www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Parallel Programming

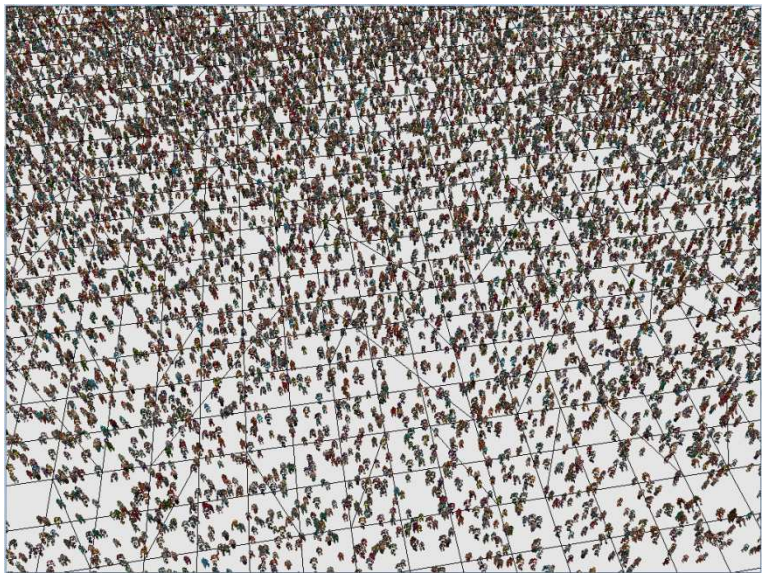
Isaac Rudomin
BSC

Instructor: Dr. Isaac Juan Rudomín Goldberg

Born: January 3, 1959, México D.F.

Address:
Barcelona Supercomputing Center
Torre Girona
c/ Jordi Girona, 31
08034 Barcelona (Spain)

Email:
rudomin.isaac@gmail.com
isaac.rudomin@bsc.es



Studies:
1990 PhD Computer Science, University of Pennsylvania
1987 MSE Computer Science, University of Pennsylvania
1984 Maestría en Ciencias de la Computación, IIMAS UNAM
1981 Licenciatura en Matemáticas, UAM Iztapalapa

Employment:
- March 2012-, Senior Researcher Barcelona
- December 1990-Jan 2012, Supercomputing Center (BSC) Professor-Researcher ITESM-CEM:

Interests:
- Human, facial, hair animation.
- Crowd modeling, simulation animation and render.
- GPU compute.
- High Performance Computing
- Mixed reality,
- Natural interfaces

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

BSC-CNS (Barcelona Supercomputing Center – Centro Nacional de Supercomputación)

BSC-CNS objectives:

- R&D in Computer Sciences, Life Sciences and Earth Sciences
- Supercomputing support to external research

MareNostrum III

- Peak Performance of 1 Petaflops
- 48,448 Intel SandyBridge-EP E5-2670 cores at 2.6 GHz (3,028 compute nodes)
- 94.625 TB of main memory (32 GB/node)
- 1.9 PB of disk storage
- Interconnection networks:
  - Infiniband
  - Gigabit Ethernet
- Operating System: Linux - SuSe Distribution

Minotauro

Cluster with 128 Bull B505 blades each:

- **2 Intel E5649 (6-Core) processor at 2.53 GHz**
- **2 M2090 NVIDIA GPU Cards**
- **24 GB of Main memory**
- **Peak Performance: 185.78 Tflops**
- **250 GB SSD (Solid State Disk) as local storage**
- **2 Infiniband QDR (40 Gbit each) to a non-blocking network**
- **RedHat Linux**
- **14 links of 10 GbitEth to connect to BSC GPFS Storage**

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Abacus:

- ABACUS-CINVESTAV is a World Class Space for Science and Technology Specialized in Applied Mathematics and High Performance Computing of the “Centro de Investigación y de Estudios Avanzados del IPN” (CINVESTAV).
  - The Cinvestav was created in 1961 by presidential decree as a public agency with legal personality and its own assets. The Cinvestav has twenty-eight research departments that are distributed by the nine campuses throughout Mexico.
  - The ABACUS-CINVESTAV project is an initiative of COMECYT, CONACYT and CINVESTAV formally launched in October 2011.
  - The ABACUS-CINVESTAV project was conceived as a forward-looking and frontier knowledge development proposal for the strengthening and reinforcement of two fundamental aspects applied mathematics and high performance computing.
  - ABACUS-CINVESTAV enrolls prestigious national and international researchers and a very considerable number of post doctorate and graduate students from various regions of the country and the world in an interdisciplinary and inter-institutional scheme. ABACUS-CINVESTAV implements lines of research in strategic areas of scientific and technological knowledge with emphasis in Applied Mathematics and High Performance Computing.

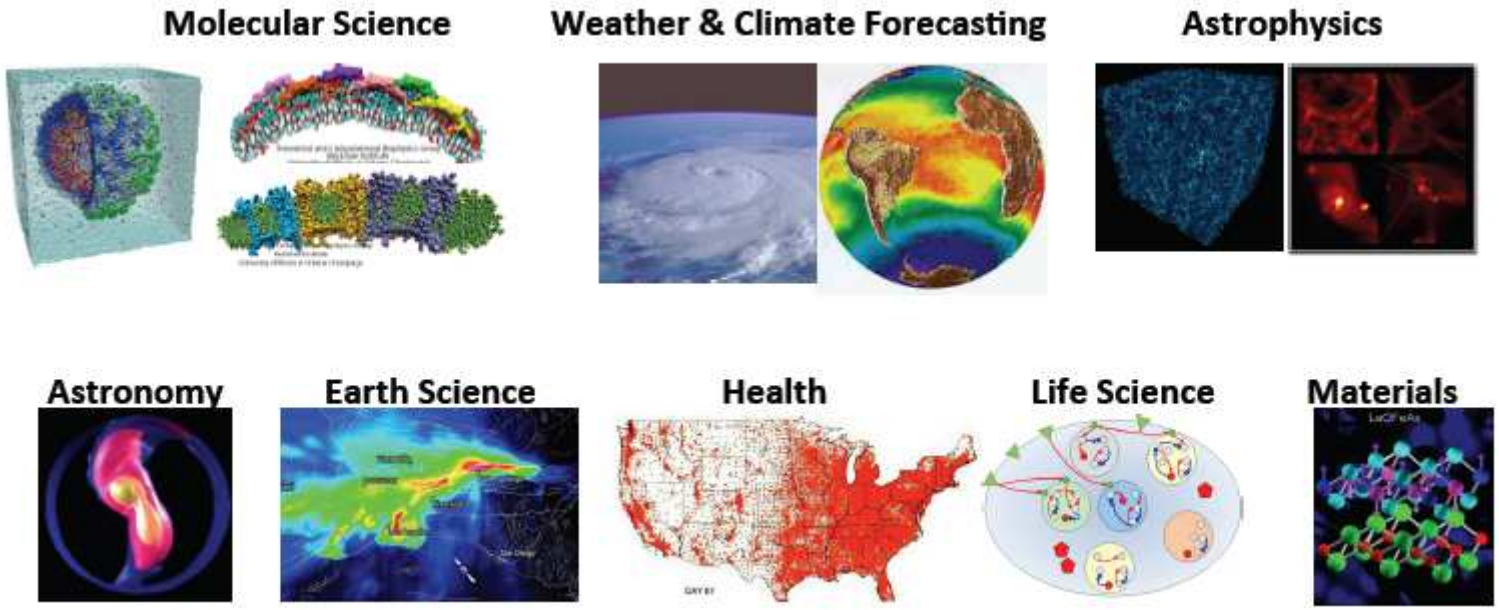
- IBM PureFlex1
- 4 x240 nodes, each with: * 1 Sandy Bridge E5-2690 de 8C @ 2.9Ghz Processor

  * 96GB RAM, * 146 GB HD Raid 1, * 2 10Gb ports
- 2 iDataplex nodes, each with: * 2 Intel E5-2603 4C 1.8Ghz 10Mb Cache Processors

  * 16Gb RAM, * 2 NVDIA Tesla K20, * 2 1 Gb Ports,

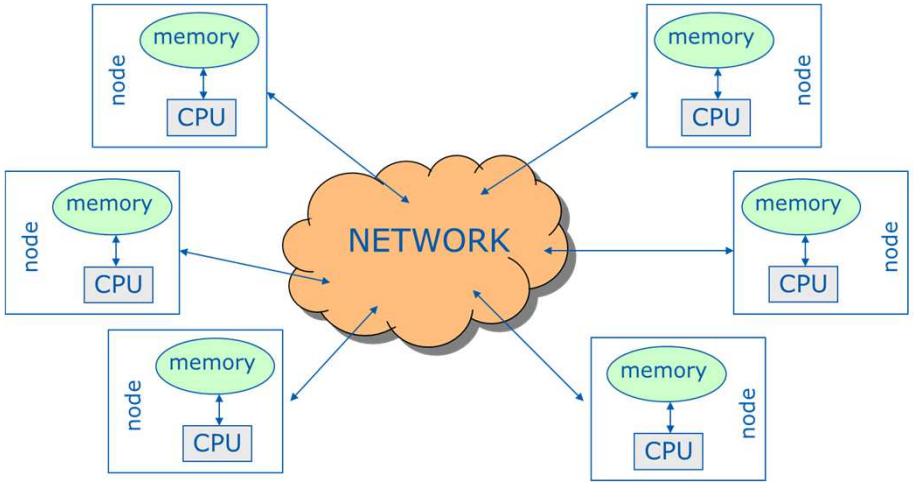
Why parallel programming?

- Solve larger problems
- Run memory demanding codes
- Solve problems with greater speed

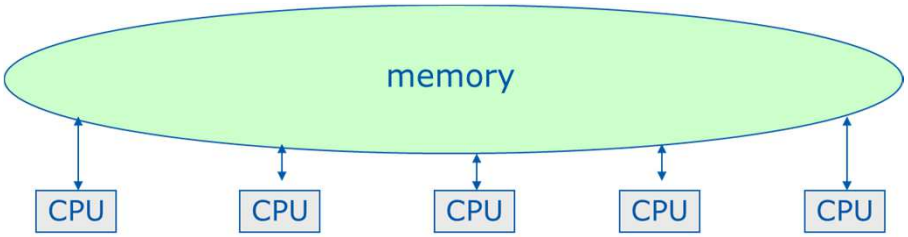


Modern Parallel Architectures

- Two basic architectural schemes:
  - **Distributed Memory**

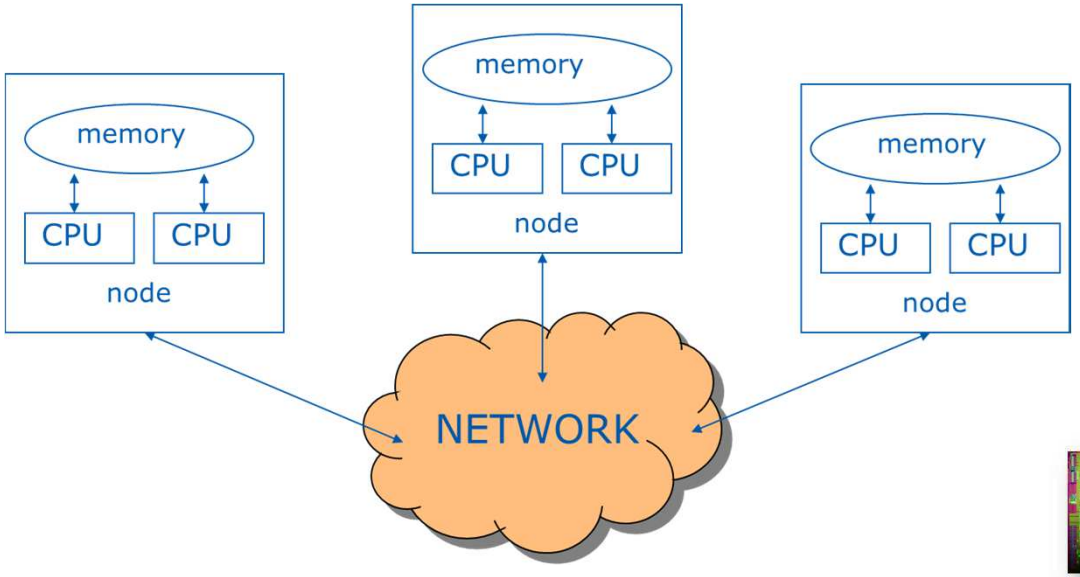


  - **Shared Memory**

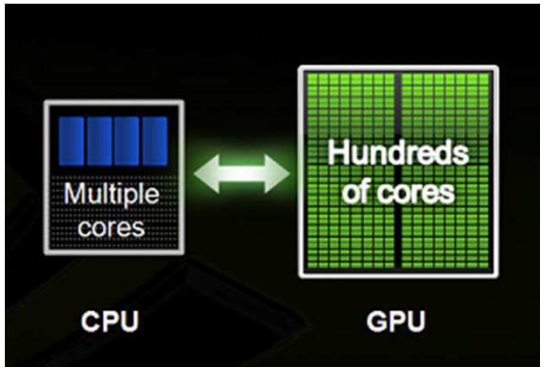


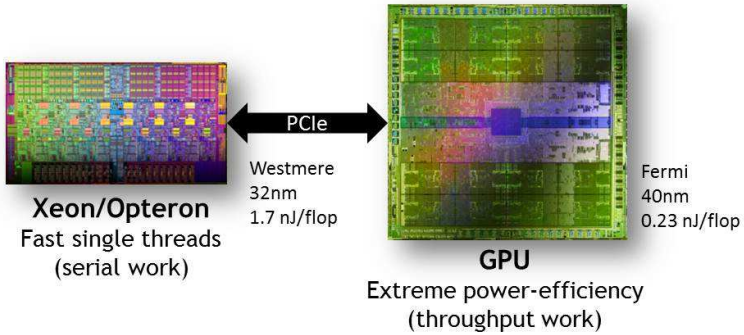
Modern Parallel Architectures

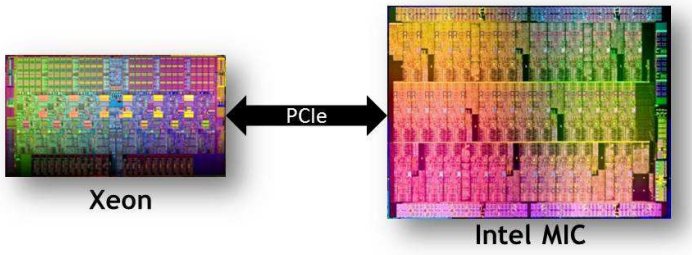
- Now most computers have a mixed architecture



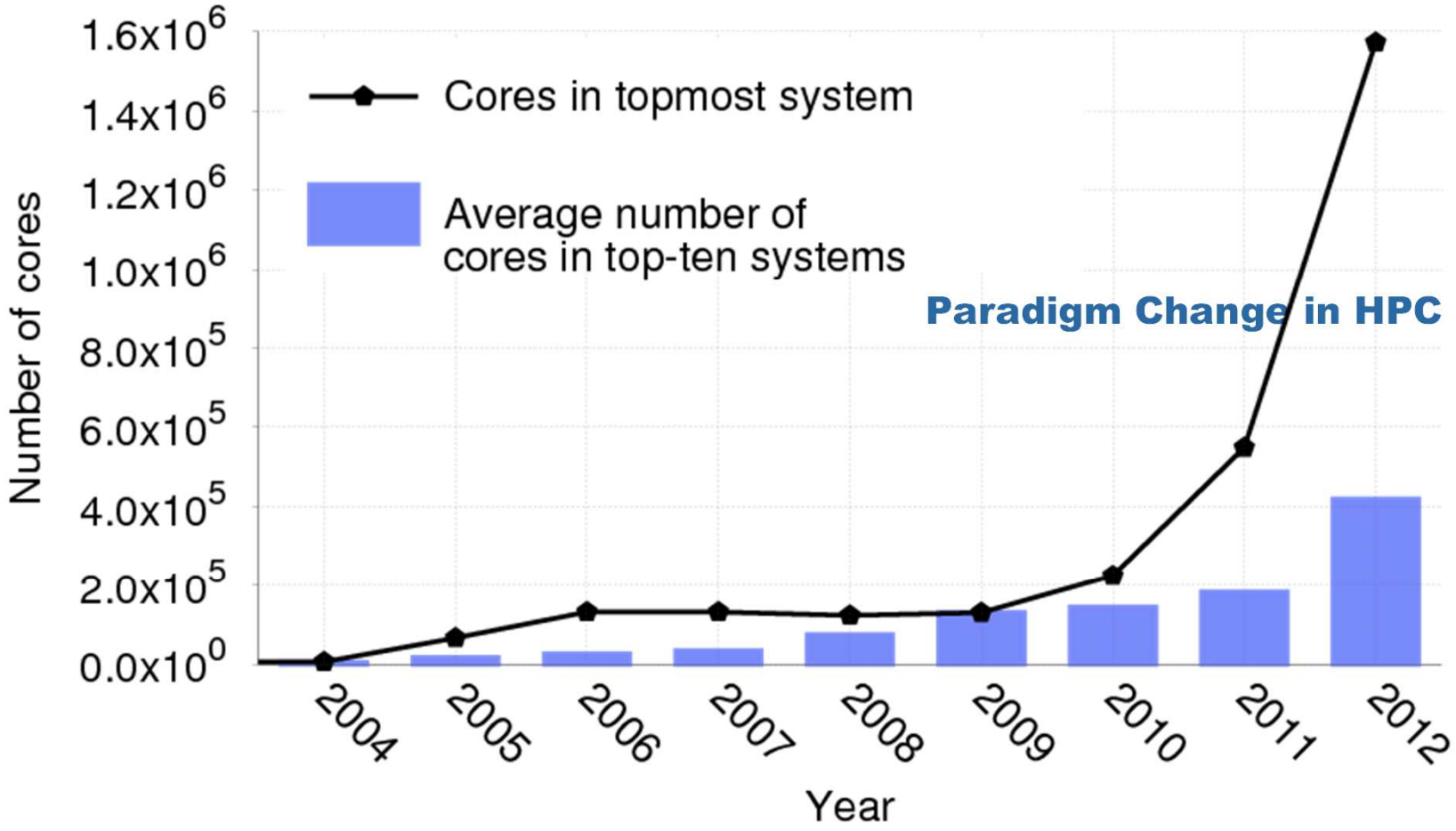
  - + accelerators -> hybrid architectures







Top500



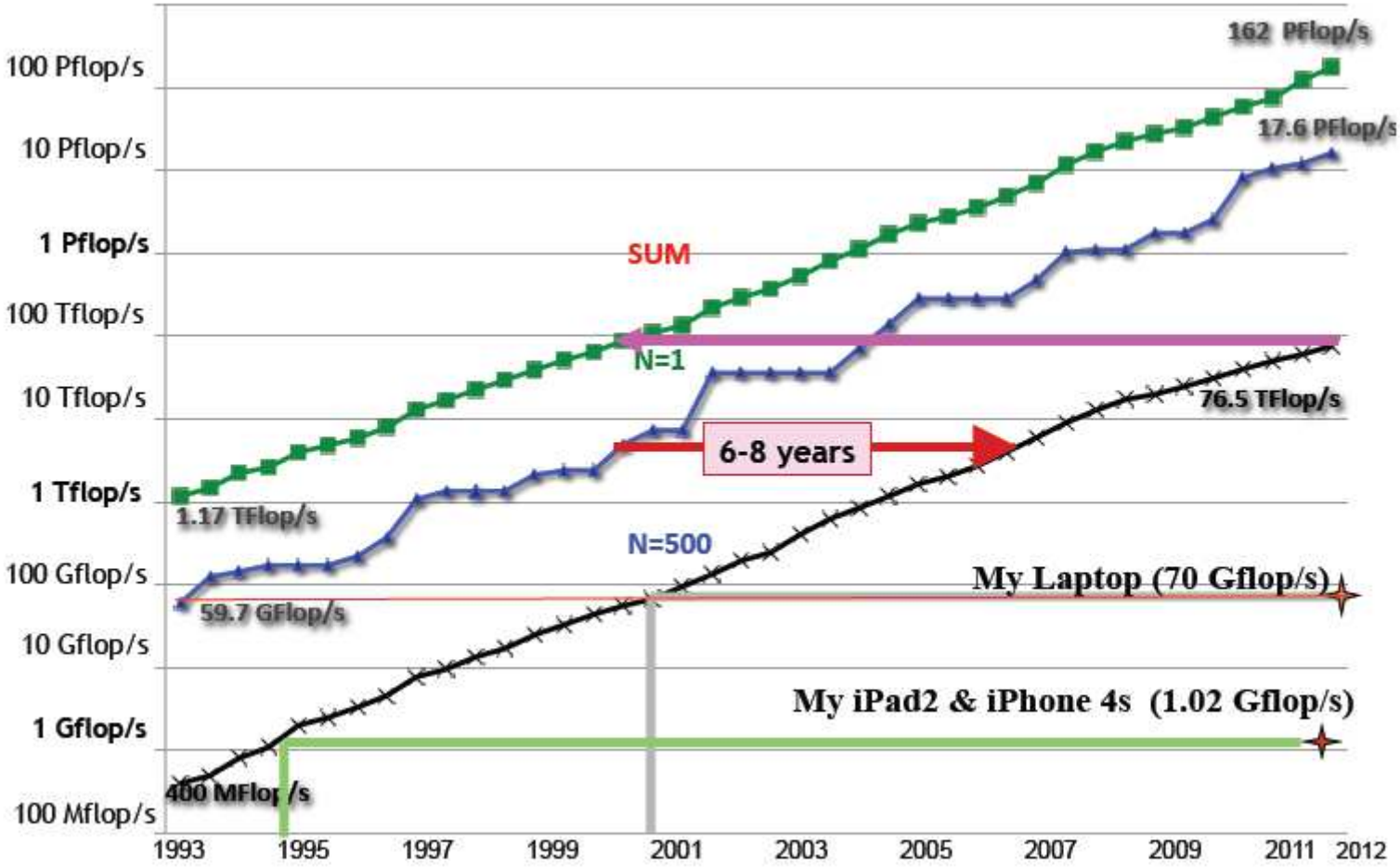
What about applications?

- http://www.top500.org/

Titan (currently #1)

- A Cray XK7 system containing 18,688 nodes, each with
  - 16-core AMD Opteron 6274 processor
  - NVIDIA Tesla K20 GPU accelerator.
- Titan also has more than 700 terabytes of memory.
- The combination of central processing units (CPUs), the traditional foundation of high-performance computers, and more recent GPUs will allow Titan to occupy the same space as its Jaguar predecessor while using only marginally more electricity.

Performance



Top 10 2012

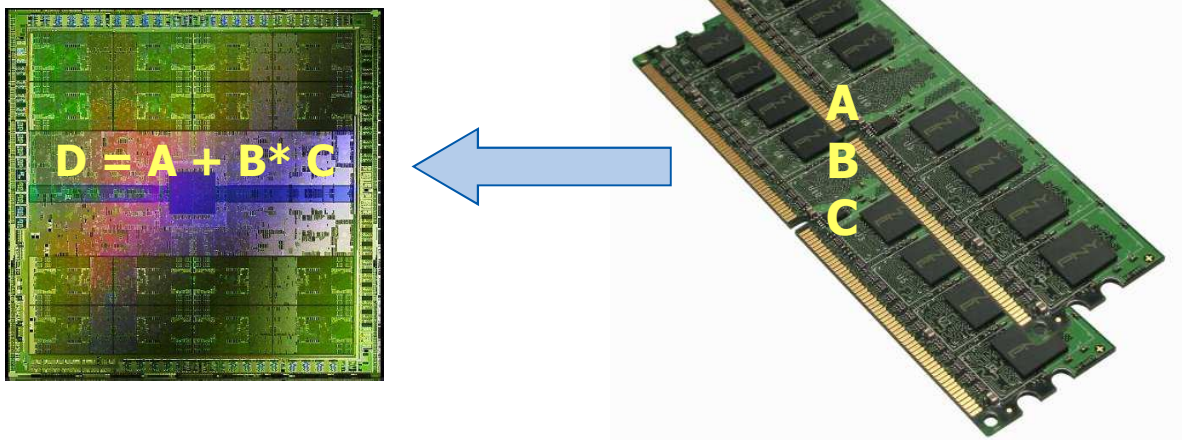
Rank	Site	Computer	Country	Cores	Rmax [Pflops]	% of Peak	Power [MW]	MFlops /Watt
1	DOE / OS Oak Ridge Nat Lab	Titan, Cray XK7 (16C) + Nvidia Kepler GPU (14c) + custom	USA	560,640	17.6	66	8.3	2120
2	DOE / NNSA L Livermore Nat Lab	Sequoia, BlueGene/Q (16c) + custom	USA	1,572,864	16.3	81	7.9	2063
3	RIKEN Advanced Inst for Comp Sci	K computer Fujitsu SPARC64 VIIIfx (8c) + custom	Japan	705,024	10.5	93	12.7	827
4	DOE / OS Argonne Nat Lab	Mira, BlueGene/Q (16c) + custom	USA	786,432	8.16	81	3.95	2066
5	Forschungszentrum Juelich	JuQUEEN, BlueGene/Q (16c) + custom	Germany	393,216	4.14	82	1.97	2102
6	Leibniz Rechenzentrum	SuperMUC, Intel (8c) + IB	Germany	147,456	2.90	90*	3.42	848
7	Texas Advanced Computing Center	Stampede, Dell Intel (8) + Intel Xeon Phi (61) + IB	USA	204,900	2.66	67	3.3	806
8	Nat. SuperComputer Center in Tianjin	Tianhe-1A, NUDT Intel (6c) + Nvidia Fermi GPU (14c) + custom	China	186,368	2.57	55	4.04	636
9	CINECA	Fermi, BlueGene/Q (16c) + custom	Italy	163,840	1.73	82	.822	2105
10	IBM	DARPA Trial System, Power7 (8C) + custom	USA	63,360	1.51	78	.358	422

Roadmap to Exascale (architectural trends)

Systems	2009	2011	2015	2018
System Peak Flops/s	2 Peta	20 Peta	100-200 Peta	1 Exa
System Memory	0.3 PB	1 PB	5 PB	10 PB
Node Performance	125 GF	200 GF	400 GF	1-10 TF
Node Memory BW	25 GB/s	40 GB/s	100 GB/s	200-400 GB/s
Node Concurrency	12	32	O(100)	O(1000)
Interconnect BW	1.5 GB/s	10 GB/s	25 GB/s	50 GB/s
System Size (Nodes)	18,700	100,000	500,000	O(Million)
Total Concurrency	225,000	3 Million	50 Million	O(Billion)
Storage	15 PB	30 PB	150 PB	300 PB
I/O	0.2 TB/s	2 TB/s	10 TB/s	20 TB/s
MTTI	Days	Days	Days	O(1Day)
Power	6 MW	~10 MW	~10 MW	~20 MW

Where Watts are burnt?

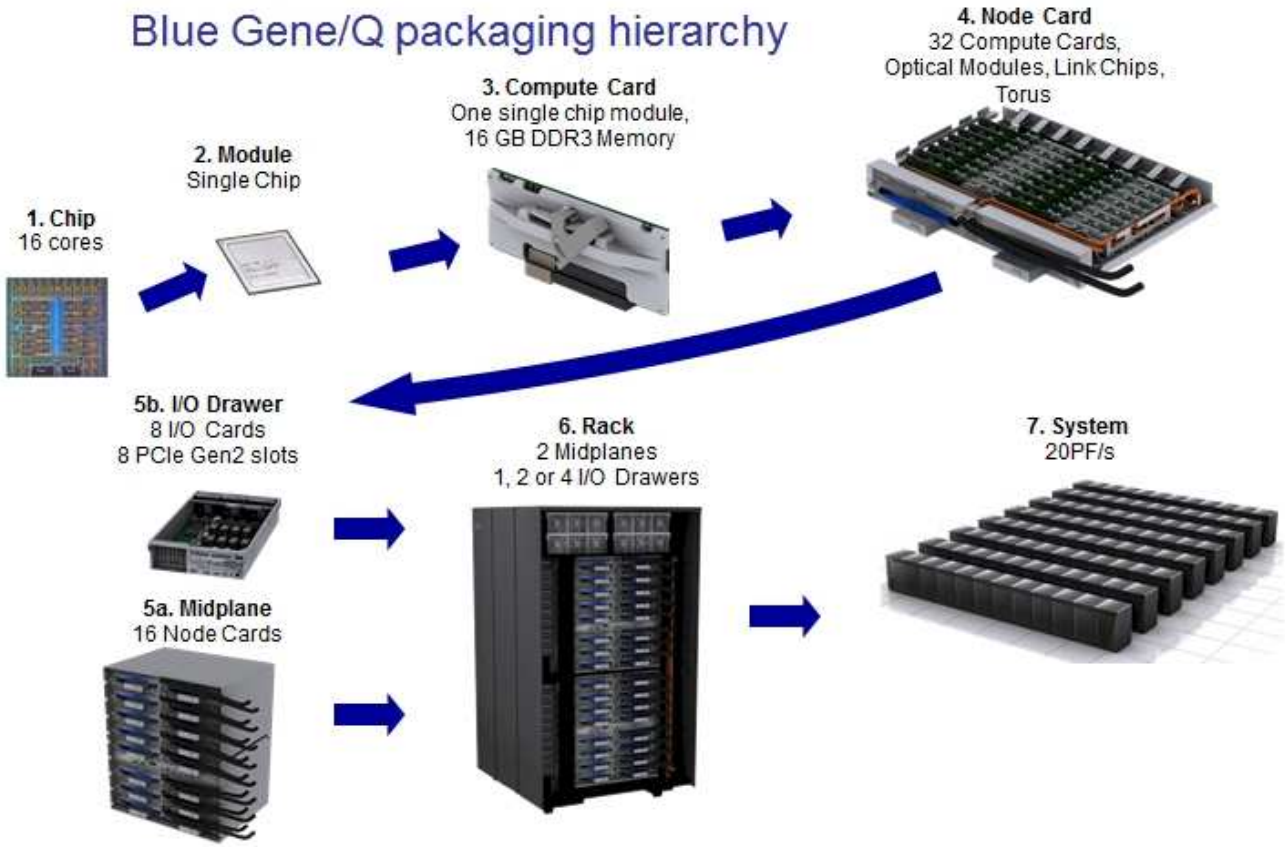
- Today (at 40nm) moving 3 64bit operands to compute a 64bit floating-point FMA takes 4.7x the energy with respect to the FMA operation itself



Extrapolating down to 10nm integration, the energy required to move date Becomes 100x !

MPP System

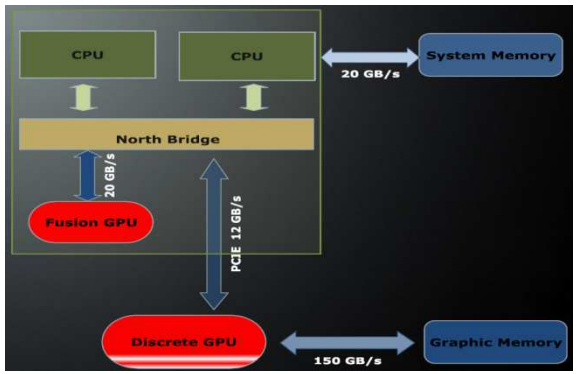
Arch	Option for BG/Q
When?	2012
PFlop/s	>2
Power	>1MWatt
Cores	>150000
Threads	>500000

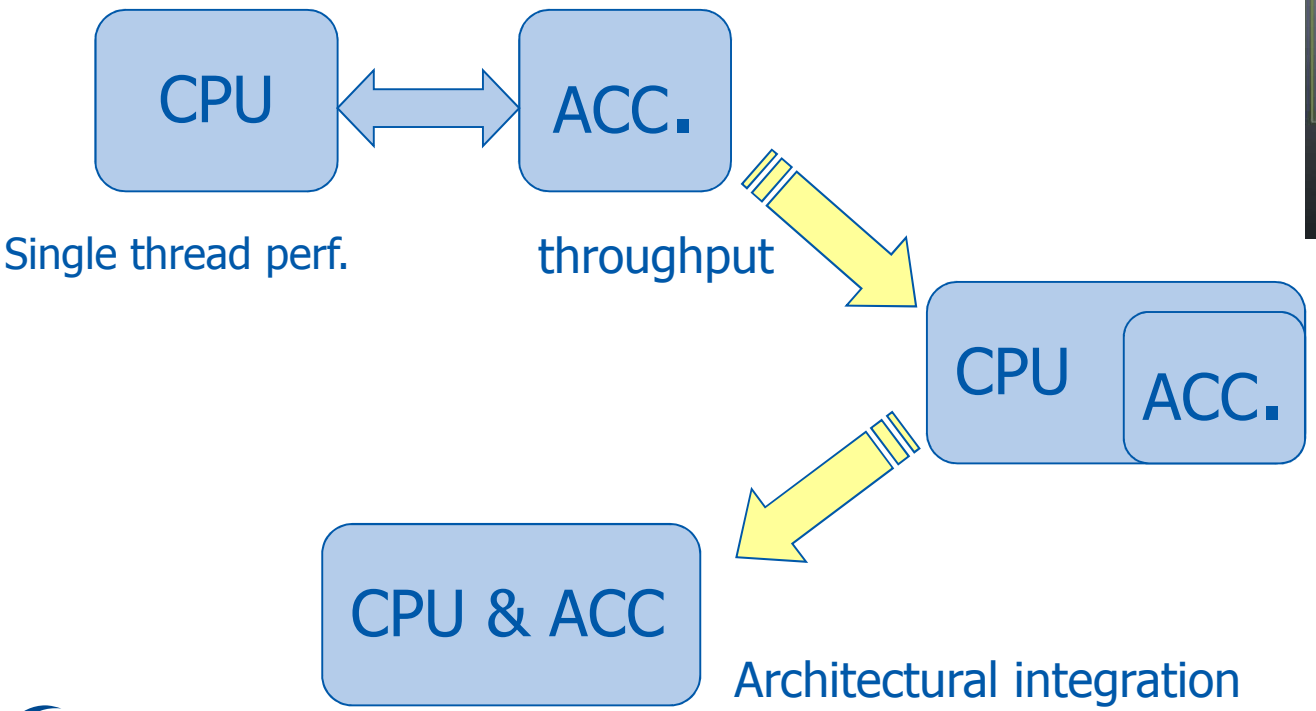


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Accelerator

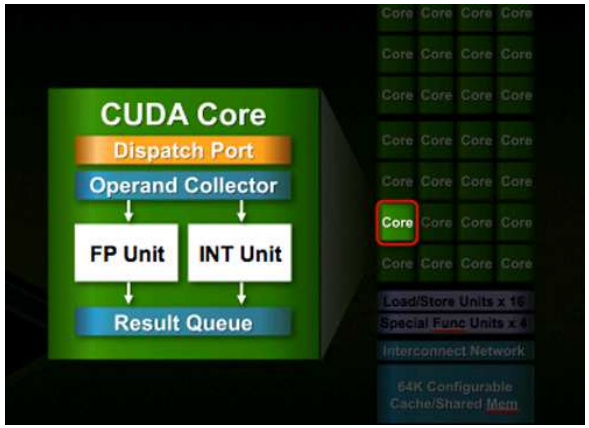
A set (one or more) of very simple execution units that can perform few operations (with respect to standard CPU) with very high efficiency. When combined with full featured CPU (CISC or RISC) can accelerate the “nominal” speed of a system. *(Carlo Cavazzoni)*





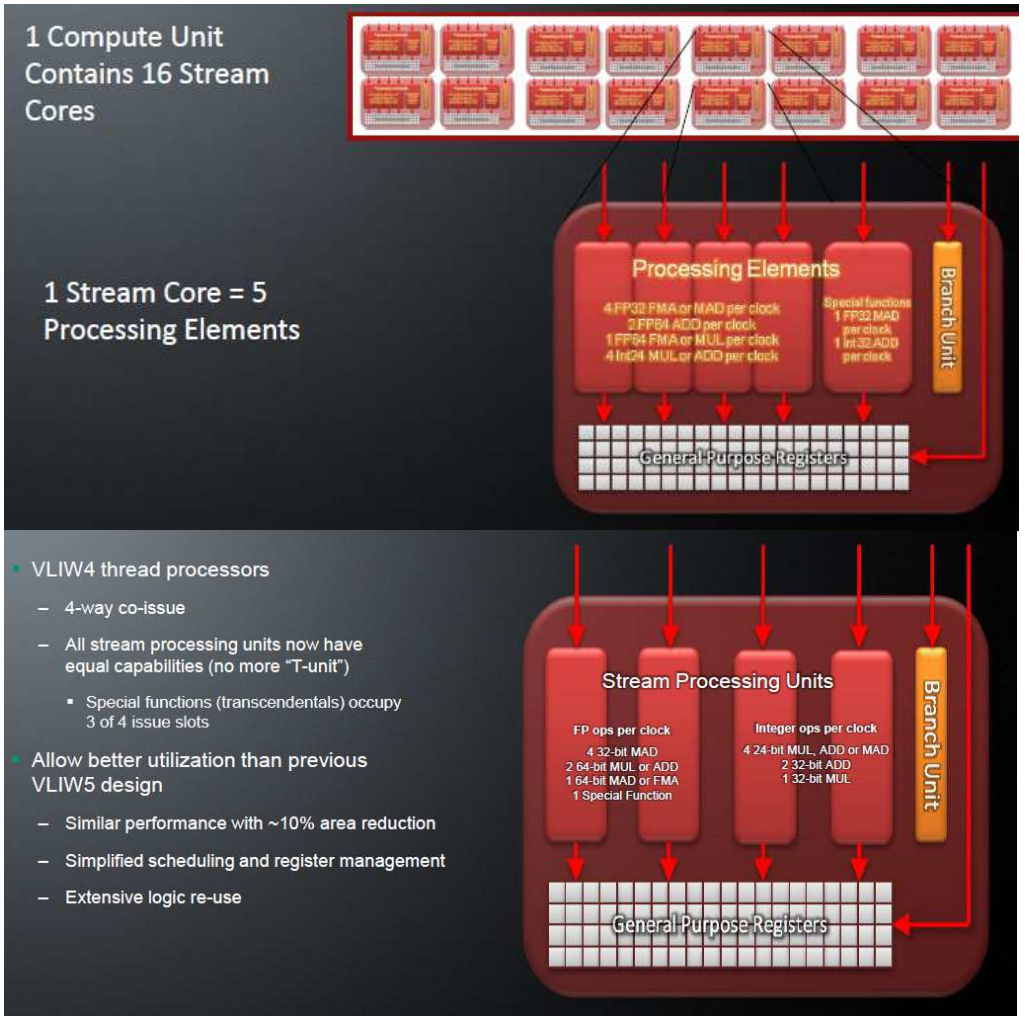
nVIDIA GPU

Tesla packs 2496 CUDA cores

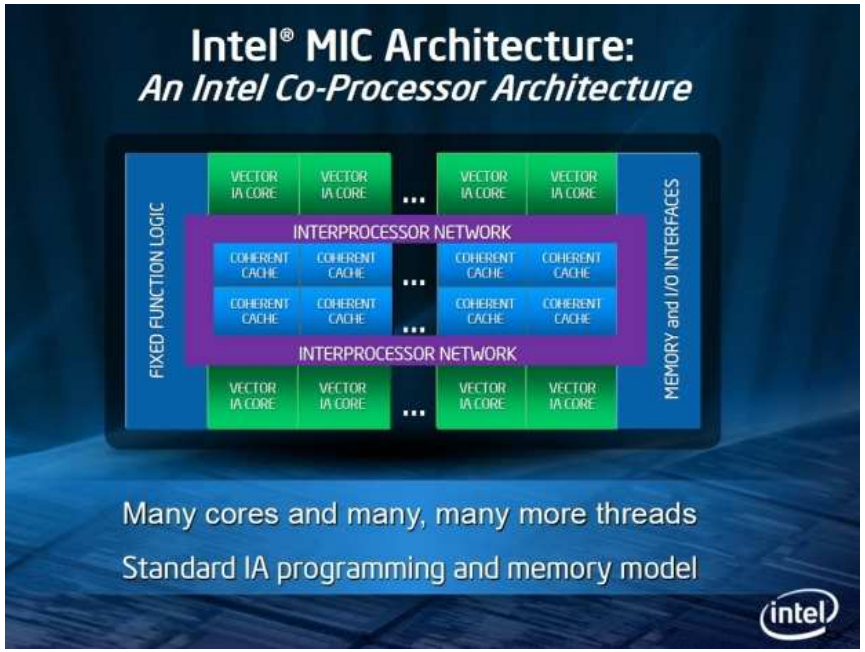


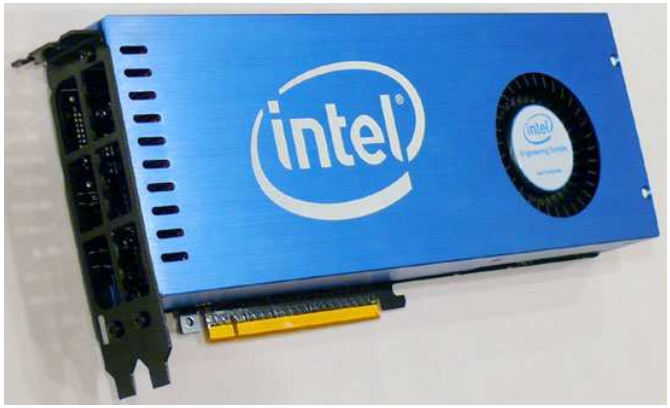
ATI FireStream, AMD GPU

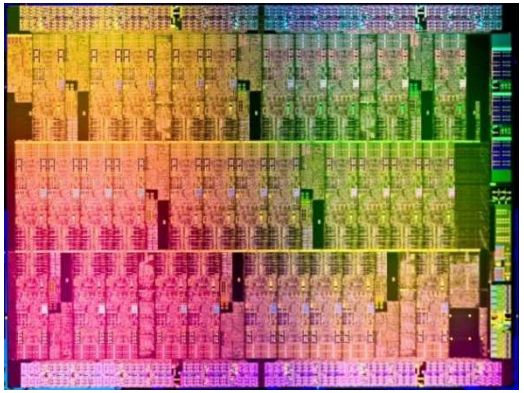
2012
New Graphics Core Next
"**GCN**"
With new instruction set and new SIMD design

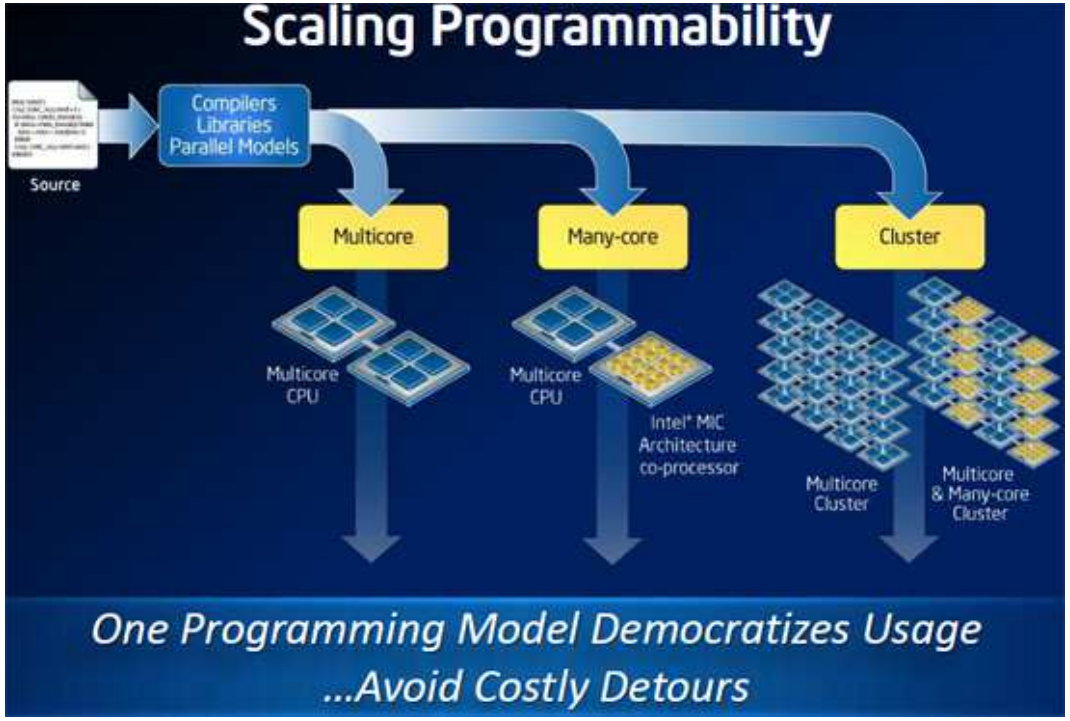


Intel MIC (Knight Ferry)



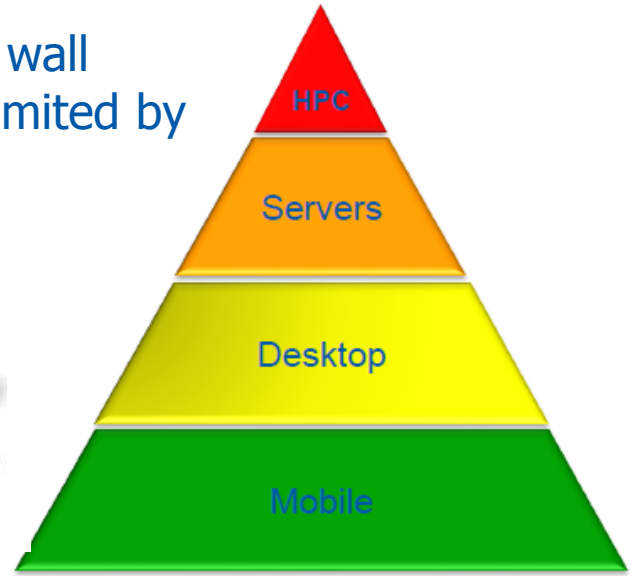


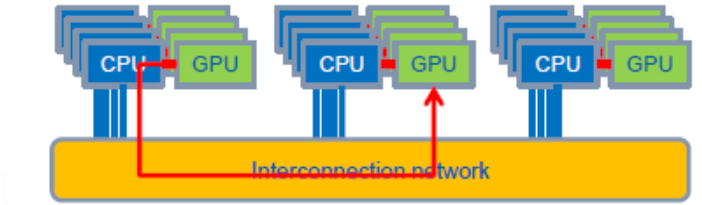


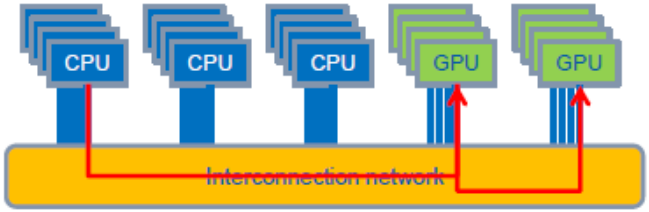


ARM + GPU

- We've hit the power wall
- ALL computers are limited by power consumption





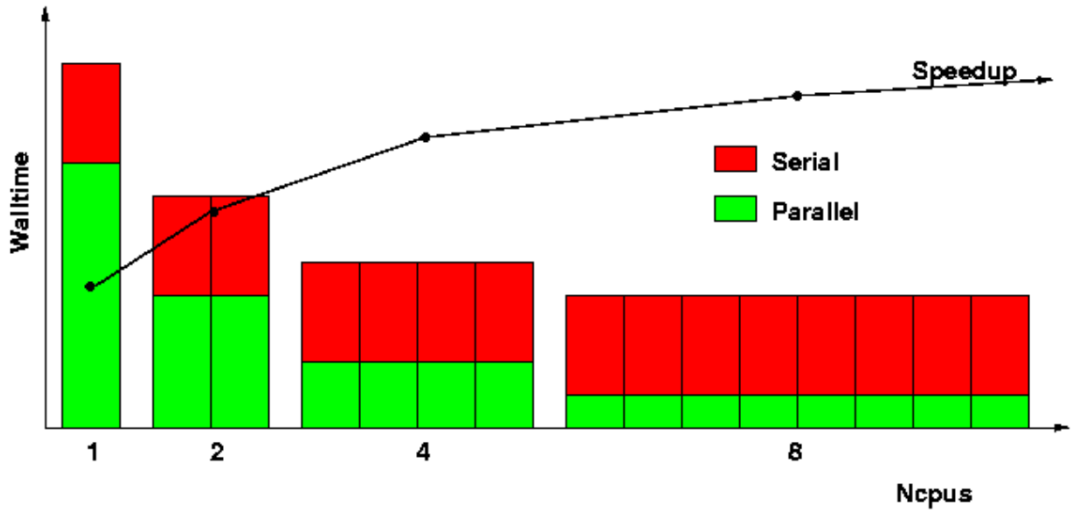


- Build the next HPC system on commodity and super-commodity components
  - 100M tablets in 2012
  - 750M smartphones in 2012

- **BSC Pedraforca:** a First ARM + GPU Cluster for HPC
- GPU-accelerated cluster vs. GPU-accelerator cluster

What about parallel App?

In a massively parallel context, an upper limit for the scalability of parallel applications is determined by the fraction of the overall execution time spent in non-scalable operations (Amdahl's law).



maximum speedup tends to
$1 / (1 - P)$
P= parallel fraction

1000000 core

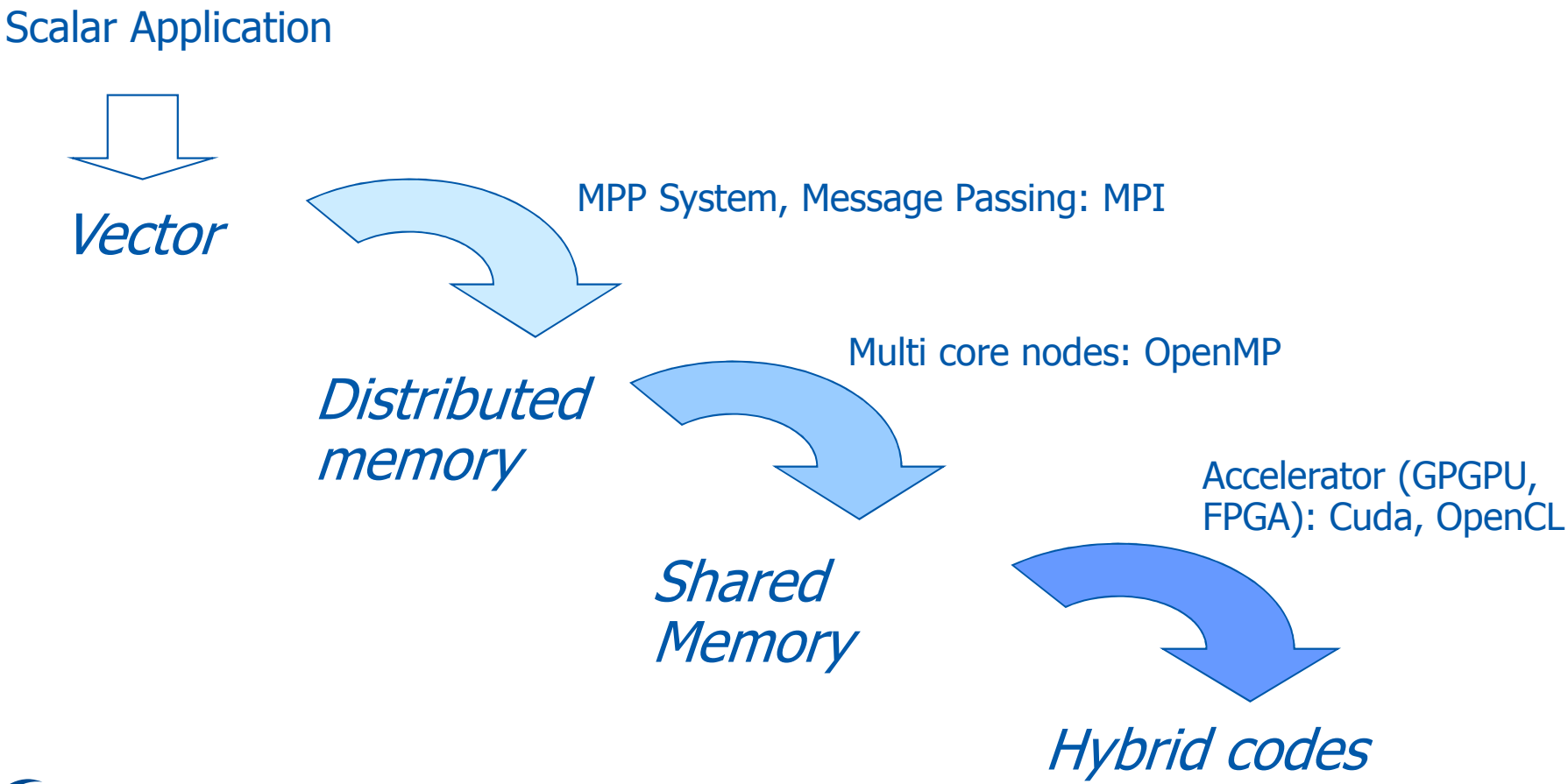
$P = 0.999999$

serial fraction= 0.000001

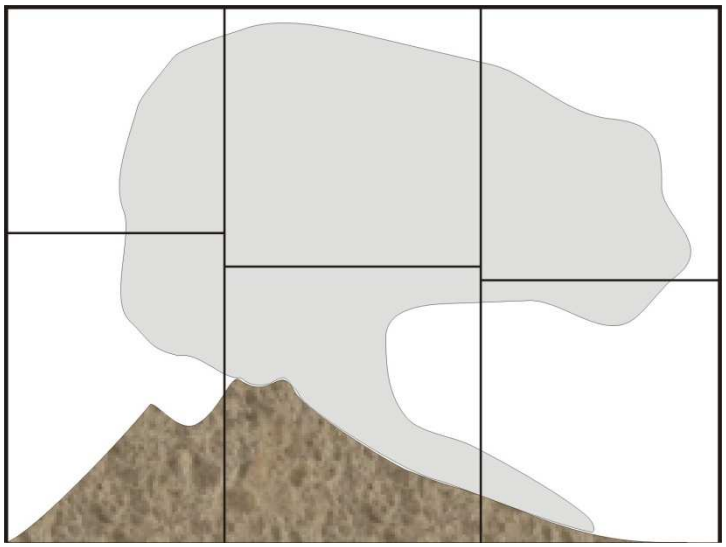
Programming Models

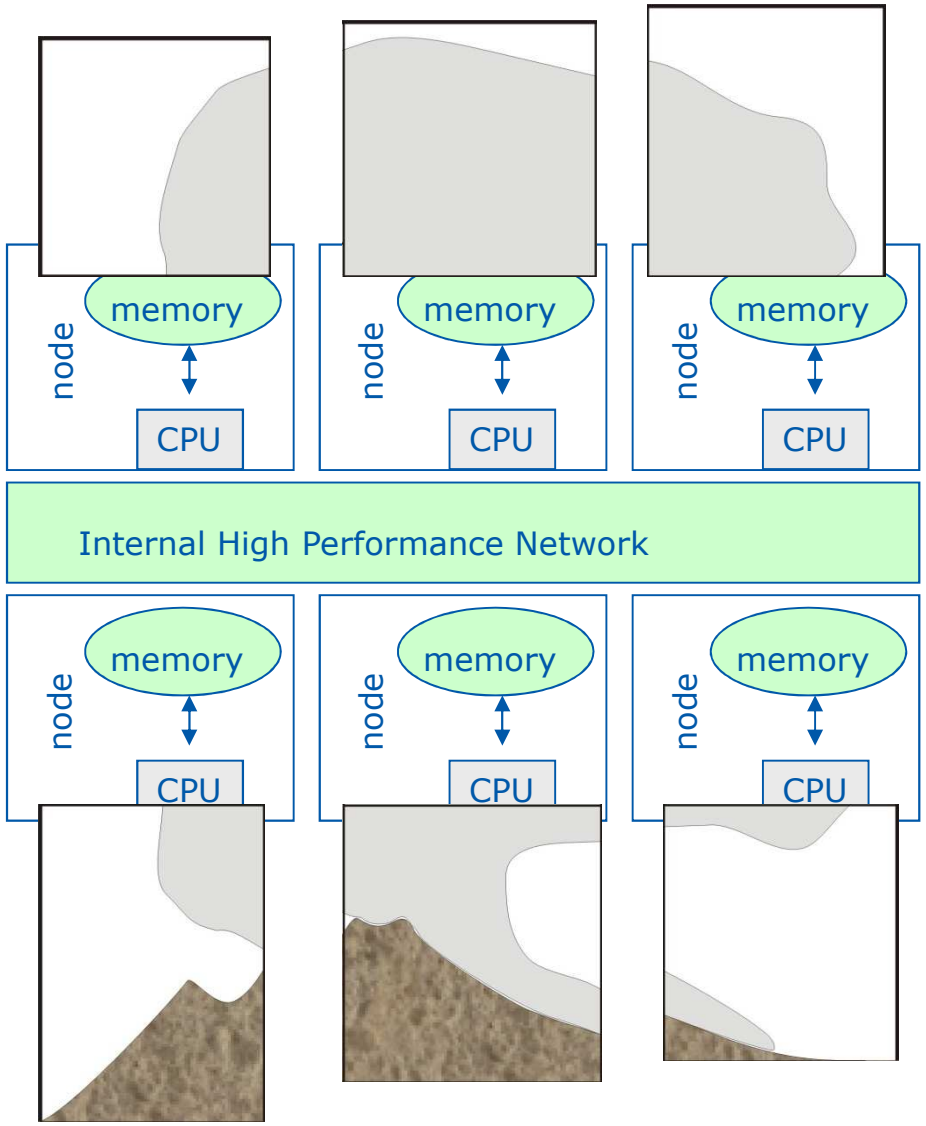
- At present, one model still dominates HPC application codes
  - **Message Passing (MPI)** for internode communication
  - **Shared Memory (OpenMP)** for intra-node parallelism
- Rapid adoption of GPUs in Top500 supercomputers has been met within by augmenting the MPI/OpenMP hybrid model with an additional third model that targets the Single Instruction Multiple Data (SIMD) architecture of GPUs.
  - The most popular **CUDA**
  - **OpenCL** an alternative open standard framework
  - directive-based open standards **OpenACC**
- **OmpSs**
  - extends OpenMP with directives for asynchronous parallelism and heterogeneity (devices like GPUs).
  - specifies data dependencies and MPI communication as tasks.
  - supports shared memory systems, systems with CUDA GPUs (other accelerator devices Intel MIC, OpenCL GPUs FPGAs) and clusters of these

trends


Scalar Application
Vector
MPP System, Message Passing: MPI
Distributed memory
Multi core nodes: OpenMP
Shared Memory
Accelerator (GPGPU, FPGA): Cuda, OpenCL
Hybrid codes

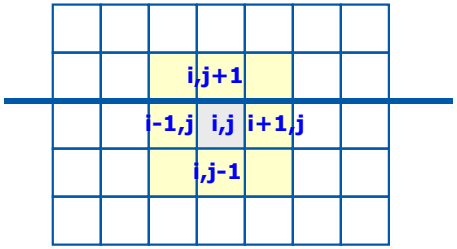
Message Passing domain decomposition



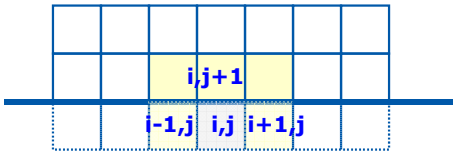


Ghost Cells - Data exchange

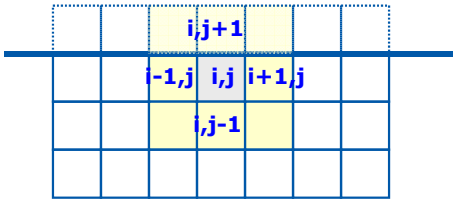
sub-domain boundaries



Processor 1

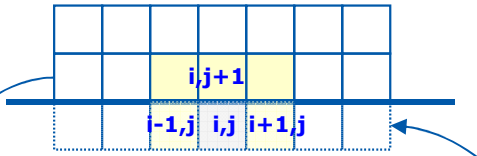


Ghost Cells

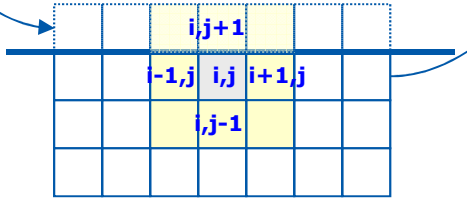


Processor 2

Processor 1



Ghost Cells exchanged
between processors
at every update



Processor 2

Message Passing: MPI

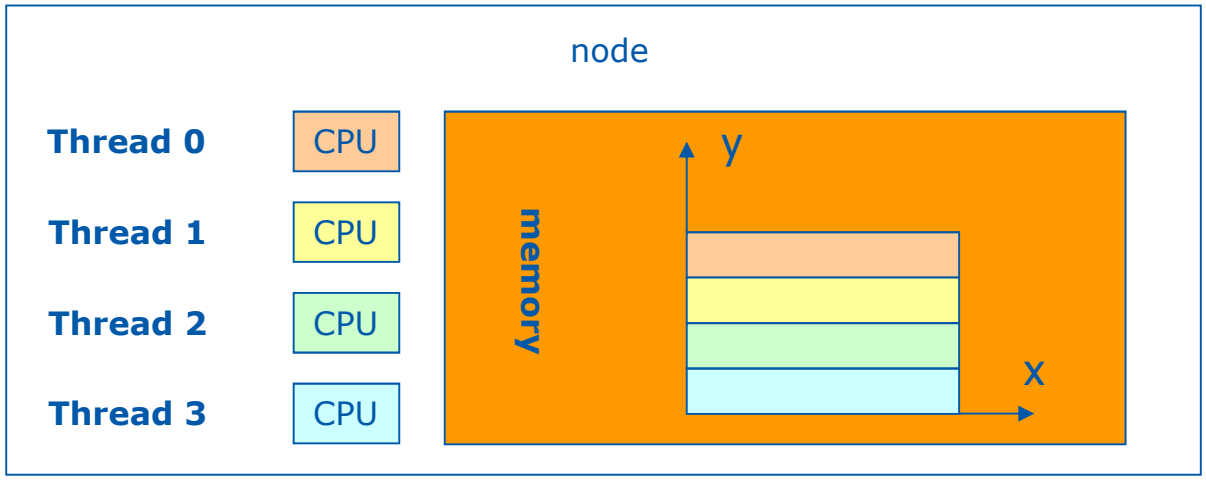
Main Characteristic

- Library
- Coarse grain
- Inter node parallelization (few real alternative)
- Domain partition
- Distributed Memory
- Almost all HPC parallel App

Open Issue

- Latency
- OS jitter
- Scalability

Shared memory



Shared Memory: OpenMP

Main Characteristic

- Compiler directives
- Medium grain
- Intra node parallelization (pthreads)
- Loop or iteration partition
- Shared memory
- Many HPC App

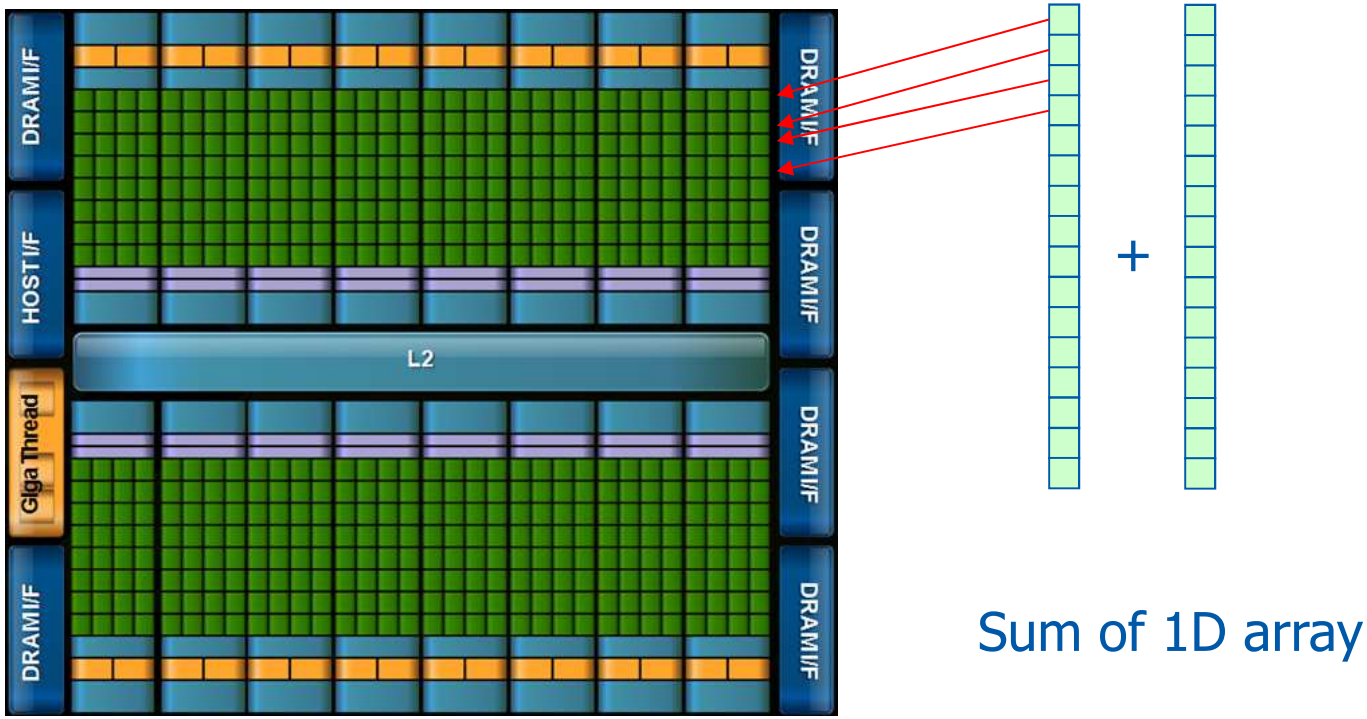
Open Issue

- Thread creation overhead
- Memory/core affinity
- Interface with MPI

OpenMP

```
(!$omp parallel do
(do i = 1 , nsl
(   call 1DFFT along z ( f [ offset( threadid ) ] )
(end do
(!$omp end parallel do
(call fw_scatter ( . . . )
(!$omp parallel
(do i = 1 , nzl
(!$omp parallel do
(  do j = 1 , Nx
(    call 1DFFT along y ( f [ offset( threadid ) ] )
(  end do
(!$omp parallel do
(  do j = 1, Ny
(    call 1DFFT along x ( f [ offset( threadid ) ] )
(  end do
(end do
(!$omp end parallel
```

Accelerator/GPGPU



Sum of 1D array

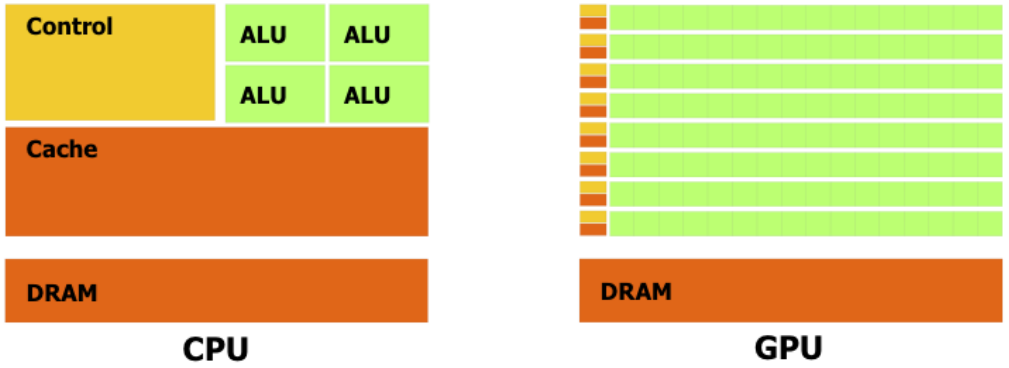
CUDA sample

```
void  CPUCode( int* input1, int* input2, int* output, int length) {
                for ( int  i = 0; i < length; ++i ) {
                      output[ i ] = input1[ i ] + input2[ i ];
               }
}
```

```
__global__void  GPUCode( int* input1, int*input2, int* output, int length) {
              int idx = blockDim.x * blockIdx.x + threadIdx.x;
               if ( idx < length ) {
                     output[ idx ] = input1[ idx ] + input2[ idx ];
              }
}
```

Each thread executes one loop iteration

CUDA
OpenCL



Main Characteristic

- Ad-hoc compiler
- Fine grain
- offload parallelization (GPU)
- Single iteration parallelization
- Ad-hoc memory
- Few HPC App

Open Issue

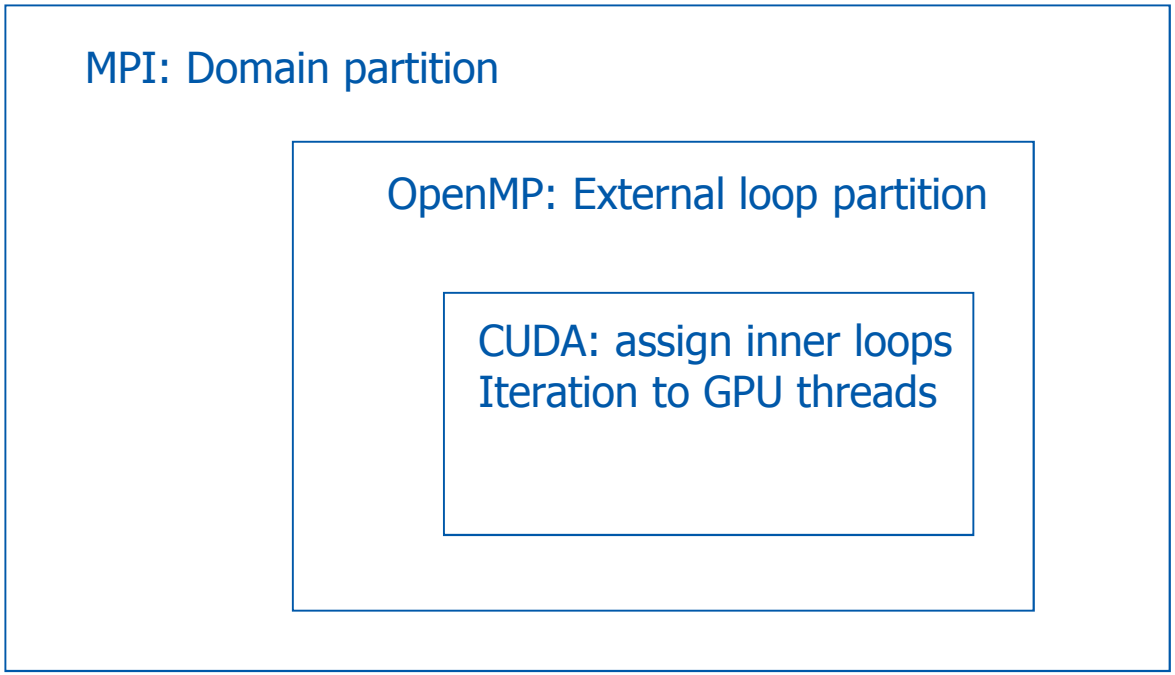
- Memory copy
- Standard
- Tools
- Integration with other languages

Hybrid (MPI+OpenMP+CUDA+…)

- Take the positive off all models
  - Exploit memory hierarchy
  - Many HPC applications are adopting this model
  - Mainly due to developer inertia
  - Hard to rewrite million of source lines

- **Must rethink the design of our algorithms and software**
  - **Manycore and Hybrid architectures are disruptive technology**
  - **Similar to what happened with cluster computing and message passing**
  - **Rethink and rewrite the applications, algorithms, and software**
  - **Data movement is expensive**
  - **Flops are cheap**

Hybrid parallel programming



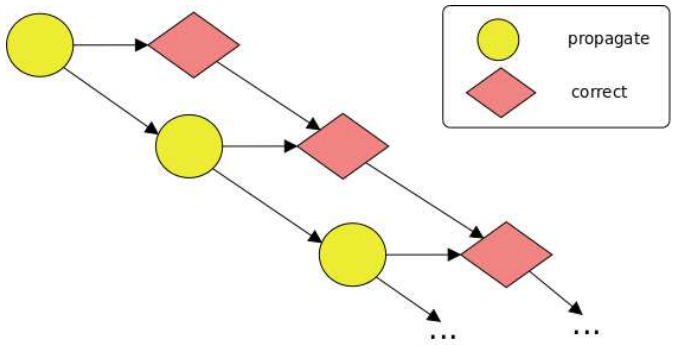
- Hybridization is the process of converting an application with a single level of parallelism to an application with multiple levels of parallelism.
- Over the past 15 years a majority of the applications that run on High Performance Computing systems have employed MPI for all of the parallelism within the application.
- In the Peta-Exascale computing regime, effective utilization of the hardware requires multiple levels of parallelism matched to the macro architecture of the system to achieve good performance.
- OmPSS and OpenACC allow a unified code base to be deployed for either (Manycore CPU or Manycore CPU+GPU) while permitting architecture specific optimizations to expose new dimensions of parallelism to be utilized.

OmPSS

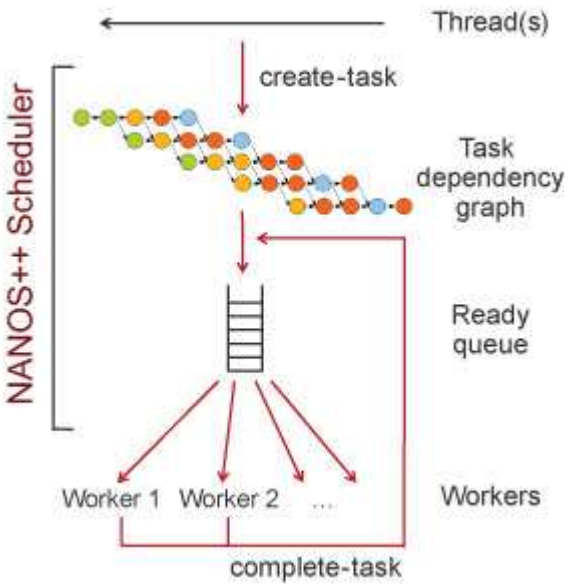
- Extend OpenMP with new directives to support asynchronous parallelism and heterogeneity (devices like GPUs).
- Asynchronous parallelism is enabled in OmpSs by the use data-dependencies between the different tasks of the program. The OpenMP task construct is extended with **input**, **output** and **inout** clauses to this end. They allow to specify for each task in the program what data a task is waiting for and signaling is readiness.

Example and dependency graph:

```
void foo ( int *a, int *b )
{
   for ( i  = 1; i < N; i++ ) {
      #pragma omp task input(a[i-1]) inout(a[i]) output(b[i])
         propagate(&a[i-1],&a[i],&b[i]);

      #pragma omp task input(b[i-1]) inout(b[i])
         correct(&b[i-1],&b[i]);
   }
}
```



OmPSS



- The proposed solution is to use support for tasks and efficient task scheduling to exploit parallelism and simplify programming.

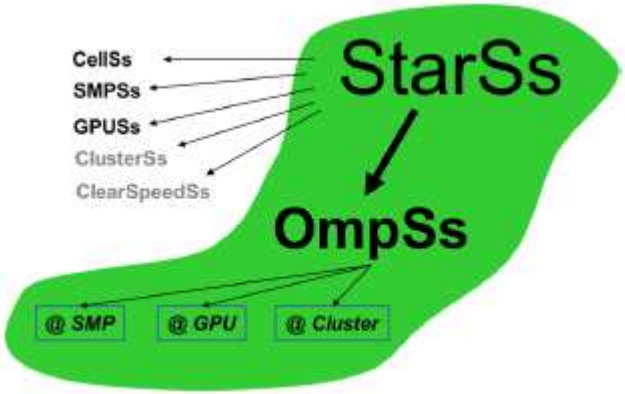
- To support heterogeneity a new construct is introduced: the **target** construct. The intent of the **target** construct is to specify that a given element can be run in a set of devices.
  The **target** construct can be applied to either a **task** construct or a function definition.

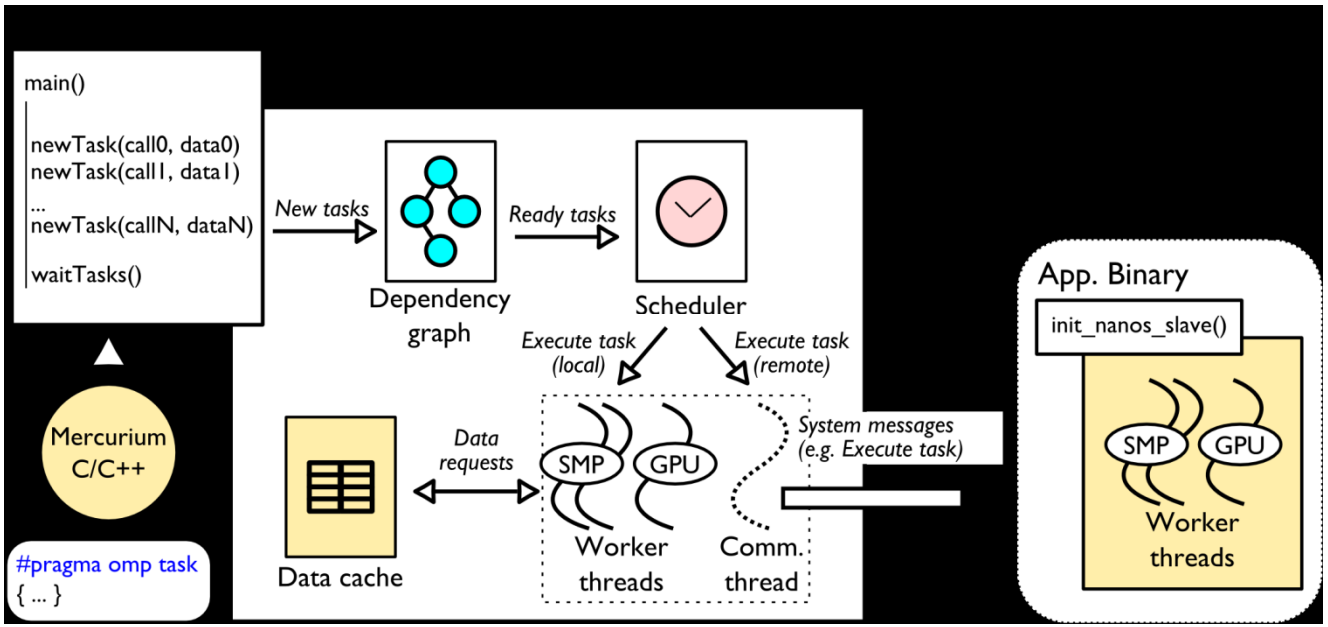
Example:

```
##pragma omp target device ({ smp | cuda }) \
[ implements ( function_name )] \
{ copy_deps | [ copy_in ( array_spec ,...)]
[ copy_out (...)] [ copy_inout (...)] }
```

OmPSS MultiGPU/Clusters

- Automatic handling of Multi-GPU execution
  - One manager thread per GPU: data transfers, task execution, synchronization
- Clusters: One runtime instance per node
  - One master image
  - N-1 worker images
  - Communication thread
  - Data transfers





OpenACC

- Accelerator programming API standard to program accelerators Portable across operating systems and various types of host CPUs and accelerators.
- Allows parallel programmers to provide simple hints, known as “directives,” to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself.
- Aimed at incremental development of accelerator code

- Directives facilitate code development for accelerators
  - Provide the functionality to: Initiate accelerator startup/shutdown
  - Manage data or program transfers between host (CPU) and accelerator
  - Scope data between accelerator and host (CPU)
  - Manage the work between the accelerator and host.
  - Map computations (loops) onto accelerators
  - Fine-tune code for performance

OpenACC

Parallelism:

- Support coarse-grain parallelism
  - Fully parallel across execution units
  - Limited synchronizations across
  - coarse-grain parallelism
- Support for fine-grain parallelism often implemented as SIMD
  - Vector operations
  - Programmer need to understand the differences between them. Efficiently map parallelism to accelerator
  - Understand synchronizations available
- Memory Model
  - Host + Accelerator memory may have completely separate memories Host may not be able to read/write device memory that is not mapped to a shared virtual addressed.
  - All data transfers must be initiated by host Typically using direct memory accesses (DMAs)
- Data movement is implicit and managed by compiler
- Device may implement weak consistency memory model

OpenACC

OpenACC Parallel Directive

- Starts parallel execution on accelerator
  - Specified by:

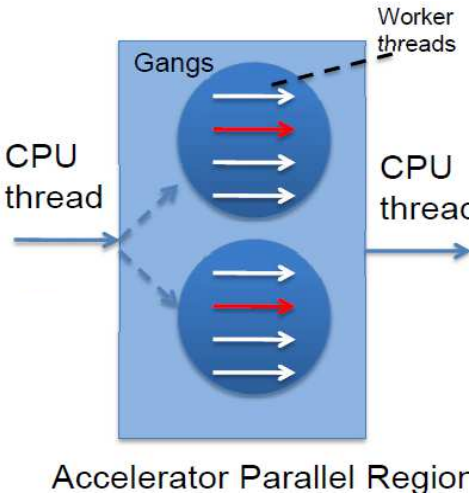
```
#pragma acc parallel [clause
[,clause]…] new-line
structured block
```

  - When encountered: Gangs of workers threads are created to execute on accelerator
  - One worker in each gang begins executing the code following the structured block
  - Number of gangs/workers remains constant in parallel region

OpenACC Kernels Directive

- Defines a region of a program that is to be compiled into a sequence of kernels for execution on the accelerator
  - Each loop nest will be a different kernel
  - Kernels launched in order in device
  - Specified by:

```
#pragma acc kernels [clause
[,clause]…] new-line
structured block
```



OpenACC Data Directive

- The data construct defines scalars, arrays and subarrays to be allocated in the accelerator memory for the duration of the region.
- Can be used to control if data should be copied-in or out from the host
  - Specified by:

```
#pragma acc data [clause
[,clause]…] new-line
structured block
```

OpenACC Loop Directive

- Used to describe what type of parallelism to use to execute the loop in the accelerator.
- Can be used to declare loop-private variables, arrays and reduction operations.
  - Specified by:

```
#pragma acc loop [clause
[,clause]…] new-line
for loop
```

Conclusion

Parallel programming trends in extremely scalable architectures

- Traditional
  - MPI
  - OpenMP
- GPU
  - CUDA/OpenCL
- Hybrid
  - MPI + OpenMP +CUDA/OpenCL
  - OmPSS
  - OpenACC

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI

Based on presentation by Janko Strassburg, BSC

Isaac Rudomin
BSC

Motivation

High Performance Computing

Processing speed

+

Memory capacity

Software development for:
Supercomputers, Clusters, Grids

Programming MPP/Clusters

Possibilities:

- Special parallel programming languages
- Extensions of existing sequential programming languages
- Usage of existing sequential programming languages + libraries with external functions for message passing

Usage of Existing Sequential Programming Languages

Approach:

- Use FORTRAN/C
- Function calls to message passing library

Explicit parallelism:

User defines

- which processes to execute,
- when and how to exchange messages, und
- which data to exchange within messages.

MPI Intention

- Specification of a standard library for programming message passing systems
- Interface: practical, portable, efficient, and flexible
- ⇨ *Easy to use*
- For vendors, programmers, and users

MPI Goals

- Design of an API (Application Programming Interface)
- Possibilities for efficient communication (Hardware-Specialities, ...)
- Implementations for heterogeneous environments
- Definition of an interface in a traditional way (comparable to other systems)
- Availability of extensions for increased flexibility
- Definition, that is easy to be realized on different kinds of hardware platforms.

MPI Forum

Collaboration of 40 Organisations (world-wide):

- IBM T.J. Watson Research Center
- Intels NX/2
- Express
- nCUBE's VERTEX
- p4 - Portable Programs for Parallel Processors
- PARMACS
- Zipcode
- PVM
- Chameleon
- PICL
- ...

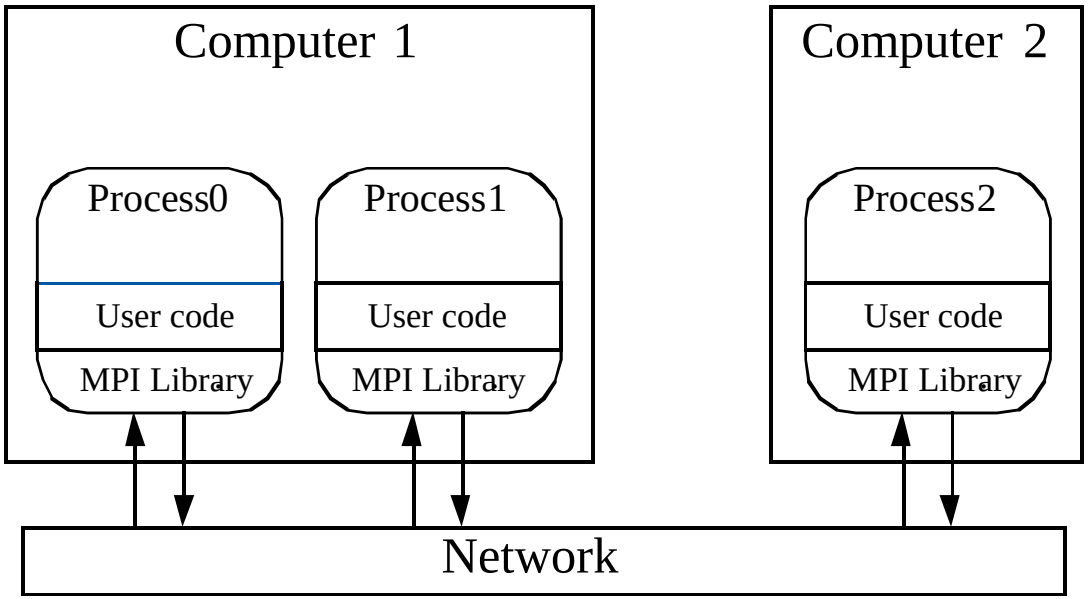
Available Implementations

- MPICH
- CHIMP
- LAM
- FT-MPI
- Open MPI:
  - Combined effort from FT-MPI, LA-MPI, LAM/MPI, PACX-MPI
  - De facto standard; used on many TOP500 systems
- Vendor specific implementations:
  - Bull, Fujitsu, Cray, IBM, SGI, DEC, Parsytec, HP, ...

MPI Programming Model

- **Parallelization**:
  - Explicit parallel language constructs (for communication)
  - Library with communication functions
- **Classification of Flynn**:
  - MIMD (multiple instructions streams over multiple data streams)
  - No (automatic) synchronization of processes
- **Programming Model**:
  - SPMD (single program multiple data)
  - All processes load the same source code
  - Distinction through process number

MPI Program



2 Parts:

- User code
- MPI Functionality (from MPI Library)

MPI Functionality

- Process Creation and Execution
- Queries for system environment
- Point-to-point communication (Send/Receive)
- Collective Operations (Broadcast, ...)
- Process groups
- Communication context
- Process topologies
- Profiling Interface

Processes

Characteristics:

- For *Parallelism*, computation must be partitioned into multiple processes (or tasks)
- Processes are assigned to processors ⇨ **mapping**
  - 1 Process = 1 Processor
  - n Processes = 1 Processor
- *Multitasking* on one processor:
  - Disadvantage: Longer execution time due to time-sharing
  - Advantage: Overlapping of communication latency

Granularity

- The size of a process defines its granularity

- **Coarse Granularity**: each process contains many sequential execution blocks ⇨ execution time
- **Fine Granularity**: each process contains only few (sometimes one) instructions

Granularity

- Granularity =
  Size of computational blocks between communication and synchronization operations

- The higher the granularity, the
  - smaller the costs for process creation
  - smaller the number of possible processes and the achievable parallelism

Parallelization

- Data Partitioning:
  SPMD = Single Program Multiple Data
- Task Partitioning:
  MPMD = Multiple Program Multiple Data
- “Chinese Army Technique”

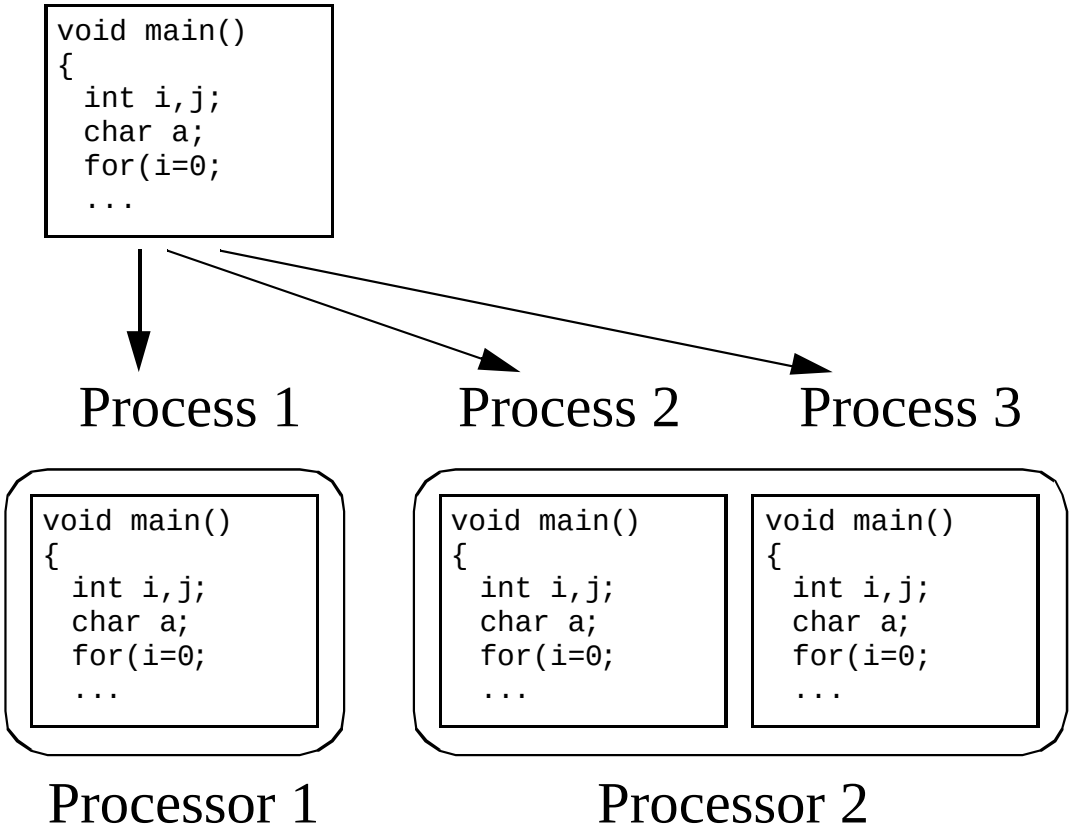
Types of Process-Creation:

- Static
- Dynamic

Data Partitioning (SPMD)

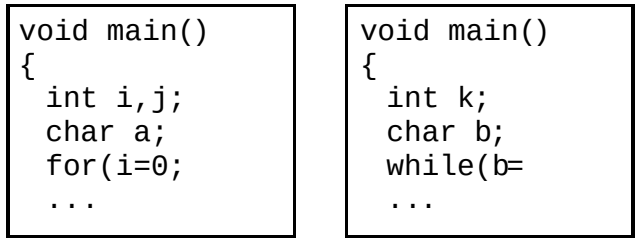
Implementation:
1 Source code

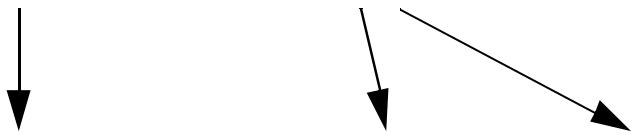
Execution:
n Executables



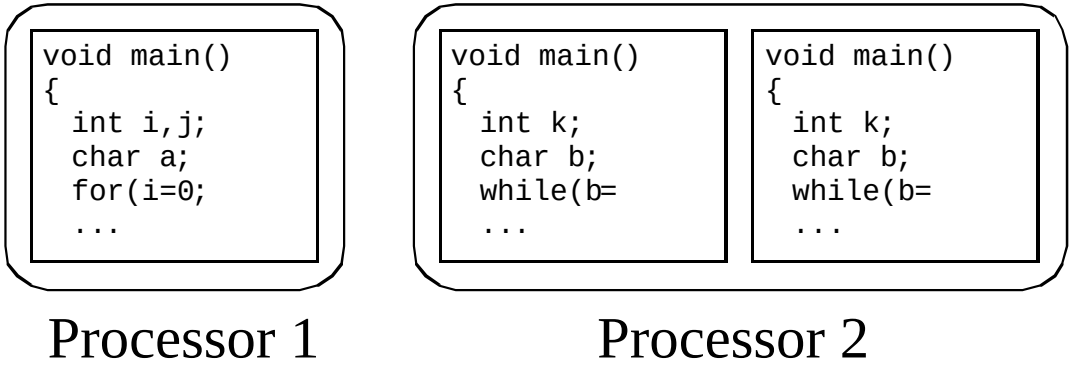
Task-Partitioning (MPMD)

Implementation:
m Source codes





Execution:
n Executables



Comparison: SPMD/MPMD

SPMD:

- One source code for all processes
- Distinction in the source code through control statements

```
if (pid() == MASTER) { ... }
else { ... }
```

- Widely used

Comparison: SPMD/MPMD

MPMD:

- One source for each process
- Higher flexibility and modularity
- Administration of source codes difficult
- Additional effort during process creation
- Dynamic process creation possible

Process Creation

Static:

- All processes are defined before execution
- System starts a fixed number of processes
- Each process receives same copy of the code

Process Creation

Dynamic:

- Processes can creation/execute/terminate other processes during execution
- Number of processes changes during execution
- Special constructs or functions are needed

- **Advantage**:
  higher flexibility than SPMD
- **Disadvantage**:
  process creation expensive ⇨ overhead

Process Creation/Execution

Commands:

- Creation and execution of processes is not part of the standard, but instead depends on the chosen implementation:

  Compile: `mpicc -o <exec> <file>.c`

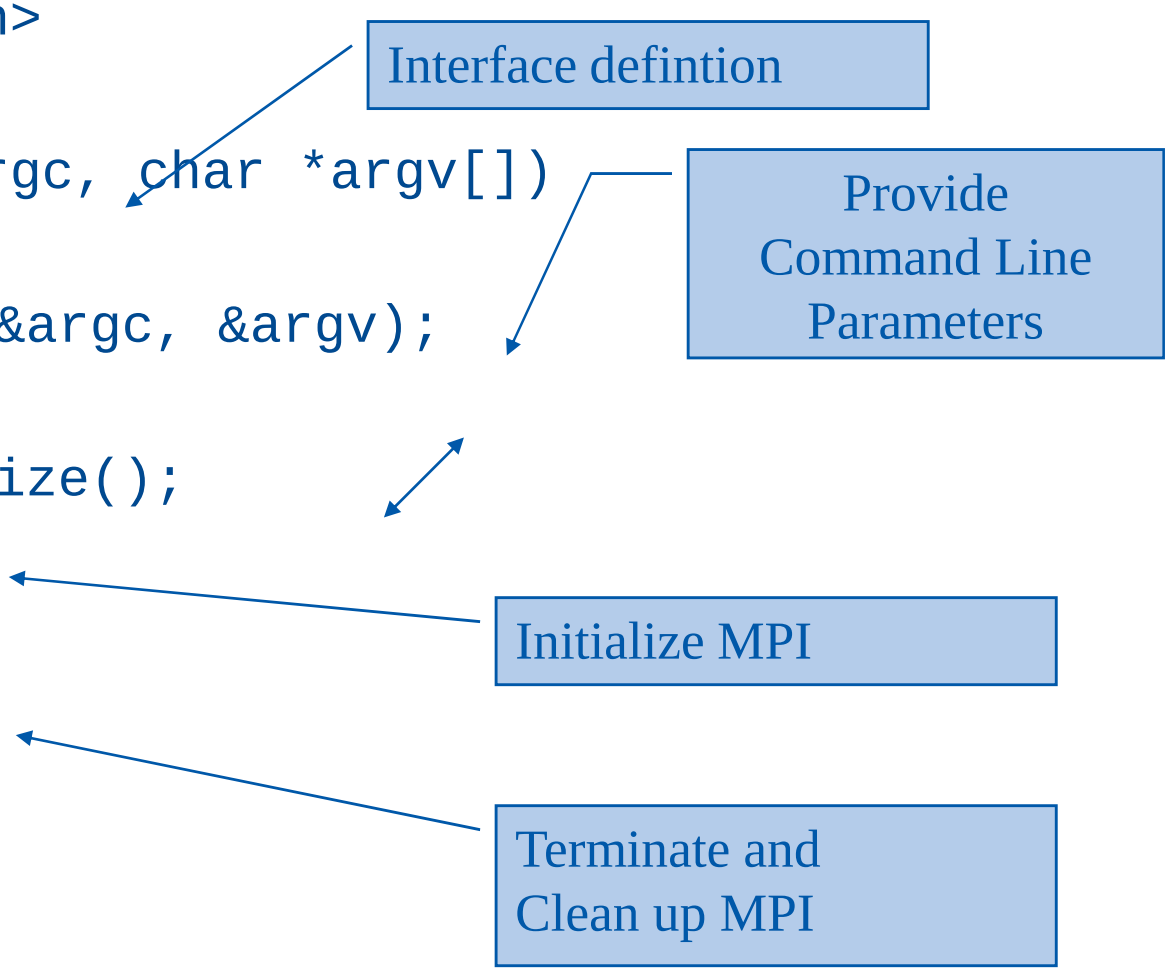
  Execute: `mpirun -np <proc> <exec>`

- Process Creation: only static (before MPI-2)
- SPMD programming model
- Mapping of process to processes not part of the standard (permits optimal automatic mapping)

Basic-Code-Fragment

Initialization and Exit:

```
1.  #include <mpi.h>
2.  ...
3.  int main(int argc, char *argv[])
4.  {
5.        MPI_Init(&argc, &argv);
6.  ...
7.        MPI_Finalize();
8.  }
```



Structure of MPI Functions

General:

```
1.  result = MPI_Xxx(…);
```

Example:

```
1.  result = MPI_Init(&argc, &argv);
2.  if(result!=MPI_SUCCESS) {
3.     fprintf(stderr,”Problem”);
4.     fflush(stderr);
5.     MPI_Abort(MPI_COMM_WORLD,result);
6.  }
```

Query Functions

Identification:

- *Who am I?*
- Which process number has the current process?

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
```

- *Who else is there?*
- How many processes have been started?

```
MPI_Comm_size(MPI_COMM_WORLD, &mysize)
```

- Characteristics: 0 <= myrank < mysize

MPI & SPMD Restrictions

- *Ideally*: Each process executes the same code.
- *Usually*: One (or a few) processes execute slightly different codes.

- Preliminaries:
  Statements to distinguish processes and the subsequent code execution

- Example: Master-slave program
  ⇨ complete code within one program/executable

Master-Slave Program

```
1.  int main(int argc, char *argv[])
2.  {
3.     MPI_Init(&argc, &argv);
4.     ...
5.     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6.     if(myrank == 0)
7.           master();
8.     else
9.           slave();
10.    ...
11.    MPI_Finalize();
12. }
```

Global Variables

Problem:

```
1.  int main(int argc, char *argv[])
2.  {  float big_array[10000];
```

Solution:

```
1.  int main(int argc, char *argv[])
2.  {
3.     if(myrank == 0) {
4.           float big_array[10000];
```

Global Variables

Problem:

```
1.  int main(int argc, char *argv[])
2.  { float big_array[10000];
```

Solution:

```
1.  int main(int argc, char *argv[])
2.  {
3.    float *big_array;
4.    if(myrank == 0) {
5.          big_array = (float *)malloc(...)
```

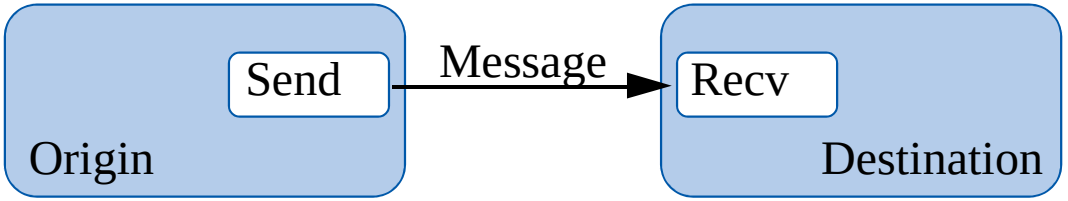
Guidelines for Using Communication

- Try to avoid communication as much as possible: more than a factor of 100/1000 between transporting a byte and doing a multiplication
  - Often it is faster to replicate computation than to compute results on one process and communicate them to other processes.
- Try to combine messages before sending.
  - It is better to send one large message than several small ones.

Message Passing

Basic Functions:

- send(parameter_list)
- recv(parameter_list)



Send Function:

- In origin process
- Creates message

Receive Function:

- In destination process
- Receives transmitted message

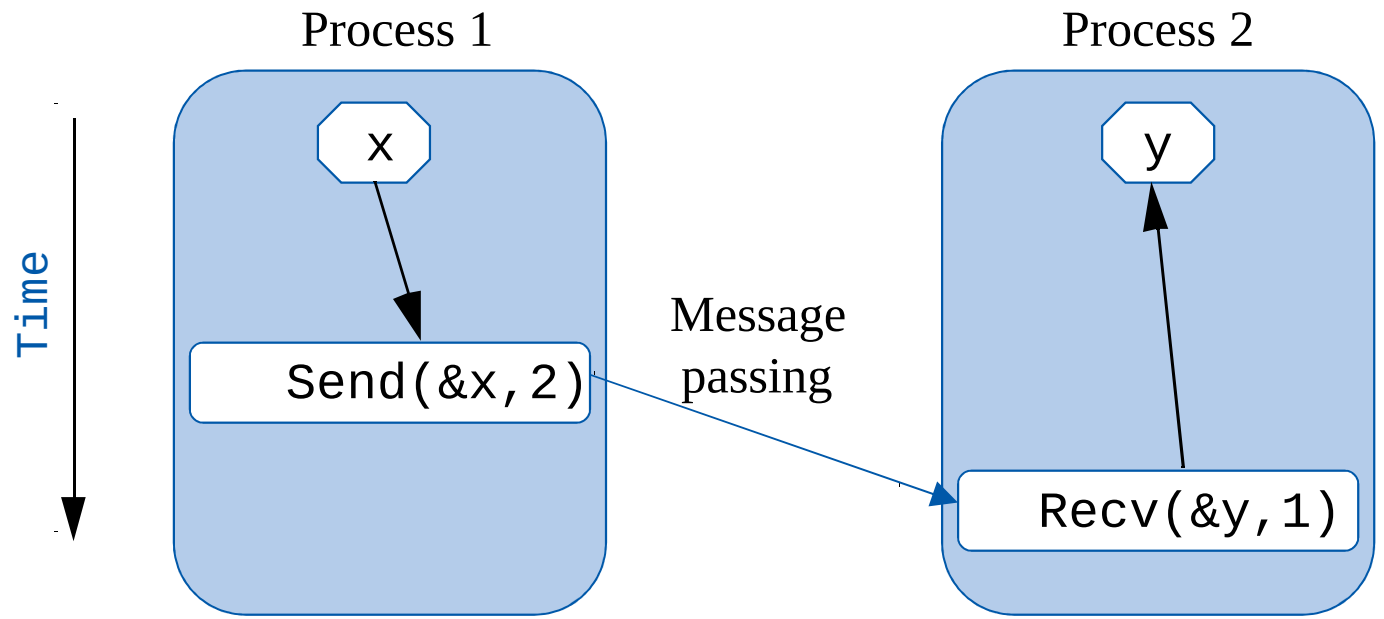
Simple Functions

On the origin process:

```
send(&x,destination_id)
```

On the destination process:

```
recv(&y, source_id)
```



Standard Send

```
int MPI_Send (void *buf, int count,
   MPI_Datatype datatype, int dest,
      int tag, MPI_Comm comm)
```

- `buf` Address of message in memory
- `count` Number of elements in message
- `datatype` Data type of message
- **`dest`** Destination process of message
- `tag` Generic message tag
- `comm` Communication handler

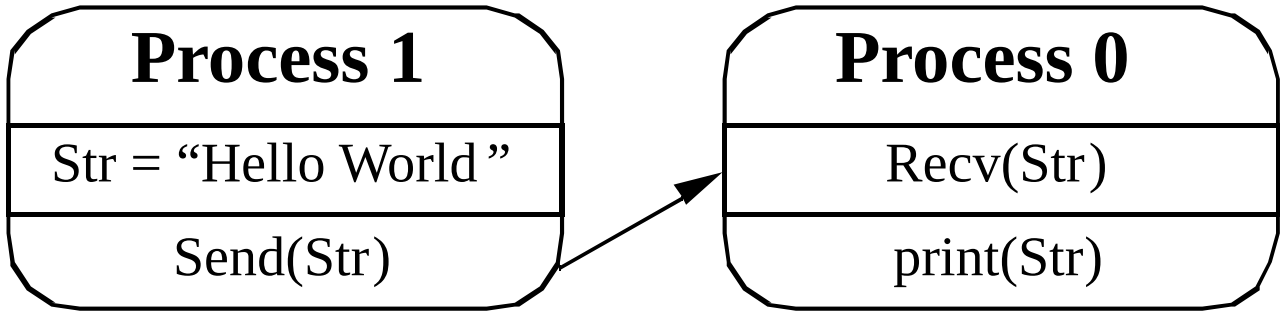
```
MPI_Datatype   MPI_CHAR,MPI_INT,MPI_FLOAT,...
MPI_Comm       MPI_COMM_WORLD
```

Standard Receive

```
int MPI_Recv (void *buf, int count,
MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

- `buf` Address of message im memory
- `count` *Expected* number of elements in message
- `datatype` Data type of message
- **`source`** Origin process of message
- `tag` Generic message tag
- `comm` Communication handler
- `status` Status-Information

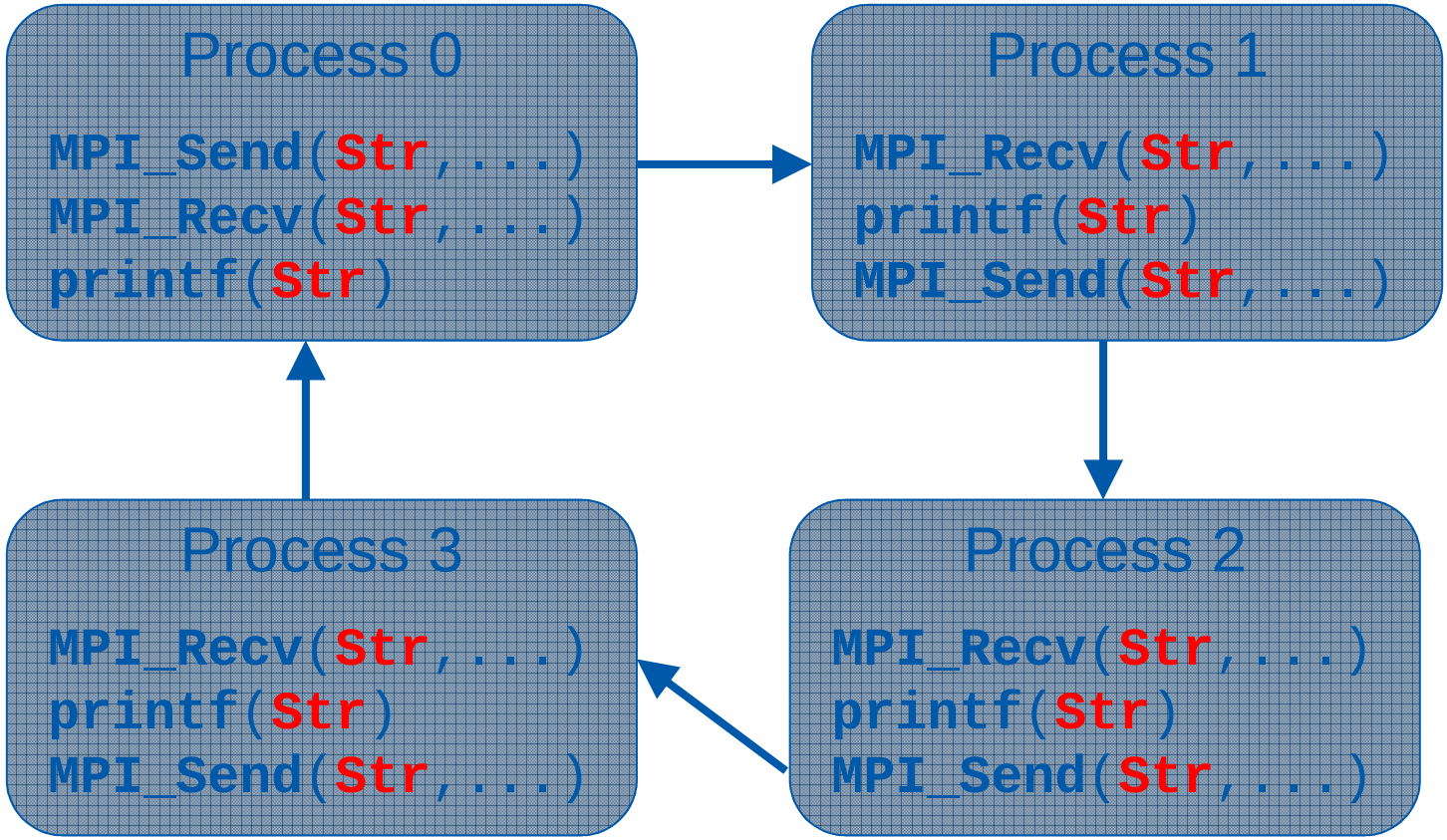
Example: *Hello World*



Example: *Hello World*

```
1.  if (myrank == 1) {
2.     char sendStr[] = “Hello World”;
3.     MPI_Send(sendStr, strlen(sendStr)+1, MPI_CHAR,
4.            0 ,3, MPI_COMM_WORLD );
5.     }
6.  else {
7.     char recvStr[20];
8.     MPI_Recv(recvStr, 20, MPI_CHAR, 1, 3,
9.            MPI_COMM_WORLD, &stat );
10.    printf(“%s\n”,recvStr);
11. }
```

Example: *Round Robin*



Standard Receive

Remark:

Maximum message length is fixed:

- If message is bigger ➔ overflow error
- If message is smaller ➔ unused memory

➔ Allocate sufficient space before calling MPI_Recv

Standard Receive

How many elements have been received?

```
int MPI_Get_count (MPI_Status *status, MPI_Datatype
                   datatype, int *count)
```

Status-Information:

```
1.  struct MPI_Status {
2.     int   MPI_SOURCE;
3.     int   MPI_TAG;
4.     int   MPI_ERROR;
5.     int   count;
6.     ...
7.  };
```

in case of MPI_ANY_SOURCE

in case of MPI_ANY_TAG

Number of Elements

MPI Specifics

- Communicators: Scope of communication operations
- Structure of messages: complex data types
- Data transfer:
  - Synchronous/asynchronous
  - Blocking/non-blocking
- Message tags/identifiers
- Communication partners:
  - Point-to-point
  - Wild card process and message tags

Communicators

Scope of processes

- Communicator group processes
- A group defines the set of processes, that can communicate with each other
- Used in point-to-point and collective communication
- After starting a program, its processes subscribe to the "Universe" ==> `MPI_COMM_WORLD`
- Each program has its own "Universe"

Usage of Communicators

- Fence off communication environment
- Example: Communication in library
  What happens, if a program uses a parallel library that uses MPI itself?
- 2 Kinds of communicators:
  - Intra-communicator: inside a group
  - Inter-communicator: between groups
- Processes in each group are always numbered *0* to *m-1* for *m* processes in a group

MPI Specifics

- Communicators: Scope of communication operations
- Structure of messages: complex data types
- Data transfer:
  - Synchronous/asynchronous
  - Blocking/non-blocking
- Message tags/identifiers
- Communication partners:
  - Point-to-point
  - Wild card process and message tags

Structure of Messages

- Standard data types:
  - Integer, Float, Character, Byte, ...
  - (Continuous) arrays
- Complex data types:
  - Messages including different data: counter + elements
  - Non-continuous data types: sparse matrices
- Solutions:
  - Pack/unpack functions
  - Special (common) data types:
    - Array of data types
    - Array of memory displacements
  - Managed by the message-passing library

Point-to-Point Communication

MPI:

- Data types for message contents:
  - Standard types:
    - `MPI_INT`
    - `MPI_FLOAT`
    - `MPI_CHAR`
    - `MPI_DOUBLE`
    - ...
  - User defined types: derived from standard types

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI Cont.

Based on presentation by Janko Strassburg, BSC

Isaac Rudomin
BSC

Data Transfer

Blocking:

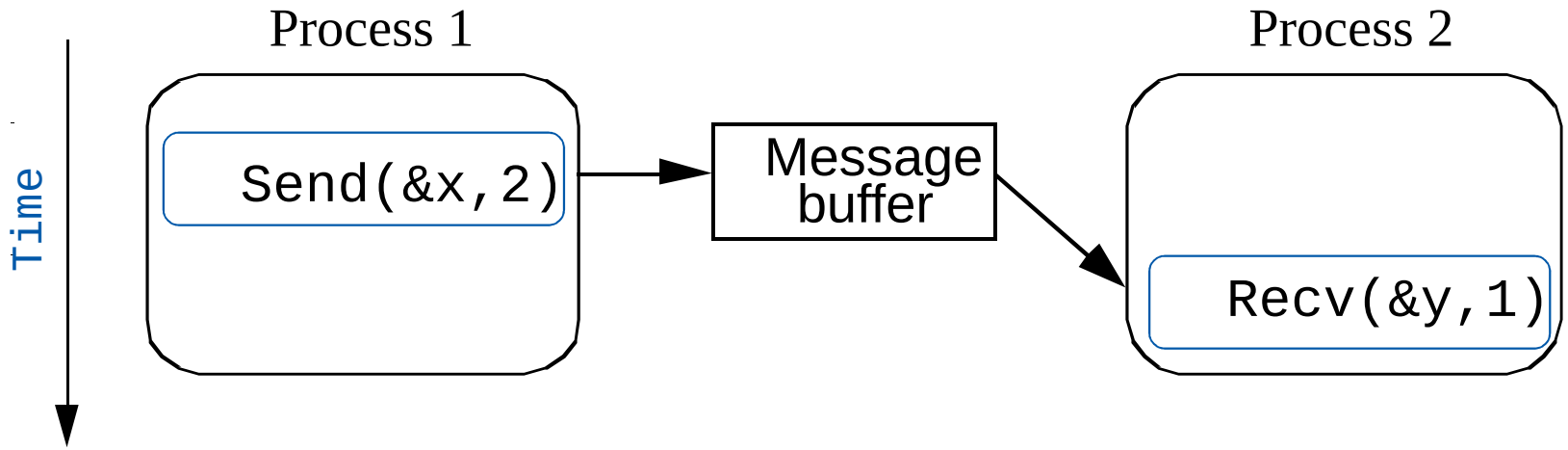
- Function does not return,
  before message can be accessed again
- Process is „blocked“

Non-blocking:

- Function returns, whether data transfer is finished or not
- Requires function to query the status of the data transfer
- Message buffers are needed
  - Length of message is limited
- Overlapping of communication and computation is possible
  - ⇨ Reduction of execution time

Data Transfer with Message Buffer

Non-blocking send:



MPI Viewpoint

Concepts for blocking:

- *Locally blocking:*
  - Function is blocked, until messages has been copied into buffer
  - Transfer needs not be completed

- *Locally non-blocking:*
  - Function returns immediately, whether message has been copied or not
  - User is responsible for message

Standard Send/Receive

- **`MPI_Send`**:
  - Is locally complete as soon as the message is free for further processing
  - The message needs not be received
    ⇨ most likely it will have been transferred to communication buffer

- **`MPI_Recv`**:
  - Is locally complete, as soon as the message has been received

Pitfall: Deadlock

Cyclic message exchange in a ring:

```
if (rank == 0) {
  MPI_Send(buffer,length,MPI_CHAR,1,…);
  MPI_Recv(buffer,length,MPI_CHAR,1,…);
} else if (rank == 1) {
  MPI_Send(buffer,length,MPI_CHAR,0,…);
  MPI_Recv(buffer,length,MPI_CHAR,0,…);
}
```

- Problem: both processes are blocked, since each process is waiting on receive to complete send.
- Cyclic resource-dependencies

Deadlock Solution

No cyclic dependencies:

```
if (rank == 0) {
  MPI_Send(buffer,length,MPI_CHAR,1,…);
  MPI_Recv(buffer,length,MPI_CHAR,1,…);
} else if (rank == 1) {
  MPI_Recv(buffer,length,MPI_CHAR,0,…);
  MPI_Send(buffer,length,MPI_CHAR,0,…);
}
```

Blocking Test

```
int MPI_Probe (int source, int tag
 MPI_Comm comm, MPI_Status *status)
```

- `source` Origin process of message
- `tag` Generic message tag
- `comm` Communication handler
- `status` Status information

- Is locally complete,
  as soon as a message has been received
- Does not return the message,
  but provides only status information about it

MPI_Sendrecv

Performs send and receive in one single function call:

```
MPI_Sendrecv (
 pointer to send buffer                    void *sendbuf,
 size of send message (in elements)        int sendcount,
 datatype of element                       MPI_Datatype sendtype,
 destination                               int dest,
  tag                                      int sendtag,
 pointer to receive buffer                 void *recvbuf,
 size of receive message (in elem.)        int recvcount,
 datatype of element                       MPI_Datatype recvtype,
 source                                    int source,
  tag                                      int recvtag,
  communicator                             MPI_Comm communicator,
 return status                             MPI_Status *status);
```

MPI_Sendrecv_replace

Performs send and receive in one single function call and operates only one one single buffer:

```
MPI_Sendrecv_replace (
 pointer to buffer                 void *buf,
 size of message (in elements)     int count,
 datatype of element               MPI_Datatype type,
 destination                       int dest,
  tag                              int sendtag,
 source                            int source,
  tag                              int recvtag,
  communicator                           MPI_Comm communicator,
 return status                     MPI_Status *status);
```

Non-blocking Functions

- **`MPI_Isend`**:
  - Returns immediately,
    whether function is locally complete or not
  - Message has not been copied
    ⇨ Changes may affect contents of message

- **`MPI_Irecv`**:
  - Returns immediately,
    whether a message has arrived or not

- **`MPI_Iprobe`**:
  - Non-blocking test for a message

Auxiliary Functions

Is an operation completed or not?

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Waits until operation is completed

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_status *status)
```

- Returns immediately.
  `flag` contains status of request (true/false).

Additional Wait-Functions

```
int MPI_Waitany(int count,
MPI_Request *array_of_requests, int *index, MPI_Status
*status)
```

```
int MPI_Waitall(int count,
MPI_Request *array_of_requests,
MPI_Status *status)
```

```
int MPI_Waitsome(int incount,
MPI_Request *array_of_requests, int *outcount, int
*array_of_indices,
MPI_Status *array_of_statuses)
```

Additional Test-Functions

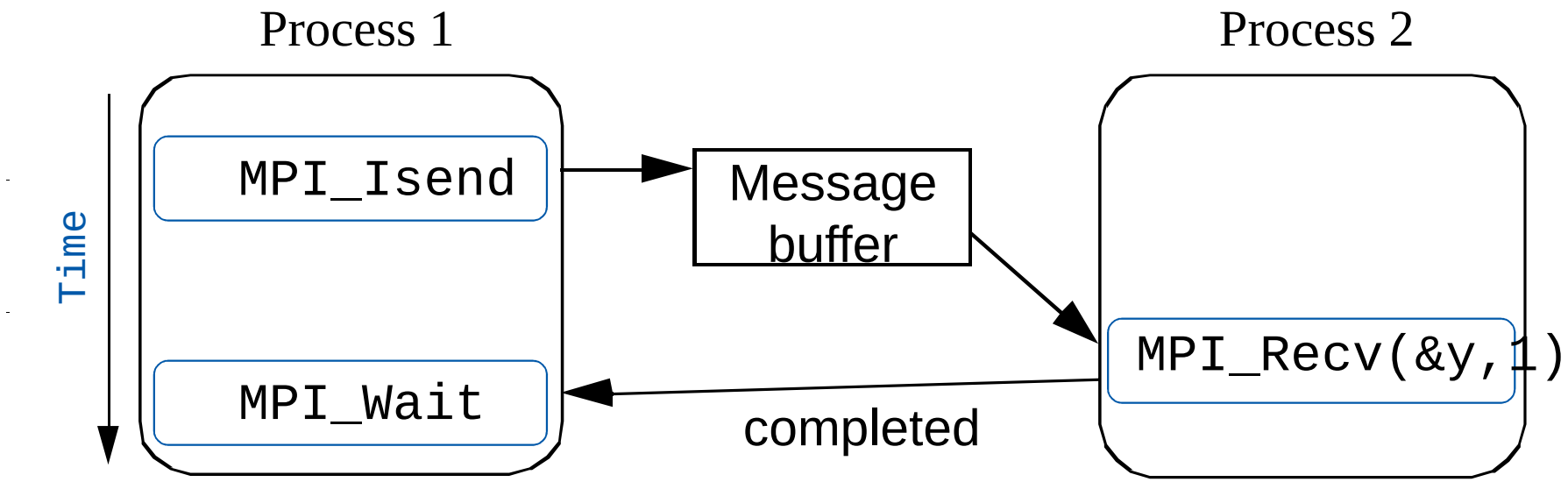
```
int MPI_Testany(int count,
MPI_Request *array_of_requests, int *index,
int *flag, MPI_Status *status)
```

```
int MPI_Testall(int count,
MPI_Request *array_of_requests,
int *flag, MPI_Status *status)
```

```
int MPI_Testsome(int incount,
MPI_Request *array_of_requests, int *outcount, int
*array_of_indices,
MPI_Status *array_of_statuses)
```

Non-Blocking Functions

Example: Overlapping of Computation and Communication



Example: *Overlapping*

```
1.  if (myrank == 0) {
2.     int x;
3.     MPI_Isend(&x,1,MPI_INT,1,3,MPI_COMM_WORLD,
                  req)
4.     compute();
5.     MPI_Wait(req,status);
6.     }
7.  else {
8.     int x;
9.     MPI_Recv(&x,1,MPI_INT,0,3,MPI_COMM_WORLD,
                  stat)
10.    }
```

Additional Send-Modes

Possibilities:

	Blocking	*Non-blocking*
Standard	`MPI_Send`	`MPI_Isend`
Synchronous	`MPI_Ssend`	`MPI_Issend`
Buffered	`MPI_Bsend`	`MPI_Ibsend`
Ready	`MPI_Rsend`	`MPI_Irsend`

Additional Send-Modes

All functions are available blocking & non-blocking

- **Standard Mode**:
  - No assumption about corresponding receive function
  - Buffers depend on implementation
- **Synchronous Mode**:
  - Send/Receive can be started independently but must finish together

Data Transfer

Synchronous communication: *Rendezvous*

- Return from function represents end of transfer
- Message buffers are not required
- Send function waits until receive finished
- Recv function waits until message arrives
- Side effect: synchronization of processes

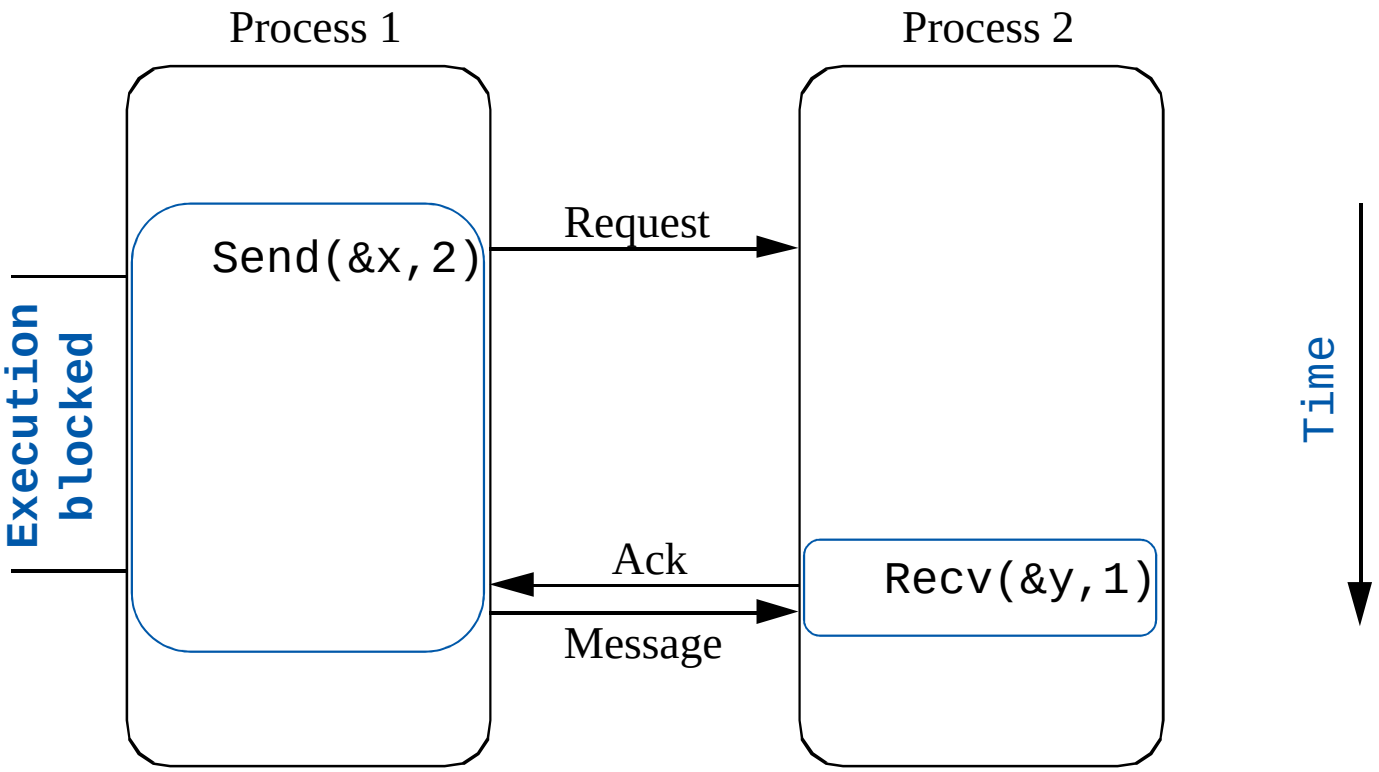
Data Transfer

Asynchronous Communication:

- Send and receive have no temporal connection
- Message buffers are required
- Buffers located at sender or receiver
- Send process does not know,
  whether message actually arrived or not
- Target process may not receive a message

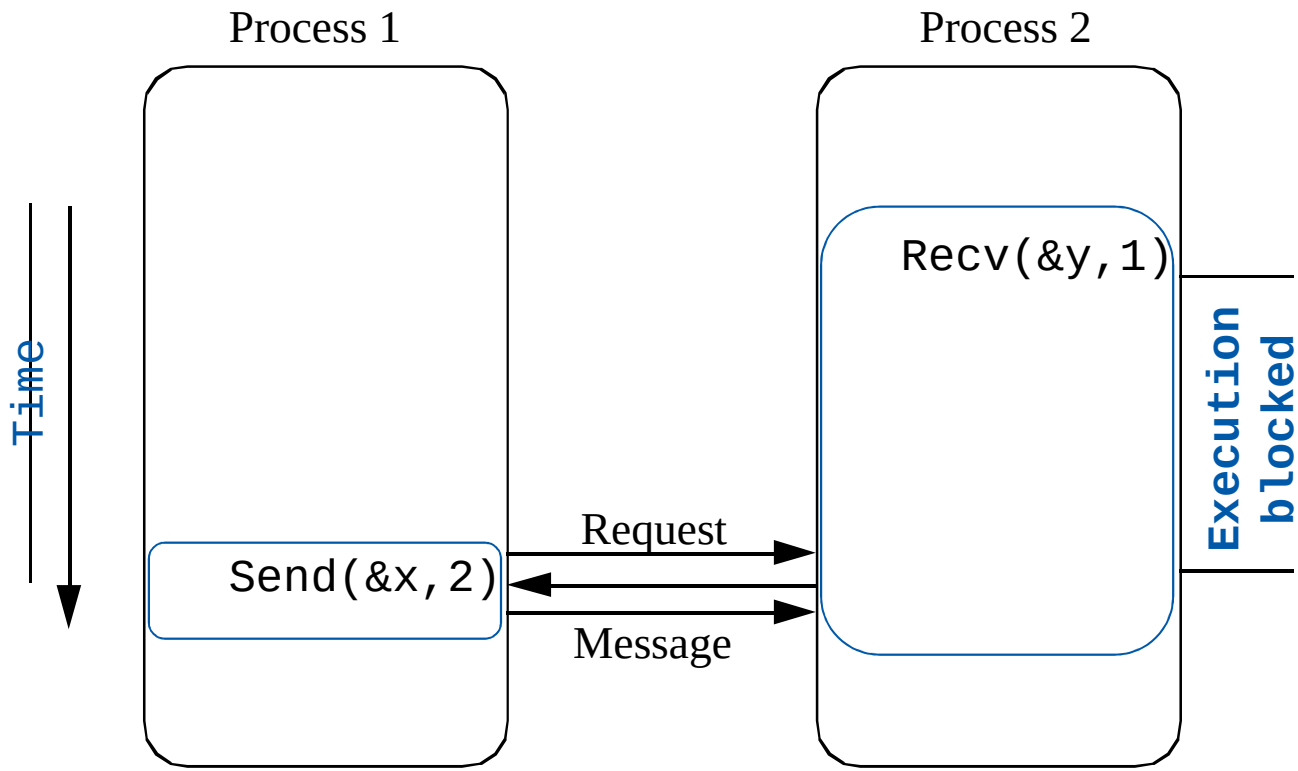
Synchronous Data Transfer

Case 1: Send is called before receive



Synchronous Data Transfer

Case 2: Recv is called before send



Additional Send-Modes

Possibilities:

	Blocking	*Non-blocking*
Standard	`MPI_Send`	`MPI_Isend`
Synchronous	`MPI_Ssend`	`MPI_Issend`
Buffered	`MPI_Bsend`	`MPI_Ibsend`
Ready	`MPI_Rsend`	`MPI_Irsend`

Message Tags

Additional Parameter:

- Identifier for message contents
- Supports distinction of different messages (e.g. commands, data, ...)
- Increases flexibility
- *msgtag* is usually arbitrarily chosen integer

Example:

```
send(&x,2,5) → recv(&y,1,5)
```

Wildcard-Identifiers

Receive-Function:

- Defines message origin and and message tag
- Only corresponding messages are accepted
- All other messages are ignored

Wild card == *Joker*

- Permits messages from arbitrary origin
- Permits messages with arbitrary tag

Wild Card

`recv(&y,a,b)`	origin = a tag = b
`recv(&y,?,b)`	arbitrary origin tag = b
`recv(&y,a,?)`	origin = a arbitrary tag
`recv(&y,?,?)`	arbitrary origin arbitrary tag

Point-to-Point Communication

MPI Specifics:

- *Wild Card* at receive operation:
  - for message origin: `MPI_ANY_SOURCE`
  - for message tag: `MPI_ANY_TAG`

Problem:

Race Conditions/Nondeterminism

Collective Operations

Until now:

- Point-to-point operations ==> 1 Sender, 1 Receiver

Now:

- Functions and operations involving multiple processes

Collective Operations

Possibilities:

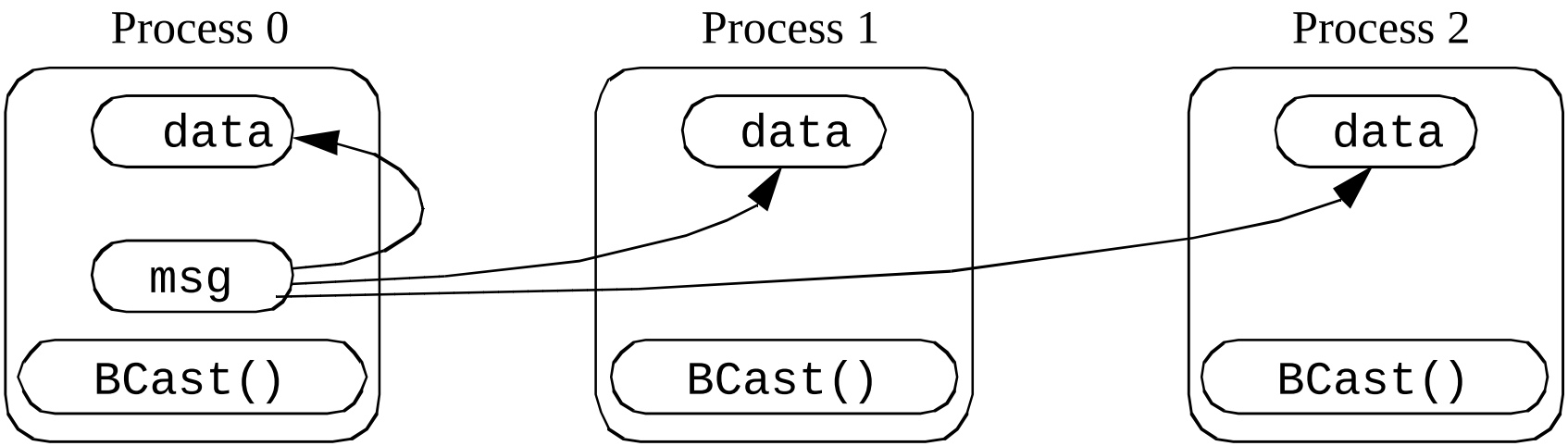
- `MPI_Barrier`: has to be passed by all processes
- `MPI_Bcast`: one process to all others
- `MPI_Gather`: collect data of other processes
- `MPI_Scatter`: distribute data onto other processes
- `MPI_Reduce`: combine data of other processes
- `MPI_Reduce_scatter`: combine and distribute
- ...

Barrier Synchronization

```
int MPI_Barrier(MPI_Comm comm)
```

- Communicator *comm* defines a group of processes, that has to wait
  until each process has arrived at the barrier

Broadcast/Multicast



MPI Broadcast

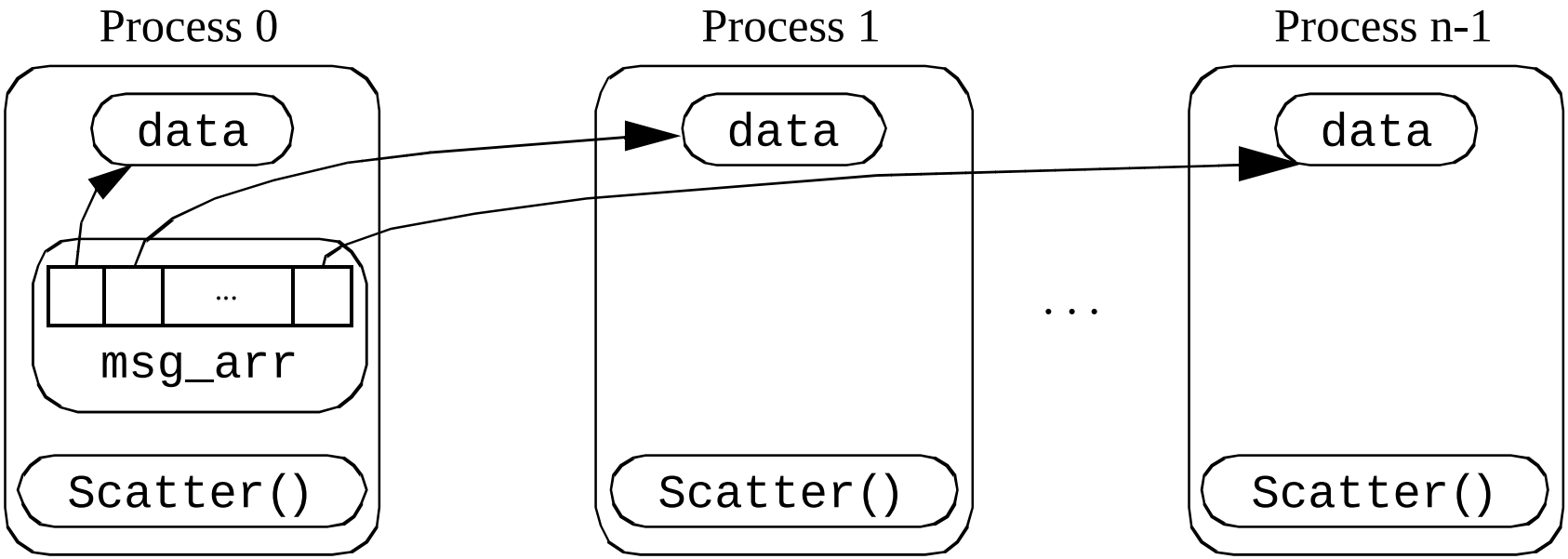
```
int MPI_Bcast(
void *buffer, int count,
 MPI_Datatype datatype,
int root, MPI_Comm comm)
```

- Message *buf* of process *root* is distributed to all processes within communicator *comm*

Scatter

- Distribute the array *msg_arr* of process *root* to all other processes
  - Contents at index i is sent to process i
  - Different implementations possible: Data may be returned to *root*, ...
  - Widely used in SPMD Model

Scatter


Process 0
Process 1
Process n-1
data
data
data
...
msg_arr
Scatter()
Scatter()
Scatter()

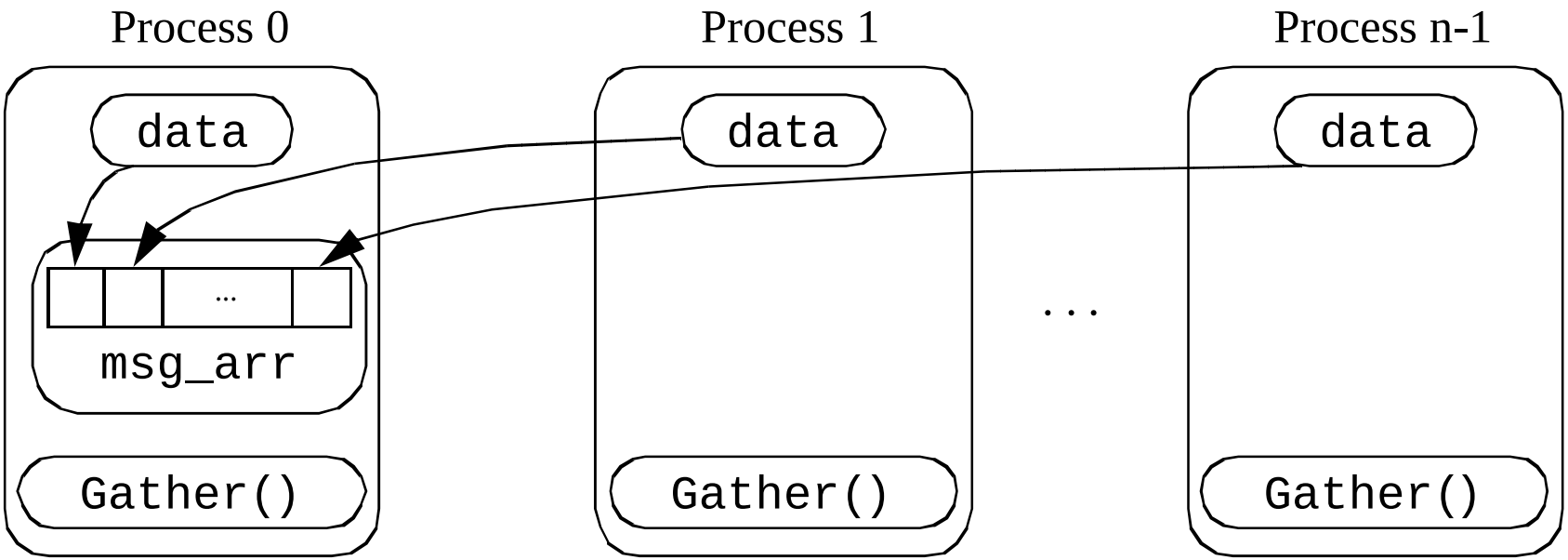
MPI Scatter

```
int MPI_Scatter (
void *sendbuf, int sendcount, MPI_Datatype
                    sendtype,
void *recvbuf, int recvcount, MPI_Datatype
                    recvtype,
              int root, MPI_Comm comm)
```

Gather

- Collect data of all processes on process *root* in array *msg_arr*
  - Data of process i is stored at index i
  - Opposite of Scatter-Operation
  - Usually at the end of a distributed computation
  - Different implementations possible

Gather


Process 0
Process 1
Process n-1
data
data
data
...
msg_arr
Gather()
Gather()
Gather()

MPI Gather

```
int MPI_Gather(
void *sendbuf, int sendcount, MPI_Datatype
                    sendtype,
void *recvbuf, int recvcount, MPI_Datatype
                    recvtype,
              int root, MPI_Comm comm)
```

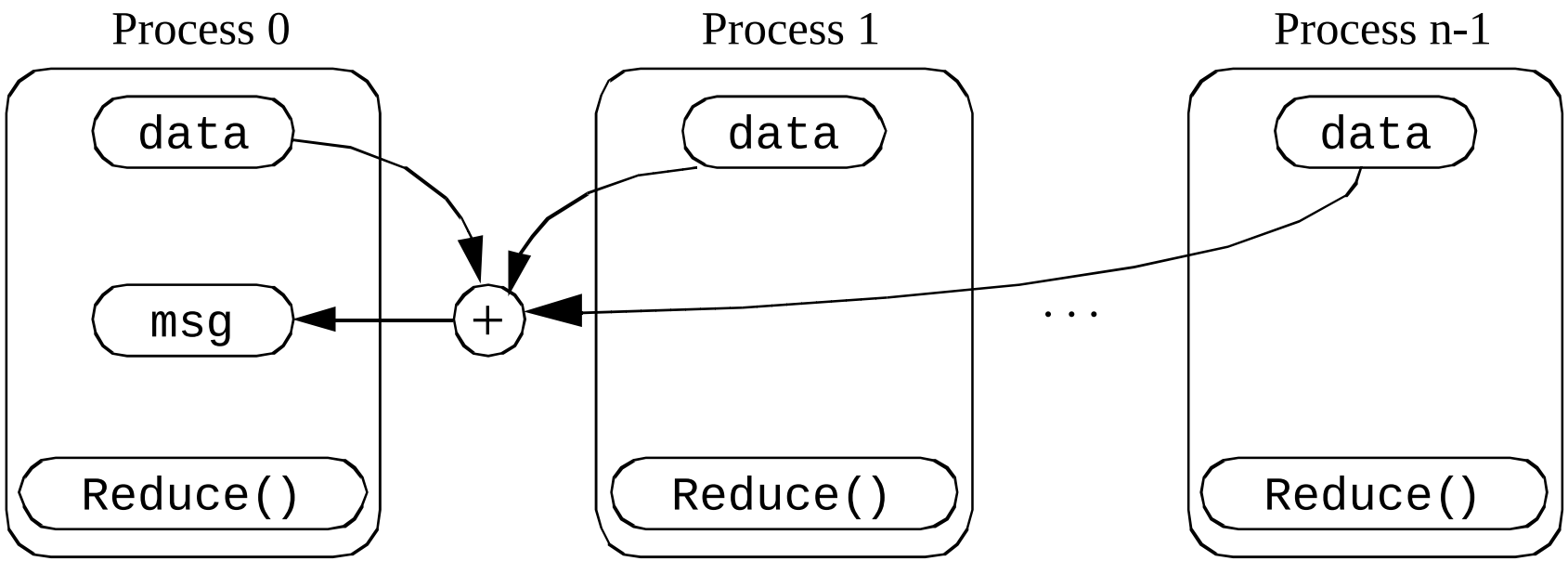
Example: *Data Collection*

```
1.  int data[10];
2.  ...
3.  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
4.  if (myrank == 0) {
5.     MPI_Comm_size(MPI_COMM_WORLD,&grp_size);
6.     buf = (int*)malloc(grpsize*10*sizeof(int));
7.     }
8.  MPI_Gather(data,10,MPI_INT,
9.     buf,grpsize*10,MPI_INT,0,MPI_COMM_WORLD);
```

Reduce

- Global operation on process *root* during data collection
  - Combination of **Gather** + global operation
  - logical or arithmetic operation possible
  - Different implementations possible:
    operation on *root*,
    partial, distributed operations, ...

Reduce


Process 0
Process 1
Process n-1
data
data
data
msg
+
...
Reduce()
Reduce()
Reduce()

MPI Reduce

```
int MPI_Reduce(
    void *sendbuf, void *recvbuf,
int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_COMM comm)
```

Operations:

`MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, …`

Selected Features

- Communicators:
  How to create process groups?
- Topologies:
  How to create virtual topolgies?
- General data types:
  How to use your own data types?

Selected Features

- Communicators:
  How to create process groups?
- Topologies:
  How to create virtual topolgies?
- General data types:
  How to use your own data types?

Communicators

Standard intra-communicator:

- MPI_COMM_WORLD =
  All processes of a program

Functions:

- MPI_Comm_group (comm, group)
- MPI_Group_excl (group, n, ranks, newgroup)
- MPI_Comm_create (comm, group, comm_out)
- MPI_Comm_free (comm)
- ...

Example: Communicator

```
#include <mpi.h>

int main(int argc, char *argv[])
{
    int rank,size;
    int array[8] = {2,3,0,0,0,0,0,0};
    int i,subrank;
    MPI_Status status;
    MPI_Group group;
    MPI_Comm  comm;

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

Example: Communicator

```
...

    MPI_Comm_group(MPI_COMM_WORLD,&group);
    MPI_Group_excl(group,2,array,&group);
    MPI_Group_rank(group,&subrank);
    MPI_Group_size(group,&size);

    MPI_Comm_create(MPI_COMM_WORLD,group,&comm);
    if(subrank != MPI_UNDEFINED) {
       MPI_Gather(&rank,1,MPI_INT,&array,1,
                  MPI_INT,0,comm);
       MPI_Comm_free(&comm);
    }
```

Example: Communicator

```
...

    if(rank == 0) {
       for(i=0;i<size;i++) printf("%d ",array[i]);
       printf("\n");
       }
    MPI_Finalize();
}
```

```
mpirun –np 8 group
0 1 4 5 6 7
```

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI

Based on presentation by Janko Strassburg, BSC

Isaac Rudomin
BSC

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenMP

Based on presentation by Xavier Martorell, BSC

Isaac Rudomin
BSC

OpenMP fundamentals, parallel regions

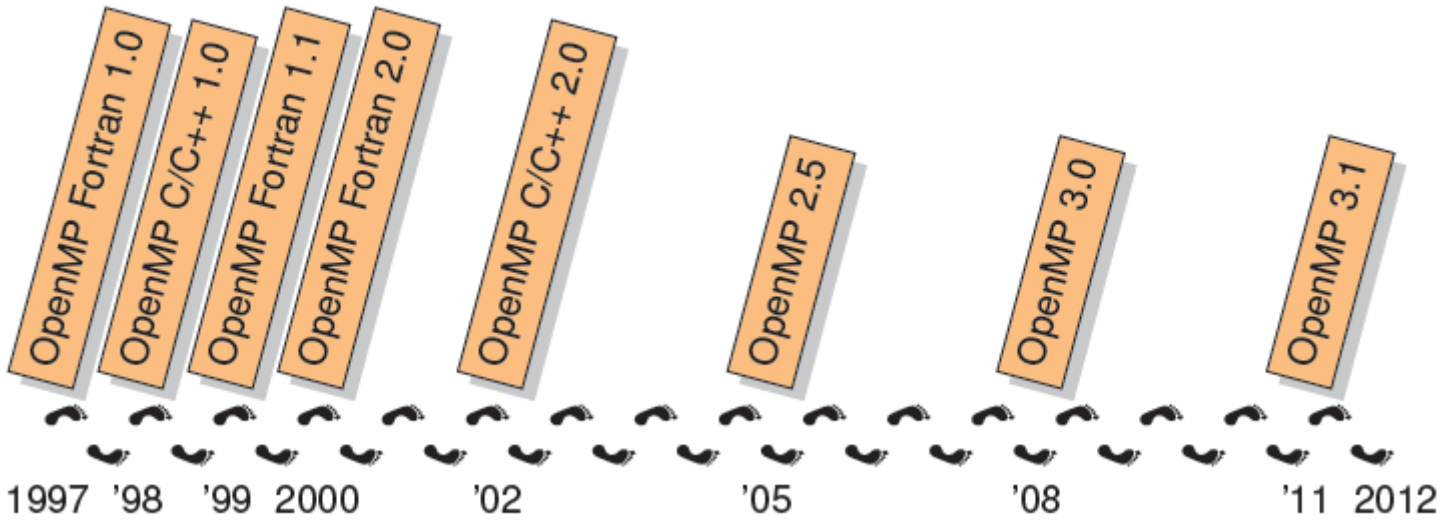
- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- Creating Threads
- Data-sharing attributes

What is OpenMP?

- It's an API extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines
  - Current version is 3.1 (July 2011)
    - ... 4.0 is open for public comments
  - Supported by most compiler vendors
    - Intel, IBM, PGI, Sun, Cray, Fujitsu, HP, GCC...
- Maintained by the Architecture Review Board (ARB), a consortium of industry and academia

  http://www.openmp.org

A bit of history



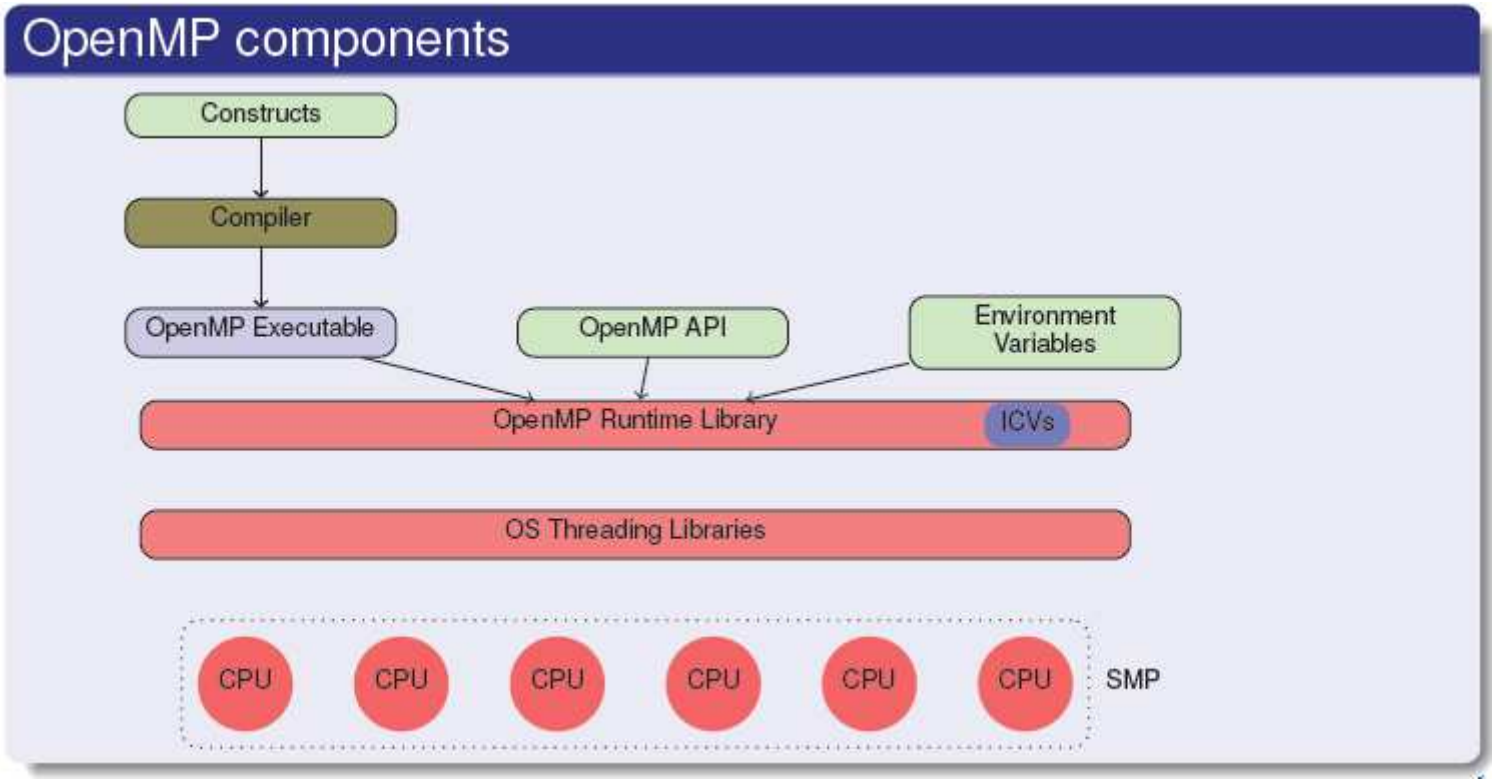
Advantages of OpenMP

- Mature standard and implementations
  - Standardizes practice of the last 20 years
- Good performance and scalability
- Portable across architectures
- Incremental parallelization
- Maintains sequential version
  - (mostly) High level language
- Some people may say a medium level language :-)
- Supports both task and data parallelism
- Communication is implicit

Disadvantages of OpenMP

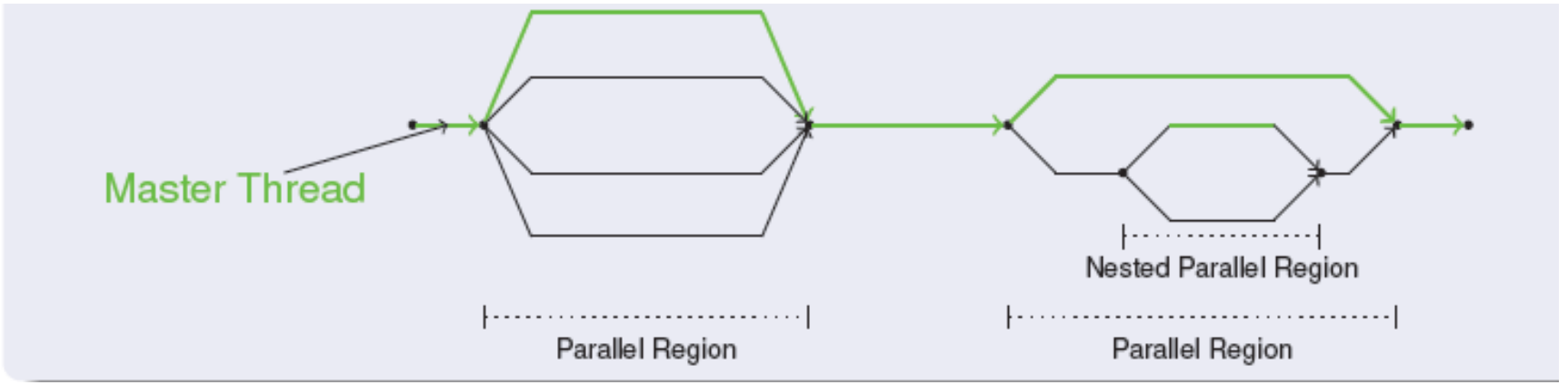
- Communication is implicit
- Flat memory model
- Incremental parallelization creates false sense of glory/failure
- No support for accelerators (...yet, maybe in 4.0)
- No error recovery capabilities (...yet, 4.0)
- Difficult to compose
- Lacks high-level algorithms and structures
- Does not run on clusters

OpenMP at a glance



Execution Model

- OpenMP uses a fork-join model
  - The master thread spawns a team of threads that joins at the end of the parallel region
  - Threads in the same team can collaborate to do work



Memory model

- OpenMP defines a relaxed memory model
  - Threads can see different values for the same variable
  - Memory consistency is only guaranteed at specific points
  - Luckily, the default points are usually enough
- Variables can be shared or private to each thread

OpenMP directives syntax

- In Fortran

Through a specially formatted comment:

s e n t i n e l construct [clauses]

where sentinel is one of:

- `!$OMP` or `C$OMP` or `*$OMP` in fixed format
- `!$OMP` in free format

- In C/C++

Through a compiler directive:

```
#pragma omp construct [ clauses ]
```

OpenMP syntax is ignored if the compiler does not recognize OpenMP (we'll be using C/C++ syntax through this tutorial)

Headers/Macros

- C/C++ only
  - omp.h contains the API prototypes and data types definitions
  - The _OPENMP is defined by the OpenMP enabled compilers
    - Allows conditional compilation of OpenMP
- Fortran only
  - The omp_lib module contains the subroutine and function definitions

Structured Block

- Definition

Most directives apply to a structured block:

- Block of one or more statements
  - One entry point, one exit point
- No branching in or out allowed
- Terminating the program is allowed

The parallel construct

Directive

#pragma omp parallel [clauses]

s t r u c t u r e d block

where clauses can be:

- **num_threads(expression)**
- **if(expression)**
- shared(var-list)
- private(var-list)
- firstprivate(var-list)
- default(none|shared| private | firstprivate)
- reduction(var-list)
- copyin(var-list)

The parallel construct

Specifying the number of threads

- The number of threads is controlled by an internal control variable (**ICV**) called **nthreads-var**
- When a parallel construct is found a parallel region with a maximum of **nthreads-var** is created
  - Parallel constructs can be nested creating nested parallelism
- The **nthreads-var** can be modified through
  - the **omp_set_num_threads** API called
  - the **OMP_NUM_THREADS** environment variable
- Additionally, the **num_threads** clause causes the implementation to ignore the ICV and use the value of the clause for that region

The parallel construct

Avoiding parallel regions

- Sometimes we only want to run in parallel under certain conditions
  - E.g., enough input data, not running already in parallel, ...
- The **if** clause allows to specify an expression. When it evaluates to false the **parallel** construct will only use 1 thread
  - Note that it still creates a new team and data environment

Putting it together

Example

```
void main ( ) {
#pragma omp parallel
 . . .
omp_set_num_threads ( 2 ) ;
#pragma omp parallel
. . .
#pragma omp parallel num_threads( random()%4+1) if
(N>=128)
. . .
}
```

Putting it together

Example

```
void main ( ) {
#pragma omp parallel
 . . . An unknown number of threads here. Use
 OMP_NUM_THREADS
omp_set_num_threads ( 2 ) ;
#pragma omp parallel
. . . A team of two threads here
#pragma omp parallel num_threads( random()%4+1) if
(N>=128)
. . . A team of [1..4] threads here
}
```

API calls

- Other useful routines
- int **omp_get_num_threads**() Returns the number of threads in the current team
- int **omp_get_thread_num**() Returns the id of the thread in the current team
- int **omp_get_num_procs**() Returns the number of processors in the machine
- int **omp_get_max_threads**() Returns the maximum number of threads that will be used in the next parallel region
- double **omp_get_wtime**() Returns the number of seconds since an arbitrary point in the past

Data environment

- A number of clauses are related to building the data environment that the construct will use when executing
- **shared**
- **private**
- **firstprivate**
- **default**
- **threadprivate**
- lastprivate
- reduction
- copyin
- copyprivate

Data-sharing attributes

Shared

- When a variable is marked as **shared**, the variable inside the construct is the same as the one outside the construct
  - In a parallel construct this means all threads see the same variable
    - but not necessarily the same value
  - Usually need some kind of synchronization to update them correctly
    - OpenMP has consistency points at synchronizations
- Example

```
int x =1;
#pragma omp parallel shared( x ) num_threads( 2 )
{
   x++;
   printf("%d\n" , x ) ;
}
printf("%d\n" , x ) ;
```

Prints 2 or 3 (three printfs in total)

Data-sharing attributes

Private

- When a variable is marked as **private**, the variable inside the construct is a new variable of the same type with an undefined value
  - In a parallel construct this means all threads have a different variable
  - Can be accessed without any kind of synchronization
- Example

```
int x =1;
#pragma omp parallel private( x ) num_threads( 2 )
{
   x++;
   printf("%d\n" , x ) ;
}
printf("%d\n" , x ) ;
```

Can print anything (twice, same or different)

Prints 1

Data-sharing attributes

Firstprivate

- When a variable is marked as **firstprivate**, the variable inside the construct is a new variable of the same type but it is initialized to the original value of the variable
  - In a parallel construct this means all threads have a different variable with the same initial value
  - Can be accessed without any kind of synchronization

- Example

```
int x =1;
#pragma omp parallel firstprivate( x ) num_threads( 2 )
{
   x++;
   printf("%d\n" , x ) ;
}
printf("%d\n" , x ) ;
```

Prints 2 twice

Prints 1

Data-sharing attributes

What is the default?

- Static/global storage is **shared**
- Heap-allocated storage is **shared**
- Stack-allocated storage inside the construct is **private**
- Others
  - If there is a **default** clause, what the clause says
    - **none** means that the compiler will issue an error if the attribute is not explicitly set by the programmer
  - Otherwise, depends on the construct
    - For the **parallel** region the default is **shared**

- Example

```
int x , y ;
#pragma omp parallel private( y )
{
  x =                x is shared
  y =                y is private
  #pragma omp parallel private( x )
  {
    x =              x is private
    y =              y is shared
  }
}
```

Data-sharing attributes

Threadprivate storage

Static/global storage is **shared**

```
#pragma omp t h r e a d p r i v a t e ( var-l i s t )
```

- Can be applied to:
  - Global variables
  - Static variables
  - Class-static members
- Allows to create a per-thread copy of “global” variables
- **threadprivate** storage persist across **parallel** regions if the number of threads is the same
- Threadprivate persistence across nested regions is complex

- Example

```
#char foo ( )
{
  s t a t i c char b u f f e r [BUF_SIZE ] ;
  #pragma omp t h r e a d p r i v a t e ( b u f f e r )
        Creates one static copy of buffer per thread
        Now foo can be called by multiple threads at the same time
  . . .
  return  b u f f e r ;          foo returns correct address to caller
}
```

Worksharing constructs

- Worksharing constructs divide the execution of a code region among the threads of a team
  - Threads cooperate to do some work
  - Better way to split work than using thread-ids
  - Lower overhead than using **tasks**
    - But, less flexible
- In OpenMP, there are four worksharing constructs:
  - single
  - loop worksharing
  - section
  - workshare
- Restriction: worksharings cannot be nested

Loop parallelism

The for construct

#pragma omp for [clauses]

for (init -expr ; test -expr ; inc -expr)

where clauses can be:

- private
- firstprivate
- **lastprivate(variable-list)**
- **reduction(operator:variable-list)**
- **schedule(schedule-kind)**
- **nowait**
- **collapse(**n**)**
- ordered

The for construct

- How does it work?

The iterations of the loop(s) associated to the construct are divided among the threads of the team

  - Loop iterations must be independent
  - Loops must follow a form that allows to compute the number of iterations
  - Valid data types for induction variables are: integer types, pointers and random access iterators (in C++)
    - The induction variable(s) are automatically privatized
  - The default data-sharing attribute is **shared**

It can be merged with the **parallel** construct:

```
#pragma omp parallel for
```

The for construct

- Example

```
void foo ( int m, int N, int M)
{
  int i ;
  #pragma omp parallel for private( j )
     New created threads cooperate to execute all the iterations of the loop
   for ( i = 0; i < N; i ++ )  The i variable is automatically privatized
       for ( j = 0; j < M; j ++ )  j Must be explicitly privatized
          m[ i ] [ j ] = 0;
}
```

- Example 2

```
void foo ( std : : vector <int > &v )
{
   #pragma omp parallel for
   for ( std : : vector <int > : : iterator it = v . begin ( ) ;
     random access iterators (and pointers) are valid types
      it < v . end ( ) ;  != cannot be used in the test expression
      it ++ )
  *it = 0;
}
```

Removing dependences

- Example

```
x = 0;
for ( i = 0; i < n ; i ++ )
{
   v [ i ] = x ;
    x += dx ;
}
```

Each iteration x depends on the previous one. Can't be parallelized

- Example 2

```
x = 0;
for ( i = 0; i < n ; i ++ )
{
   x = i * dx ;
   v [ i ] = x ;
}
```

But x can be rewritten in terms of i. Now it can be parallelized

The lastprivate clause

- When a variable is declared **lastprivate**, a private copy is generated for each thread. Then the value of the variable in the last iteration of the loop is copied back to the original variable
  - A variable can be both **firstprivate** and **lastprivate**

The reduction clause

A very common pattern is where all threads accumulate some values into a single variable

- E.g., n += v[i], our heat program, ...
- Using **critical** or **atomic** is not good enough
  - Besides being error prone and cumbersome

Instead we can use the **reduction** clause for basic types

- Valid operators are: +, -, *, |, ||, &, &&,^, min, max
  - User-defined reductions coming soon...
- The compiler creates a **private** copy that is properly initialized
- At the end of the region, the compiler ensures that the **shared** variable is properly (and safely) updated

We can also specify **reduction** variables in the **parallel** construct

The reduction clause

- Example

```
int vector_sum ( int n , int v [ n ] )
{
   int i , sum = 0;
   #pragma omp parallel for reduction ( + : sum)
   {
      for ( i = 0; i < n ; i ++ )
         sum += v [ i ] ;
  }
return sum;
}
```

Private copy initialized here to the identity value

Shared variable updated here with the partial values of each thread

The schedule clause

The **schedule** clause determines which iterations are executed by each thread

- If no **schedule** clause is present then is implementation defined

There are several possible options as schedule:

- **STATIC**
- **STATIC,chunk**
- **DYNAMIC[,chunk]**
- **GUIDED[,chunk]**
- **AUTO**
- **RUNTIME**

The schedule clause

Static schedule

- The iteration space is broken in chunks of approximately size N/num - threads. Then these chunks are assigned to the threads in a Round-Robin fashion

Static, N schedule (Interleaved)

- The iteration space is broken in chunks of size N. Then these chunks are assigned to the threads in a Round-Robin fashion

Characteristics of static schedules

- Low overhead
- Good locality (usually)
- Can have load imbalance problems

The schedule clause

Dynamic, N schedule

- Threads dynamically grab chunks of N iterations until all iterations have been executed. If no chunk is specified, N = 1.

Guided, N schedule

- Variant of **dynamic**. The size of the chunks deceases as the threads grab iterations, but it is at least of size N. If no chunk is specified, N = 1.

Characteristics of dynamic schedules

- Higher overhead
- Not very good locality (usually)
- Can solve imbalance problems

The schedule clause

Auto schedule

- In this case, the implementation is allowed to do whatever it wishes
  - Do not expect much of it as of now

Runtime schedule

- The decision is delayed until the program is run through the **sched-nvar** ICV. It can be set with:
  - The **OMP_SCHEDULE** environment variable
  - The **omp_set_schedule**() API call

The nowait clause

- When a worksharing has a **nowait** clause then the implicit **barrier** at the end of the loop is removed
  - This allows to overlap the execution of non-dependent loops/tasks/worksharings

- Example

```
#pragma omp for nowait
```

First and second loop are independent, so we can overlap them Side note: you would better fuse the loops in this case

```
for ( i = 0; i < n ; i ++ )
   v [ i ] = 0;
#pragma omp for
for ( i = 0; i < n ; i ++ )
    a [ i ] = 0;
```

The nowait clause

Example

`#pragma omp for nowait` First and second loops are dependent! No guarantees that the previous iteration is finished

```
for ( i = 0; i < n ; i ++ )
   v [ i ] = 0;
#pragma omp for
for ( i = 0; i < n ; i ++ )
   a [ i ] = v [ i ] v [ i ] ;
```

The nowait clause

- Exception: static schedules
  - If the two (or more) loops have the same **static** schedule and all have the same number of iterations
- Example

```
#pragma omp for schedule( stat ic , M) nowait
for ( i = 0; i < n ; i ++ )
   v [ i ] = 0;
#pragma omp for schedule( stat ic , M)
for ( i = 0; i < n ; i ++ )
   a [ i ] = v [ i ] v [ i ] ;
```

The collapse clause

- Allows to distribute work from a set of n nested loops
  - Loops must be perfectly nested
  - The nest must traverse a rectangular iteration space

- Example

```
#pragma omp for collapse( 2 )
for ( i = 0; i < N; i ++ )
   for ( j = 0; j < M; j ++ )
      foo ( i , j ) ;
```

i and j loops are folded and iterations distributed among all threads. Both i and j are privatized

Basic Synchronizations

- Why synchronization?

- Mechanisms

Threads need to synchronize to impose some ordering in the sequence of actions of the threads. OpenMP provides different synchronization mechanisms:

- **barrier**
- **critical**
- **atomic**
- taskwait
- ordered
- locks

Thread Barrier

- The barrier construct

`#pragma omp barrier`

  - Threads cannot proceed past a barrier point until all threads reach the barrier AND all previously generated work is completed
  - Some constructs have an implicit **barrier** at the end
    - E.g., the **parallel** construct

- Example

```
#pragma omp parallel
{
   foo ( ) ;
 #pragma omp barrier
```

Forces all foo occurrences to happen before all bar occurrences

```
   bar ( ) ;
}
```

Implicit barrier at the end of the parallel region

Exclusive access

- The critical construct

```
#pragma omp critical [ ( name ) ]
  structured block
```

  - Provides a region of mutual exclusion where only one thread can be working at any given time.
  - By default all critical regions are the same, but you can provide them with names
    - Only those with the same name synchronize

- Example

```
int x =1;
#pragma omp parallel num_threads( 2 )
{
#pragma omp critical
    x++; Only one thread at a time here
}
printf ("%d\n" , x ) ; Prints 3!
```

Exclusive access

- Example

```
int x=1 , y =0;
#pragma omp parallel num_threads( 4 )
{
#pragma omp critical ( x )
```

Different names: One thread can update x while another updates y

```
   x++;
#pragma omp critical ( y )
   y++;
}
```

Exclusive access

- The atomic construct

```
#pragma omp atomic
    expression
```

  - Provides an special mechanism of mutual exclusion to do read & update operations
  - Only supports simple read & update expressions
    - E.g., x += 1, x = x - foo()
  - Only protects the read & update part
    - foo() not protected
  - Usually much more efficient than a **critical** construct
  - Not compatible with **critical**

- Example

```
int x =1;
#pragma omp parallel num_threads( 2 )
{#
pragma omp atomic
   x++; Only one thread at a time updates x here
}
p r i n t f ("%d\n" , x ) ; Prints 3!
```

Exclusive access

- Example

```
int x =1;
#pragma omp parallel num_threads( 2 )
{
#pragma omp critical
```

Different threads can update x at the same time!

```
    x++;
#pragma omp atomic
    x++;
}
p r i n t f ("%d\n" , x ) ;
```

Prints 3,4 or 5 :(

Heat diffusion

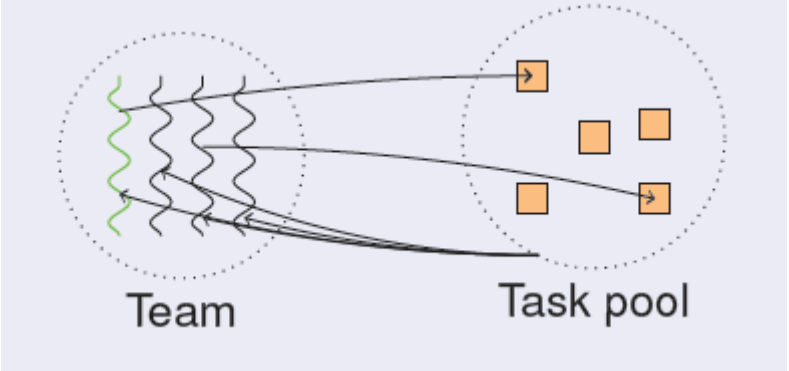
- Parallel loops
- The file solver.c implements the computation of the Heat diffusion

1. Annotate the jacobi, redblack, and gauss functions with OpenMP
2. Execute the application with different numbers of processors

- compare the results
- evaluate the performance

Task Parallelism in OpenMP

- Task parallelism in OpenMP
- Task parallelism model



- Parallelism is extracted from “several” pieces of code
- Allows to parallelize very unstructured parallelism
  - Unbounded loops, recursive functions, ...

Task Parallelism in OpenMP

- What is a task in OpenMP ?
- Tasks are work units whose execution may be deferred
  - they can also be executed immediately
- Tasks are composed of:
  - code to execute
  - a data environment
    - Initialized at creation time
  - internal control variables (ICVs)
- Threads of the team cooperate to execute them

Creating tasks

- The task construct

```
#pragma omp task [ clauses ]
        s t r u c t u r e d block
```

- Where clauses can be:
  - shared
  - private
  - firstprivate
    - Values are captured at creation time
  - default
  - **if(expression)**
  - **untied**

When are tasks created?

- **Parallel** regions create tasks
  - One implicit task is created and assigned to each thread
    - So all task-concepts have sense inside the parallel region
- Each thread that encounters a **task** construct
  - Packages the code and data
  - Creates a new explicit task

Default task data-sharing attributes

- If no default clause
  - Implicit rules apply
    - e.g., global variables are shared
- Otherwise...
  - **firstprivate**
  - **shared** attribute is lexically inherited

Default task data-sharing attributes

- In Practice
- Example

```
int a ;
void foo ( ) {
   int b , c ;
   #pragma omp parallel shared( b )
   #pragma omp parallel private( b )
   {
      int d ;
      #pragma omp task
      {
         int e ;
         a = shared
         b = firstprivate
         c = shared
         d = firstprivate
         e = private
} } }
```

Tip: default(none) is your friend if you do not see it clearly

List traversal

- Example

```
void traverse_list ( List l )
{
   Element e ;
   for ( e = l->first ; e ; e = e->next )
      #pragma omp task
         process ( e ) ; e is firstprivate
}
```

Task synchronization

- There are two main constructs to synchronize tasks:
  - **barrier**
    - Remember: all previous work (including tasks) must be completed
  - **taskwait**
- The taskwait construct

```
#pragma omp taskwait
```

Suspends the current task until all children tasks are completed

  - Just direct children, not descendants

List Traversal

- Example

```
void traverse_list ( List l )
{
Element e ;
for ( e = l->first ; e ; e = e->next )
   #pragma omp task
      process ( e ) ;
#pragma omp taskwait
}
```

All tasks guaranteed to be completed here:

Now we need some threads to execute the tasks

- Example 2

```
List  l
#pragma omp parallel
   traverse_list( l ) ;
```

This will generate multiple traversals;

We need a way to have a single thread execute traverse_list

Giving work to just one thread

The single construct

```
#pragma omp single [ clauses ]
    s t r u c t u r e d block
```

- where clauses can be:
  - private
  - firstprivate
  - nowait
  - copyprivate
- Only one thread of the team executes the structured block
- There is an implicit **barrier** at the end

- Example

```
int main ( int argc , char argv )
{
   #pragma omp parallel
   {
      #pragma omp single
         {
           printf ("Hello world!\n" ) ;
}}}
```

This program outputs just one “Hello world”

List traversal

- Example

```
List l
#pragma omp parallel
#pragma single
  traverse_list ( l ) ;
```

One thread creates the tasks of the traversal

All threads cooperate to execute them

Task scheduling

Tasks are tied by default

- Tied tasks are executed always by the same thread
  - Not necessarily the creator
- Tied tasks have scheduling restrictions
  - Deterministic scheduling points (creation, synchronization, ...)
    - Tasks can be suspended/resumed at these points
  - Another constraint to avoid deadlock problems
- Tied tasks may run into performance problems

The untied clause

- A task that has been marked as **untied** has none of the previous scheduling restrictions:
  - Can potentially switch to any thread
  - Can potentially switch at any moment
  - Bad mix with thread based features
    - thread-id, critical regions, threadprivate
  - Gives the runtime more flexibility to schedule tasks

Task scheduling

The if clause

- If the the expression of an **if** clause evaluates to false
  - The encountering task is suspended
  - The new task is executed immediately
    - with its own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
  - Allows implementations to optimize task creation
    - For very fine grain task you may need to do your own if

Common tasking problems

Search problem

```
void search ( int n , int j , bool s ta t e )
{
   int i , res ;
   if ( n == j ) {
     /* good solution , count it */
     solutions ++;
     return ;
   }
   /* try each possible solution */

   for ( i = 0; i < n ; i ++)
   {
      state [ j ] = i ;
      if ( ok ( j +1 , state ) ) {
         search ( n , j +1 , state ) ;
}}}
```

Common tasking problems

Search problem

```
void search ( int n , int j , bool s ta t e )
{
   int i , res ;
   if ( n == j ) {
     /* good solution , count it */
     solutions ++;
     return ;
   }
   /* try each possible solution */
   #pragma omp task
   for ( i = 0; i < n ; i ++)
   {
      state [ j ] = i ;
      if ( ok ( j +1 , state ) ) {
         search ( n , j +1 , state ) ;
}}}
```

- Data scoping: Because it's an orphaned task all variables are firstprivate
- State is not captured: Just the pointer is captured not the pointed data

Common tasking problems

Search problem

```
void search ( int n , int j , bool s ta t e )
{
   int i , res ;
   if ( n == j ) {
     /* good solution , count it */
     solutions ++;
     return ;
   }
   /* try each possible solution */
   #pragma omp task
   for ( i = 0; i < n ; i ++)
   {
      state [ j ] = i ;
      if ( ok ( j +1 , state ) ) {
         search ( n , j +1 , state ) ;
}}}
```

- Problem #1: Incorrectly capturing pointed data
- firstprivate does not allow to capture data through pointers
- Solutions

1 Capture it manually

2 Copy it to an array and capture the array with firstprivate

Common tasking problems

Search problem

```
void search ( int n , int j , bool s ta t e )
{
   int i , res ;
   if ( n == j ) {
     /* good solution , count it */
     solutions ++;
     return ;
   }
   /* try each possible solution */
   #pragma omp task
   for ( i = 0; i < n ; i ++)
   {
      bool new_state = a l l o c a ( sizeof ( bool )n ) ;
      memcpy( new_state , state , sizeof ( bool )n ) ;
      new_state [ j ] = i ;
      if ( ok ( j +1 , new_state ) ) {
        search ( n , j +1 , new_state ) ;
}}}
```

- Caution! Will new_state still be valid by the time memcpy is executed?
- Problem #2: Data can go out of scope!

Common tasking problems

- Problem: Stack-allocated parent data can become invalid before being used by child tasks
  - Only if not captured with firstprivate
- Solutions

1 Use firstprivate when possible
2 Allocate it in the heap: Not always easy (we also need to free it)
3 Put additional synchronizations: May reduce the available parallelism

Search problem

```
void search ( int n , int j , bool s ta t e )
{
   int i , res ;
   if ( n == j ) {
     /* good solution , count it */
     solutions ++; Shared variable needs protected access
     return ;
   }
   /* try each possible solution */
   #pragma omp task
   for ( i = 0; i < n ; i ++)
   {
       bool new_state = a l l o c a ( sizeof ( bool )n ) ;
      memcpy( new_state , state , sizeof ( bool )n ) ;
      new_state [ j ] = i ;
      if ( ok ( j +1 , new_state ) ) {
        search ( n , j +1 , new_state ) ;
   }}
 #pragma omp taskwait
 }
```

- Solutions: Use **critical,** Use **atomic,** Use **threadprivate**

Common tasking problems

- Example

```
int solution s =0;
int mysolutions=0;
#pragma omp threadprivate (mysolutions)
void start_sear ch ( )
   {
   #pragma omp parallel
      {
         #pragma omp single
         {
           bool initial_state [ n ] ;
           search ( n ,0 , initial_state ) ;
         }
         #pragma omp atomic
           solutions += mysolutions ;
      }
   }
```

Use a separate counter for each thread

Accumulate them at the end

Common tasking problems

- Example

```
void search ( int n , int j , bool *state )
{
   int i , res ;
   if ( n == j ) {
   /* good solution , count it */
   mysolutions++;
   return ;
   }
   /* try each possible solution */
   for ( i = 0; i < n ; i ++)
      #pragma omp task
      {
       bool *new_state = a l l o c a ( sizeof ( bool )n ) ;
       memcpy( new_state , state , sizeof ( bool )n ) ;
       new_state [ j ] = i ;
       if ( ok ( j +1 , new_state ) ) {
          search ( n , j +1 , new_state ) ;
       }
   }
   #pragma omp taskwait
}
```

Programming using a hybrid MPI/OpenMP approach

- Alternatives
- MPI + computational kernels in OpenMP
  - Use OpenMP directives to exploit parallelism between communication phases
    - OpenMP parallel will end before new communication calls
- MPI inside OpenMP constructs
  - Call MPI from within for-loops, or tasks
    - MPI needs to support multi-threaded mode
- MPI compiler driver gets the proper OpenMP option
  - mpicc -openmp
  - mpicc -fopenmp

Practical: heat diffusion

- Heat diffusion using OpenMP tasks

Enter the OpenMP directory to do the following exercises

  - part2 contains the version to be annotated with tasks
  - part3 contains the multisort example to be annotated with tasks

- MPI+OpenMP Heat diffusion

  Parallel loops

  - The file solver.c implements the computation of the Heat diffusion
    1 Use MPI to distribute the work across nodes
    2 Annotate the jacobi, redblack, and gauss functions with OpenMP tasks
    3 Execute the application with different numbers of nodes/processors, and compare the results

www.bsc.es

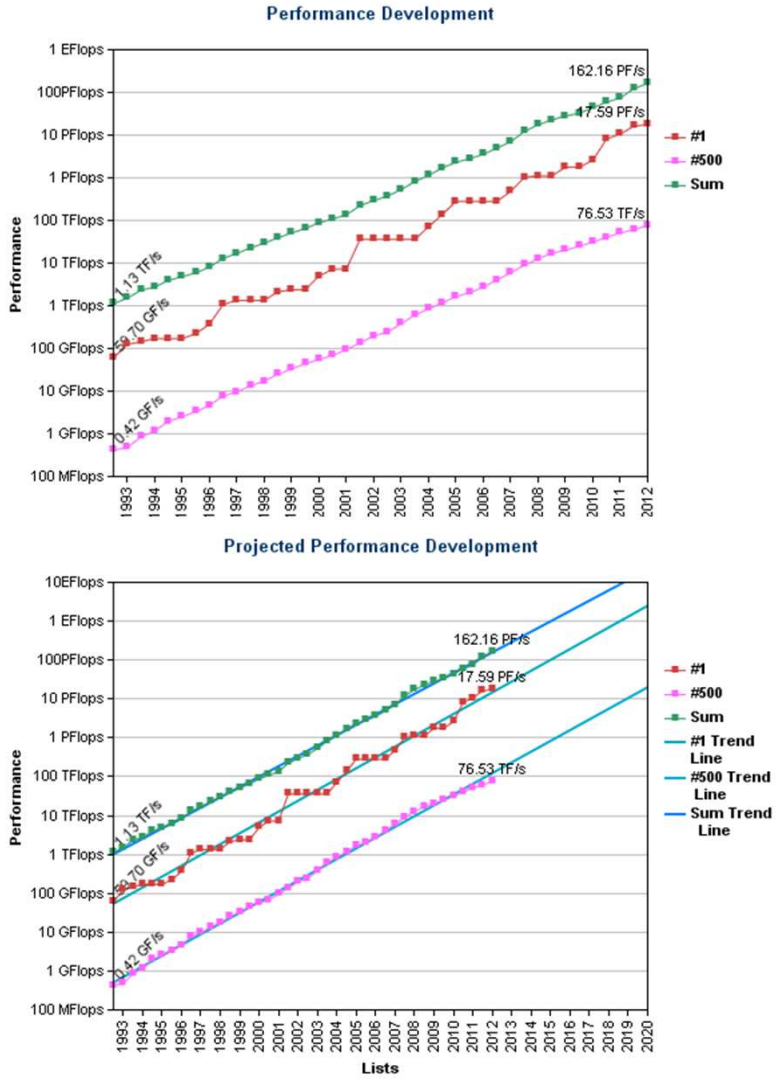
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs

Based on presentation by Rosa M Badia, Xavier Martorell

Isaac Rudomin
BSC

Evolution of computers

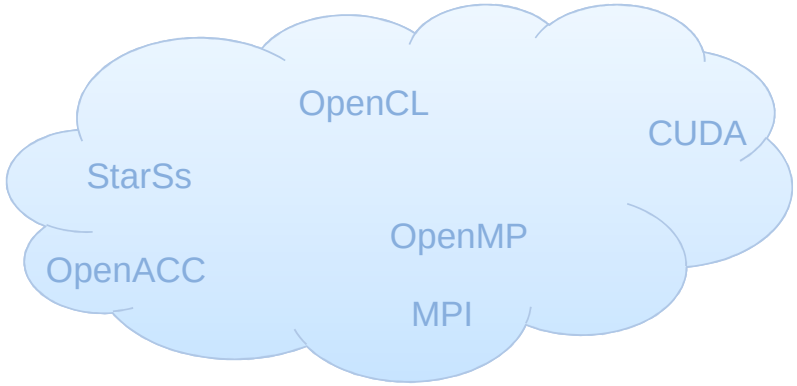


All include multicore or GPU/accelerators

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
2	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	16324.8	20132.7	7890
3	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
4	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8162.4	10066.3	3945
5	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393216	4141.2	5033.2	1970
6	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147456	2897.0	3185.1	3423
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi Dell	204900	2660.3	3959.0	
8	National Supercomputing Center in Tianjin China	**Tianhe-1A** - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT	186368	2566.0	4701.0	4040
9	CINECA Italy	**Fermi** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	163840	1725.5	2097.2	822
10	IBM Development Engineering United States	**DARPA Trial Subset** - Power 775, POWER7 8C 3.836GHz, Custom Interconnect IBM	63360	1515.0	1944.4	3576

Parallel programming models

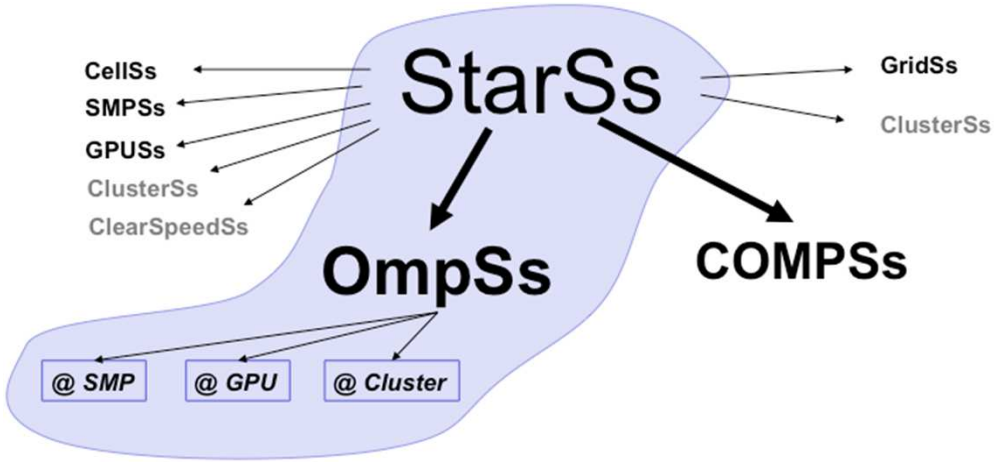
- Traditional programming models
  - Message passing (MPI)
  - OpenMP
  - Hybrid MPI/OpenMP
- Heterogeneity
  - CUDA
  - OpenCL
  - OpenACC
- ...



Simple programming paradigms that enable easy application development are required

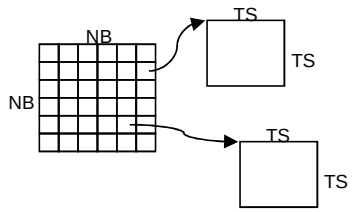
StarSs principles

- StarSs: a family of **task based** programming models
  - Basic concept: **write sequential on a flat single address space** + **directionality annotations**
    - Order **IS** defined
    - Dependence and data access related **information** (NOT specification) in a single mechanism
    - Think global, specify local
    - Intelligent runtime



Example: Cholesky OpenMP

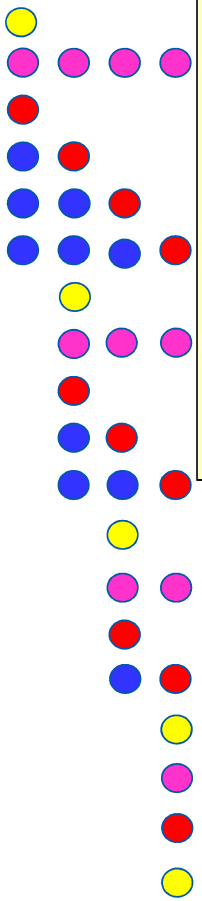
```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
         ssyrk (A[k*NT+i], A[i*NT+i]);
   }
}
```




```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]);
      #pragma omp parallel for
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++) {
            #pragma omp task
            sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
         }
         #pragma omp task
         ssyrk (A[k*NT+i], A[i*NT+i]);
         #pragma omp taskwait
      }
   }
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Execution in OpenMP



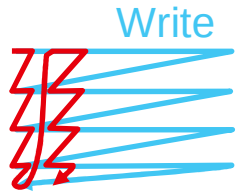
```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]);
      #pragma omp parallel for
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      for (i=k+1; i<NT; i++) {
         #pragma omp parallel for
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
         ssyrk (A[k*NT+i], A[i*NT+i]);
   }
}
```

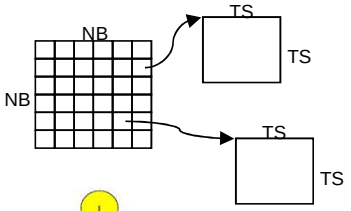
```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]);
      #pragma omp parallel for
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         #pragma omp task
         {
         #pragma omp parallel for
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
          }
         #pragma omp task
         ssyrk (A[k*NT+i], A[i*NT+i]);
         }
         #pragma omp taskwait
  }}
```

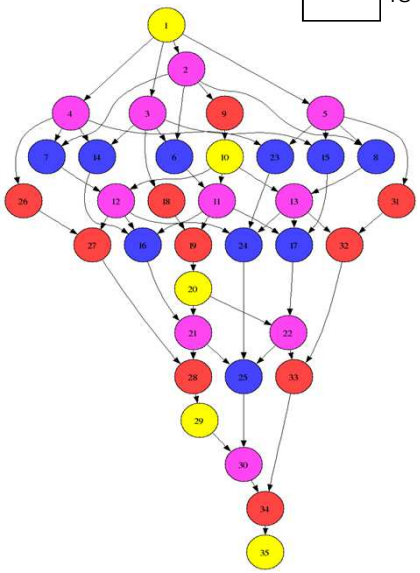
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

StarSs: data-flow execution of sequential programs

Decouple
how we write
form
how it is executed





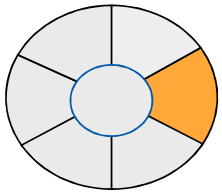


```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
         ssyrk (A[k*NT+i], A[i*NT+i]);
   }
}
```

```
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A);
#pragma omp task input ([TS][TS]T) input ([TS][TS]B)
void strsm (float *T, float *B);
#pragma omp task input ([TS][TS]A,[TS][TS]B) inout ([TS][TS]C )
void sgemm (float *A, float *B, float *C);
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs syntax

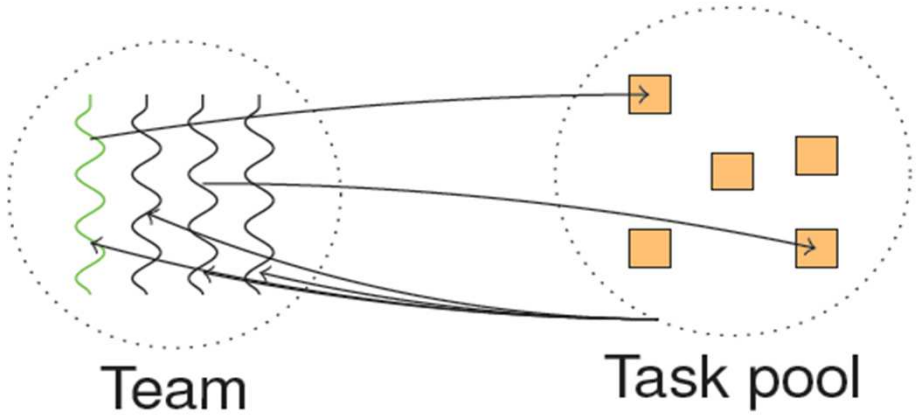


OmpSs = OpenMP + StarSs extensions

- OmpSs is based on OpenMP + StarSs with some differences:
  - Different execution model
  - Extended memory model
  - Extensions for point-to-point inter-task synchronizations
    - data dependencies
  - Extensions for heterogeneity
  - Other minor extensions

Execution Model

- Thread-pool model
  - OpenMP parallel “ignored”
- All threads created on startup
  - One of them starts executing main
- All get work from a task pool
  - And can generate new work



OmpSs: Directives

Task implementation for a GPU device
The compiler parses CUDA/OpenCL kernel invocation syntax

Provides configuration for CUDA/OpenCL kernel

```
#pragma omp target device ({ smp | cuda | opencl })     \
                [ndrange (…)]\
                [ implements ( function_name )]          \

                { copy_deps | [ copy_in ( array_spec ,...)] [ copy_out (...)] [ copy_inout (...)] }
```

Support for multiple implementations of a task

To compute dependences

Ask the runtime to ensure data is accessible in the address space of the device

```
#pragma omp task [ input (...)] [ output (...)] [ inout (...)] [ concurrent (...)] [ commutative (…)] [priority(…)]

{ function or code block }
```

To set priorities to tasks

To relax dependence order allowing concurrent execution of tasks

To relax dependence order allowing change of order of execution of commutative tasks

```
#pragma omp taskwait [on (...)] [noflush]
```

Wait for sons or specific data availability

Relax consistency to main program

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs: Directives

Alternative syntax towards new OpenMP dependence specification

#pragma omp task [in (...)] [out (...)] [inout (...)] [concurrent (...)] [commutative (…)] [priority(…)]

{ function or code block }

To relax dependence order allowing concurrent execution of tasks

To relax dependence order allowing change of order of execution of commutative tasks

To set priorities to tasks

OpenMP: Directives

OpenMP dependence specification

```
#pragma omp task [ depend (in: …)] [ depend(out:…)] [ depend(inout:...)]
{ function or code block }
```

Direct contribution of BSC at OpenMP promoting dependences and heterogeneity clauses

Main element: tasks

- Task
  - Computation unit. Amount of work (granularity) may vary in a wide range (µsecs to msecs or even seconds), may depend on input arguments,…
  - Once started can execute to completion independent of other tasks
  - Can be declared inlined or outlined
- States:
  - **Instantiated**: when task is created. Dependences are computed at the moment of instantiation. At that point in time a task may or may not be ready for execution
  - **Ready**: When all its input dependences are satisfied, typically as a result of the completion of other tasks
  - **Active**: the task has been scheduled to a processing element. Will take a finite amount of time to execute.
  - **Completed**: the task terminates, its state transformations are guaranteed to be globally visible and frees its output dependences to other tasks.

Main element: inlined tasks

- Pragmas inlined
  - Applies to a statement
  - The compiler outlines the statement (as in OpenMP)

```
int   main   (    )
{
   int X[100];

   #pragma  omp   task
   for (int i =0; i< 100; i++) X[i]=i;
#pragma omp taskwait

   ...
}
```

Main element: inlined tasks

- Pragmas inlined
  - Standard OpenMP clauses private, firstprivate, ... can be used

```
int   main   (    )
{
   int X[100];

   int i=0;
   #pragma omp task firstprivate (i)
   for ( ; i< 100; i++) X[i]=i;
}
```

```
int   main   (    )
{
   int X[100];

   int i;
   #pragma omp task private(i)
   for (i=0; i< 100; i++) X[i]=i;
}
```

Main element: outlined tasks

- Pragmas outlined: attached to function definition
  - All function invocations become a task
  - The programmer gives a name, this enables later to provide several implementations

```
#pragma  omp   task
void   foo (int Y[size], int size) {
int j;

   for (j=0; j < size; j++) Y[j]= j;
}

int main()
{
int X[100];

foo (X, 100) ;
#pragma omp taskwait
...
}
```

Main element: outlined tasks

- Pragmas attached to function definition
  - The semantic is capture value
    - For scalars is equivalent to firstprivate
    - For pointers, the address is captured

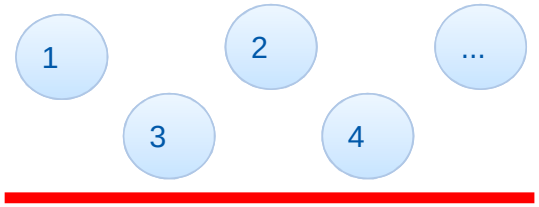
```
#pragma  omp   task
void   foo (int Y[size], int size) {
int j;

   for (j=0; j < size; j++) Y[j]= j;
}

int main()
{
int X[100];

foo (X, 100) ;
#pragma omp taskwait
...
}
```

Synchronization

#pragma omp taskwait

- Suspends the current task until all children tasks are completed

```
void traverse_list ( List l )
{
   Element e ;
   for ( e = l-> first; e ; e = e->next )
   #pragma omp task
    process ( e ) ;

#pragma omp taskwait
}
```



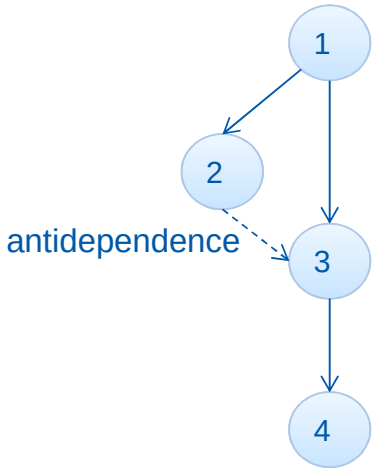
Without taskwait the subroutine will return immediately after spawning the tasks allowing the calling function to continue spawning tasks

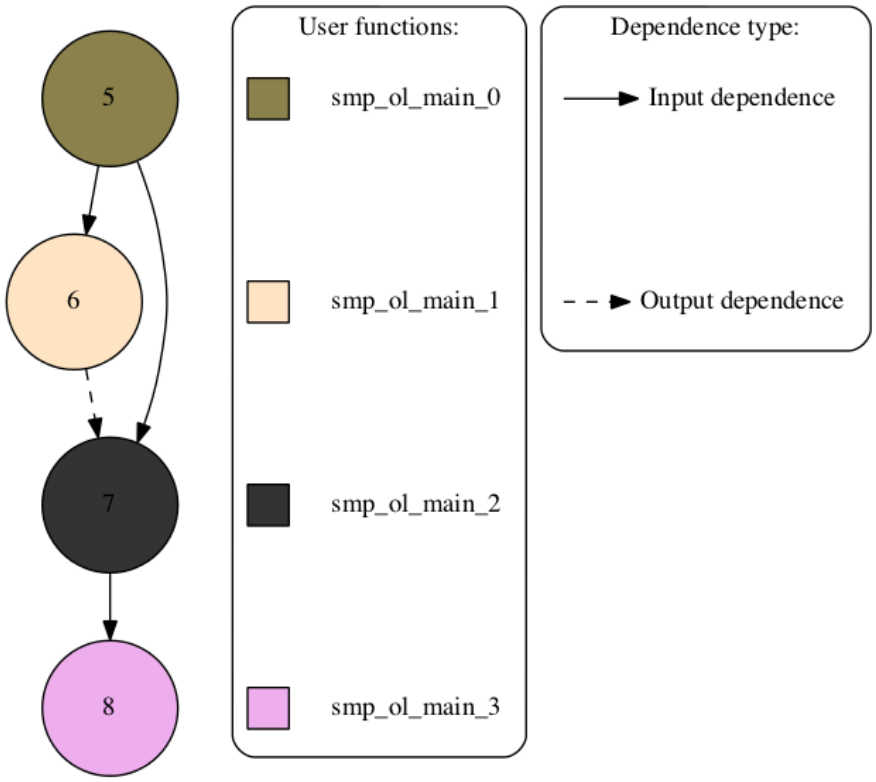
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Defining dependences

- Clauses that express data direction:
  - in
  - out
  - inout
- Dependences computed at runtime taking into account these clauses

```
#pragma omp task output( x )
x = 5;                          //1
#pragma omp task input( x )
printf("%d\n" , x ) ;           //2
#pragma omp task inout( x )
x++;                            //3
#pragma omp task input( x )
printf ("%d\n" , x ) ;          //4
```

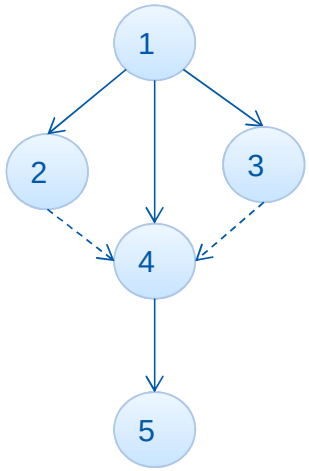


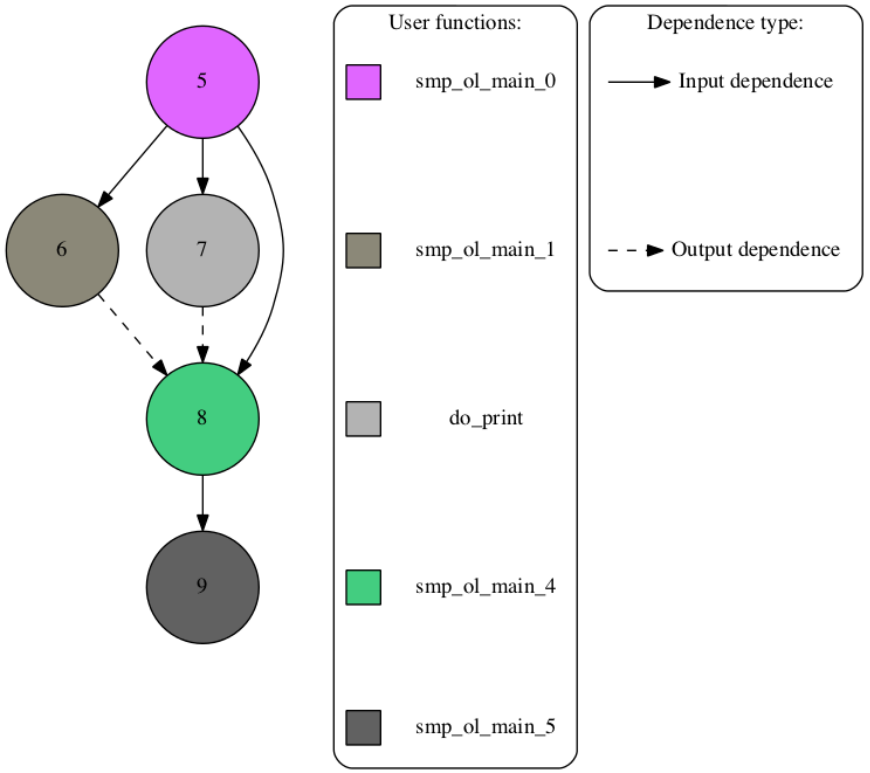

5
6
7
8
User functions:
smp_ol_main_0
smp_ol_main_1
smp_ol_main_2
smp_ol_main_3
Dependence type:
Input dependence
Output dependence

Defining dependences

non-taskified: executed sequentially

```
#pragma omp task input (*px)
void do_print (int *px) {
   printf("from do_print %d\n" , *px ) ;
}

int main()
{
int x;

   x=3;

   #pragma omp task output( x )
   x = 5;                                  //1
   #pragma omp task input( x )
   printf("from main %d\n" , x );          //2
   do_print(&x);                           //3
   #pragma omp task inout( x )
   x++;                                    //4
   #pragma omp task input( x )
   printf ("from main %d\n" , x ); //5
}
```



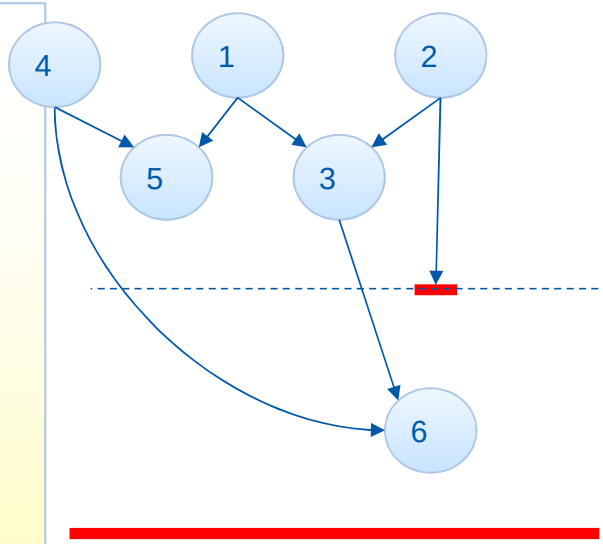

5
6
7
8
9
User functions:
smp_ol_main_0
smp_ol_main_1
do_print
smp_ol_main_4
smp_ol_main_5
Dependence type:
Input dependence
Output dependence

Synchronization

#pragma taskwait on (expression)

- Expressions allowed are the same as for the dependency clauses
- Blocks the encountering task until the data is available

```
#pragma omp task input([N][N]A, [N][N]B) inout([N][N]C)
void dgemm(float *A, float *B, float *C);
main() {
(
 ...
dgemm(A,B,C); //1
dgemm(D,E,F); //2
dgemm(C,F,G); //3
dgemm(A,D,H); //4
dgemm(C,H,I); //5

#pragma omp taskwait on (F)
prinft (“result F = %f\n”, F[0][0]);


dgemm(H,G,C); //6


#pragma omp taskwait
prinft (“result C = %f\n”, C[0][0]);
}
```



Task directive: array regions

- Indicating as input/output/inout subregions of a larger structure:
  - **input (A[i])**
    - → the input argument is element *i* of A
- Indicating an array section:
  - **input ([BS]A)**
    - → the input argument is a block of size BS from address A
  - **input (A[i;BS])**
    - → the input argument is a block of size BS from address &A[i]
    - → the lower bound can be omitted (default is 0)
    - → the upper bound can be omitted if size is known (default is N-1, being N the size)
  - **input (A[i:j])**
    - → the input argument is a block from element A[i] to element A[j] (included)
    - → A[i:i+BS-1] equivalent to A[i; BS]

Examples dependency clauses, array sections

```
int a[N];
#pragma omp task input(a)
```

=

```
int a[N];
#pragma omp task input(a[0:N-1])
//whole array used to compute dependences
```

=

```
int a[N];
#pragma omp task input([N]a)
//whole array used to compute dependences
```

=

```
int a[N];
#pragma omp task input(a[0;N])
//whole array used to compute dependences
```

```
int a[N];
#pragma omp task input(a[0:3])
//first 4 elements of the array used to compute dependences
```

=

```
int a[N];
#pragma omp task input(a[0;4])
//first 4 elements of the array used to compute dependences
```

Examples dependency clauses, array sections (multidimensions)

```
int a[N][M];
#pragma omp task input(a[0:N-1][0:M-1])
//whole matrix used to compute dependences
```

=

```
int a[N][M];
#pragma omp task input(a[0;N][0;M])
//whole matrix used to compute dependences
```

```
int a[N][M];
#pragma omp task input(a[2:3][3:4])
// 2 x 2 subblock of a at a[2][3]
```

=

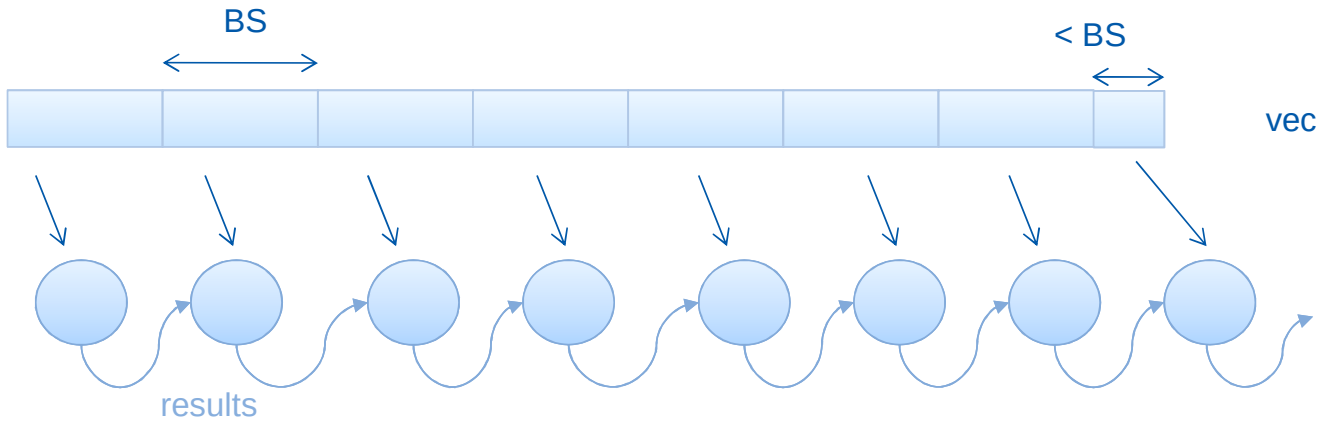
```
int a[N][M];
#pragma omp task input(a[2;2][3;2])
// 2 x 2 subblock of a at a[2][3]
```

```
int a[N][M];
#pragma omp task input(a[2:3][0:M-1])
//rows 2 and 3
```

=

```
int a[N][M];
#pragma omp task input(a[2;2][0;M])
//rows 2 and 3
```

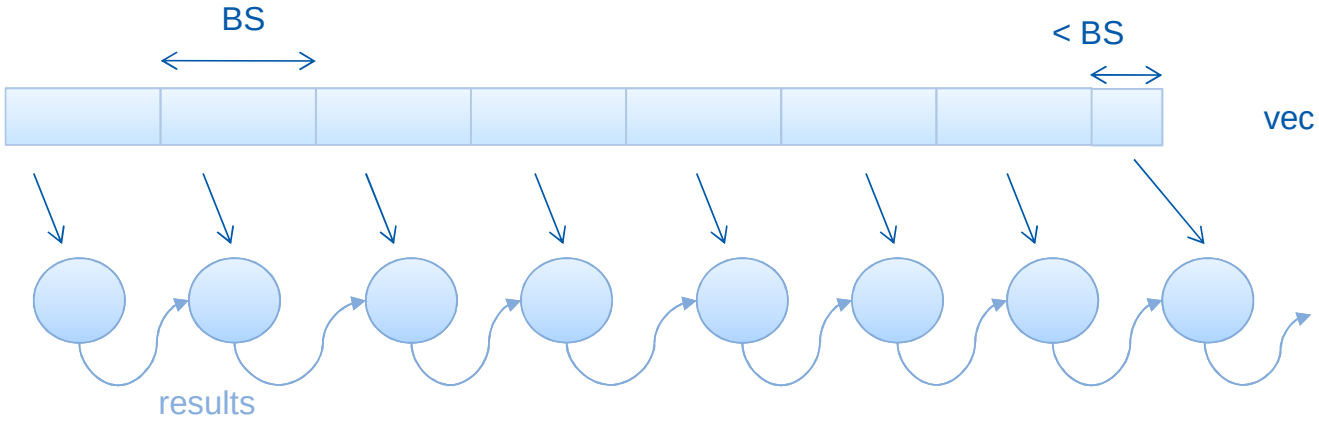
Examples dependency clauses, array sections




```
#pragma omp task input ([n]vec) inout (*results)
void sum_task ( int *vec , int n , int  *results);

void main(){
int actual_size;
   for (int j; j<N; j+=BS){
    actual_size = (N- j> BS ? BS: N-j);
    sum_task (&vec[j], actual_size, &total);
   }
}
```

dynamic size of argument

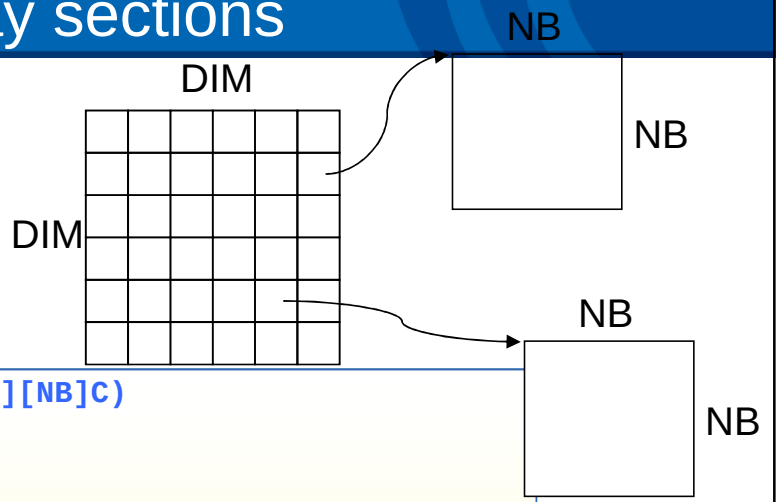
Examples dependency clauses, array sections




```
for (int j; j<N; j+=BS){
   actual_size = (N- j> BS ? BS: N-j);
#pragma omp task input (vec[j;actual_size]) inout(results) firtprivate(actual_size,j)
for (int count = 0; count < actual_size; count++)
   results += vec [j+count] ;
}
```

dynamic size of argument

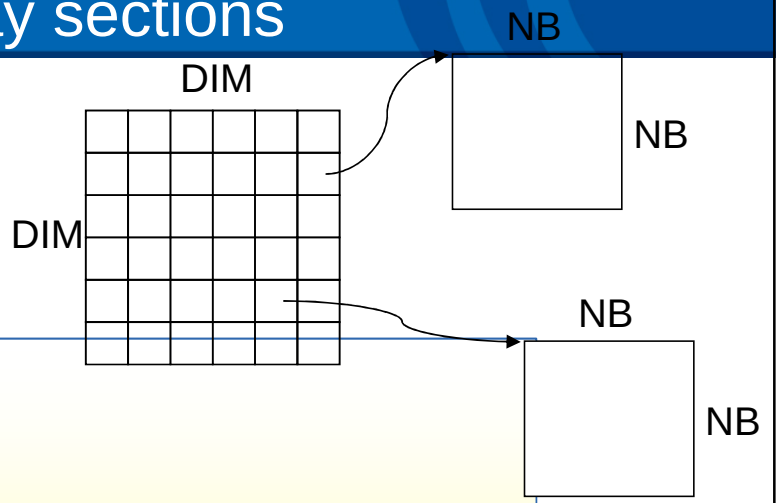
Examples dependency clauses, array sections




```
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void matmul(double *A, double *B, double *C,
unsigned long NB)
{
  int i, j, k;

  for (i = 0; i < NB; i++)
   for (j = 0; j < NB; j++)
    for (k = 0; k < NB; k++)
       C[i][j] +=A[i*NB+k]*B[k*NB+j];
}
```

```
void compute(unsigned long NB, unsigned long DIM,
double *A[DIM][DIM], double *B[DIM][DIM], double *C[DIM][DIM])
{
  unsigned i, j, k;

  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      for (k = 0; k < DIM; k++)
        matmul (A[i][k], B[k][j], C[i][j], NB);
}
```

Examples dependency clauses, array sections




```
void matmul(double *A, double *B, double *C,
unsigned long NB)
{
  int i, j, k;

  for (i = 0; i < NB; i++)
   for (j = 0; j < NB; j++)
    for (k = 0; k < NB; k++)
        C[i][j];+=A[i*NB+k]*B[k*NB+j];
}
```

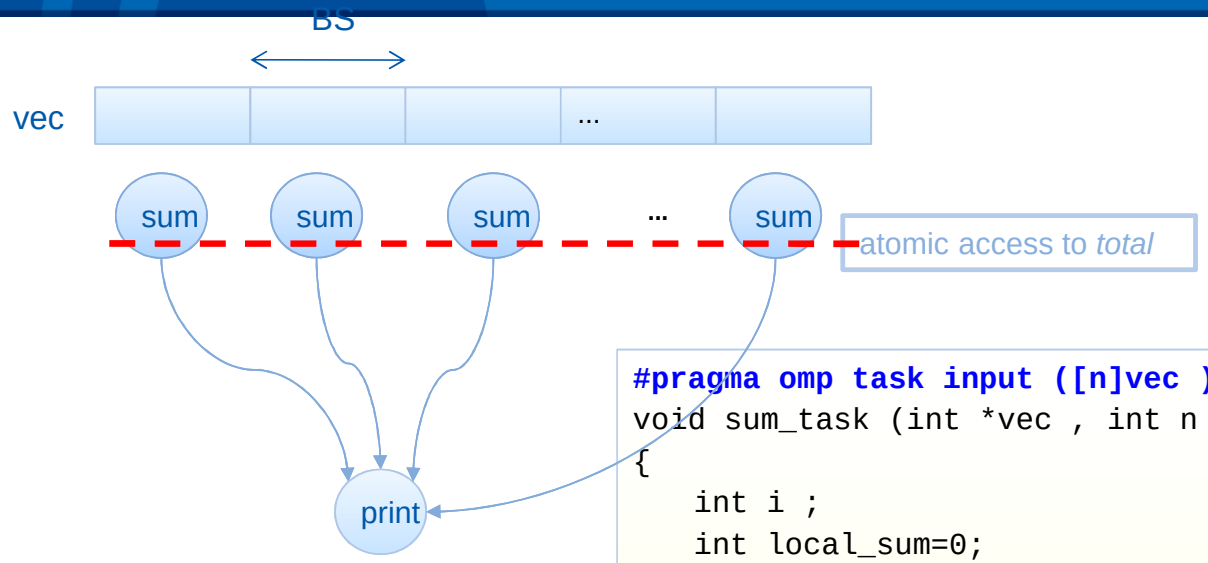
```
void compute(unsigned long NB, unsigned long DIM,
double *A[DIM][DIM], double *B[DIM][DIM], double *C[DIM][DIM])
{
  unsigned i, j, k;

  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      for (k = 0; k < DIM; k++)
#pragma omp task input([NB][NB]A[i][k], [NB][NB]B[k][j]) inout(NB][NB]C[i][j])\
firstprivate (i, j, k)
        matmul (A[i][k], B[k][j], C[i][j], NB);
}
```

Concurrent

```
#pragma omp task input ( ...) output (...) concurrent (var)
```

- Less-restrictive than regular data dependence
  - → Concurrent tasks can run in parallel
  - – Enables the scheduler to change the order of execution of the tasks, or even execute them concurrently
    - → alternatively the tasks would be executed sequentially due to the inout accesses to the variable in the concurrent clause
  - – Dependences with other tasks will be handled normally
    - → Any access input or inout to *var* will imply to wait for all previous *concurrent* tasks
- The task may require additional synchronization
  - – i.e., atomic accesses
  - – programmer responsibility: with pragma atomic, mutex, ...

Concurrent




```
#pragma omp task input ([n]vec ) concurrent (*results)
void sum_task (int *vec , int n , int  *results)
{
    int i ;
    int local_sum=0;

    for ( i = 0; i < n ; i ++)
    local_sum += vec [i] ;

#pragma omp atomic
    *results += local_sum;
}

void main(){
   for (int j=0; j<N; j+=BS) sum_task (&vec[j], BS, &total);
   #pragma omp task input (total)
printf ("TOTAL is %d\n", total);
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

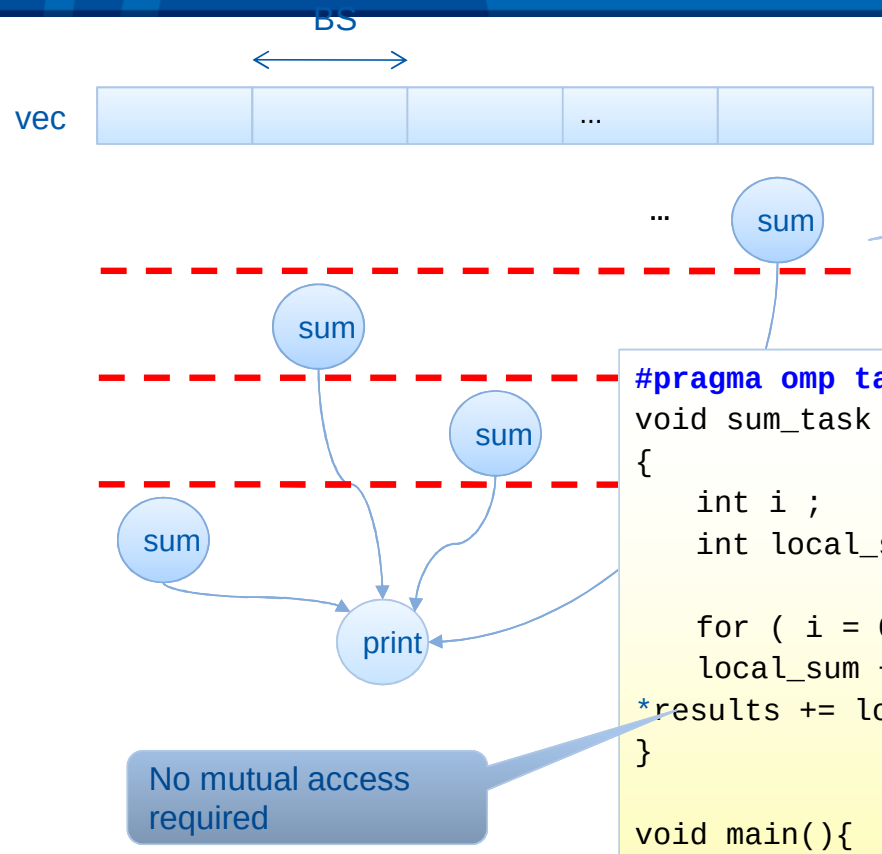
Commutative

```
#pragma omp task input ( ...) output (...) commutative(var)
```

- Less-restrictive than regular data dependence
  - → denoting that tasks can execute in any order but not concurrently

  Enables the scheduler to change the order of execution of the tasks, but without executing them concurrently
    - → alternatively the tasks would be executed sequentially in the order of instantiation due to the inout accesses to the variable in the commutative clause
  - Dependences with other tasks will be handled normally
    - →Any access input or inout to *var* will imply to wait for all previous *commutative* tasks

Commutative



Tasks executed out of order but not concurrently

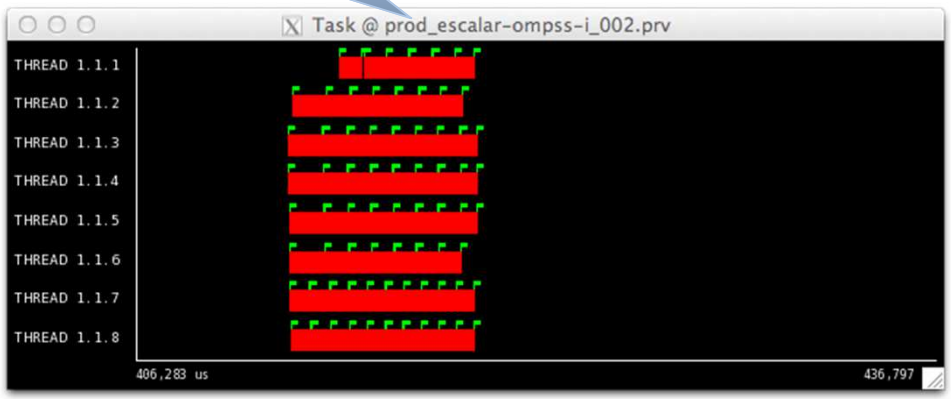
No mutual access required

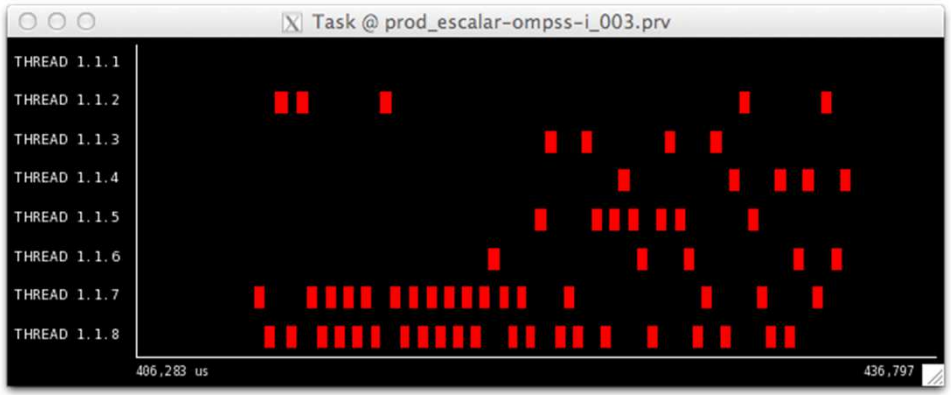
```
#pragma omp task input ([n]vec ) commutative(*results)
void sum_task (int *vec , int n , int  *results)
{
    int i ;
    int local_sum=0;

    for ( i = 0; i < n ; i ++)
    local_sum += vec [i] ;
*results += local_sum;
}

void main(){
   for (int j=0; j<N; j+=BS) sum_task (&vec[j], BS, &total);
   #pragma omp task input (total)
printf ("TOTAL is %d\n", total);
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Differences between concurrent and commutative

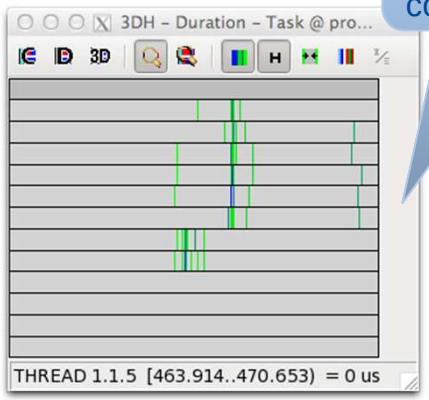
Tasks timeline: views at same time scale

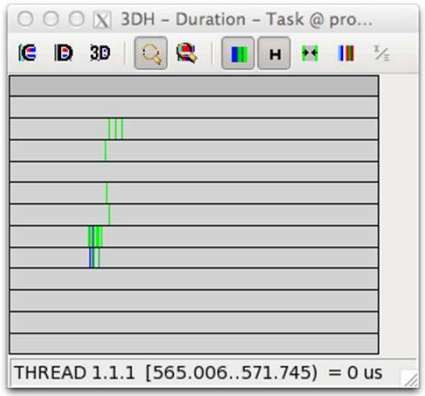




In this case, concurrent is more efficient

Histogram of tasks duration: at same control scale





… but tasks have more duration and variability

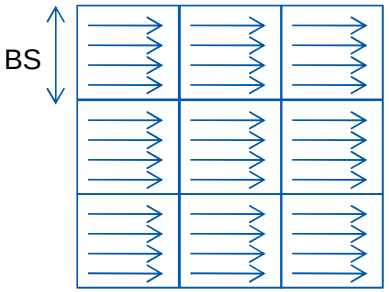
Hierarchical task graph

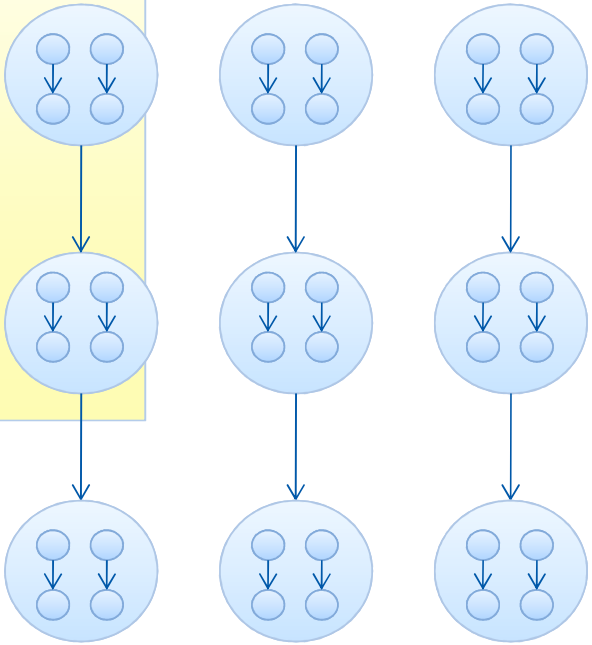
- Nesting
  - Tasks can generate tasks themselves
- Hierarchical task dependences
  - Dependences only checked between siblings
    - Several task graphs
    - Hierarchical
    - There is no implicit taskwait at the end of a task waiting for its children
  - Different level tasks share the same resources
    - When ready, queued in the same queues
    - Currently, no priority differences between tasks and its children

Hierarchical task graph

```
#pragma omp task input([BS][BS]A, [BS][BS] B) inout([BS][BS]C)
void block_dgemm(float *A, float *B, float *C);

#pragma omp task input([N]A, [N]B) inout([N]C)
void dgemm(float (*A)[N], float (*B)[N], float (*C)[N]){
int i, j, k;
int NB= N/BS;

for (i=0; i< N; i+=BS)
for (j=0; j< N; j+=BS)
   for (k=0; k< N; k+=BS)
      block_dgem(&A[i][k*BS], &B[k][j*BS], &C[i][j*BS]);
}

main() {
(
 ...
dgemm(A,B,C);
dgemm(D,E,F);
#pragma omp taskwait

}
```

Block data-layout

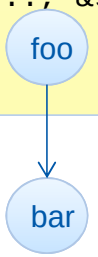




Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Example sentinels

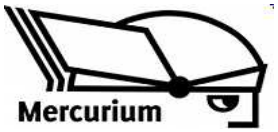
```
#pragma omp task output (*sentinel)
void foo ( .... , int *sentinel){ // used to force dependences under complex structures
    (graphs, ... )

...
}

#pragma omp task input (*sentinel)
void bar ( .... , int *sentinel){

...
}
main () {
   int sentinel;

   foo (..., &sentinel);
   bar (..., &sentinel)

}
```

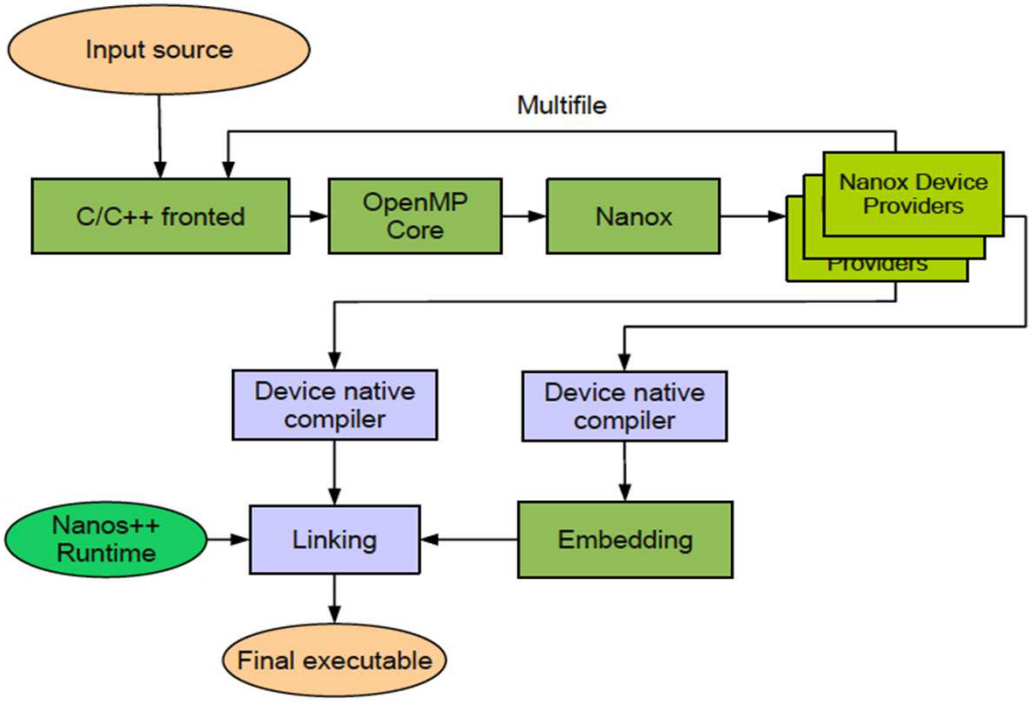


- Mechanism to handle complex dependences
  - when difficult to specify proper input/output clauses
- To be avoided if possible
  - the use of an element or group of elements as sentinels to represent a larger data-structure is valid
  - however might made code non-portable to heterogeneous platforms if copy_in/out clauses cannot properly specify the address space that should be accessible in the devices

Mercurium Compiler

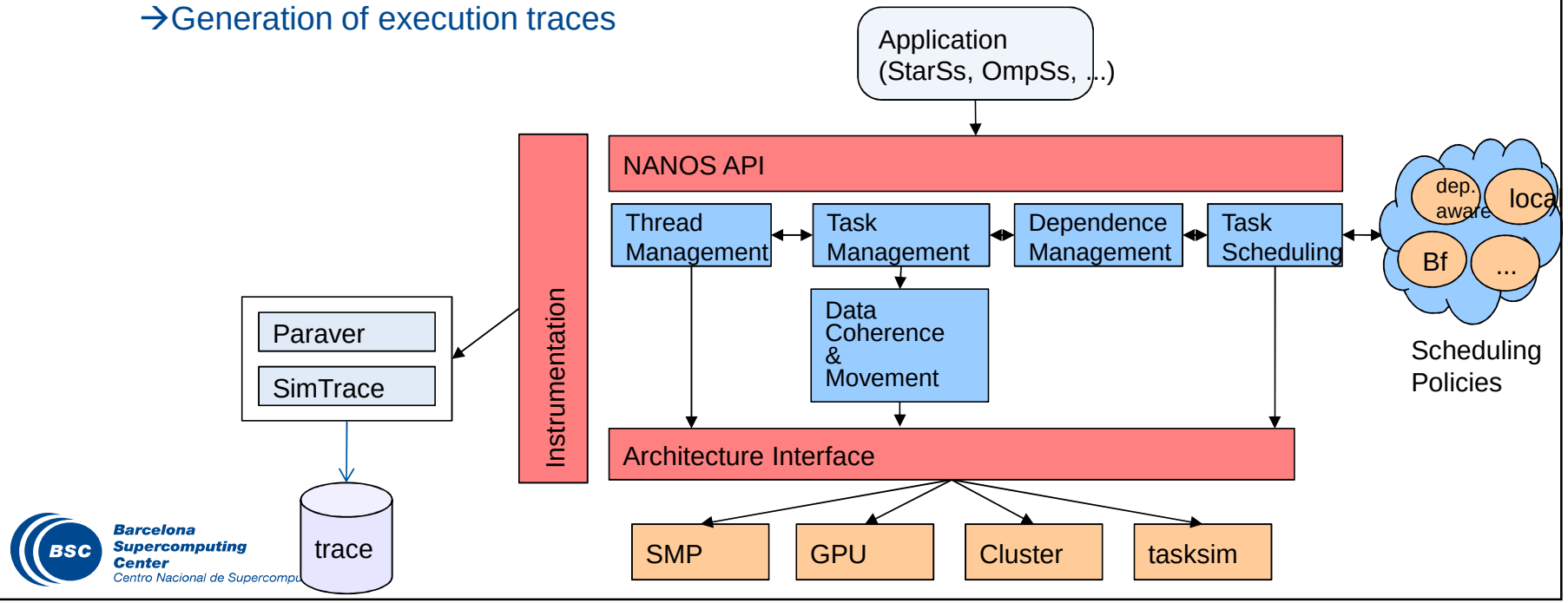
- Recognizes constructs and transforms them to calls to the runtime
- Manages code restructuring for different target devices
  - Device-specific handlers
  - May generate code in a separate file
  - Invokes different back-end compilers
    → nvcc for NVIDIA





Runtime structure

- Support to different programming models: OpenMP (OmpSs), StarSs, Chapel
- Independent components for thread, task, dependence management, task scheduling, ...
- Most of the runtime independent of the target architecture: SMP, GPU, tasksim simulator, cluster
- Support to heterogeneous targets
  - → i.e., threads running tasks in regular cores and in GPUs
- Instrumentation
  - →Generation of execution traces



Barcelona Supercomputing Center
Centro Nacional de Supercomput

Compiling

- Compiling

```
frontend --ompss  -c bin.c
```

- Linking

```
frontend --ompss -o bin bin.o
```

- where frontend is one of:

mcc	C
mcxx	C++
mnvcc	CUDA & C
mnvcxx	CUDA & C++
mfc	Fortran (In development)

Compiling

- Compatibility flags:
  - -I, -g, -L, -l, -E, -D, -W
- Other compilation flags:

-k	Keep intermediate files
--debug	Use Nanos++ debug version
--instrumentation	Use Nanos++ instrumentation version
--version	Show Mercurium version number
--verbose	Enable Mercurium verbose output
--Wp,flags	Pass flags to preprocessor (comma separated)
--Wn,flags	Pass flags to native compiler (comma separated)
--Wl,flags	Pass flags to linker (comma separated)
--help	To see many more options :-)

Executing

- No LD_LIBRARY_PATH or LD_PRELOAD needed

```
./bin
```

- Adjust number of threads with OMP_NUM_THREADS

```
OMP_NUM_THREADS=4 ./bin
```

Nanos++ options

- Other options can be passed to the Nanos++ runtime via NX_ARGS

```
NX_ARGS="options" ./bin
```

--schedule=name	Use name task scheduler
--throttle=name	Use name throttle-policy
--throttle-limit=limit	Limit of the throttle-policy (exact meaning depends on the policy)
--instrumentation=name	Use name instrumentation module
--disable-yield	Nanos++ won't yield threads when idle
--spins=number	Number of spin loops when idle
--disable-binding	Nanos++ won't bind threads to CPUs
--binding-start=cpu	First CPU where a thread will be bound
--binding-stride=number	Stride between bound CPUs

Nanox helper

- Nanos++ utility to
  - list available modules:

```
nanox --list-modules
```

  - list available options:

```
nanox --help
```

Tracing

- Compile and link with --instrument

```
mcc --ompss --instrument -c bin.c
mcc -o bin --ompss --instrument bin.o
```

- When executing specify which instrumentation module to use:

```
NX_INSTRUMENTATION=extrae ./bin
```

- Will generate trace files in executing directory
  - 3 files: prv, pcf, rows
  - Use paraver to analyze

Reporting problems

- Compiler problems
  - http://pm.bsc.es/projects/mcxx/newticket
- Runtime problems
  - http://pm.bsc.es/projects/nanox/newticket
- Support mail
  - pm-tools@bsc.es
- Please include snapshot of the problem

Programming methodology

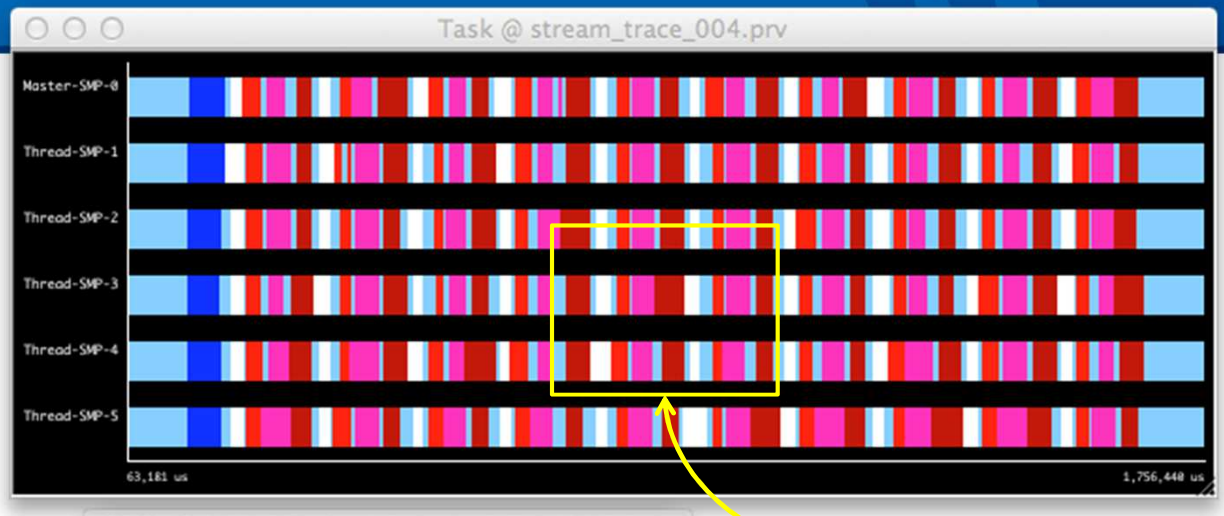
- Correct sequential program
- Incremental taskification
  - Test every individual task with forced sequential in-order execution
    - → 1 thread, scheduler = FIFO, throtle=1
- Single thread out-of-order execution
- Increment number of threads
  - Use taskwaits to force certain levels of serialization

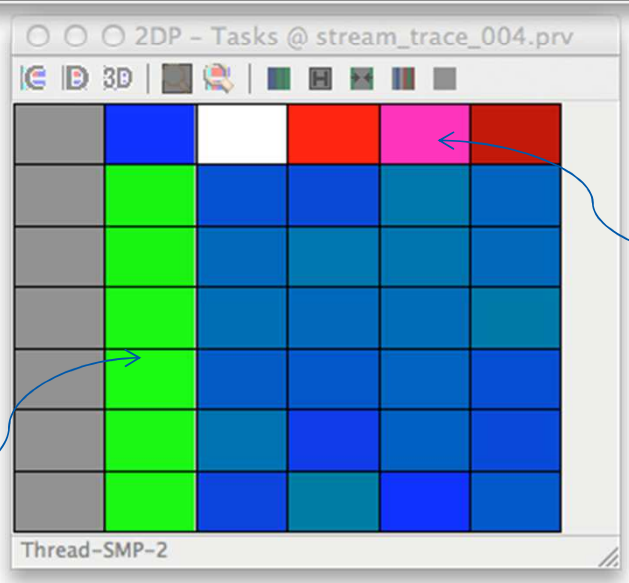
Visualizing Paraver tracefiles

- Set of Paraver configuration files ready for OmpSs. Organized in directories
  - **Tasks: related to application tasks**
  - Runtime, nanox-configs: related to OmpSs runtime internals
  - **Graph_and_scheduling: related to task-graph and task scheduling**
  - DataMgmgt: related to data management
  - CUDA: specific to GPU

Tasks' profile

- 2dp_tasks.cfg
- Tasks' profile





control window: timeline where each color represent the task been executed by each thread

light blue: not executing tasks

threads

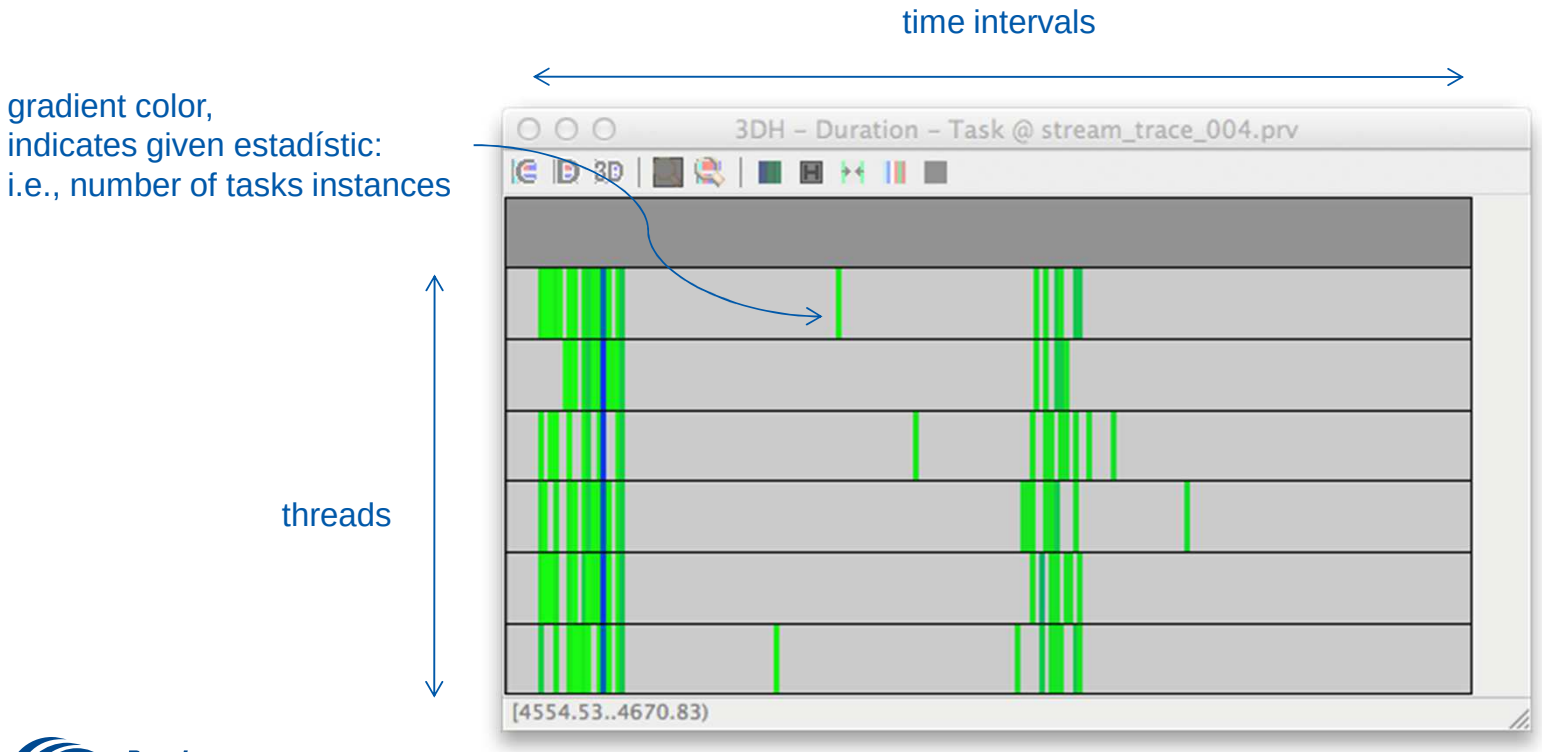
different colours represent different task type

gradient color, indicates given estadístic: i.e., number of tasks instances

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Tasks duration histogram

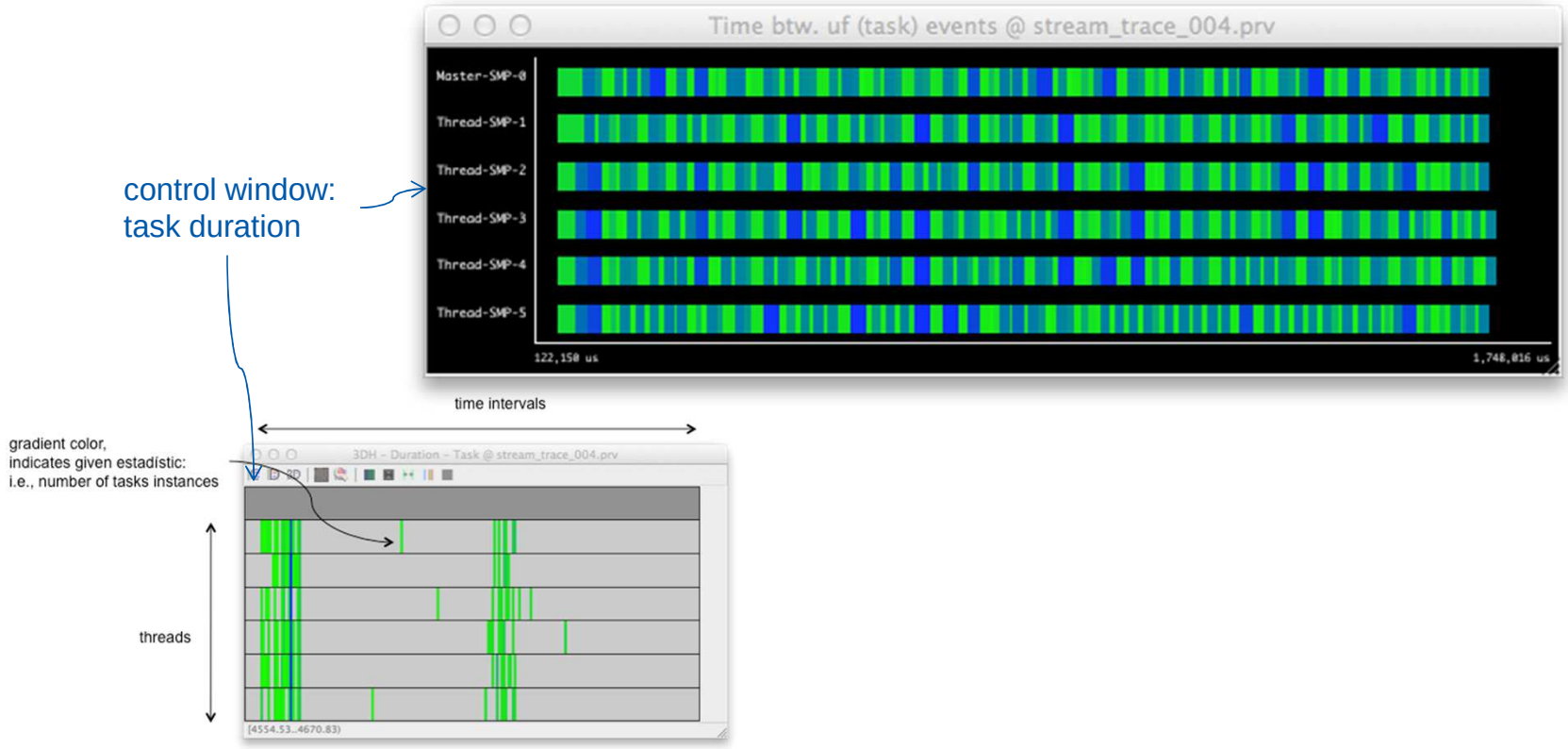
- 3dh_duration_task.cfg



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

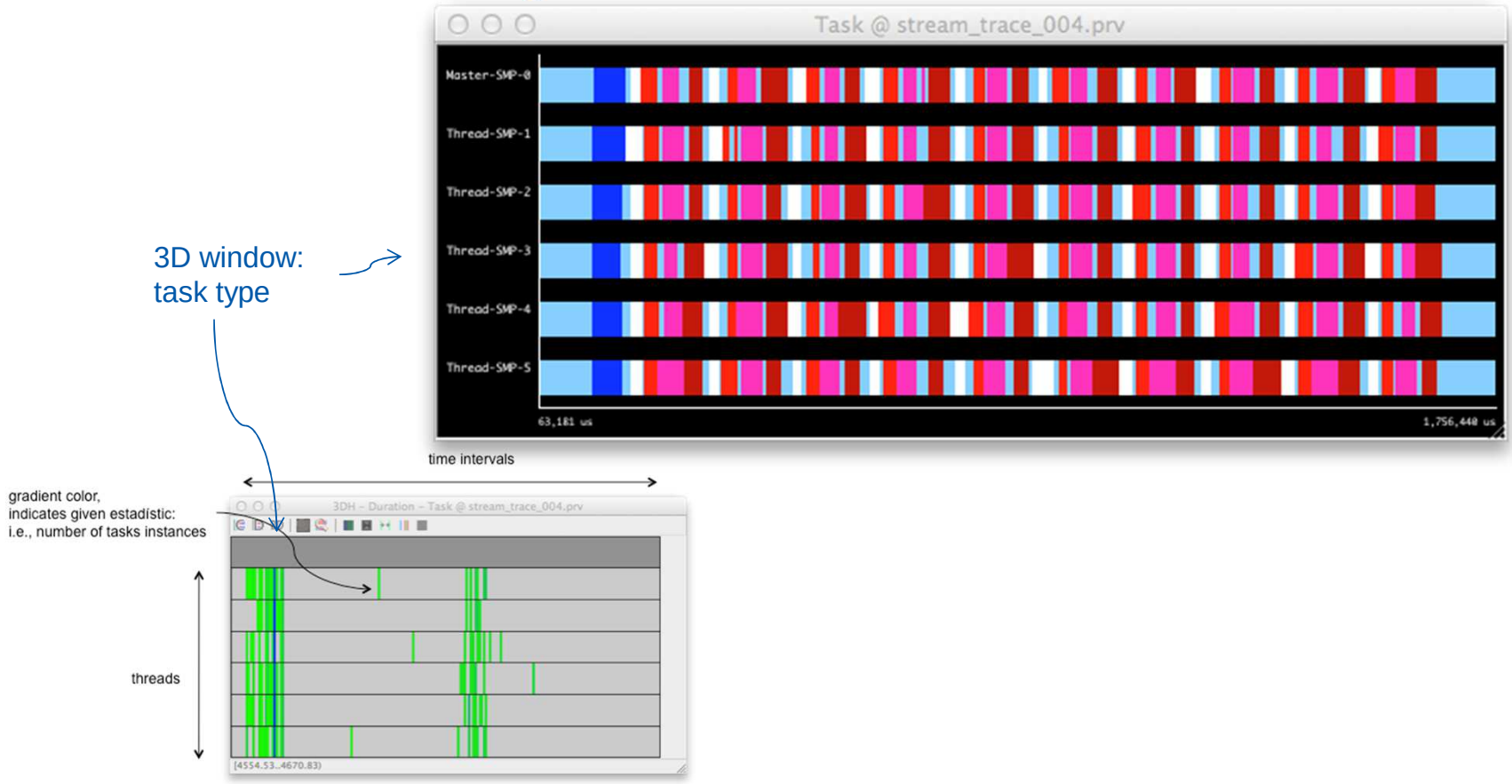
Tasks duration histogram

- 3dh_duration_task.cfg



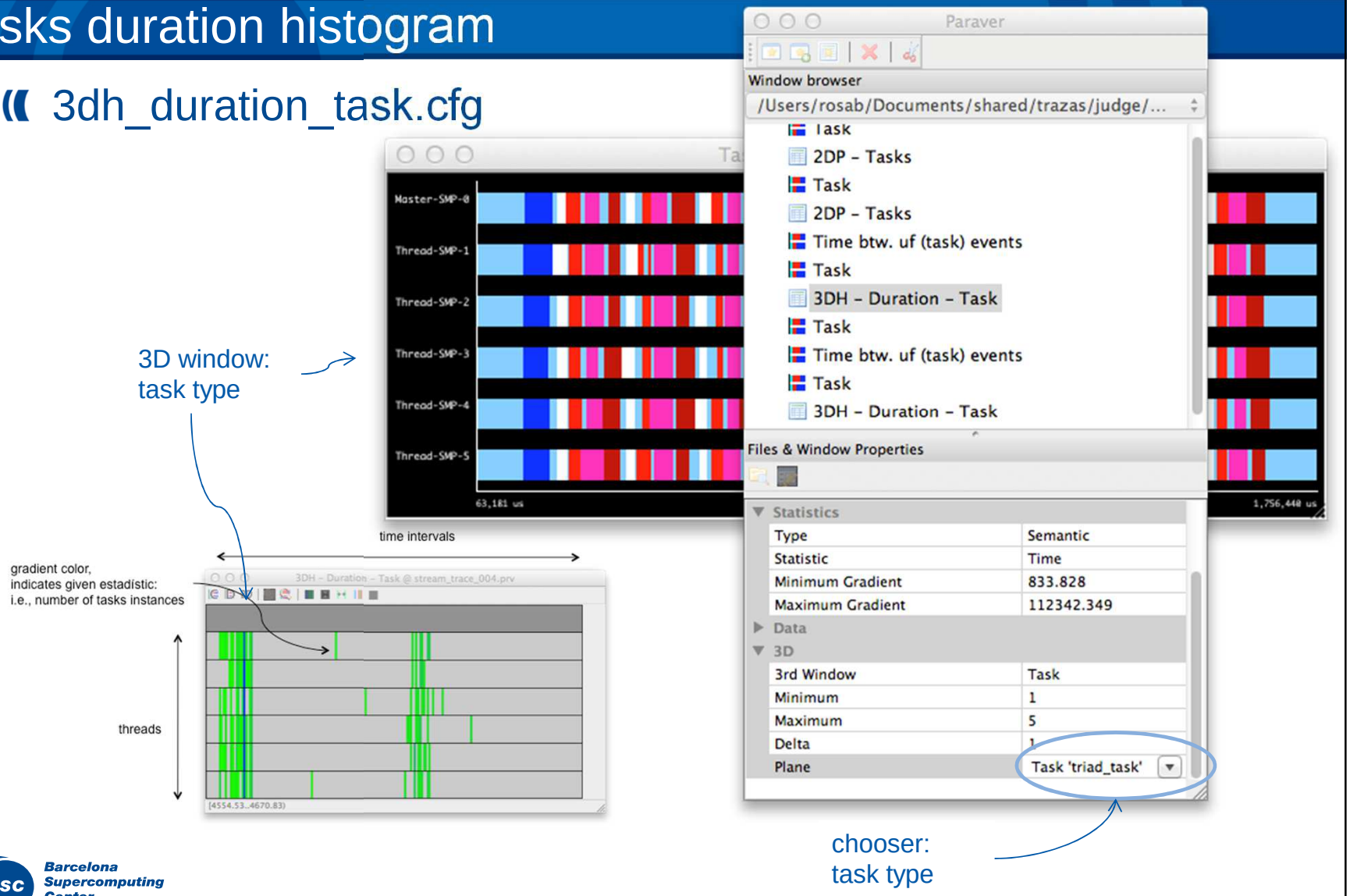
Tasks duration histogram

- 3dh_duration_task.cfg



Tasks duration histogram

- 3dh_duration_task.cfg

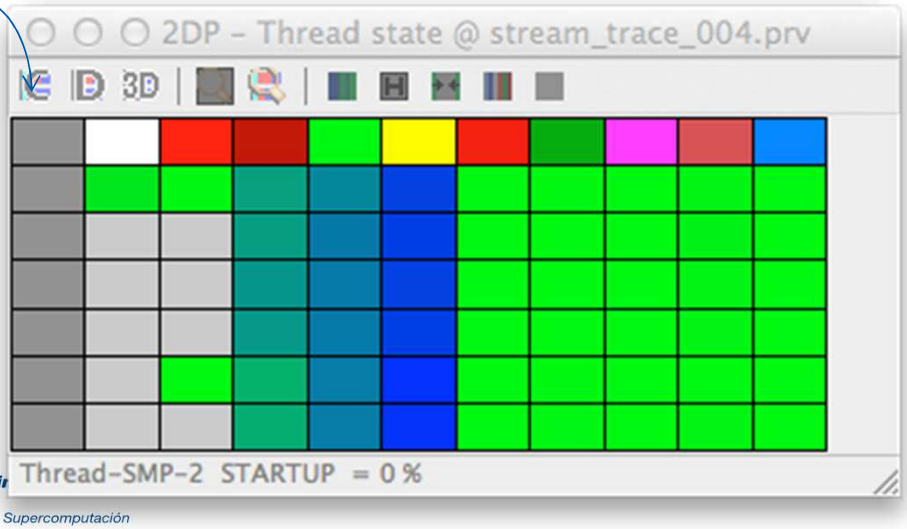


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Threads state profile

- 2dp_threads_state.cfg

control window: timeline where each color represent the runtime state of each thread



threads

Barcelona Supercomputing Center
Centro Nacional de Supercomputación


www.bsc.es
BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
OmpSs
Based on presentation by Rosa M Badia, Xavier Martorell
Isaac Rudomin
BSC