Parallel Computing: a brief discussion

Marzia Rivi

Astrophysics Group

Department of Physics & Astronomy

Research Programming Social – Nov 10, 2015

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 potential cost savings.
- **Solve larger problems**:
  - when the problems are so large and complex, it is impossible to solve them on a single computer, e.g. "Grand Challenge" problems requiring PetaFLOPS and PetaBytes of computing resources (en.wikipedia.org/wiki/Grand_Challenge).
  - Many scientific problems can be tackled only by **increasing processor performances**.
  - Highly complex or memory greedy problems can be solved only with **greater computing capabilities**.
- **Limits to serial computing**: physical and practical reasons

Who needs parallel computing?

Researcher 1	• has a large number of independent jobs (e.g. processing video files, genome sequencing, parametric studies) • uses serial applications	**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)	
Researcher 2	• developed serial code and validated it on small problems • to publish, needs some “big problem” results		**High Performance Computing (HPC) (single parallel computer):** • static environment • single large scale problems • tightly coupled parallelism
Researcher 3	• needs to run large parallel simulations fast (e.g. molecular dynamics, computational fluid dynamics, cosmology)		

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

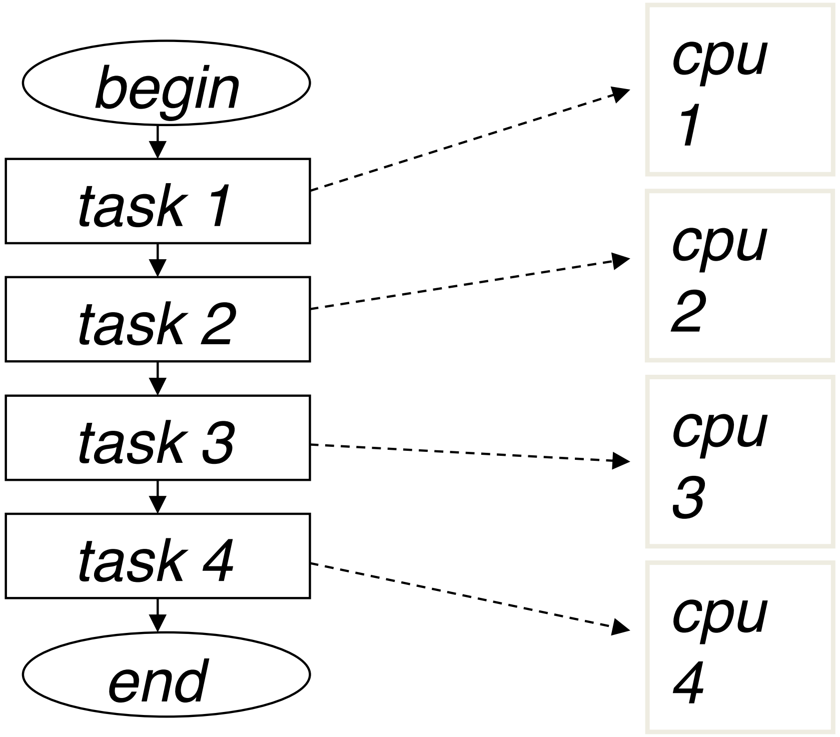
- Then, make it work **well**!
  - Pay attention to the key aspects of an optimal parallel execution:
    - **data locality** (computation vs. communication)
    - **scalability** (linear scaling is the holy grail: execution time is inversely proportional with the number of processors)
  - Use profilers and performance tools to identify problems.

Task Parallelism

Thread (or task) parallelism is based on **executing concurrently different parts** of the algorithm.

Features

- Different independent sets of instructions applied to single set (or multiple sets) of data.
- May lead to *work imbalance* and may not scale well and *performance limited by the slowest process*.

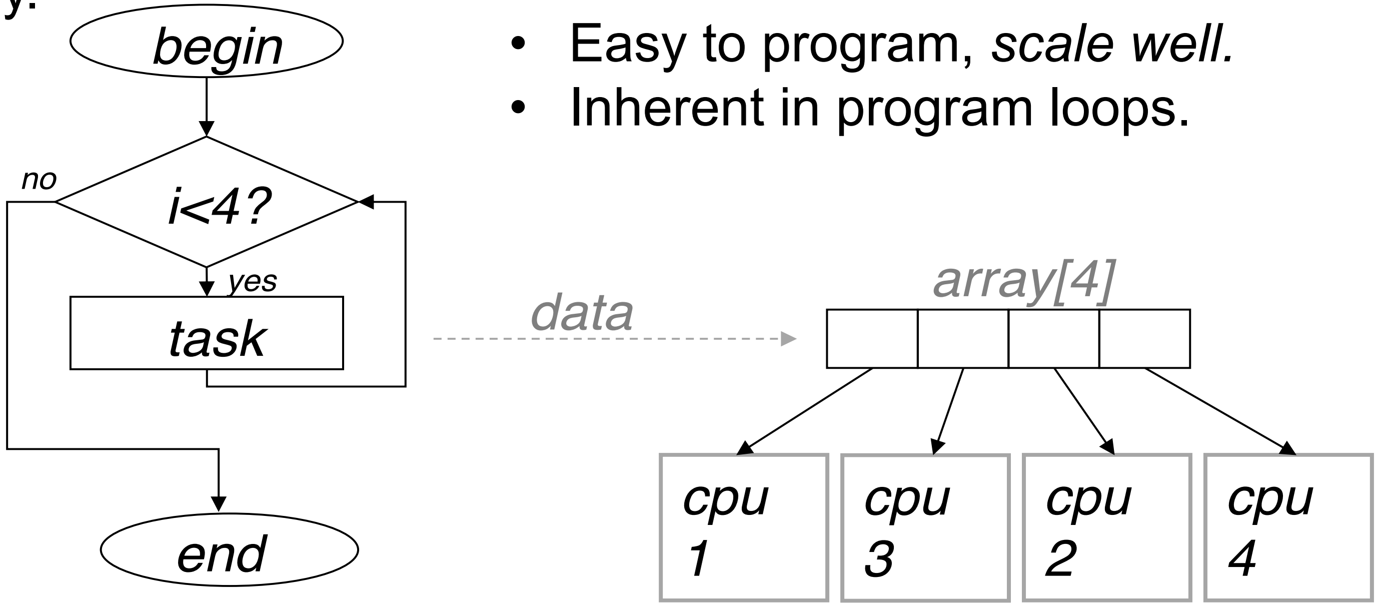


Data Parallelism

Data parallelism means **spreading data** to be computed through the processors.

Features

- The same sets of instructions applied to different (parts of the) data.
- Processors work only on data assigned to them and communicate when necessary.



- Easy to program, *scale well.*
- Inherent in program loops.

Granularity

- **Coarse**
  - parallelise large amounts of the total workload
  - in general, the coarser the better
  - minimal inter-processor communication
  - can lead to imbalance
- **Fine**
  - parallelise small amounts of the total workload (e.g. inner loops)
  - can lead to unacceptable parallel overheads (*e.g.* communication)

Dependencies

Dictate the order of operations, imposes limits on parallelism and requires parallel **synchronisation**.

In this example the *i* index loop can be parallelized:

fortran
```
DO I = 1, N
         DO J = 1, N
            A(J,I) = A(J-1,I) + B(J,I)
         END DO
END DO
```

c/c++
```
For (i=1; i<n; i++){
    for (j=1 ;j<n; j++){
        a[j][i] = a[j-1][i] + b[j][i];
        }
    }
```

In this loop parallelization is dependent on the *k* value:

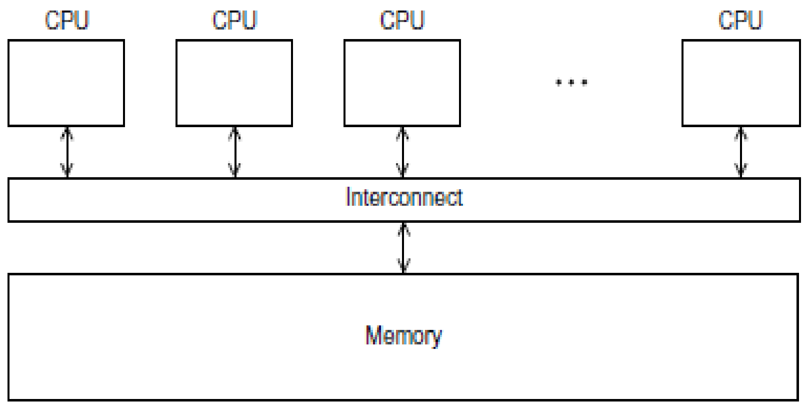
fortran
```
DO I = M, N
            A(I) = A(I-K) + B(I)/C(I)
END DO
```

c/c++
```
For (i=m; i<n; i++){
    a[i] = a[i-k] + b[i]/c[i];
    }
```

If k > M-N or k < N-M parallelization is straightforward.

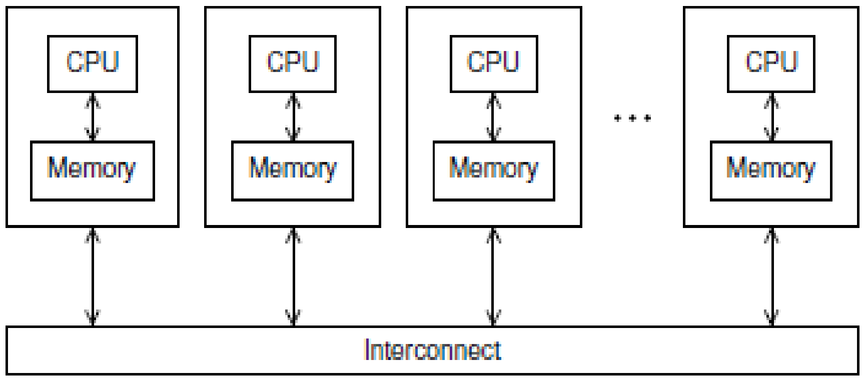
Parallel computing models

Shared memory



- Each processor has direct access to **common physical memory** (e.g. multi-processors, cluster nodes).
- Agent of parallelism: the **thread** (program = collection of threads).
- Threads exchange information *implicitly* by **reading/writing shared variables.**
- Programming standard: **OpenMP**.

Distributed memory



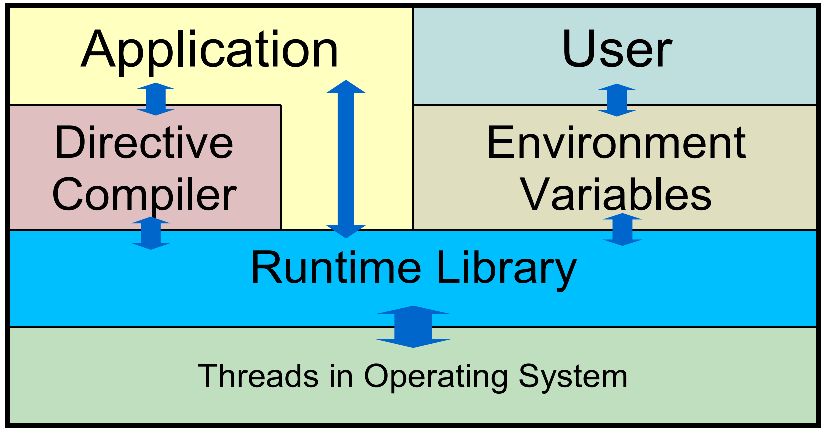
- Local processor memory is invisible to all other processors, **network based memory access** (e.g. computer clusters).
- Agent of parallelism: the **process** (program = collection of processes).
- Exchanging information between processes requires **communications**.
- Programming standard: **MPI**.

OpenMP

http://www.openmp.org

API instructing the compiler what can be done in parallel (**high-level programming**).

- Consisting of:
  - compiler **directives**
  - runtime **library functions**
  - environment **variables**

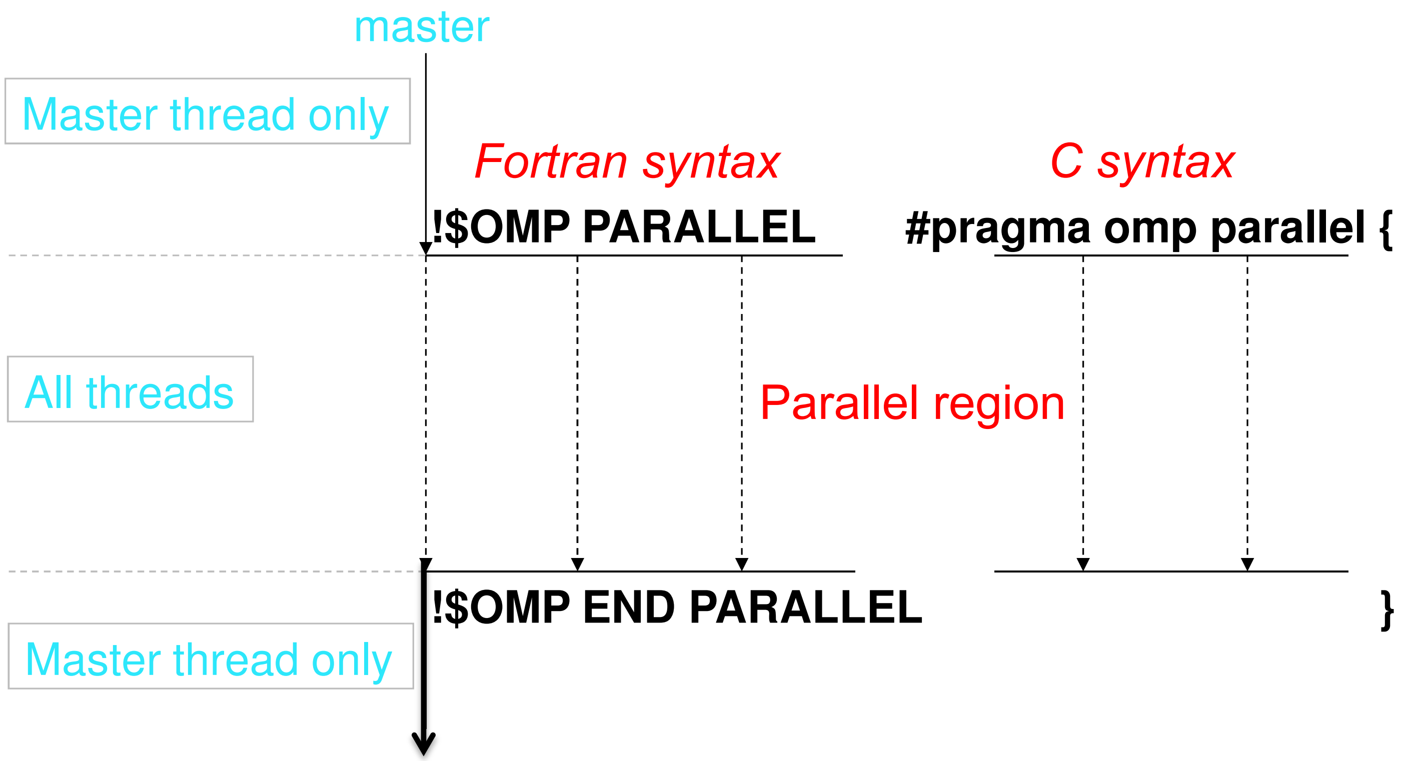


- Supported by most compilers for Fortran and C/C++.
- Usable as serial code (threading ignored by serial compilation).
- By design, suited for data parallelism.

OpenMP

Threads are generated automatically at runtime and scheduled by the OS.

- **Thread creation / destruction overhead.**
- Minimise the number of times parallel regions are entered/exited.



OpenMP - example

Objective: vectorise a loop, to map the sin operation to vector x in parallel.

Idea: instruct the compiler on what to parallelise (the loop) and how (private and shared data) and let it do the hard work.

In C:

```
#pragma omp parallel for shared(x, y, J) private(j)
for (j=0; j<J; j++) {
   y[j] = sin(x[j]);
}
```

In Fortran:

```
$omp parallel do shared(x, y, J) private(j)
do j = 1, J
y(j) = sin(x(j))
end do
$omp end parallel do
```

Number of threads is set by environment variable `OMP_NUM_THREADS` or programmed for using the RTL function `omp_set_num_threads()`.

MPI - Message Passing Interface

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 1.0, quickly and universally adopted (most used and useful)
  - MPI 2.0, is a superset of MPI 1.0 (adding parallel I/O, dynamic process management and direct remote memory operations) but is not so popular.

- **Many implementations**
  - open software (MPICH, MVAPICH, OpenMPI)
  - vendor (HP/Platform, SGI MPT, Intel).

MPI implementation components

- **Libraries** covering the functionality specified by the standard.
- **Header files**, specifying interfaces, constants etc.
  - C/C++: `mpi.h`
  - Fortran: `mpif.h`

- **Tools** to compile and link MPI applications (wrappers around serial compilers)
  - Fortran: `mpif77, mpif90`
  - C: `mpicc`
  - C++: `mpicxx, mpiCC`

- An MPI application **launcher** (mapping processes to CPUs)

```
mpirun -np <n processes> <executable>
```

MPI - overview

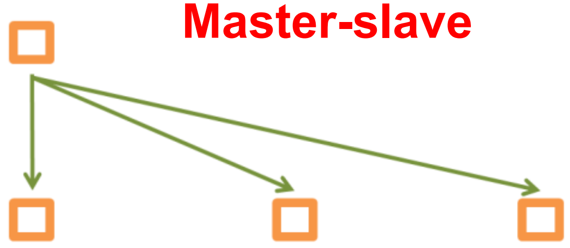
- **Processes** (MPI tasks) are mapped to **processors** (CPU cores).
- **Start/stop mechanisms:**
  - `MPI_Init()` to initialise processes
  - `MPI_Finalize()` to finalise and clean up processes
- **Communicators:**
  - a communicator is a collection (network) of processes
  - default is `MPI_COMM_WORLD`, 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

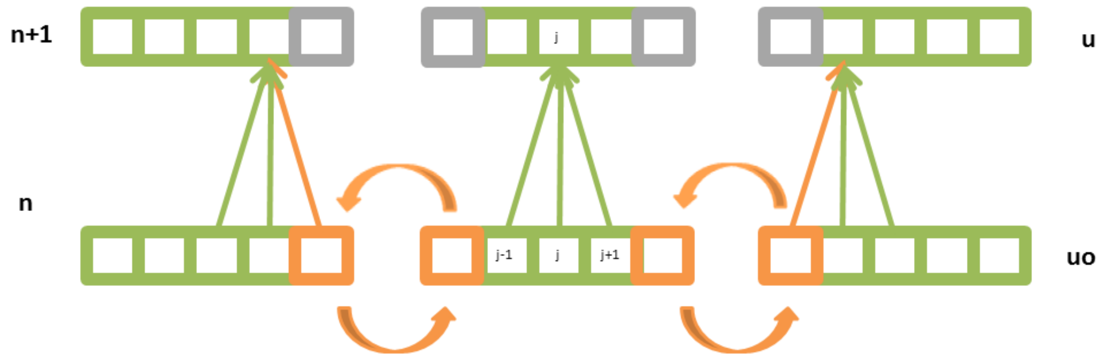
Inter-process communication (the cornerstone of MPI programming):

- **one-to-one** communication (*send, receive*)
- **one-to-many** communication (*broadcasts, scatter*)
- **many-to-one** communication (*gather*)
- **many-to-many** communication (*allgather*)
- **reduction** (*e.g.* global sums, global max/min) (a special many-to-one!)
- process **synchronisation** (*barriers*)

Master-slave



Domain decomposition



MPI - example

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[])
{
   int my_rank, numprocs; char message[100]; int dest, tag, source; MPI_Status
status;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   if (my_rank != 0)
   {
         sprintf(message,"Greetings from process %d !\0",my_rank); dest = 0;
         tag = 0;
         MPI_Send(message, sizeof(message), MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else {
        for (source = 1; source <= (numprocs-1); source++)
        {
          MPI_Recv(message, 100, MPI_CHAR,  source, tag, MPI_COMM_WORLD, &status);
          printf("%s\n",message);
        }
    }
 MPI_Finalize();
}
```

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```

Distributed vs shared memory

application feature	shared memory / OpenMP	distributed memory / MPI
parallelisation	• easy, incremental (parallelising small parts of the code at a time) • mostly parallelise loops	• relatively difficult (tends to require a all-or-nothing approach) • can be used in a wider range of contexts
scaling (hardware view)	both expensive (few vendors provide scalable solutions) and cheap (multi-core workstations)	• relatively cheap (most vendors provide systems with 1000's of cores) • runs on both shared and distributed systems
scaling (programming view)	small/simple programs are easy and fast to implement	even small/simple programs involve large programming complexity
maintainability	code is relatively easy to understand and maintain	code is relatively difficult to understand
readability	small increase in code size, readable code	tends to add a lot of extra coding for message handling, code readable with difficulty
debugging	• requires special compiler support • debuggers are extension of serial ones	• no special compiler support (just libraries) • specialised debuggers

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 localised data, e.g. PDEs solved using finite elements/volumes (CFD, CHMD, etc.)
  - other models with distributed data: molecular dynamics, etc.

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

- **Communication bound** problems (much data shared between threads)
  - models with non-local data: e.g. Newtonian particle dynamics
  - Fourier transform, convolutions.

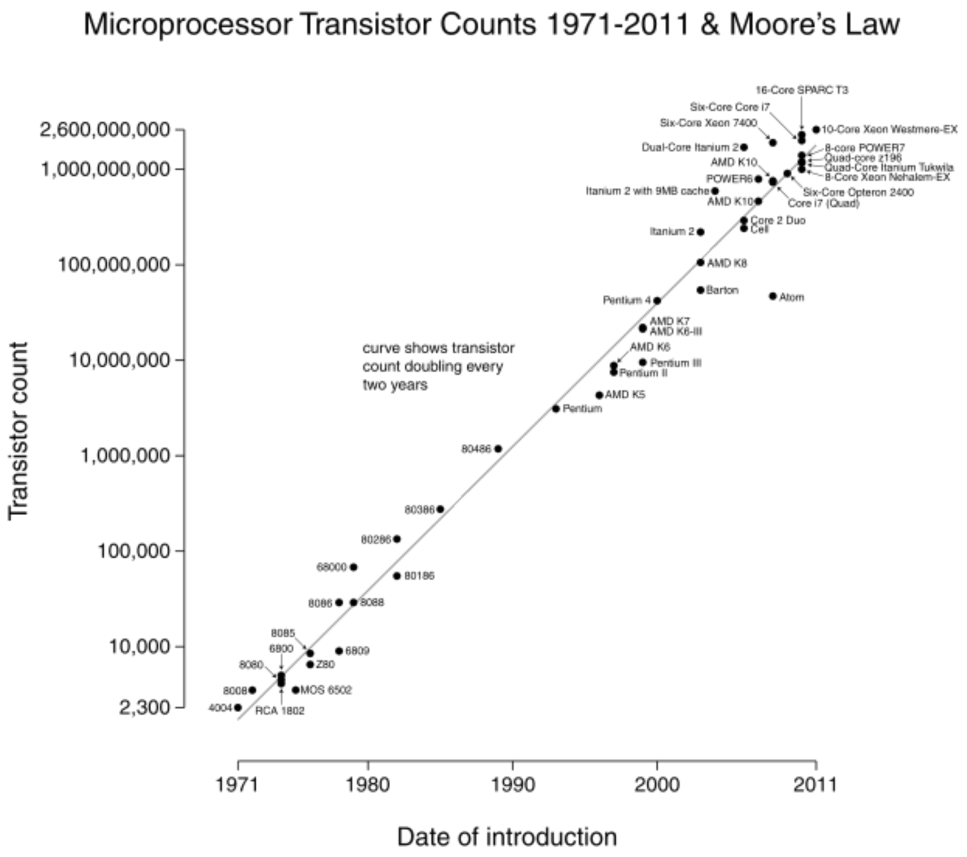
Accelerators - motivation

Moore's Law (1965):

- the number of transistors in CPU design doubles roughly every 2 years
- backed by clock speed increase, this has correlated with exponentially increasing CPU performance for at least 40 years.

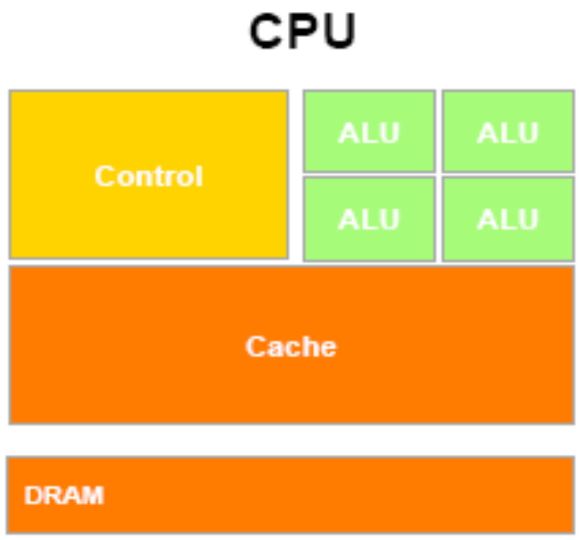
This meant the same old (single-threaded) **code just runs faster on newer hardware. No more!**

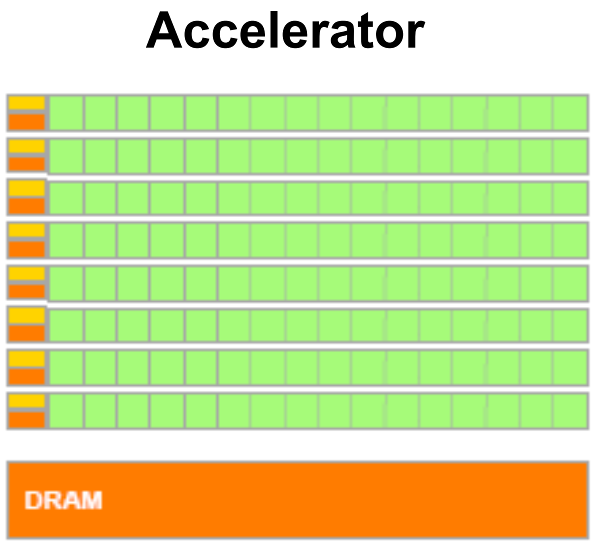
While the "law" still holds, clock frequency of general purpose CPUs was "frozen" in 2004 at around 2.5-3.0 GHz and **design has gone multicore**.



Performance improvements are now coming from the increase in the number of cores on a processor.

Accelerators – different philosophies





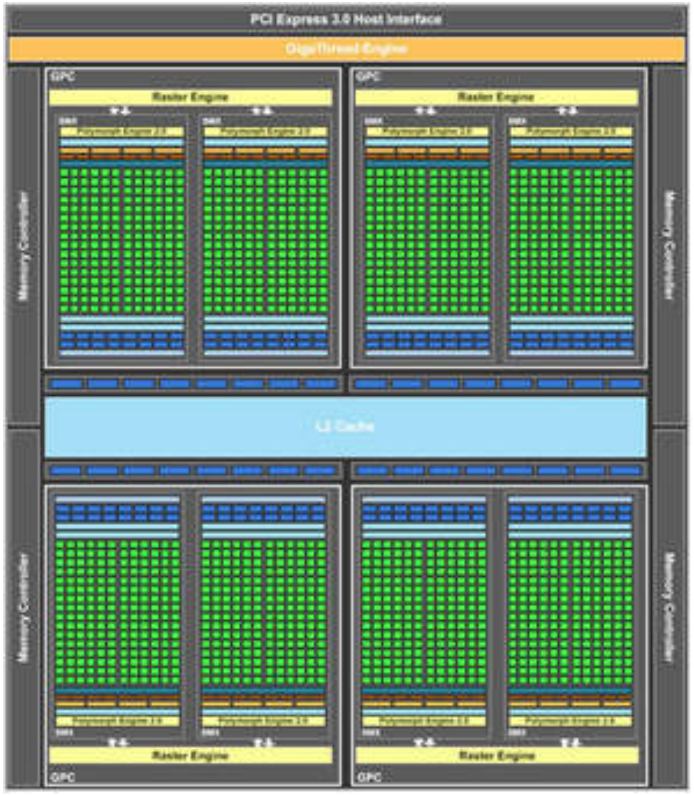
Design of CPUs optimized for **sequential** code and **coarse grained parallelism**:

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

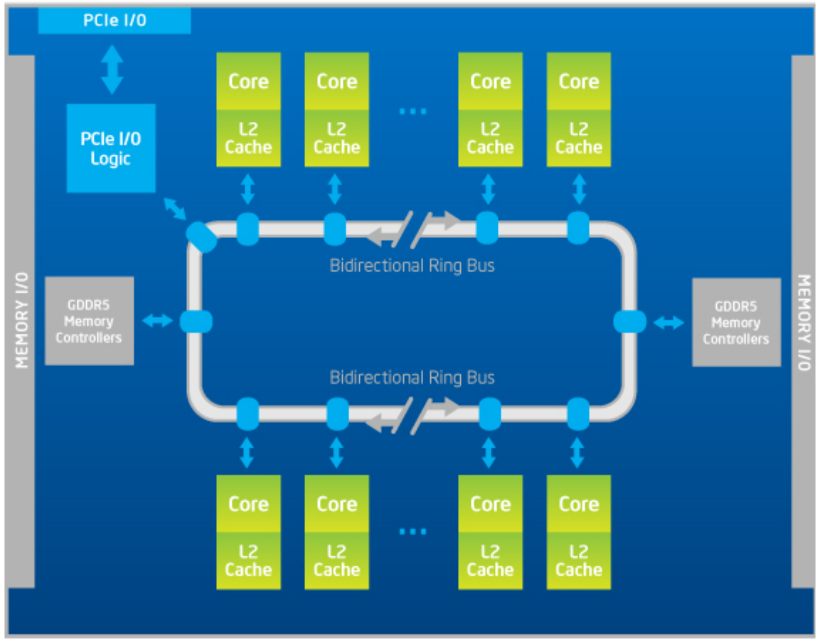
Design of accelerators optimized for *numerically intensive* computation by a **massive fine grained parallelism**:

- 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

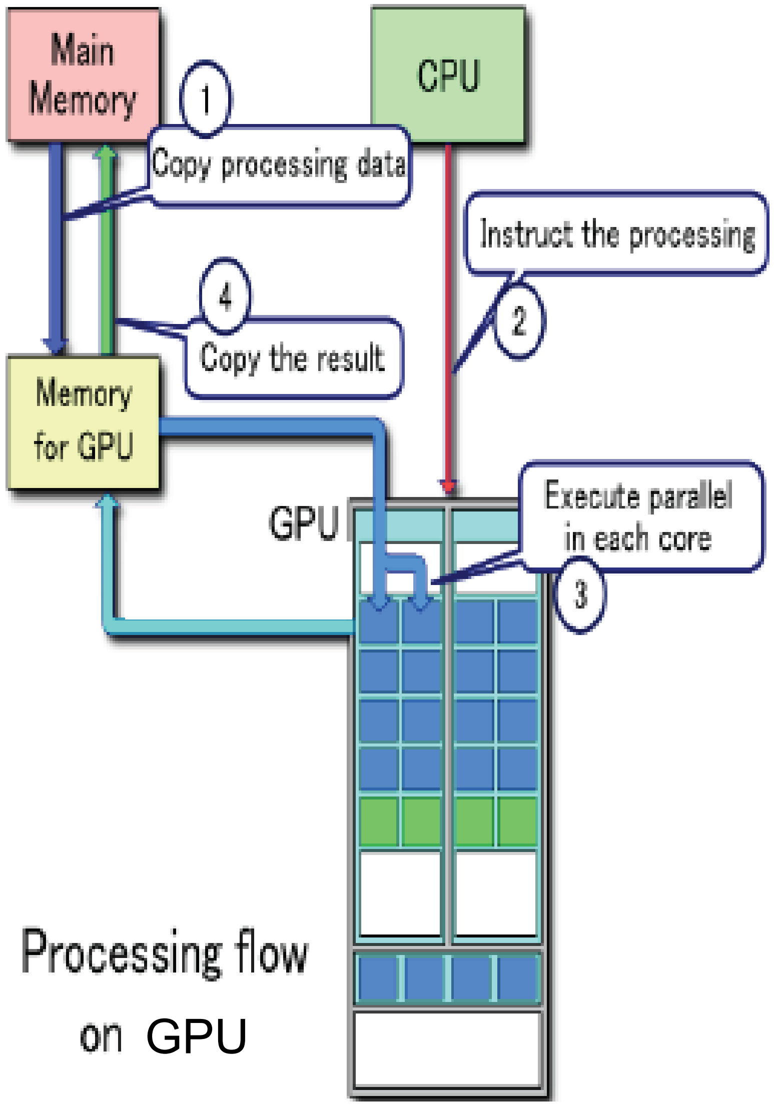
Accelerators – programming model

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

CPU code

```
void increment_cpu(float *a, float b, int
N)
{
 for (int idx = 0; idx<N; idx++)
 a[idx] = a[idx] + b;
}
void main()
{
  .....
 increment_cpu(a, b, 16);
}
```

DEVICE

CUDA code

```
__global__ void increment_gpu(float *a,
float b, int N)
{
 int idx = blockIdx.x * blockDim.x +
threadIdx.x;
 if( idx < N)
 a[idx] = a[idx] + b;
}
```

One thread per iteration!

HOST

```
void main()
{
  ….
  cudaMalloc(da,sizeof(da));
  cudaMemcpy(da,a,N,cudaMemcpyHostToDevice);
  increment_gpu<<<4,4>>>(da,b,16);
  cudaMemcpy(a,da,N,cudaMemcpyDeviceToHost);
  ….
}
```

OpenACC

http://www.openacc-standard.org/

GPU directive based API (corresponds to "OpenMP" for CPU parallel programming).

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop
!$acc parallel loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
  <stuff>
END PROGRAM main
```

- ✓ 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.
- ✓ Can be "combined" with CUDA code.

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.

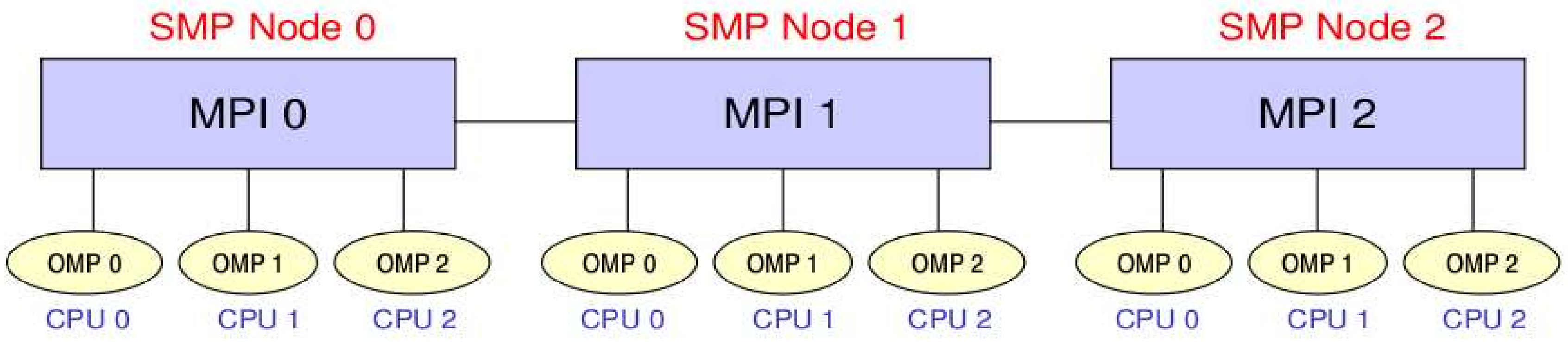
- The **effort** to get codes efficiently running on accelerators is, in general, **big, irrespectively of the programming model** adopted. However portability and maintainability of the code push toward directive based approaches (at the expenses of some performance).

- 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?