Parallel Computing: a brief discussion

Marzia Rivi

What is parallel computing?

Traditionally, software has been written for **serial computation**:

- To be run on a single computer having a single core.
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently.
- Instructions from each part executed simultaneously on different cores.

Why parallel computing?

- **Save time and/or money**:
  - in theory, more resources we use, shorter the time to finish, with

Who needs parallel computing?

<table>
<tr><td>Researcher 1</td><td>• has a large number of independent jobs (e.g. processing video files, genome sequencing, parametric studies)
• uses serial applications</td><td rowspan="2">High Throughput Computing (HTC)
(many computers):
• dynamic environment
• multiple independent small-to-medium jobs
• large amounts of processing over long time
• loosely connected resources (e.g. grid)</td><td rowspan="2"></td></tr>
<tr><td>Researcher 2</td><td>• developed serial code and validated it on small problems
• to publish, needs some “big problem” results</td></tr>
<tr><td>Researcher 3</td><td>• needs to run large parallel simulations fast (e.g. molecular dynamics, computational fluid dynamics, cosmology)</td><td></td><td>High Performance Computing (HPC)
(single parallel computer):
• static environment
• single large scale problems
• tightly coupled parallelism</td></tr>
</table>

How to parallelise an application?

Automatic parallelisation tools:

- compiler support for vectorisation of operations (SSE and AVX) and threads parallelisation (OpenMP)
- specific tools exists but limited practical use
- all successful applications require intervention and steering

Parallel **code development** requires:

- programming **languages** (with support for parallel libraries, APIs)
- parallel programming **standards** (such as MPI and OpenMP)
- **compilers**
- performance **libraries/tools** (both serial and parallel)

But , more that anything, it requires **understanding:**

- **the algorithms** (program, application, solver, etc.):
- the factors that influence **parallel performance**

How to parallelise an application?

- First, make it **work**!
  - analyse the key features of your parallel algorithms:
    - **parallelism**: the type of parallel algorithm that can use parallel agents
    - **granularity**: the amount of computation carried out by parallel agents
    - **dependencies**: algorithmic restrictions on how the parallel work can be scheduled
  - re-program the application to run in parallel and validate it
-

Task Parallelism

Data Parallelism

Dependencies

Parallel computing models

OpenMP - example

Objective: vectorise

http://www.mpi-forum.org/

MPI is a specification for a Distributed-Memory API designed by a committee for Fortran, C and C++ languages.

- **Two versions**:
  -

MPI implementation components

- **Libraries** covering the functionality specified by the standard.
- **Header files**, specifying interfaces, constants etc.
  -

MPI - overview

- **Processes** (MPI tasks) are mapped to **processors** (CPU cores).
- **Start/stop mechanisms:**
  - `MPI_Init()` to initialise processes
  - `MPI_Finalize()` to finalise and clean up processes
- **Communicators:**
  - a communicator is a collection (network) of processes
  - default is `MPI_COMM_WORLD`, which is always present and includes all processes requested by `mpirun`
  - only processes included in a communicator can communicate
- **Identification mechanism:**
  - process id: `MPI_Comm_rank()`
  - communicator size (number of processes): `MPI_Comm_size()`

MPI - communication

Inter-process communication (the cornerstone of MPI programming):

-

Distributed vs shared memory

Distributed vs shared memory paradigm

Which problems are suited to **Distributed Memory Processing**?

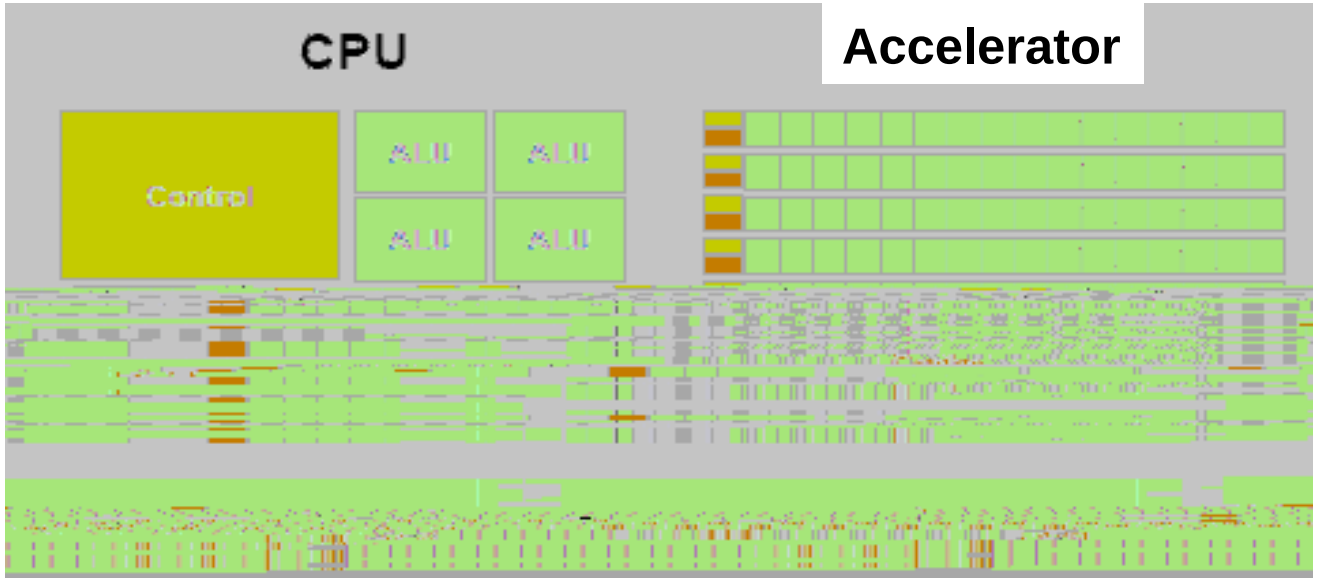
- **Embarrassingly parallel** problems (independent tasks), e.g. Monte Carlo methods.
- **Computation bound** problems (heavy local computation with little data exchange between processes).
  - models with

Accelerators - motivation

Moore's Law (1965):

- the number of transistors in CPU design doubles roughly every 2

Accelerators – different philosophies



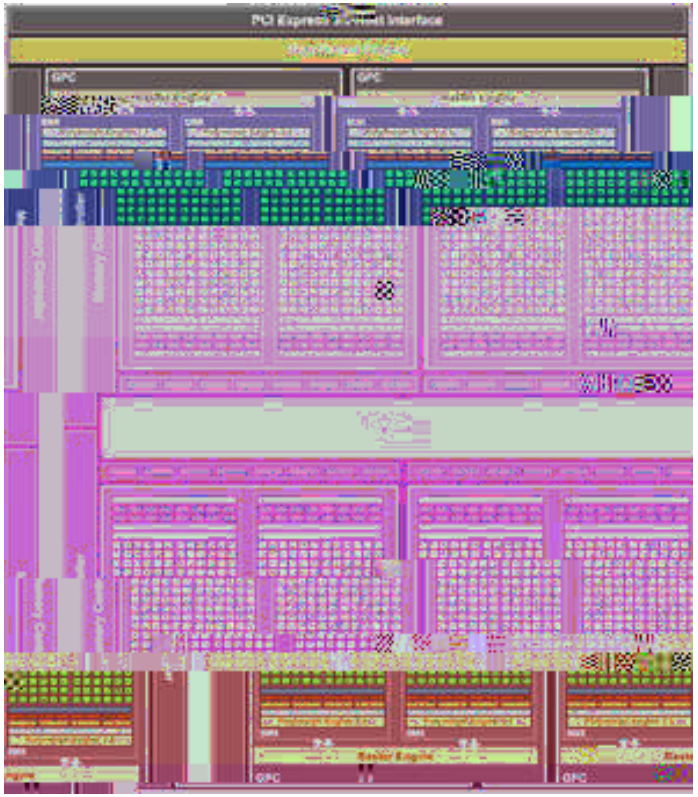
Design of CPUs optimized for **sequential** code and **coarse grained parallelism**:

- multi-core
- sophisticated control logic unit
- large cache memories to reduce access latencies.

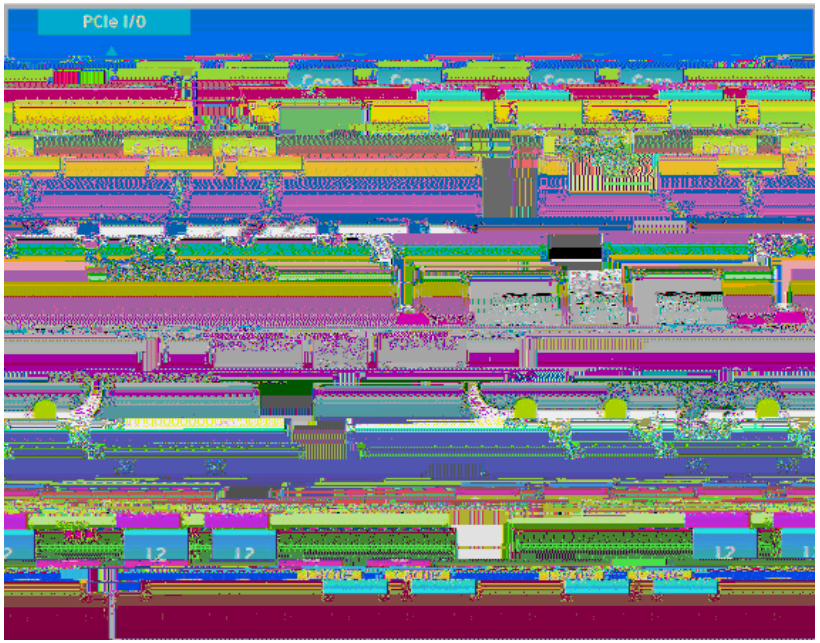
Design of accelerators optimized for *numerically intensive* computation by a **massive fine grained parallelism**:

- many-cores (several hundreds)
- leightweight threads and high execution throughput
- large number of threads to overcome long-latency memory accesses.

Accelerators - examples



NVIDIA Tesla K20X GPU
2688 cores
6GB GDDR5 memory
250 GB/sec memory bandwidth
3.95Tflops/sec of peak SP



Intel Xeon Phi 5110 MIC
60 cores
8GB GDDR5
320 GB/s memory bandwidth
240 HW threads (4 per core)
512-bit wide SIMD capability

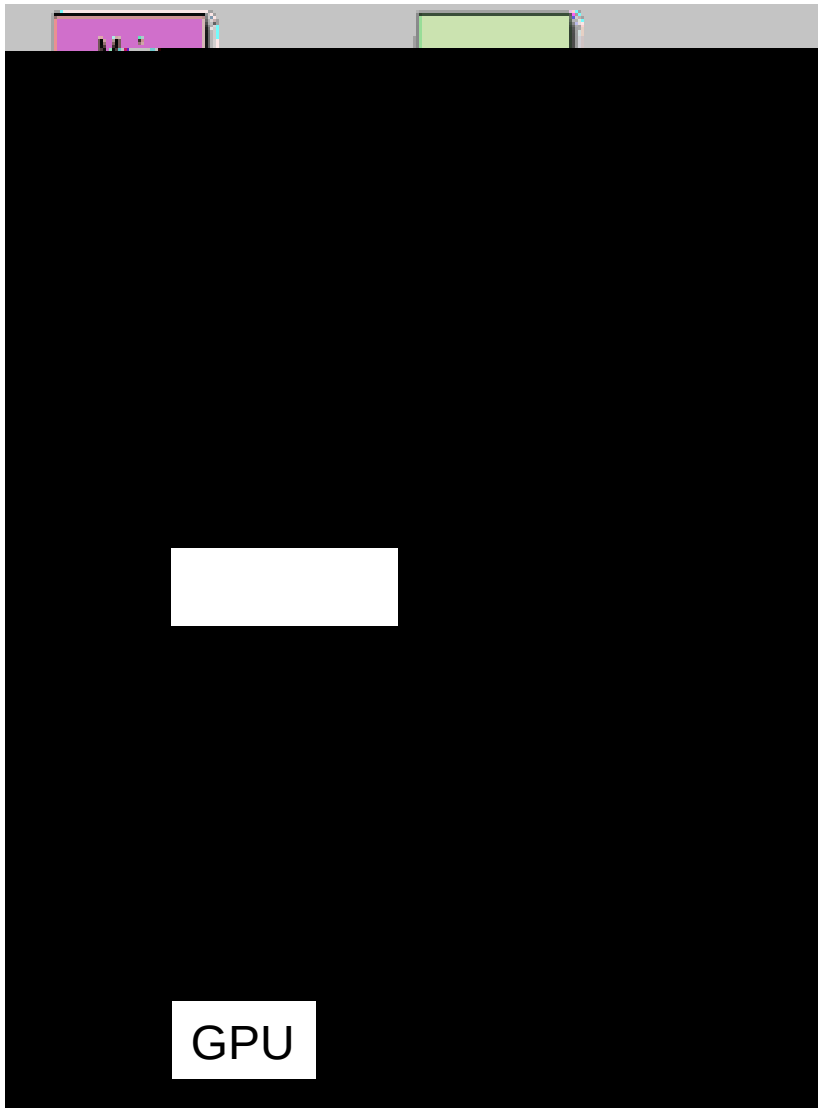
Accelerators – programming model

Applications should use both CPUs and the accelerator, where the latter is exploited as a **coprocessor**:

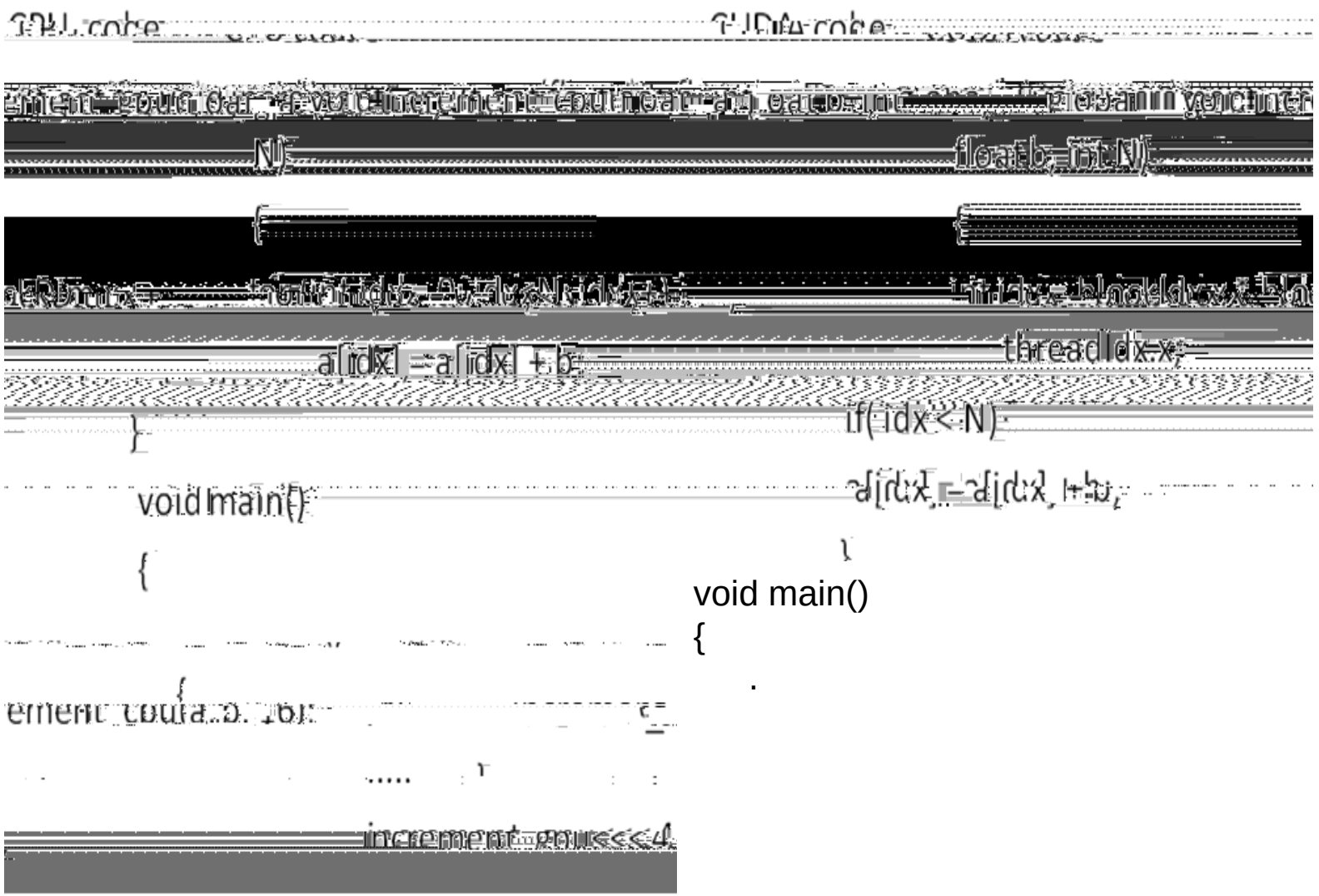
- Serial sections of the code are performed by CPU (host).
- The parallel ones (that exhibit rich amount of *data parallelism*) are performed by accelerator (device).
- Host and device have separate memory spaces: **need to transfer data** in a manner similar to one-sided message passing.

Several **languages/API**:

- GPU: CUDA, pyCUDA, OpenCL, OpenACC
- Xeon Phi: OpenMP, Intel TBB, Cilk



Example – CUDA



```
void main()
{
    .
```

OpenACC

- ✓ Supported by CRAY and PGI (slightly different implementations, but converging) and soon GCC.
- ✓ “Easier” code development – supports incremental development.
- ✓ possible performance loss – about 20% compared to CUDA.
- ✓

Accelerators programming

- Accelerators suitable for **massively parallel** algorithms and require **low-level programming** (architecture bound) to have good performances.
- They can effectively **help** in reducing the **time to solution**. However the effectiveness is **strongly dependent** on the algorithm and the amount of computation.
- The **effort** to get codes efficiently running on accelerators is, in general, **big, irrespectively of the programming model** adopted. However portability and maintainability of the code push toward directive based approaches (at the expenses of some performance).
- All the (suitable) computational demanding parts of the code should be ported. Data transfer should be minimized or hidden. Host-Device **overlap is hard to achieve**.

Hybrid parallel programming

Hybrid programming (MPI+OpenMP, MPI+CUDA) is a **growing trend.**

- Take the positive of all models.
- Suits the memory hierarchy on “fat-nodes” (nodes with large memory and many cores).
- Scope for better scaling than pure MPI (less inter-node communication) on modern clusters.

Questions?