cachegrind arguments

***** Matrix transpose exercise**
traccia esercizio
slides memory hierarchy del prof

Indexing a matrix as an array
slides interessanti
esercizio svolto 1
esercizio svolto 2

if you use fortran *cclock* does not work, you have to use

```
CALL CPU_TIME(tstar)
```

se la matrice ha una sola entrata, coincide con la sua trasposta

```
if( argc < 2 ){...
```

il primo argomento è la dimensione della matrice, atoi converte una stringa in un intero

```
MATRIXDIM=atoi(argv[1]);
```

```
void* malloc (size_t size);
```

Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.
A è un puntatore, sto allocando la memoria per la dimensione $MATRIXDIM^2$

```
A = ( double * ) malloc( MATRIXDIM * MATRIXDIM * sizeof( double ) );
```

Do I really need casting malloc?
malloc returns *void**, which is a generic pointer that can point to any type of data. The *(double*)* is an explicit type conversion, converting the pointer returned by malloc from a pointer to anything, to a pointer to char. This is unnecessary in C, since it is done implicitly, and it is actually recommended not to do this, since it can hide some errors.

il codice genera la matrice di componenti $a_{ij} = i * MATRIXDIM + j$

```
AT[ ( j * MATRIXDIM ) + i ] = A[ ( i * MATRIXDIM ) + j ];
```

sto definendo $A_{ji} = A_{ij}$

Identify the minima:
there's a lot of noise on small matrix, because running time is too small

Cache misses

L1 cache misses is correlated with cache minima

Our L1 cache cache is usually 32KB

A cache *miss*, generally, is when something is looked up in the cache and is not found – the cache did not contain the item being looked up. The cache *hit* is when you look something up in a cache and it *was* storing the item and is able to satisfy the query.

```
./faster_transpose.x $N 2>>faster_transpose.csv
```

stampa solo lo stderr

***** matrix transpose - D5**

Ridimensionare le immagini:

```
$ file lstopo.png
```

lstopo.png: PNG image data, 832 x 915, 8-bit/color RGB, non-interlaced

```
$ convert -size 1600x915 xc:white lstopo.png -gravity center -composite added_white_space.png
$ convert -resize 50% prova.png lstopo_final.png
```

Install HPL

download

```
wget http://www.netlib.org/benchmark/hpl/hpl-2.1.tar.gz
```

unpack

```
tar -xvzf hpl-2.2.tar.gz
```

```
cd hpl-2.2/
less README
less INSTALL
cp setup/Make.Linux_PII_CBLAS make.Ginevra
nano make.Ginevra
mpicc
```

install mpicc

```
sudo apt install lam4-dev
sudo apt install libmpich-dev
sudo apt install libopenmpi-dev
mpicc
```

install blass

```
sudo apt-get install libblas-dev liblapack-dev
locate blas
cd /usr/lib/
```

exercise HPL profiling

```
-pg
```

```
gmon.out
stock
gprof -s $exe gmon.out stack
```

this way gprof accumulates data in stack

Hacks di Alberto

rinominare

```
 mv make.Ginevra Make.Ginevra
```

copiare

```
cp [origine] [destinazione]
```

rimuovere

```
rm
```

```
–help //alternativa a man
```

schede terminale CTRL+MAIUSC+T
chiudere scheda CTRL+MAIUSC+W

```
cat /proc/cpuinfo
```

to know informations about your machine

```
top
```

per definire i mount point basta usare il file contente gli alias

```
$ nano ~/.bashrc
$ source ~/.bashrc
```

CTRL+R per trovare l'ultimo comando che inizia con quella stringa

ho aggiunto

```
watch_Ulysses='watch -n 300 sshfs Ulysses:/home/gcarbone/ /home/ginevracoal/Dropbox/DSSC/mount_point'
```

quindi se in un terminale eseguo *watch_Ulysses*, ogni 300 secondi viene eseguito il comando ' '

```
$ mount
```

per vedere i mount point attivi

```
$ sudo umount /home/ginevracoal/mount_igirotto
```

per eliminare un mount attivo

This will list all processes that have sshd in their names:

```
$ ps aux | grep ssh
```

```
ps aux | grep ssh
  764  ssh-add
  765  ps aux | grep ssh
  766  watch -n 300 sshfs Ulysses:/home/gcarbone/ /home/ginevracoal/Dropbox/DSSC/mount_point
  767  nano ~/.bashrc
  768  source ~/.bashrc
  769  watch_Ulysses
  770  alias
  771  watch -n 300 sshfs Ulysses:/home/gcarbone/ /home/ginevracoal/Dropbox/DSSC/mount_point
```

23/10/17

Multicore architecture

Lesson 7

Difference between latency and bandwidth

what is a 64 bit address space?

when you connect more memory controllers you increase the bandwidth, but the latency remains the same because the path is the same

Opteron 6xxx: NUMA machine, 4 units, 2 sockets, 12 cores per socket, 24 cores in total

communication can has different levels within the same architecture

in the multicore era you have to parallelize

Parallel computing uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others

Cache coherence
In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local cache of each processor that requested it. When one of the copies of data is changed, the other copies must reflect that change. Cache coherence is the discipline which ensures that the changes in the values of shared operands(data) are propagated throughout the system in a timely fashion.

every time you change processor you load and clean the correspondent cache line.

Hyper threading
A *thread* is a process that runs on a multi core architecture. A single thread cannot run on more cores. A thread shares a virtual memory space, while a process has its own virtual space, so the first one is more efficient if you want to share virtual memory.
Hyper threading (Multithreading) is only available on Intel CPUs and allows multiple threads to run on each core. By HT you can fill holes in the CPU.

For each processor core that is physically present, the operating system addresses two virtual (logical) cores and shares the workload between them when possible. The main function of hyper-threading is to increase the number of independent instructions in the pipeline; it takes advantage of superscalar architecture, in which multiple instructions operate on separate data in parallel. With HTT, one physical core appears as two processors to the operating system, allowing concurrent scheduling of two processes per core.

you can see HT enables within the flags with

```
$ cat /proc/cpuinfo
```

Hyperthreading means that some parts of a core are duplicated. A core with hyperthreading is sometimes presented as an assemblage of two “virtual cores” — meaning not that each core is virtual, but that the plural is virtual because these are not actually separate cores and they will sometimes have to wait while the other core is making use of a shared part.
Thus `top` shows you 4 CPUs, because you can have 4 threads executing at the same time.
`/proc/cpuinfo` has 4 entries, one for each CPU (in that sense). The `processor` numbers (which are the number of the `cpu`*`NUMBER`* entries in `/sys/devices/system/cpu`) correspond to these 4 threads.

```
physical id : 0
siblings    : 4
core id     : 0
cpu cores   : 2
```

means that `cpu0` is one of 4 threads inside physical component (processor) number 0, and that's in core 0 among 2 in this processor.

Running omp_101 with

```
$ gcc -fopenmp omp_101.c -o omp_101.x
$ ./omp_101.x
```

gives me the number of threads

```
Hello from thread 0 out of 4
Hello from thread 3 out of 4
Hello from thread 1 out of 4
Hello from thread 2 out of 4
```

You can run your codes on how many threads you want, because you can run more threads on a single core.

Numactl
For thread placement

```
$ numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3
cpubind: 0
```

```
nodebind: 0
membind: 0
```

```
$ numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 7879 MB
node 0 free: 4277 MB
node distances:
node   0
  0:  10
```

If I have 2 nodes it tells me

```
available: 2 nodes (0-1)
```

In which case

```
$ numactl --cpunodebind=0 --membind=0,1 ./a.out
```

This puts memory on nodes 0 an 1, but threads only on node 0.

OpenMP

```
$ Export OMP_PLACES=sockets
```

increases the bandwidth

```
$ Export OMP_PLACES=cores
$ Export OMP_PROC_BIND=close
```

Ideal if you have a small array

```
$ Export OMP_PLACES=cores
$ Export OMP_PROC_BIND=spread
```

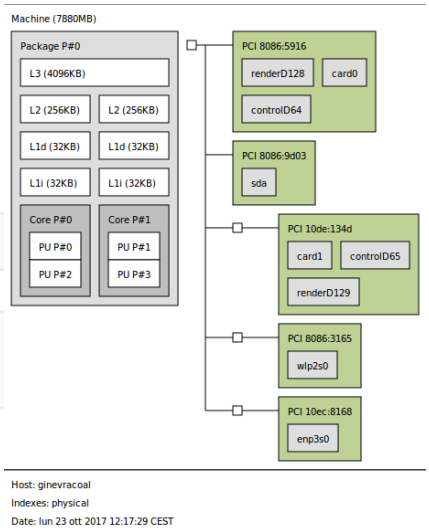
Produces the same result as the first command, but in the first case we are not specifing cores, so the operating system could choose unordered cores and also put more than one thread on a single core.

OpenMP Affinity allows you to also choose on which core you want to put each thread.

hwloc

```
$ lstopo
```

I only have 1 socket.



show object distances:

```
$ hwloc-distances
```

```
$ hwloc-bind --membind node:1 --cpubind node:
1.socket:0 ./a.out
```

the memory and the thread are located in the same node

Performance libraries are designed to use the CPU in the most efficient way.

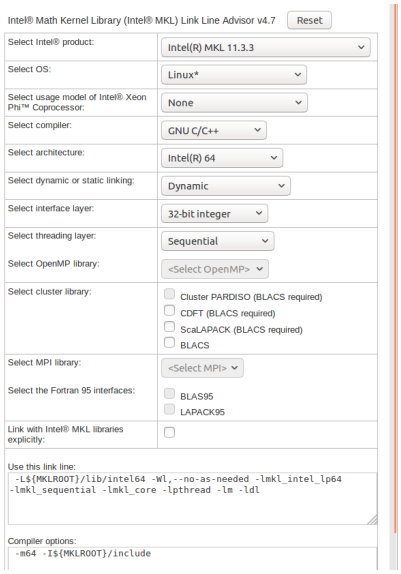
Download math kernel library

with these specs

If you have a multi thread library and run a parallel program, for example

```
$ mpirun -np 4 xhpl
$ top
```

you get 4 instances of xhpl, and each one of them runs on 24 cores, so it's not optimized



First I check the number of cores, then I

```
$ exp OMP_NUM_THREAD=1
$ mpirun -np 24 xhpl
```

***** stream benchmark**

Stream exercise

copy the folder stream into the server:

```
$ scp -r /home/ginevracoal/Dropbox/DSSC/foundations_hpc/P1.2_seed/D7-mate
rials/stream/ gcarbone@frontend2.hpc.sissa.it:/home/gcarbone/DSSC/foundat
ions_hpc
```

login into one node with execution time of 30 mins max:

```
$ qsub -l nodes=1:ppn=20 -l walltime=00:30:00 -I
```

Estimate the overall bandwidth of one node of ulyssess

Here I am into one socket:

```
$ (for i in `seq 1 10`; do OMP_NUM_THREADS=$i numactl --cpunodebind 0 --membind 0 ./stream.x; done) | grep "Triad:" | awk '{print $2}'
```

Than I use membind 1

```
$ (for i in `seq 1 10`; do OMP_NUM_THREADS=$i numactl --cpunodebind 0 --membind 1 ./stream.x; done) | grep "Triad:" | awk '{print $2}'
```

NUMA region 0 [Mb/s]	NUMA region 1 [Mb/s]
13593.0	9913.1
13586.7	9915.1
13605.2	9912.3
13534.1	9921.3
13498.4	9923.2
13484.7	9917.2
13468.2	9910.9
13496.0	9921.2
13460.7	9907.8
13550.1	9914.7

On Ulysses:

```
$ lstopo
```

2 sockets

When I have more than 4 threads in one socket it does not scale anymore, because I am saturating the bandwidth along the memory.
If I distribute threads between 2 sockets I'm able to scale better.
To find the best configuration we just have to try.

The bandwidth of one core is measured running a single process and reading from the memory. We are on a NUMA machine, so it changes depending on the memory we have to access.

In order to maximize the bandwidth the best way is to allocate threads on two sockets.

```
$ ./stream_omp.x
$ export OMP_NUM_THREADS=16
$ ./stream_omp.x
```

Compare the results. why does it work better?

```
$ make clean
```

removes all the executables in the folder

to see all the modules loaded:

```
$ module list
```

Rough estimate of the total bandwidth of a node:

```
$ export OMP_NUM_THREADS=20 //the maximum number of threads
$ ./stream_omp.x
```

Now I run stream on 1 core, so I measure the bandwidth on 1 core:

```
$ export OMP_NUM_THREADS=1
$ ./stream_omp.x
```

The result is

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	6387.4	0.200562	0.200395	0.201484
Scale:	11244.9	0.113967	0.113830	0.114222
Add:	11702.1	0.164280	0.164073	0.164613
Triad:	11776.9	0.163195	0.163031	0.163478

so the overall bandwidth of the total machine is 11x20 GB/s, where 20 is the number of cores.

On Ulysses I should get 40 GB/s

```
$ numactl --hardware
```

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 20451 MB
node 0 free: 19534 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19
node 1 size: 20480 MB
node 1 free: 19599 MB
node distances:
node 0 1
0: 10 11
1: 11 10

```
$ lscpu
```

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 20
On-line CPU(s) list: 0-19
Thread(s) per core: 1
Core(s) per socket: 10

Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 62
Stepping: 4
CPU MHz: 2800.140
BogoMIPS: 5599.17
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 25600K
NUMA node0 CPU(s): 0-9
NUMA node1 CPU(s): 10-19

accesso alla memoria dello stesso core:

```
[gcarbone@cn02-06 stream]$ (for i in seq 1 10; do OMP_NUM_THREADS=$i numac
tl --cpunodebind 0 --membind 0 ./stream_omp.x; done) | grep "Triad:" | aw
k '{print $2}'
```

13916.7
20838.3
22956.9
23584.3
23443.2
23562.1
23540.1
23556.5
23308.1
23645.9

accesso alla memoria di un core esterno:

```
[gcarbone@cn02-01 stream]$ (for i in seq 1 10; do OMP_NUM_THREADS=$i numac
tl --cpunodebind 0 --membind 1 ./stream.x; done) | grep "Triad:" | awk
'{print $2}'
```

10002.9
10061.3
9895.2
9874.5
9877.4
9880.9
9888.2
9889.6
9881.7
9887.0

***** compile and run nodeperf.c program**

***** run HPL benchmark using MKL multithread library**

Improving HPL Performance across Nodes
Very helpful notes on tuning HPL are available here. The `HPL.dat` file resides inside `hpl/bin/Ginevra`. The file contains information on the problem size, machine configuration, and algorithm.

In `HPL.dat`, you can change:
N – size of the problem. The problem size is the largest problem size fitting in memory. You should fill up around 80% of total RAM as recommended by the HPL docs. If the problem size is too large, the performance will drop. Think about how much RAM you have. For instance, let's say that I had 4 nodes with 256 MB of RAM each. In total, I have 1 GB of RAM. On our cluster, our peak performance for N is at 64000.
P – number of processes. One caveat is that P is less than Q.
Q – number of nodes. (P * Q is the total number of processes you can run on your cluster).
NBs – subset of N to distribute across nodes. NB is the block size, which is used for data distribution and data reuse. Small block sizes will limit the performance because there is less data reuse in the highest level of memory and more messaging. When block sizes are too big, we can waste space and extra computation for the larger sizes. HPL docs recommend 32 – 256. We used 256.

Creare un nuovo makefile "Make.Ginevra" e poi:

```
$ make clean arch=Ginevra
$ make arch=Ginevra
$ cd bin/Ginvera
$ ldd xhpl
$ module load mkl
$ echo $LD_LIBRARY_PATH
$ ldd xhpl #tutto è risolto correttamente
$ module load openmpi
$ OMP_NUM_THREADS=1 mpirun xhpl
```

dobbiamo usare una matrice abbastanza grande per farlo funzionare, ovvero tale che N~ 75% ram $\sqrt{6}$ GB... deve venire fuori circa 80.000

blocchi non troppo piccoli perché le cache sono grandi,provare con: 128, 256, 512

cercare di arrivare sopra l'80%

gli altri parametri sono P e Q che distribuiscono all'interno della griglia di processori.
Su ulisse ho 20 processori, ma ne uso 16 per problemi di ulisse (Quando uso la infiniband con mpi ho un massimo di 16 processori utilizzati) quindi divido in una griglia 4x4. su cosilt ne ho 24 quindi posso scegliere tra 6x4 e 4x6.

Ovviamente riesco a dividere esattamente iln blocchi quadrati solo se 80000 è una potenza perfetta di 16/24. Nel caso reale ci saranno dei resti.

In hpl dat possiamo definire i diversi valori (guardare quello di Doma)
con top possiamo verificare l'esecuzione, si può fare solo con qsub

Osservazioni:

- # CPU = # socket
- il processore è la parte fisica, il processo è il software
- # CPU instructions = # flops/instruction x # instructions/cycle, che nel caso di ulisse è 8 = 2x4.

non ho capito come si calcolano le flops/cycle

27/10/17

Introduction to benchmarking and tools

qstat -nr

```
tracejob -n 7 130655
```

7 is the number of days it looks back from the job

	latency	bandwidth
infinib	1.5-2 micros	6-12 GB/s
tcp/ip	70 micros	125 MB/s

HPL example in the slides
2.2 Tflps with 2 cores

HPL dense computations
HPCG dense and sparse computations

hpl.dat is the input file

If I have a 24 core machine it's better to choose 6x4 or 4x6 matrix instead of 12x2.
In the case of multi-processor it's better to have a raw-major mapping, so 6x4.

30/10/17

Talking about old exercises

Latency with MPI exercise
traccia

I would like to see how interacting between two nodes is slower.

on ulysses I have to load 2 modules:

```
$ module load  pgi/2016
$ module load  openmpi/1.10.2/gnu/4.8.3
```

Latency is the time it takes to open the channel, it's the third column given by the benchmark.
Now we would like to force them on some cores.
You can estimate the latency when the values on the third column are almost the same.

```
$ mpirun -np 2 /u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1
PingPong
```

Bind the two processes on core 0 and 7 and give the latency inside the socket containing 0 and 7:

```
$ mpirun -np 2 hwloc-bind core:0 core:7 /u/shared/programs/x86_64/intel/i
mpi_5.0.1/bin64/IMB-MPI1 PingPong
```

I got:

#bytes	#repetitions	t[usec]	Mbytes/sec
0	1000	0.19	0.00
1	1000	0.20	4.73
2	1000	0.20	9.37

Now we bind the processes on different sockets:

```
$ mpirun -np 2 hwloc-bind core:0 core:13 /u/shared/programs/x86_64/intel/
impi_5.0.1/bin64/IMB-MPI1 PingPong
```

I got:

#bytes	#repetitions	t[usec]	Mbytes/sec
0	1000	0.56	0.00
1	1000	0.63	1.52
2	1000	0.62	3.06

06/11/17
Ivan Girotto

Parallel programming

Parleremo di diversi protocolli di comunicazione per programmazione in parallelo.

Parallel thinking

Distributed memory vs shared memory

When we create an executable (a.out) data is moved to memory
Then we run the program into the CPU: we temporarly move all the data into the CPU, execute the program and then transfer the data back to the memory.

Communication between cpus introduces a lot of complications and worses execution time.
Concurrency: we broke a problem into single tasks that can be performed in parallel, i. e. executed simultaneouly on different processors.

- [ ] are processor and CPU the same thing?

We write a so called “single program multiple data”.

We consider two processing elements. If I want two processes I have to allow communication between the two parallel parts. This adds more time (communication time) to the total time execution of the initial problem, because we also have to divide it and merge the results.
The goal of this course will be to minimize the total time of execution.

Functional parallelism: different tasks are performed at the same time.
Data parallelism: the same task is performed on different intedenpent objects. In this case we have to define a sinchronization method.

We’ll use for parallel programming: MPI, openMP, cuda.

Writing a parallel program is different from running a serial program in parallel.

Message passing

A library is a collection of (compiled, in this case) code we can use to put up the communication.

MPI

Message passing interface.
We use MPI mainly because it’s based on **distributed memory**, so each process has its own memory. A process can’t access the data of another process, otherwise we get a segmentation fault.

Our goals is to minimize communication or overlap communication and computation for time efficiency (and of course to avoid deadlock).

Amdahl’s law

Evolution according to Amdahl's law of the theoretical speedup in latency of the execution of a program in function of the number of processors executing it, for different values of p. The speedup is limited by the serial part of the program.
Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors.
Amdahl's law can be formulated in the following way:

$$S(n) = \frac{1}{(1-p)+\frac{p}{n}}$$

where

- *S* is the theoretical speedup of the execution of the whole task;

File:AmdahlsLaw.svg

- n is the number of processes;
- *p* is the proportion of execution time that can be parallelized using n processes, while (1-p) is the non parallelizable part

Il calcolo parallelo è utile solamente o per n piccolo o per problemi con valori molto bassi di (1-p).

Link fondamentale per MPI:
http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node182.html

Every time I use an MPI function I'll have to specify a communicator, which is a label for a group of processors ready for
Every MPI program contains a call to the functions mpi_comm_size() and mpi_comm_rank.
Notice that in mpi_comm_size(MPI_COMM_WORLD, &rank) I access the rank by value, in order to change it accordingly.

The allocation of a message is similar to a malloc.

Let's suppose I have 4 processes (P0,..., P4). If the first one does a malloc and initializes the data, the other three processes have to do a malloc to receive the information but without the initialization.

The only difference between these 4 processes is in the initialization, so I know that at a certan point I'll have to write in my C code the condition if(process 0){also do the initialization}.

P0 is only reading and sending data, the other processes are receiving the data and writing the information received in their own memory.

when I do MPI_INIT() the processes start working together
when I do MPI_COMM_RANK() each process has a different rank, so the same command runs differently on the various processes.

If a.out containts an MPI_INIT() there is a difference between

```
$mpirun -np 2 ./a.out
$./a.out &./a.out
```

because in the first case my computer knows that the two processes are working together (splitting the work) while in the second case they are totally independent.
Otherwise they are equivalent.

After the call to MPI_BCAST all the imesg have value 0.

Blocking means that only when the communication is completed, the next instruction is executed. This generates a delay according to the time of synchronization, so it is dangerous by an efficiency point of view.
Any collective communication is blocking.
Of course the sequential pattern of communication can always be optimized, for example by parallel communicating.

Now we run a program using Ulysses:
point2point.c runs two processes, where the first one has buffer 0,.. the second 1.0,.. and sends the first buffer to the second

Observe that the SEND only has input values.

In MPI_COMM_WORLD I specify all communicators because I could define various communicators

mem_buf is equivalent to &mem_buf[0], so if I write

```
MPI_Receive(mem_buf+(BUF_SIZE/2), BUF_SIZE/2, MPI_DOUBLE, 0, MPITAG, MPI_
COMM_WORLD, MPI_STATUS_IGNORE);
```

I only change the second half of the buffer.

- [ ] what is a macro?

Now we compile

```
$ mpicc point2point.c
```

and run

```
$ mpirun -np 2 ./a.out
```

the program, but the output is unordered.
There is no way to control the order of the output in parallel programming.

Deadlock
We write a second program which swaps the buffers, point2point_bis.c. This program is wrong, because instead of a blocking send we are using a buffer send. If the buffer size is low this still works, but it doesn't work if the buffer size is bigger.
This situation is called deadlock, because we are expecting to first send/recv and then send/recv, but it's not happening. Both are sending but no one is receiving.

Let's do another example:

```
1 if(a>0)
2 MPI_Bcast(a, …)
3 else ….
```

If *a* differs for each process and one process enters the else, then this gives a deadlock

Now we write the correct version of the program: *point2point_bis_correct.c*

- [ ] read this: https://stackoverflow.com/questions/20448283/deadlock-with-mpi

Where are processes running?
mpirun will execute a number of "processes" on the machine. The cpu or core where these processes are executed is operating-system dependent. On a N cpu machines with M cores on each cpu, you have room for N*M processes running at full speed.
But, typically:

- If you have multiple cores, each process will run on a separate core
- If you ask for more processes than the available core*cpus, everything will run, but with a lower efficiency (yes, you can run multi-process jobs on a single-cpu single-core machine…)
- If you are using a queuing system or a preconfigured MPI system for which a list of remote machines exists, the allocation will be distributed on the remote machines

Find out the **maximum number of processes** you can use on your machine:

```
cat /proc/sys/kernel/pid_max
```

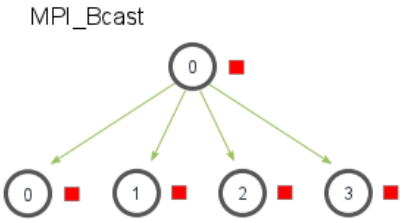
Parallel programming

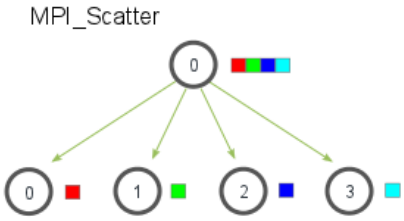
An example of collective communication is the calculation of the transpose of a matrix.

Broadcast vs Scatter

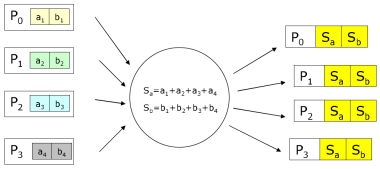
`MPI_Bcast()` is a N to 1 operation: sends the same piece of data to everyone.

`MPI_Scatter()` is 1 to N: sends each process a part of the input array.





Reduction is an N to 1 operation, in which processes have different sizes.



reduction

all-reduce = reduce + broadcast, is a N to N operation.

In **non-blocking** communication I can send from two processes at the same time because at a certain point they will start receiving. This allows to have computation and communication at the same time. Notice that this is not an example of overlapping.

Domandine da esame:
è sbagliato scrivere

```
if (rank==0)
  MPI_BARRIER(MPI_COMM_WORLD)
```

e anche

```
#pragma mpr paralel for
for(i=0, i<N, i++)
  sum =+i
```

perché la variabile sum è in comune

Invece

```
if(rank==0)
  send(...,1,...)
else
  recv(..., 0, ...)
```

è giusta se i processori sono solo due

***** PI exercise**

/home/ginevracoal/Dropbox/DSSC/mount_point/DSSC/foundations_hpc/parallel_programming/01_intro_parallel

esempio:

```
$ mpicc parallel_pi.c
$ mpirun -np 4 ./a.out
```

***** ring implementation**
exercise (third slide) only for MHPC

***** pi parallelo con mpi**
A differenza del codice della prima lezione, questo si basa su un algoritmo diverso.

10/11/17

Shared memory programming paradigm
Slides day 3 parallel

We can program different processes using message passing (mpi).

stack memory vs heap memory
The **stack** is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed.
The **heap** is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.
Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

Difference between process and thread

Inside a single node we work with threads, between more nodes we can't.
This method is faster but works only is systems with a *shared memory*. Multiple nodes are a shared memory system.

OpenMP

OpenMP vs MPI

OpenMP is a way to program on *shared memory* devices. This means that every parallel thread has access to all of your data.
You can think of it as: parallelism can happen during execution of a specific `for` loop by splitting up the work.
You can think of it as: every bit of code you've written is executed independently by every process. The parallelism occurs because you tell each process exactly which part of the global problem they should be working on based entirely on their process ID.

- [x] ~~riguardare esempio a pagina 20~~

example on how to divide program between threads:

```
cint i, n=6;
double mat[6][6]
#pragma omp parallel
start_idx= Thid * 6/ Thsize
end_idx= start_idx+6/Thsize
for (i=start_idx; i<end_idx; i++){
mat[i][i]=1;
}
```

is equal to

```
#pragma omp
int i
#pragma omp parallel for
for()
```

In the second case the for loop is shared among the threads

atomic vs critical
atomic: implemented in hardware, only for simple operations
critical: implemented in software, more complex operations. Only one thread at a time can enter a critical section.

```
for(i…){
  #pragma omp critical
    sum +=mat[i][i]
  #pragma end
}
```

is equal to

```
#pragma omp
  int loc_sum
  for(i..){
    loc_sum +=mat[i][i]
    #pragma atomic
```

```
6         sum = loc_sum
7     }
```

The second is better because the first one does n accesses to shared memory where n is the number of loops, while the second one does n accesses to shared memory where n is the number of threads.

No wait
If specified then threads do not synchronize at
the end of the parallel loop.

Reduction
Reduction(op : list)
- A local copy of each list variable is made and initialized

depending on the "op" (e.g. 0 for "+")
- pair wise "op" is updated on the local value
- Local copies are reduced into a single global copy at the end of the construct.

Barrier

```
#pragma omp barrier
```

This directive synchronises the threads in a team by causing them to wait until all of the other threads have reached this point in the code.

***** parallel pi with OpenMP**
Parallellize pi using omp and make scaling with 1...20 nodes on Ulysses. plot number of threads vs speedup.

```
#pragma omp parallel for private (i,x) reduction(+:sum)
```

x and i are private, otherwise each thread writes on its own copy of the shared variable sum.

13/11/17

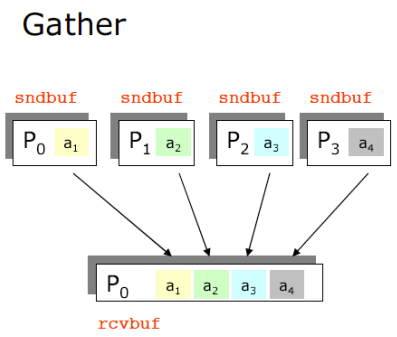
More on // programming

Blocking vs non blocking communication
Blocking communication is done using `MPI_Send()` and `MPI_Recv()`. These functions do not return (i.e., they block) until the communication is finished.
In contrast, *non-blocking communication* is done using `MPI_Isend()` and `MPI_Irecv()`. These function return immediately (i.e., they do not block) even if the communication is not finished yet. You must call `MPI_Wait()` or `MPI_Test()` to see whether the communication has finished.

Implementing matrix multiplication
Initialize the identity matrix splitting the work between processes, one row each distributing memory (replication provides no scaling).
A matrix SIZExSIZE is divided into nprocs matrices of size loc_sizexSIZE, where loc_size=SIZE/nprocs.

The problem is the changing index: A(n,n) becomes A(loc_n,n)
the process with rank=0 prints the matrix and all the others send their results to proc 0.

Gather + Broadcast = Allgather



Our purpose is the one to minimize the number of communications betw procs.
B(N,N)xnprocs increases linearly and doesn't scale, while B(N_loc,N)xprocs, where N_loc=N/nprocs, is constant so it scales.

Assignment MHPC:
Creare anche un plot #nodes vs execution time using t_COMM and t_COMP, where

t_COMM is the ex time of

```
set_send_buf();
MPI_All_Gather(send_buf,...)
```

and t_COMP is the ex time of

```
MATMUL(A,C,Recv_buf);
```

***** Fast transpose OpenMP**

- [ ] [MHPC] parallelize fast transpose code using OpenMP and do a scaling chart... (come al solito).
  You have to end a block when you do a parallel region.

***** matrix mult MPI**

- [ ] come sto misurando il tempo?
- [ ] perché il prof non usa cicli annidati?

soluzione del prof (ce ne sono due):
Nel block_nonblocking usa Isend e Irecv

17/11/17

CUDA

Slides day 6 *parallel I/O.*

CUDA è l'architettura di elaborazione in parallelo di NVIDIA che consente l'esecuzione di programmi in parallelo sulle GPU (delle schede video NVIDIA).

CPU has host memory
GPU has device memory

basic functions:
`host` is called and executed in host
`global` is called in host and executed in device
`device` is called and executed in device

Triple angle brackets in main mark a call from host code to device code:

```
kernel<<< ... >>>()
```

compile and execute cuda code:

```
nvcc -o name name.cu
./name
```

nvcc splits source file into host and device components

We are used to think that the number of threads is always less than the number of processing elements. In cuda we can also have more threads than processing elements, so we write something like:

```
#PRAGMA CUDA
{
idx = cuda_get_idx(); //this function does not exist actually
if( idx < N )
  A[idx] = B[idx] + C[idx];
}
```

Pay attention to pointers in cuda! leggere questo

`cudaMalloc` allocates device memory.

`cudaMemcpy(destination, source, size, type_of_move)` moves data between host and device. This is a blocking code, so the transfer is not synchronous.

In cuda we divide the work between multiple blocks of multiple threads:

tot_n_threads = n_threads_per_block * n_blocks

The function `cuda_get_idx()` does not exist, so in general we have to access the index of a single thread inside a specific block as:

```
idx = blockIdx.x * blockDim.x + threadIdx.x
```

(è come se fosse una matrice blockDim x blockDim)

Kernel function has the structure

```
Kernel <<<numBlocks,threadsPerBlock>>>( /* params for the kernel function */ );
```

and launches numBlocks copies of parallel kernels, using blockIdx and threadIdx to access the correct indexes.

Su cosilt

coda riservata con GPU:

```
-q jrc
```

maximum number of threads per block = 1024

```
$ /opt/cluster/cuda/6.5/samples/1_Utilities/deviceQuery/deviceQuery
```

***** matrix transpose CUDA**

- [x] ~~testare su cosilt e controllare se funziona~~
- [x] ~~capire perché qua uso gli indici i e j mentre nella matrix mult uso gli indici locali, quale dei due metodi è giusto?~~
- [ ] capire perché uso un numero maggiore per la dimensione di un blocco
- [ ] leggere questo link per capire bene come funziona la suddivisione del lavoro sui thread

***** array reversal (MHPC)**

- [ ] pag. 38 slides

Debugging flag

se nel programma definisco una flag

```
#ifdef DEBUG
...
#endif
```

dopo devo compilare in questo modo:

```
nvcc -D DEBUG name.cu
```

First use of CUDA

Access to the cluster and load the module for cuda

```
$ module load cuda/7.5
```

```
$ nvidia-smi
```

tells us the configuration of a node
GPU-util tells us if we are using the GPU

```
$ module load cudatoolkit
```

We are interested in the number of CUDA cores.
The clock rate is very low, usually we have 2-3 GHz. So if we serialize here we go very slow.

ECC support enabled = the arithmetic of the GPU is the same ad the arithmetic of the CPU

parallel communication:

[TA]

[TB]
[sum A+B]
[TA']
[TB'] [TC]
[sum A+B]

compile the cuda code with

```
$ nvcc -c my_cuda_first.cu
```

create an executable

```
$ nvcc -o my_cuda_first.x my_cuda_first.o
```

run it

```
./my_cuda first
```

mount_igirotto
copiare file nel mount point

```
$ cp source_file destination_folder
```

aprire sublime text da terminale:

```
$ subl file
```

20/11/2017
slides day 7 Girotto

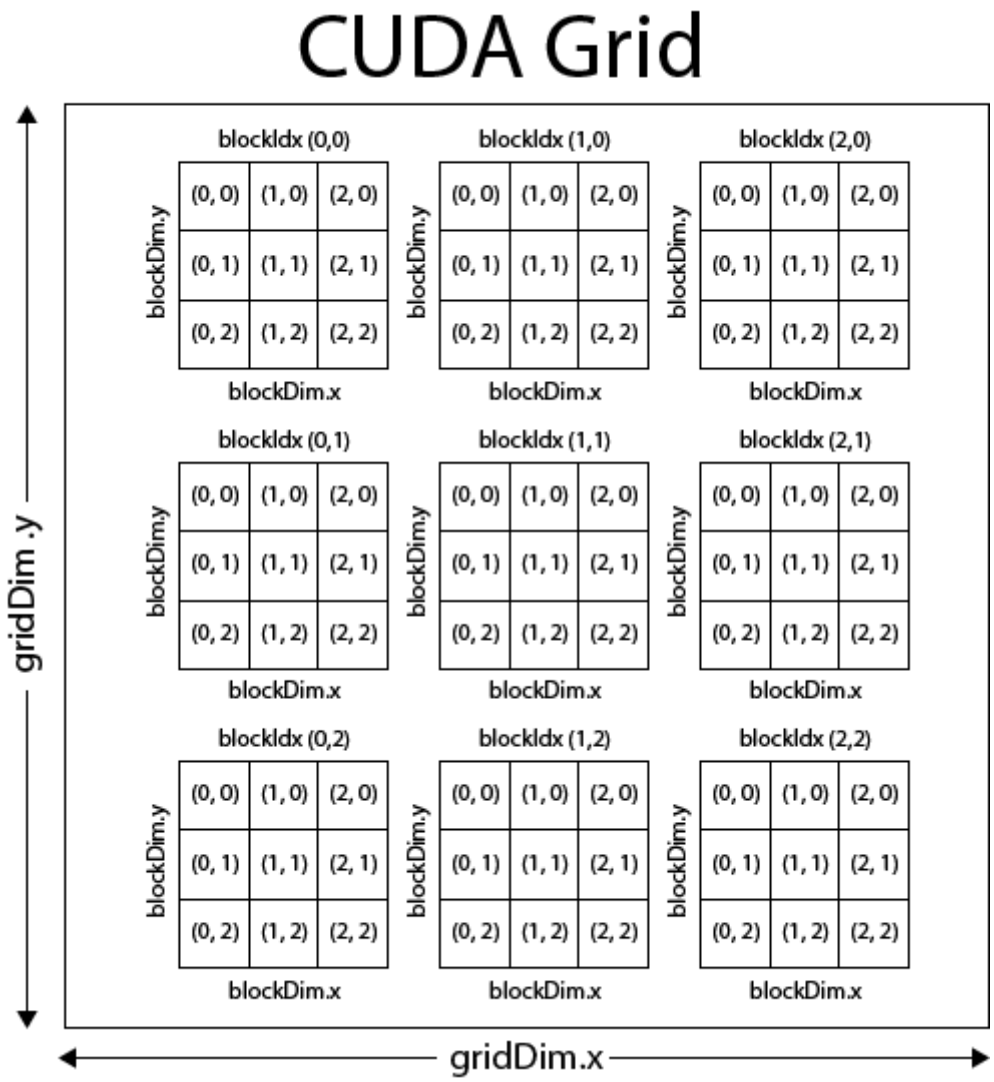
We have SPMD between processors and SIMD within a processor.

understanding GPU

Grids of multithread blocks
For better process and data mapping, threads are grouped into thread blocks. The number of threads varies with available shared memory. The number of threads in a thread block is also limited by the architecture to a total of 512 threads per block.
Threads in the same block can communicate with each other via shared memory, barrier synchronization or other synchronization primitives such as atomic operations.

Why do we need grids?
Multiple blocks are combined to form a grid. All the blocks in the same grid contain the same number of threads. Since the number of threads in a block is limited to 512, grids can be used for computations that require a large number of thread blocks to operate in parallel.

CUDA thread indexing



enter image description here

Per conoscere tutte le informazioni sulla GPU e CUDA:

```
$ deviceQuery
```

Memory hierarchy

- registers
- local thread memory
- shared block memory
- global application (grid) memory

Multiblock dot product

```
__global__ void dot( int *a, int *b, int *c ) {
__shared__ int temp[THREADS_PER_BLOCK];
```

temp è la variabile condivisa fra i thread di ogni blocco, e sarà quella che memorizza tutte le componenti da sommare

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

individua l'indice giusto del thread all'interno del blocco in a e b

```
  temp[threadIdx.x] = a[index] * b[index];
  __syncthreads();
```

sincronizzo in modo da poter fare la somma

```
  if( 0 == threadIdx.x ) {
    int sum = 0;
    for( int i = 0; i < THREADS_PER_BLOCK; i++ ) sum += temp[i];
```

il thread 0 somma tutti elementi di temp e ottiene il valore sum per il blocco corrente

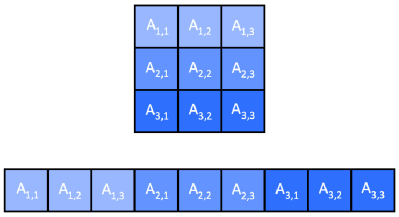
```
    atomicAdd( c , sum );
```

si aggiunge il risultante c a sum per ogni blocco. si usa la somma atomica perché c e sum sono nella memoria del thread 0.

matrici in CUDA

Matrices as 2D arrays:
The easiest way to linearize a 2D array is to stack each row lengthways.
These are row and column indexes:

```
int ROW = blockIdx.y*blockDim.y+threadIdx.y;
int COL = blockIdx.x*blockDim.x+threadIdx.x;
```



***** bidim matrix multiplication CUDA**

```
dim3 dimBlock(4,4);
```

significa che ogni blocco è di dimensioni 4x4, quindi funziona solo per matrici la cui dimensione (size) è un multiplo di 4.
In questo esempio ogni blocco opera su una riga di A ed una colonna di B, per ottenere un blocco di C.

In generale avrei

```
SIZE = blockDim.y*gridDim.y
```

e in questo caso è giusto perché

```
blockDim.y*gridDim.y = 4 * (matrixsize/4) = matrixsize
```

```
nvidia-smi
```

informazioni sulla GPU

***** shared matrix multiplication CUDA**

Alcuni esempi:

- alberto
- piero

- elisa

spiegazione più dettagliata

- [ ] here you can compare cpu vs gpu speedup
  s= t_cpu/ (t_cpu+t_gpu)
- [ ] perché non posso usare più di 31 thread per blocco?

Caso 1D con s_A
Hints del prof:

- every block of threads i stores the i-th row of A in SM
- every block of threads computes the i-th row of matrix C
  A_row x B = C_row
- when you take timing remember to do something like:

```
timing
memcopy
kernel <>
memcopy
timing
```

```
qstat gpu
```

per vedere l'attuale stato della GPU e i processi attivi

Questo significa che ogni thd calcola il dot product tra s_A e una colonna di B. Tutti i risultati sono nella SM e compongono una riga di C.
Problema a cui stare attenta: possono esserci meno thread della dimensione della matrice! Quindi devo usare il block index: Ogni riga di A è suddivisa in n° di blocchi per riga parti copiate nella shared memory del blocco. Quindi s_A è una riga di A ma non è tutta contenuta in un singolo blocco di shared memory, perché il numero di thread è inferiore alla dimensione della matrice, ovvero per ogni riga dovrò considerare più di un blocco.

NUM_BLOCKS è il numero di blocchi per riga

Posso condividere sia A che B? perché le griglie dovrebbero essere di dimensioni diverse (o comunque una per righe ed una per colonne)...

24/11/17

Test benchmarks using HPCG, MPI
Cozzini

Slides on benchmarks, *lecture 8*

*** **HCPG**

- [ ] compile
- [ ] run on single node with different size and timing

104x104x104 → 300x300x300
60(times in second) → 360

- [ ] compute ratio among your best HPC and your best HPCG
- [ ] optional: compare results with the precompiled version on HPCG coming from intel

assign only a certain number of MPI procs on one node:

```
-npernode X
```

try writing

```
$ mpirun -np 4 /bin/hostname
```

and see how many nodes and hosts you have

```
ompi_info | grep btl
```

to know all the infos about btl available

```
cd src
ll
mirun -np 2 –report-bindings ./IMB...
mirun -np 2 -npernode 1 –report-bindings ./IMB...
```

I am getting the bandwidth between two nodes

```
mirun -np 2 -mca btl tcp –report-bindings ./IMB…
```

now i am forcing the use of tcp

if it does not work use

```
mirun -np 2 -mca btl tcp,self –report-bindings ./IMB…
```

Parallel debugging

Slides *parallel*, Tornatore

Using gdb with MPI
non-stop and all-stop modes

If you have to debug 100k processes or more don't use gdb, use instead ARM or DDT.

```
gdb -q -tui
```

open a double screen where you can see the code

Hands on with gdb on an MPI code
Sartori

Ho usato il file *mpi_pi.c*

We can use several terminal emulators: xterm, sakura, ...

```
$ mpicc -g mpi_pi.c -o mpi_pi.x
$ mpirun -np 2 sakura -e gdb mpi_pi.x
```

This creates two windows, one for each process.

```
(gdb) CTRL+X
(gdb) A
(gdb) start
```

splits the gdb terminal in two parts: terminal and source code

Alternative to gdb:

```
$ mpicc -DDEBUGGER -g mpi_pi.c
$ mpirun -np 2 ./a.out
```

27/11/17
Girotto, lezione persa

Introduction to Hybrid Programming
Basic Examples of Hybrid Programming

01/12/17
Esame Girotto

***** midterm exam**

- [ ] correggere plot openmp
- [ ] prima studiare tutte le vecchie lezioni commentando i codici
- [ ] Fate un confronto CPU (con 1, 5, 10 cores) Vs GPU e mettetelo su un grafico ad istogrammi, considerando il trasferimento dei dati (per la parte GPU).
- [ ] Se qualcuno e' ambizioso puo' pensare di utilizzare la shared memory per la versione GPU.
- [ ] capire in quali casi uso GPU e CPU
- [ ] plot di confronto tra tutti i metodi
- [ ] spiegare la differenza tra i metodi
- [ ] parallelizzare con mpi (opzionale)

04/12/17
Cozzini

The energy issues on HPC system

remember Moore law

Esempi di consumi:

- 75w su un normale pc (una sola cpu)
- 400w un nodo di ulisse (2 cpu), di cui 100 per ogni cpu + la ram + il raffreddamento...
  In questo caso considero power=~500, quindi ppw=~1.

PUE

- [ ] Blas libraries

Netlib accesses the memory too often, that's why the ram consumption is high
Mkl is the most efficient library

Increasing the frequency scales the performance well, but this is no longer true for the efficiency. So it's better to spend a little bit more time on computation than on increasing the frequency, in terms of energy.
Osservare che il tempo di esecuzione è quasi sempre lo stesso.
In general the highest performance is not the most convenient one.

nodo exact lab con GPU:

```
qsub -q jrc -I
```

per vedere se c'è una GPU:

```
nvidia-smi
```

11/12/17
Tornatore

Optimizing code
Declaring variables does not take time

Infos about optimization in gcc:

```
gcc -f opt-info-vec-<option>
```

`optimized`: Print information when an optimization is successfully applied. It is up to a pass to decide which information is relevant. For example, the vectorizer passes print the source location of loops which are successfully vectorized.
`missed`: Print information about missed optimizations. Individual passes control which information to include in the output.
`note`: Print verbose information about optimizations, such as certain transformations, more detailed messages about decisions etc.
`all`: Print detailed optimization information. This includes 'optimized', 'missed', and 'note'.

- [ ] restrict type qualifier

for example this code, using an additional variable tmp, could be faster

```
swap(a,b){
  type tmp_a=a, tmp_b=b;
  a=tmp_b;
  b=tmp_a;
```

- [ ] gcc -march=native

Memory optimization

strided access in write is worse than strided access in read.
strided access

Loop unrolling is a fairly hotly contested subject. On one hand, it *can* lead to code that's much more CPU-friendly, reducing the overhead of instructions executed for the loop itself. At the same time, it can (and generally does) increase code size, so it's relatively cache unfriendly.

The data cache is also limited in size. This means that you generally want your data packed as densely as possible so as much data as possible will fit in the cache

Data structure reordering: depth first, breadth first,...

distance distortion = distance in memory / distance in space

Space filling curves

Loop optimization

For example in distribute particles you could use a specialized function and avoid the if condition inside the loop.

```
func(v,i)
  if(v==0)
    func(v,0)
  else if (v==1)
    func(v,1)
  …

double(*func_ptr)(v, … )
```

Prefetching

Esame

markdown to another folder

Parte di Cozzini: https://github.com/sissa/P1.2_seed/tree/master/Assignements

- A01: compute weak/strong scalability
- A02: loop_optimization
- A05: Transpose a matrix
- A06: Stream
- A09: HPL with MKL

Parte di Girotto: https://github.com/sissa/P1.2_seed

- today assignment is the implementation of the PI approximation using MPI. The exercise was presented in class and schematically described on today slides1. The serial code is included in the hands-on folder.
- today assignment is the implementation of the PI approximation using OpenMP. The exercise was presented in class and schematically described on today slides2. Provide the scaling curve of the multi-threaded version of the code on 1, 2, 4, 8, 16, 20 Threads performed on a Ulisse's compute node.
- today assignment is the implementation of the Parallel Distributed MatMul as presented in class. The Exercise is divided in 5 main points: 1) Distribute the Matrix, 2) Initialize the Distributed Matrix, 3) At every time step use MPI_Allgather to send at all processes a block of column of B, 4) Repeat point 3 for all blocks of column of B and 5) Sequential Print of the Matrix C with all processes sending data to P0 (example given in class and reference sent by e-mail).
- today assignment is the implementation of a naive version of Parallel Transpose of a NxN matrix in CUDA.