Programación Paralela y Distribuida

Cores, Threads and Nodes

Pedro Antonio Varo Herrero
pevahe@gmail.com

Pedro Antonio Varo Herrero

- Estudiante 4º Curso - Universidad de Sevilla
- Grado Ing. Informática –Tecnologías Informáticas, Rama de Computación.
- Investigaciones actuales:
  - Técnicas de paralelización de algoritmos.
  - Evolución de redes complejas.
  - Paralelización en GP-GPU con Cuda y OpenCl.
  - Simulación de fluidos con método SPH y Cuda/OpenCL.
  - Algoritmos de Colonias de Hormigas.

Pedro Varo Herrero @pevahe91

PyCon

Pedro Varo Herrero – pevahe@gmail.com

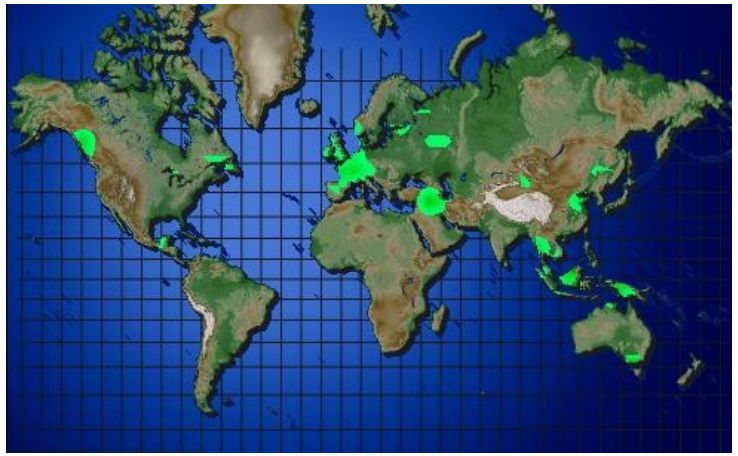
Contenidos

1. Que es la Programación Paralela.
2. Porque paralelizar.
3. Tipos de Paralelismos y arquitecturas.
4. Paradigma de Programación Paralela.
5. Librerías de Python: Cython+OpenMP, MPI4Py, PyCuda, PyOpenCL,

Que es

- Varios procesadores juntos para resolver uno o varios problemas.

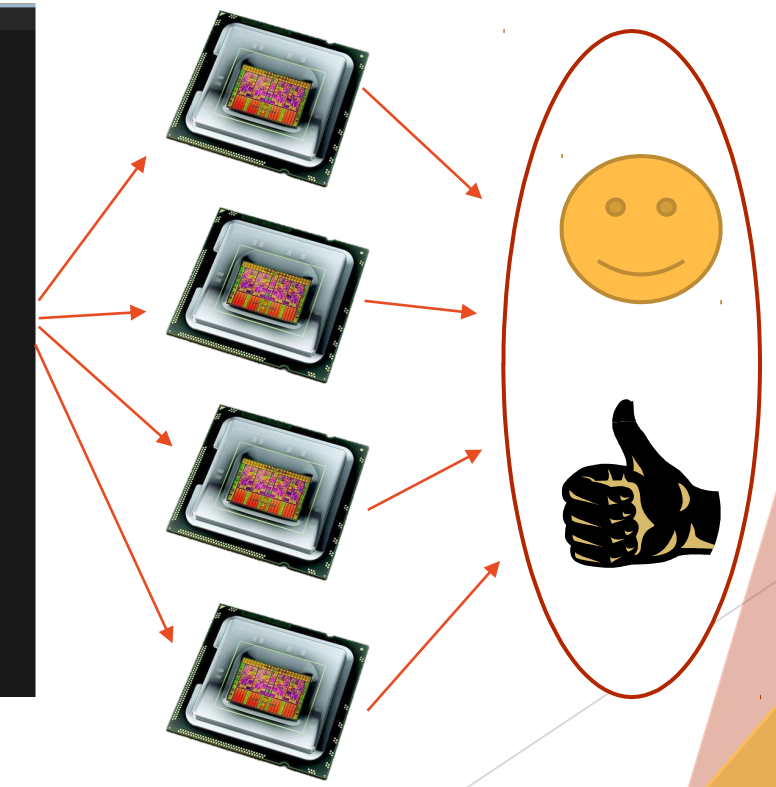
Problems



Code/Algorithm

```
buf = open('x.pdf', 'rb').read()

while True:
    pos1 = buf.find('/Length 1047768')
    if pos1 == -1:
        break
    pos2 = buf.find('endobj', pos1) + 6
    i = pos1
    while i > 0:
        if buf[i:i + 6] == 'endobj':
            pos1 = i + 6
            break
        i -= 1
    buf = buf[:pos1] + buf[pos2:]

while True:
    pos1 = buf.find('/Length 359388')
    if pos1 == -1:
        break
    pos2 = buf.find('endobj', pos1) + 6
    i = pos1
    while i > 0:
        if buf[i:i + 6] == 'endobj':
            pos1 = i + 6
            break
        i -= 1
    buf = buf[:pos1] + buf[pos2:]

open('x2.pdf', 'wb').write(buf)
```

Processesors



Results

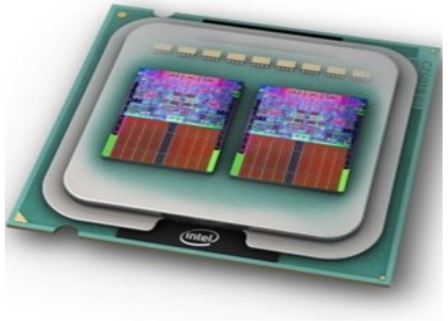
PyCon

Porque paralelizar

PyCon

Porque paralelizar

- Limitaciones físicas de sistemas secuenciales:



Topes frecuencia de reloj

Más Frecuencia -> Más Temperatura y Más Consumo

- **Problemas de alta complejidad computacional:**
  - Simulación de sistemas físicos, biológicos…
  - Volumen de datos con los que operar.
  - Ingeniería, ciencia.

PyCon

Tipos de Paralelismos y arquitecturas.

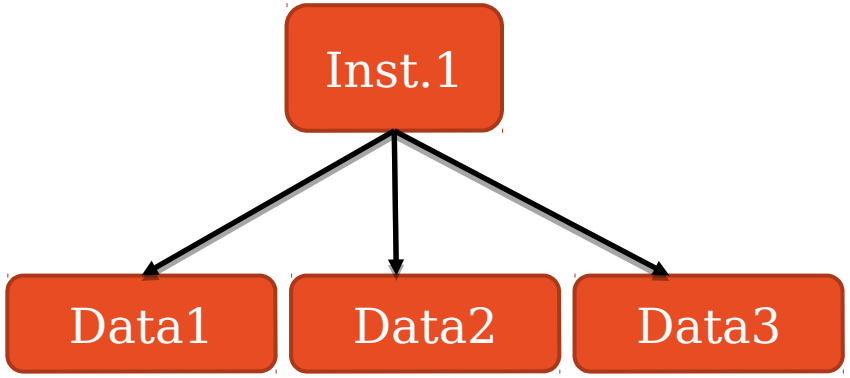
Tipos de Paralelismos y arquitecturas.

- Arquitecturas según instrucciones y datos:
  - Taxonia de Flynn, 1972:

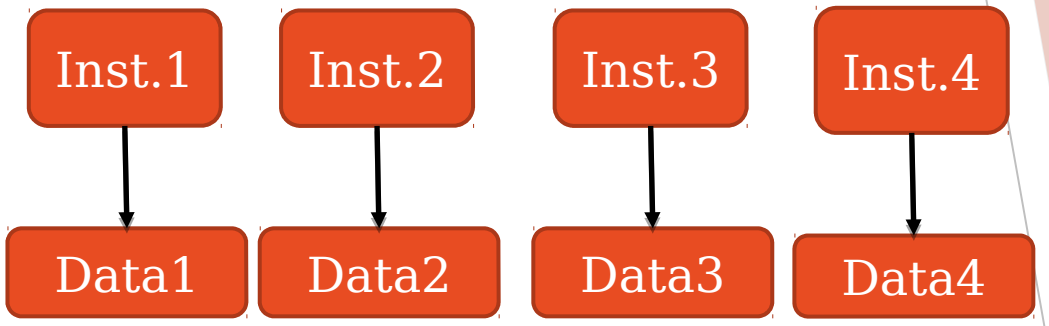
Instrucciones/Datos	Simples	Múltiples
Simples	**Single Instr. Single Data (SISD)**	**Single Instr. Multiple Data (SIMD)**
Múltiples	**Multiple Instr. Single Data (MISD)**	**Multiple Instr. Multiple Data (MIMD)**

- **Arquitecturas Paralelas -> SIMD , MIMD**
- **SISD ->** Antiguas arquitecturas de procesadores secuenciales.
- **MISD ->** distintas instrucciones a un mismo dato.

Single Instr. Multiple Data (SIMD)

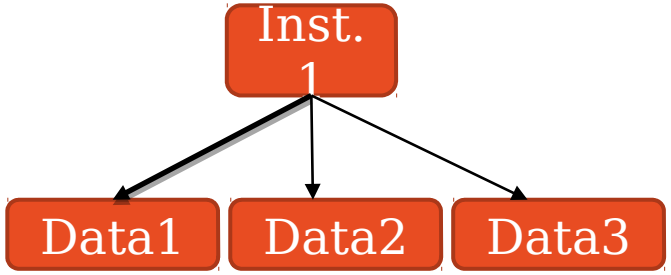


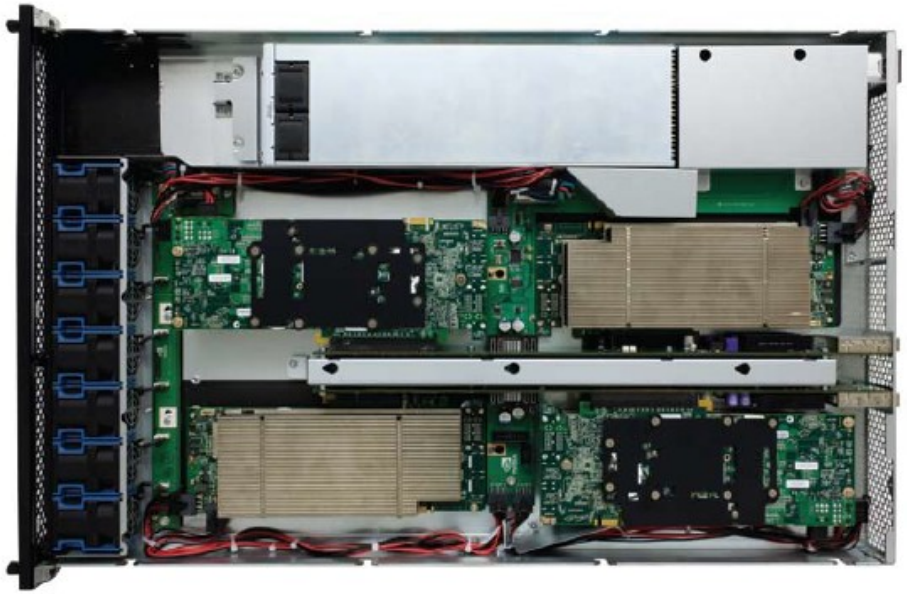
Multiple Instr. Multiple Data (MIMD)



PyCon

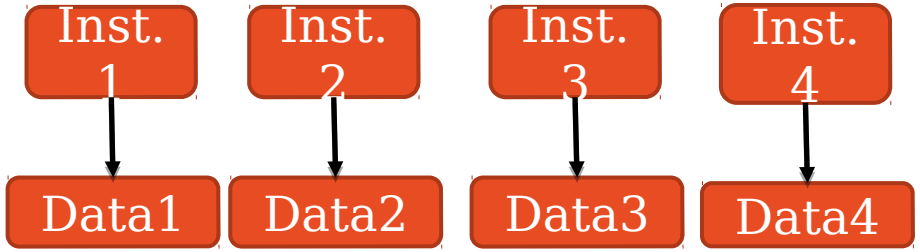
Single Instr. Multiple Data (SIMD)





PyCon

Multiple Instr. Multiple Data (MIMD)



Clúster casero

Supercomputador Marenostrum - Barcelona

PyCon

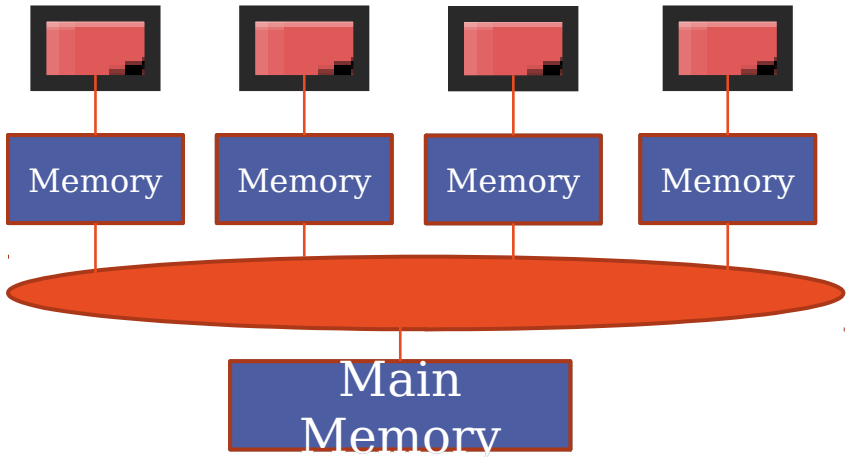
Tipos de Paralelismos y arquitecturas.

- Arquitecturas según distribución de memoria:

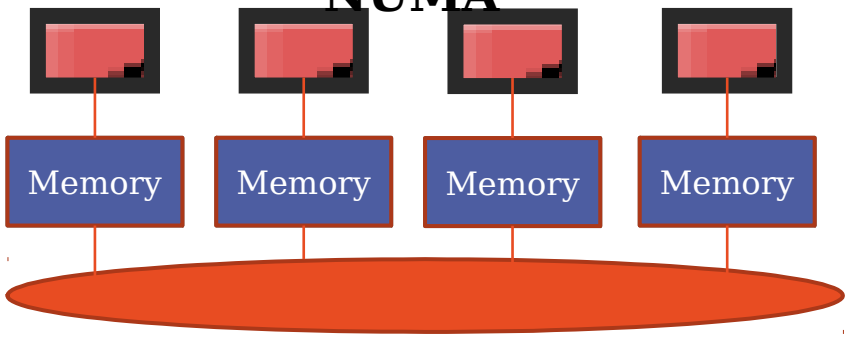
Física/Lógica	Direcciones Memoria compartida	Direcciones Memoria separadas
Memoria compartida	**Uniform Memory Access UMA**	**------**
Memoria Distribuida	**Non-Uniform Memory Access NUMA**	**Memory Passing Message MPM**

PyCon

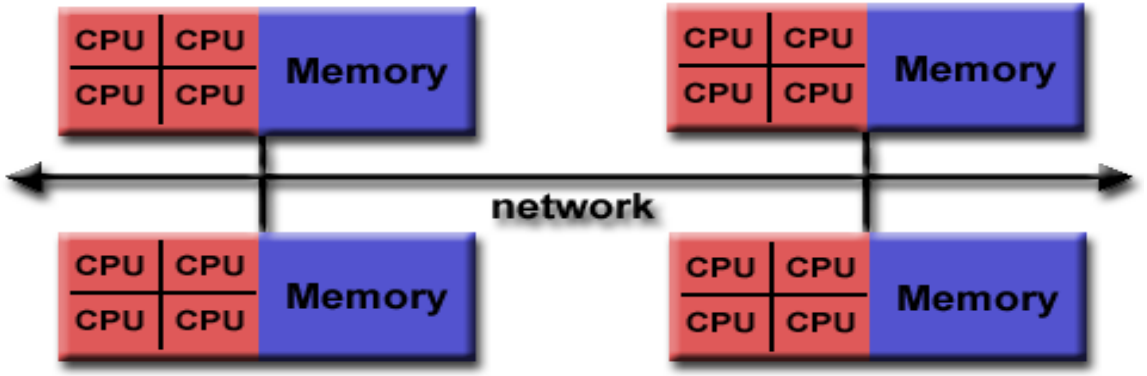
Uniform Memory Access
UMA



Non-Uniform Memory Access
NUMA

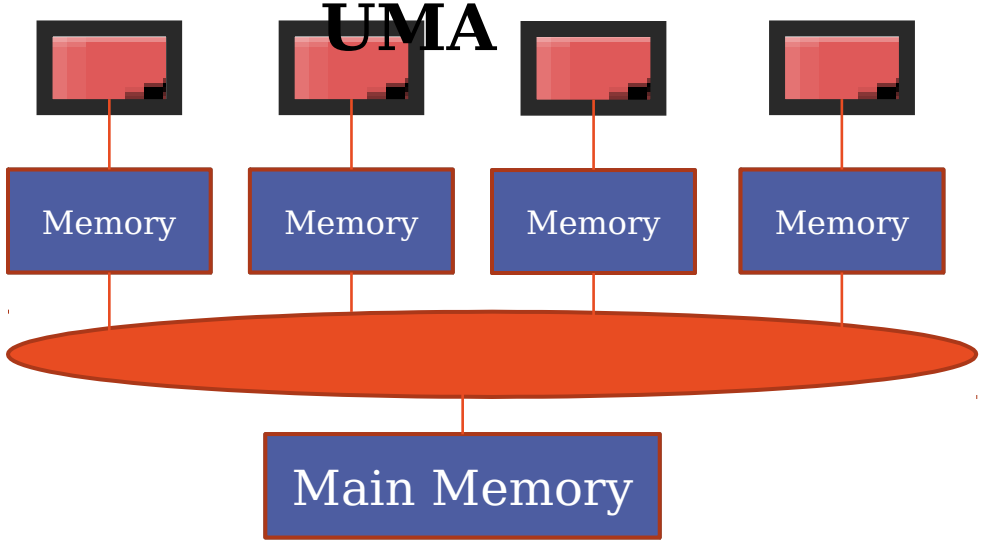


Memory Passing Message
MPM



PyCon

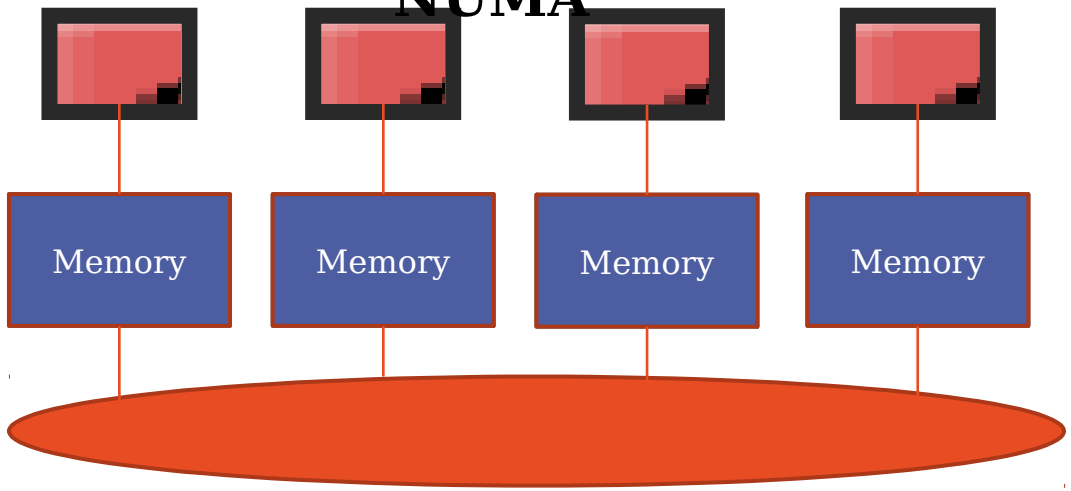
Uniform Memory Access UMA



SPARC Enterprise T5140 server

PyCon

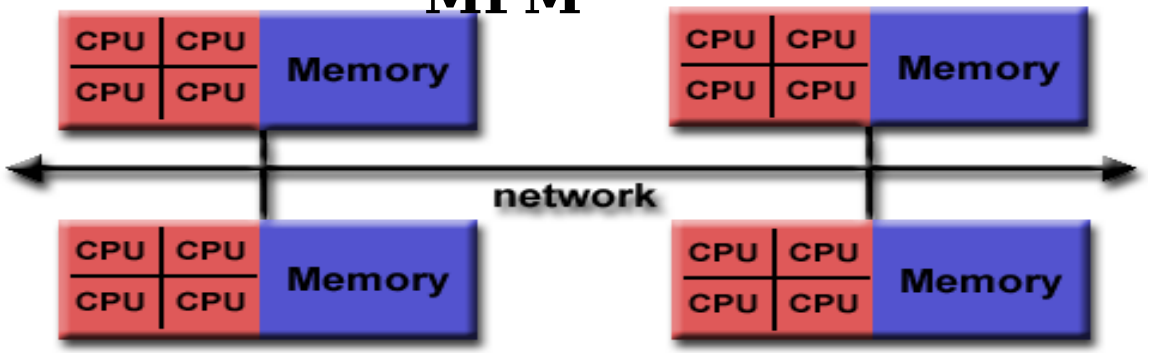
Non-Uniform Memory Access NUMA



Cray CS300

PyCon

Memory Passing Message MPM



Supercomputador Marenostrum - Barcelona

Clúster casero

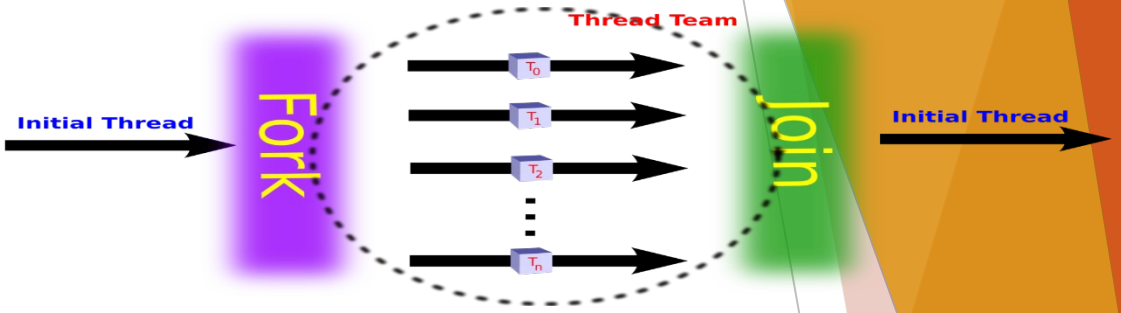
PyCon

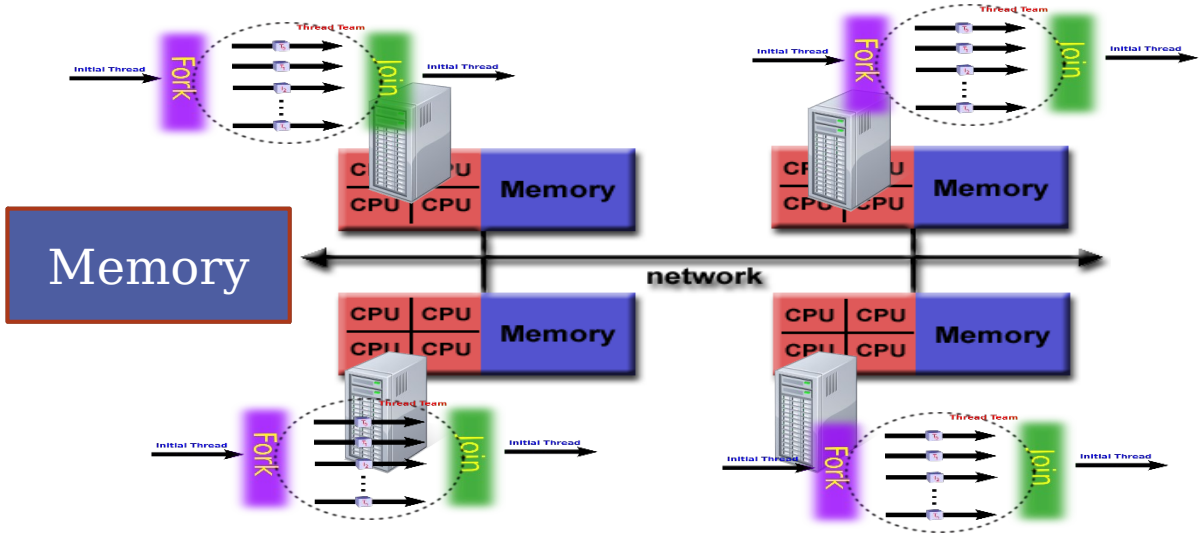
Paradigmas de Programación Paralela

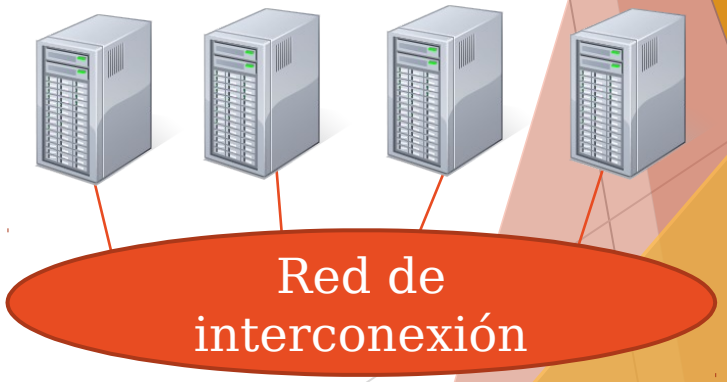
PyCon

Paradigmas de Programación Paralela

- Por manejo de Threads/Tareas.
- Por paso de mensajes.
- Hibrida: Threads + Paso de mensajes.



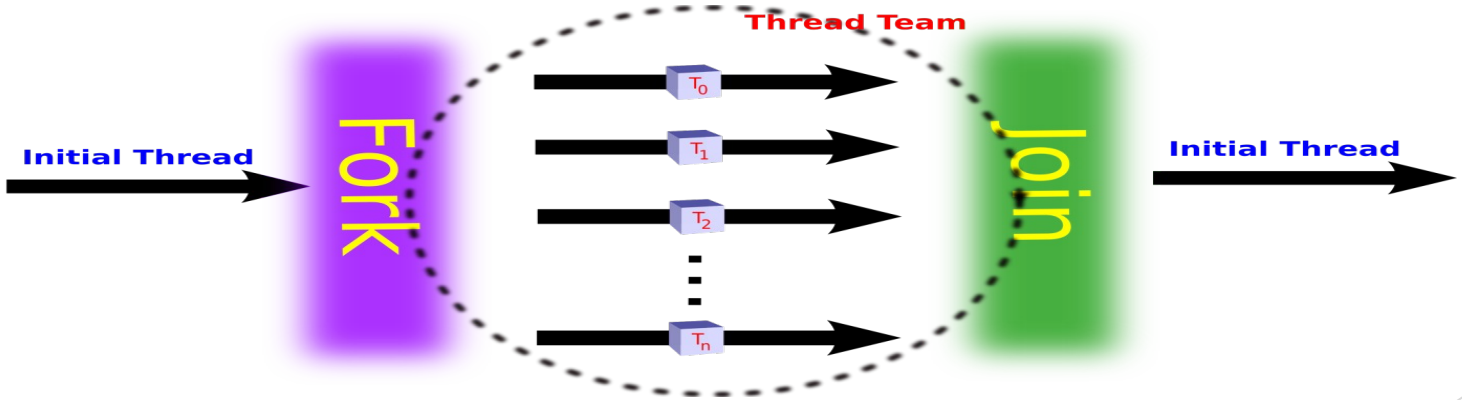




PyCon

Paradigmas de Programación Paralela

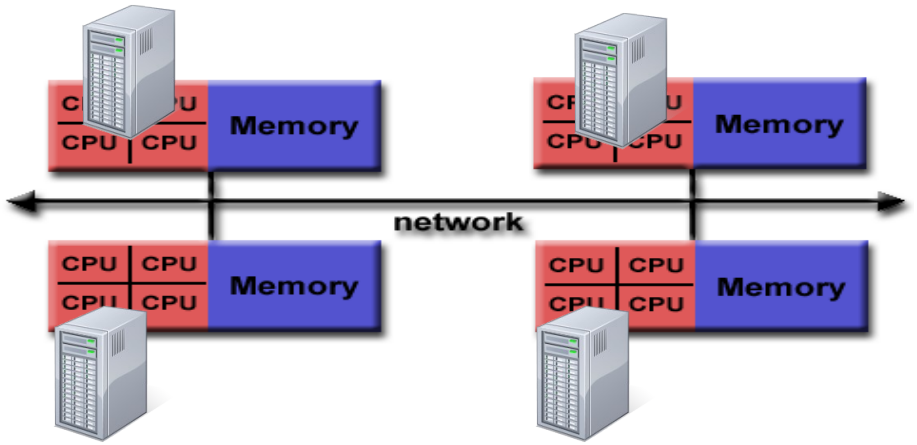
- Por manejo de Threads:
  - Usado con arquitecturas de Memoria compartida.
  - Da comunicación entre threads en un procesador.
  - Estandar : OpenMP (C/C++ ,Fortran) , CUDA, OpenCL.

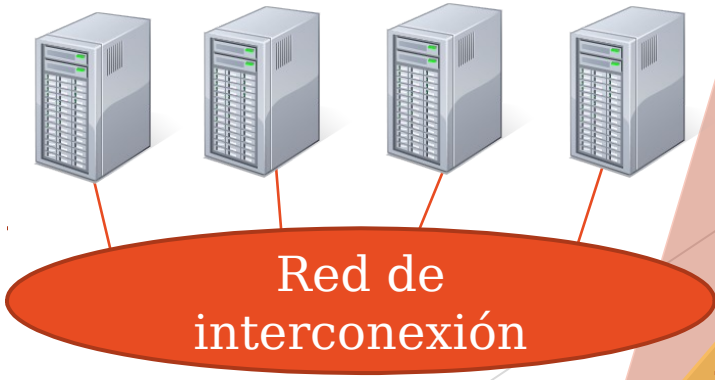


PyCon

Paradigmas de Programación Paralela

- ## Por paso de mensajes:
  - Usado en arquitecturas de memoria distribuida.
  - Da comunicación entre los distintos procesadores/nodos/maquinas del sistema.
  - Se crean distintas tareas, cada uno con su propio espacio de memoria.
  - Los datos entre tareas, se comparten en el paso del mensaje.
  - Código escalable.
  - Estandar: MPI(C/C++,Fortran).

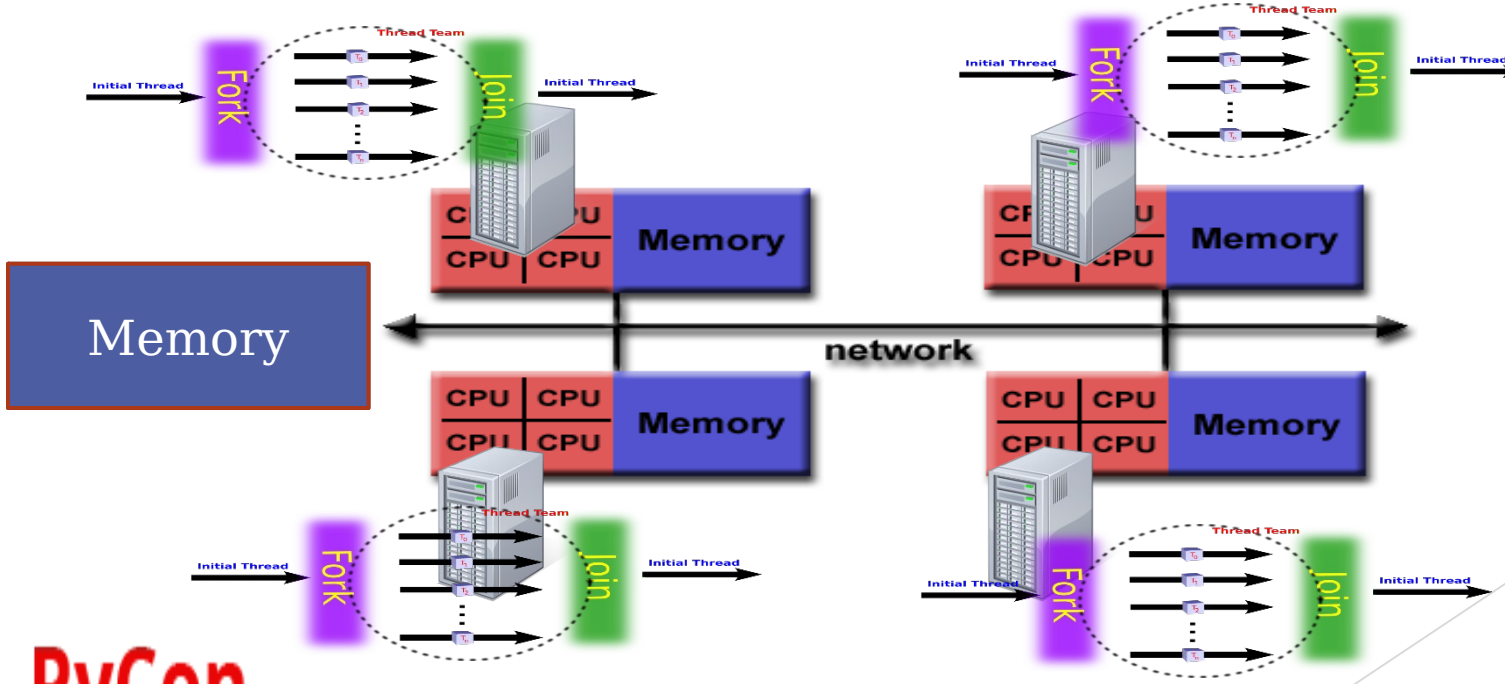




PyCon

Paradigmas de Programación Paralela

- Híbrida:
  - Usa ambas arquitecturas.
  - Para llegar desde nivel de nodo/maquina a nivel de hilo.
  - Usa ambos: OpenMP/CUDA/OpenCL+MPI



PyCon

Librerías

Librerías

Multiprocessing

MPI4Py

PyCUDA

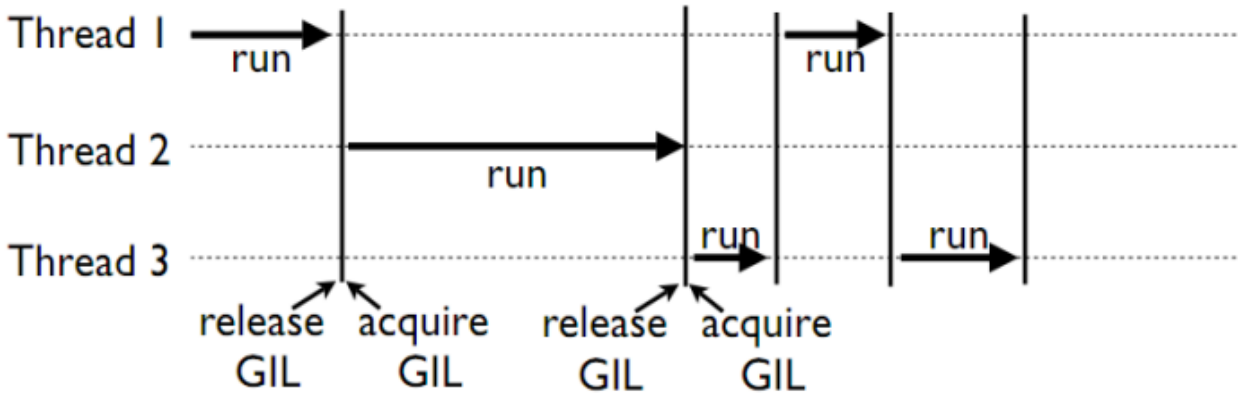
PyOpenCL

“Estándares” de librerías de programación paralela

Multiprocessing

GIL – Global Interpreter Lock

- En Python la ejecución de Threads está controlada por el GIL(Global Interpreter Lock).
- No permite que se ejecute mas de un Thread a la vez.



Sys setcheckinterval

Python Summer-School 2011 – UK University of St Andrews
Francesc Alted

https://python.g-node.org/python-summerschool-2011/_media/materials/parallel/parallelcython.pdf

Multiprocessing

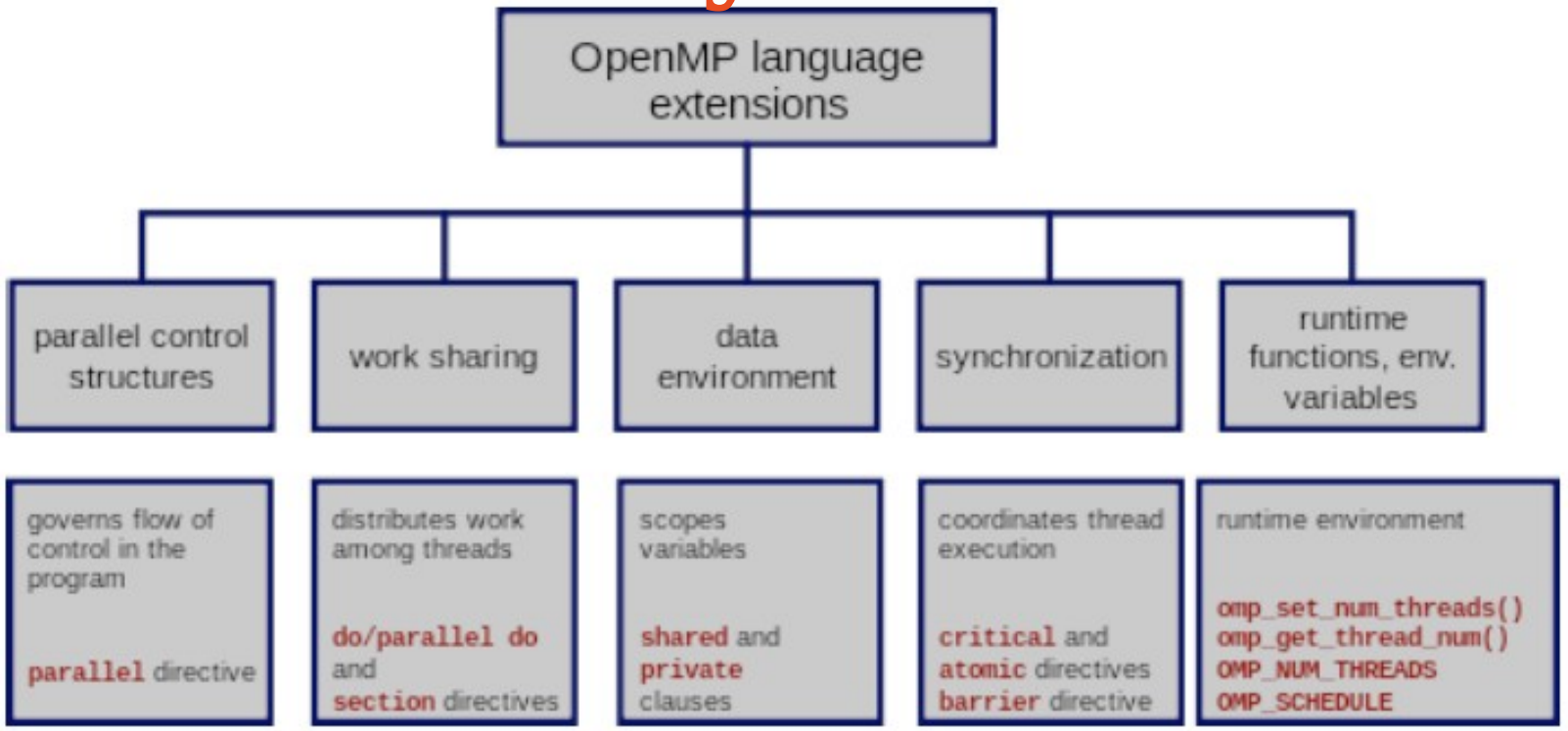
y

- Para ello Python nos ofrece el módulo Multiprocessing, basado en la ejecución de distintos procesos en distintos cores.
- Cython extensión que permite escribir funciónes/módulos Python con variaciones y compilarlo.
- Este código compilado , luego podemos llamarlo desde Python.

OpenMP

PyOpenCL

Python



PyCon

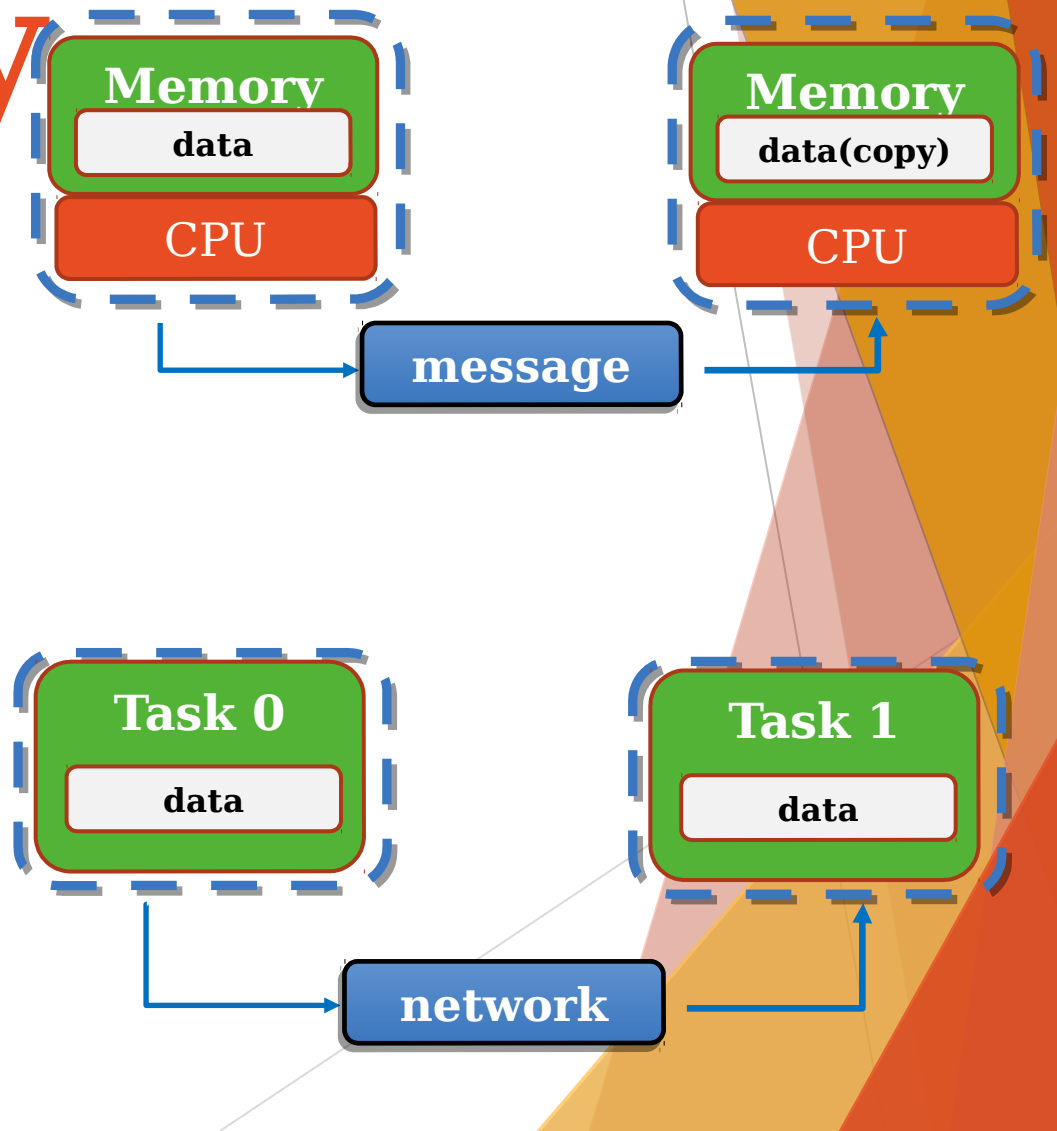
- Python con añadidos.

- Ahora podemos crearnos nuestros hilos y ejecutarlos.
- Podemos usar OpenMp importándolo en Cython.
- Y con esto nos saltamos el GIL.

- Que nos hace falta:
  - Compilador de C/C++
  - Python 2.6-3.3.2 (32 bits)
  - Setup tools: https://pypi.python.org/pypi/setuptools
  - Pypi: https://pypi.python.org/pypi
  - Añadimos variable de entorno: C:\Python33\Scripts
  - Ejecutamos **pip install cython** o **easy_install cython**

MPI4Py

- **PyPar**
  - Proyecto de la Universidad Nacional de Australia.
  - https://code.google.com/p/pypar/
- **pyMPI**
  - Proyecto hecho por investigadores del Lawrence Livermore National Laboratory , California
  - http://pympi.sourceforge.net/index.html
- **MPI4Py**
  - Proyecto de Lisandro Dalcin, basado en MPI-1/2
  - Implementa la mayoría de funciones de MPI
  - http://mpi4py.scipy.org/



PyCon

MPI4Py

MPI4

	PyPar	MPI4Py	pyMPIP	SciPy.MPI
MPI_Send	✓	✓	✓	✓
MPI_Recv	✓	✓	✓	
MPI_Sendrecv		✓	✓	✓
MPI_Isend		✓	✓	✓
MPI_Irecv		✓	✓	✓
MPI_Bcast	✓	✓	✓	✓
MPI_Reduce	✓	✓	✓	✓
MPI_Allreduce		✓	✓	✓
MPI_Gather	✓	✓	✓	
MPI_Allgather		✓	✓	
MPI_Scatter	✓	✓	✓	
MPI_Alltoall		✓		

	C	PyPar	MPI4Py	pyMPIP	SciPy.MPI
Latency	8	25	14	133	23
Bandwith	967.004	898.949	944.475	150.901	508.972

- MPI4Py implementa la mayoría de rutinas.
- PyPar, MPI4Py y SciPy dan mejores resultados.
- Con PyPar tenemos menos control.
- Si sabemos MPI, MP4Py es trivial.

Comparativas de Trabajo de Fin de Master Universidad de Oslo por:

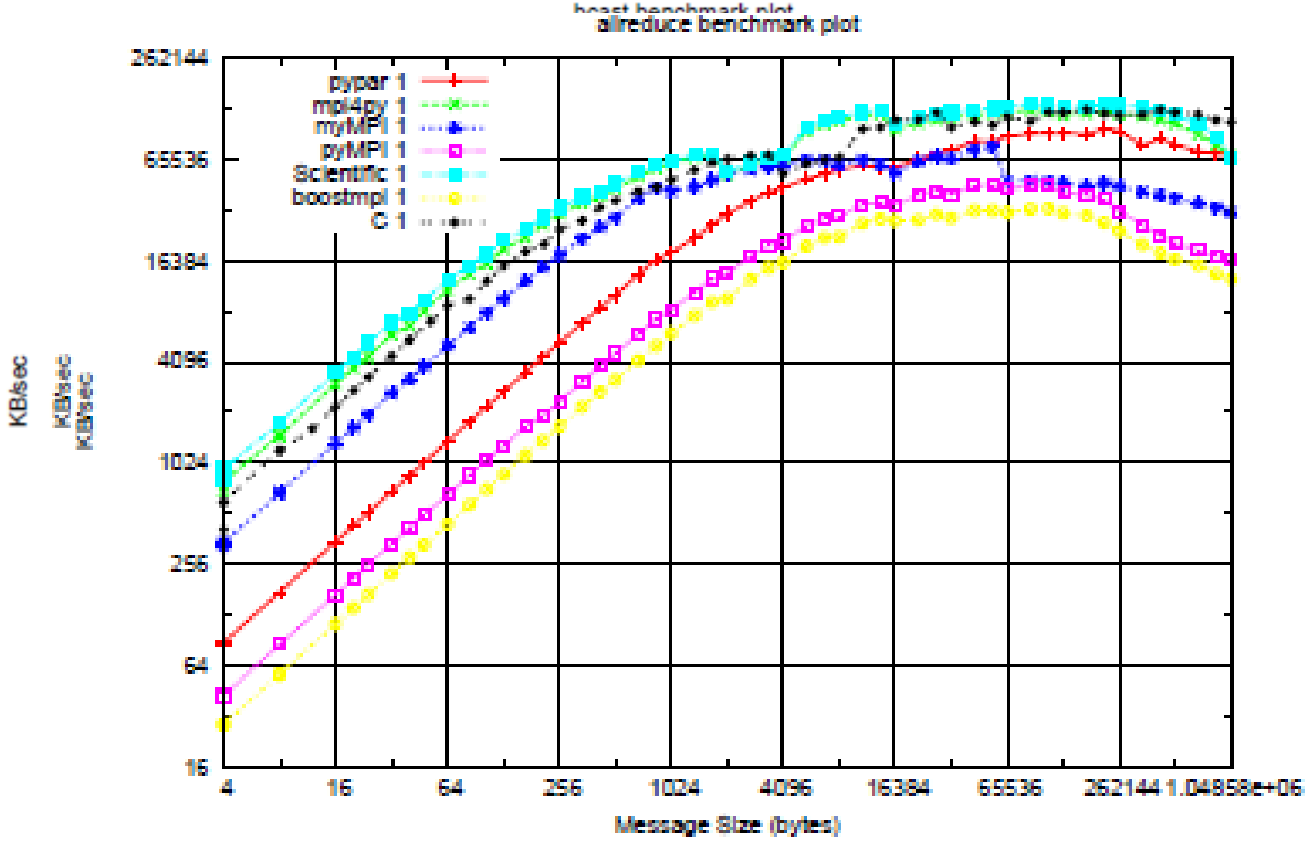
WENJING LIN – A comparison of existing python modules of MPI

Master i Anvendt matematikk og mekanikk

(Master de Matemáticas y Mecánica aplicada)

PyCon

MPI4



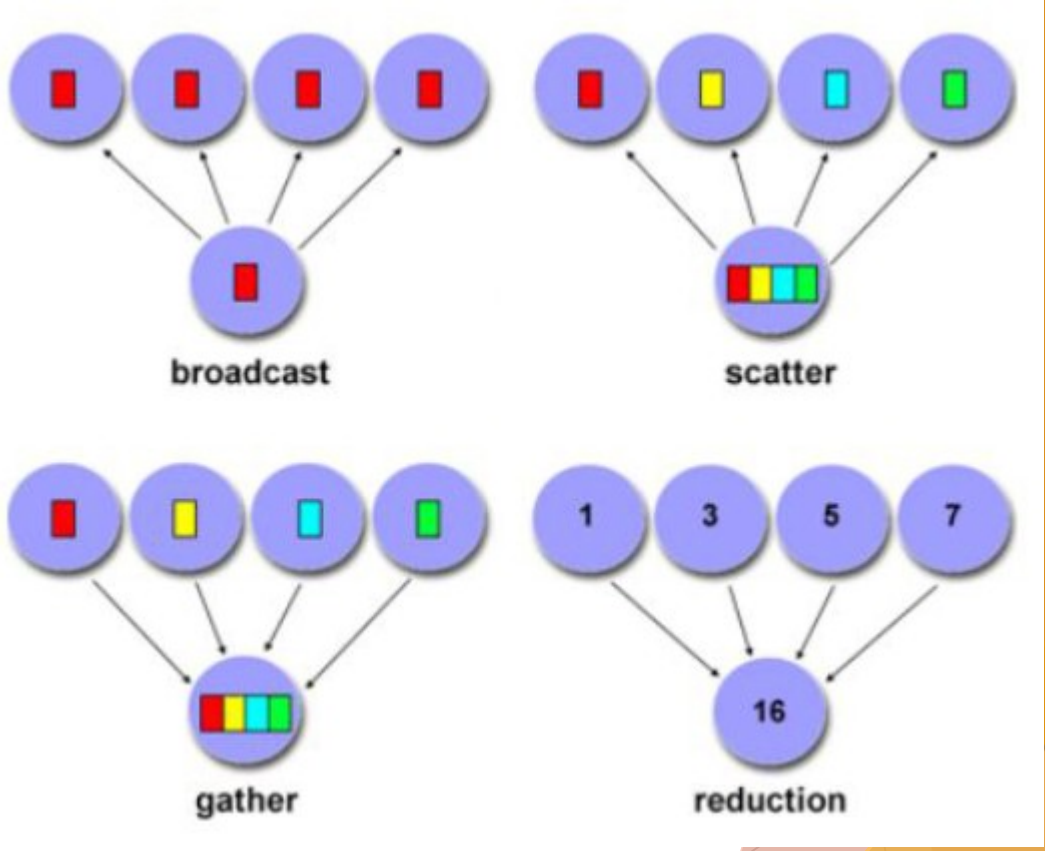
Imágenes de Trabajo de Fin de Master Universidad de Oslo por:

WENJING LIN

Master i Anvendt matematikk og mekanikk

MPI4

- MPI_Init -> Siempre y única la primera ve
- MPI_Finalize -> Última rutina
- MPI_Comm_size -> Nº procesos de un gr
- MPI_Comm_rank -> Devuelve rango(id).
- MPI_Send -> Enviar mensajes
- MPI_Recv -> Recibir mensajes



Funciones colectivas

PyCon

MPI4Py

- Que nos hace falta:
  - Una versión de MPI, por ejemplo **OpenMPI** (http://www.open-mpi.org/software/ompi/v1.6/).
  - Compilador de C/C++
  - Python 2.6-3.3.2 (32 bits)
  - Setup tools: https://pypi.python.org/pypi/setuptools
  - Pypi: https://pypi.python.org/pypi
  - Ejecutamos comando : **pip instal mpi4py** o **easy_install mpi4py**

MPI4Py

- Cálculo de Pi

Pi.py

```
from mpi4py import MPI
import numpy
import sys

print "Spawning MPI processes"
comm =
MPI.COMM_SELF.Spawn(sys.executable
          , args=['CalcPi.py'] ,maxprocs=8)

N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE],
          op=MPI.SUM, root=MPI.ROOT)

print "Calculated value of PI is: %f16" %PI
```

CalcPi.py

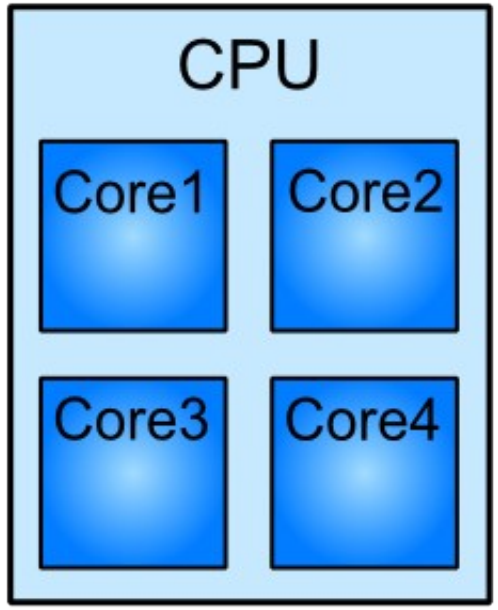
```
from mpi4py import MPI
import numpy

comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

N = numpy.array(0, dtype='i')
comm.Bcast([N, MPI.INT], root=0)
h = 1.0 / N; s = 0.0
for i in range(rank, N, size):
      x = h * (i + 0.5)
      s += 4.0 / (1.0 + x**2)
       PI = numpy.array(s * h, dtype='d')
comm.Reduce([PI, MPI.DOUBLE], None,
      op=MPI.SUM, root=0)
print "Disconnecting from rank %d"%rank
comm.Barrier()

comm.Disconnect()
```

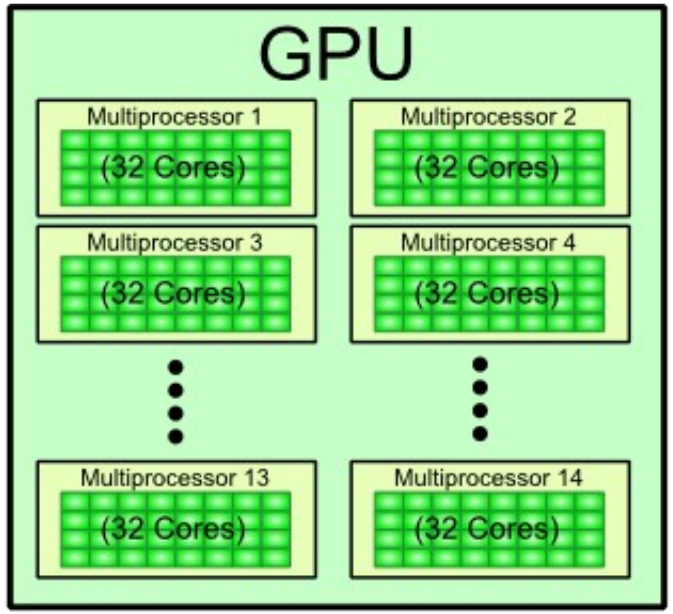

PyCUDA
PyOpenCL


PyCon

NVIDIA GPU Computing CUDA

CPU/GPU Architecture Comparison

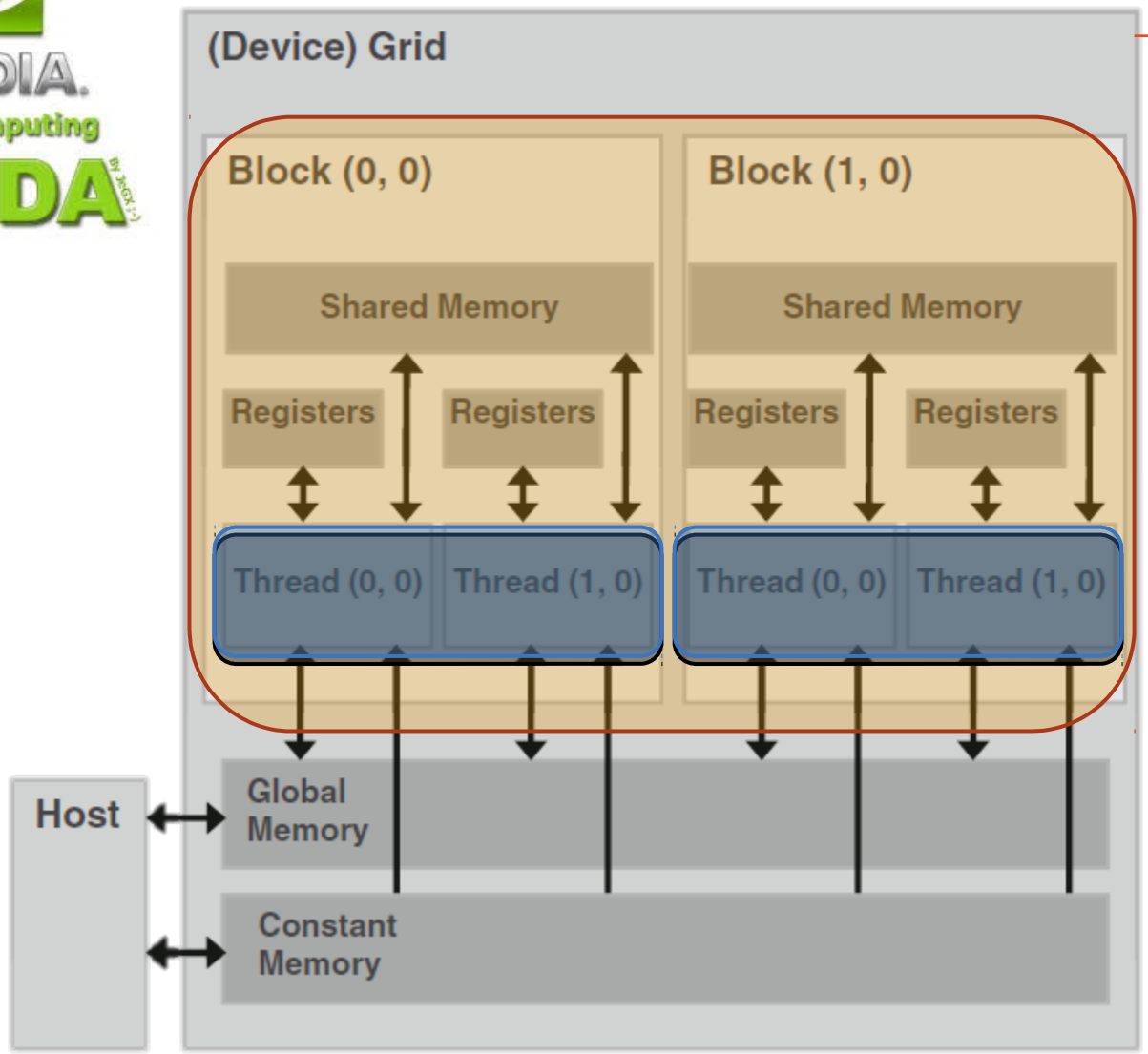




Cores complejos, con muchas instrucciones.

Cores simples, con instrucciones limitadas.

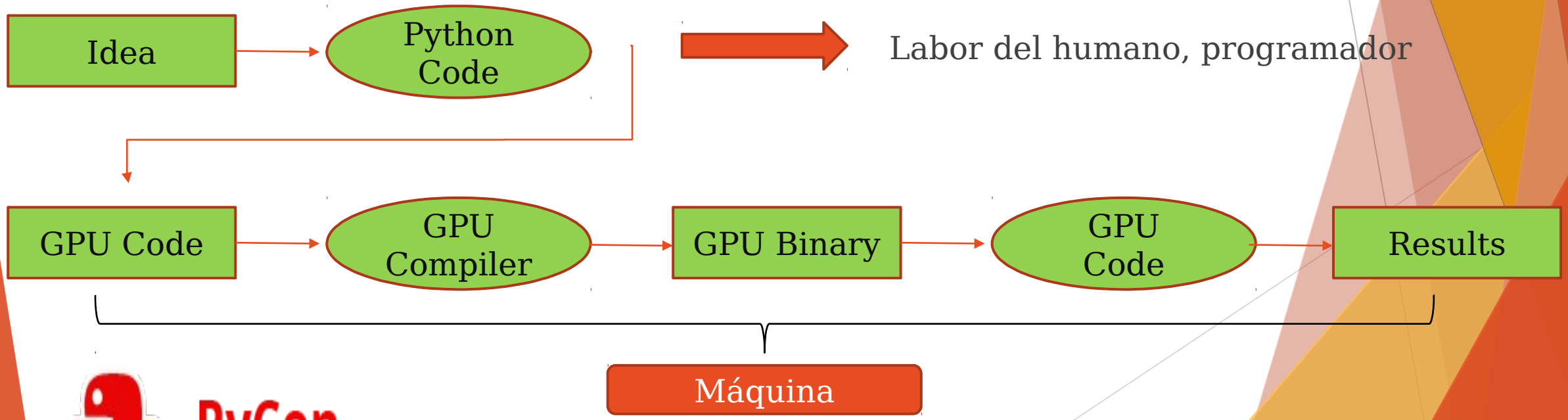
PyCon



- Grid de bloques de hilos.
- Bloques de hilos.
- Hilos máximos 512 por bloque.

PyCUDA

- CUDA(***C**ompute **U**nified **D**evice **A**rchitecture*) creado por NVIDIA.
- PyCUDA proyecto de Andreas Klöckner.
- Wrapper de CUDA -> PyCUDA.
- http://mathema.tician.de/software/pycuda/

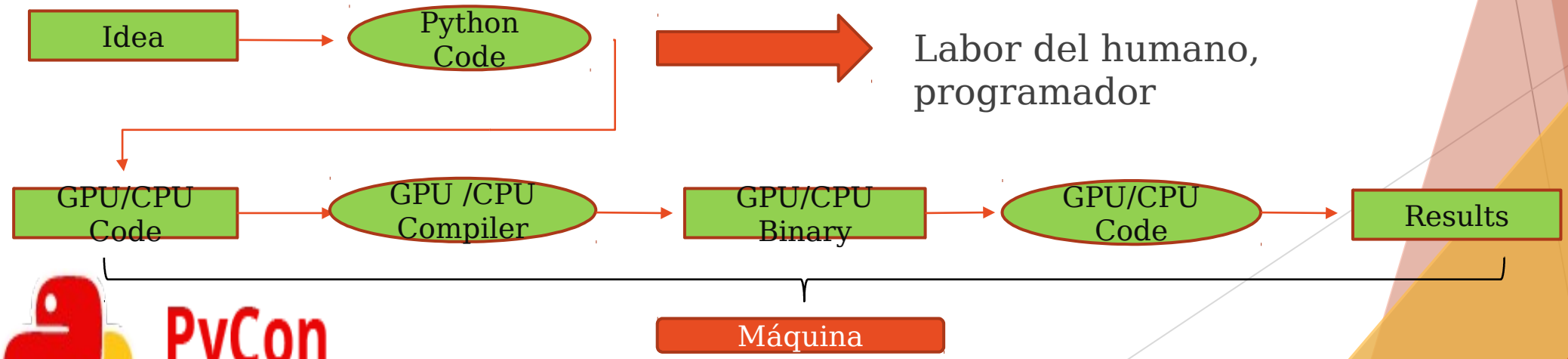


PyCon

OpenCL

PyOpenCL

- OpenCL(Open Computing Language), creado por Apple y desarrollada en conjunto AMD, Intel, IBM y Nvidia.
- Propuesta al Grupo Khronos para convertirla en estandar.
- Api para computación paralela en CPU y GPU.
- Wrapper de OpenCL -> PyOpenCL.
- http://mathema.tician.de/software/pyopencl/

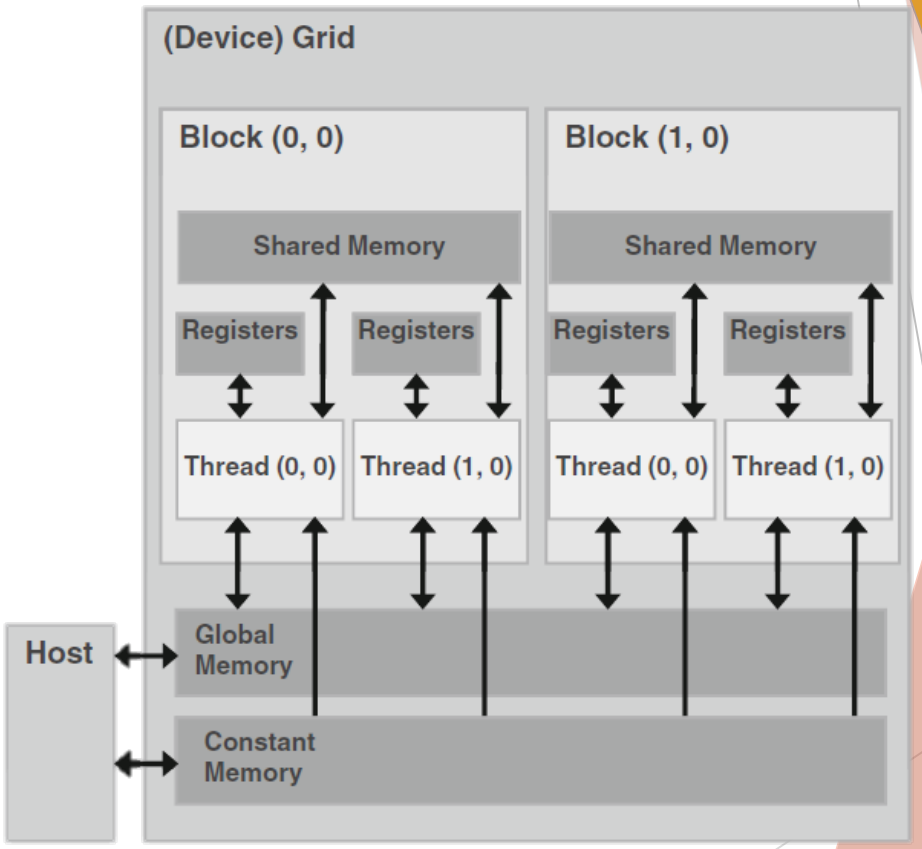


PyCon

- Que nos hace falta:
  - CUDA/OpenCL:
  - Python 3.3 (64 bits)
  - PyCUDA: http://www.lfd.uci.edu/~gohlke/pythonlibs/
  - Boost.Python: http://www.lfd.uci.edu/~gohlke/pythonlibs/#pycuda
  - NumPy: http://www.lfd.uci.edu/~gohlke/pythonlibs/
  - CUDA: https://developer.nvidia.com/cuda-downloads
  - Setuptools: https://pypi.python.org/pypi/setuptools
  - PyTools: http://pypi.python.org/pypi/pytools
  - Visual C++ 2010

PyCUDA PyOpenCL

- “Pasos a seguir”:
  - 1. Inicializar Memoria en GPU
  - 2. Configurar Grid
  - 3. Lanzar Kernel
    - 3.1 Calcular ID del Hilo.
    - 3.2 Acceder a datos y cálculo.
  - 4. Traer Resultado

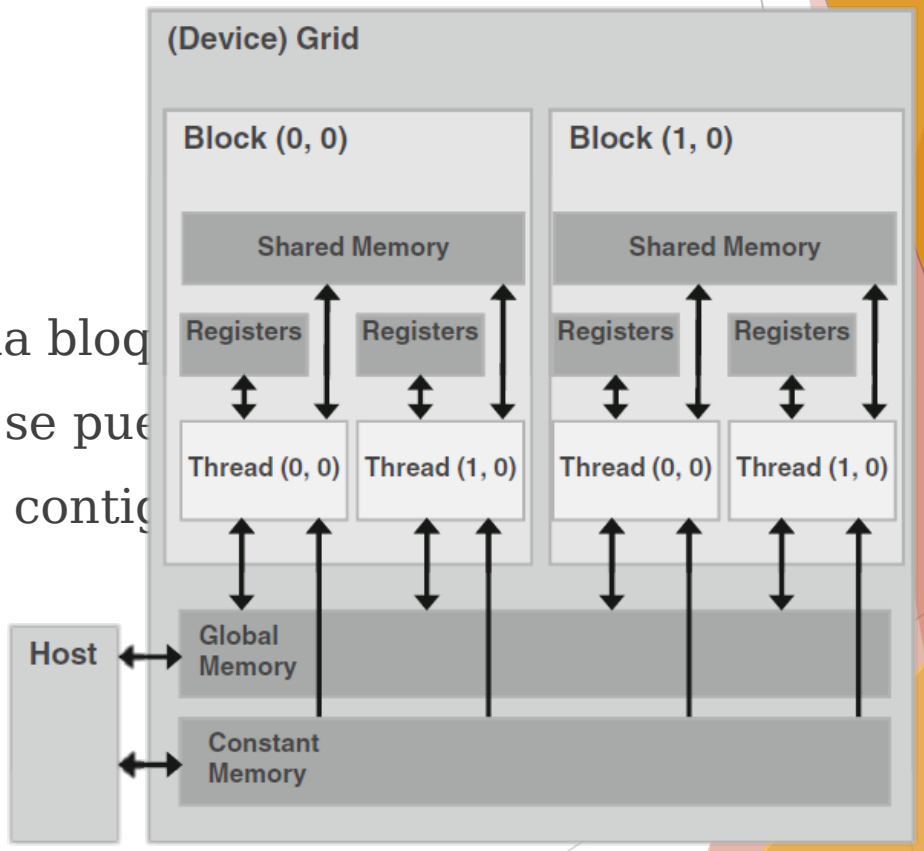


PyCon

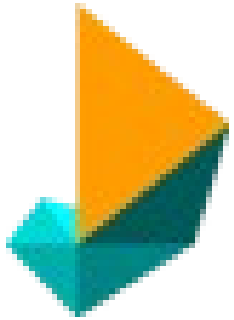
PyCUDA PyOpenCL

- Consideraciones de Rendimiento:

- 1.- Lanzar cuantos más hilos mejor.
- 2.- Mantener el SIMD/SIMT dentro de cada bloq
- 3.- Usar memoria compartida siempre que se pue
- 4.- Acceder "bien" a memoria global, datos contig



PyCon



DEAP

Distributed Evolutionary Algorithms in Python

- Nos permite diseñar fácilmente un algoritmo evolutivo.
- Tiene 4 algoritmos para usar(algorithms) o podemos hacer el nuestro (creator).
- Muy fácil de implementar y paralelizar.
- Paralelización:
  - Antes utilizaba DTM(Distribution Task Manager), basada en MPI.
  - Ahora usa Scoop.
- SCOOP (Scalable Concurrent Operation in Python):
  - Aprovecha las distintas maquinas dentro de una red.
  - Balanceador de cargas incorporado.

Scalable Concurrent Operation in Python

- Usa ØMQ en vez de MPI.

Muchas Gracias

Pedro Varo Herero

@pevahe91