www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

The OmpSs programming model and its runtime support

Jesús Labarta
BSC

13th Charm++ Workshop
Urbana-Champaign. May, 8th 2015

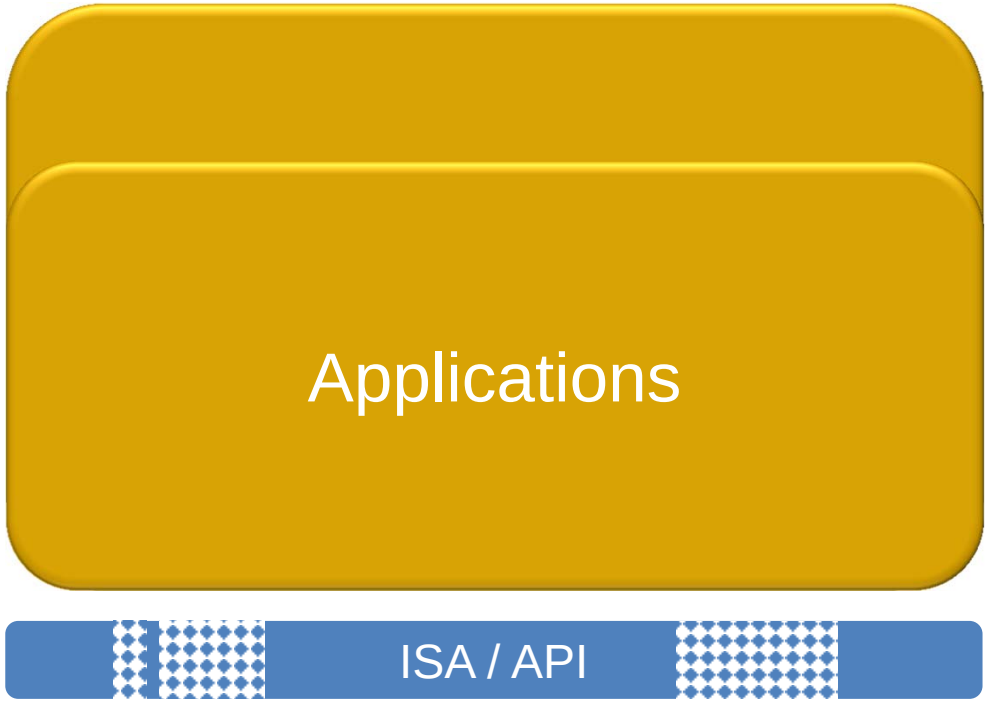

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
VISION

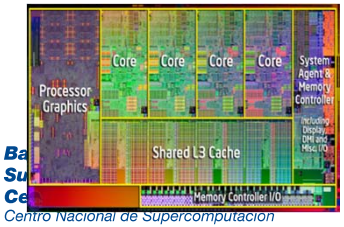
Look around …

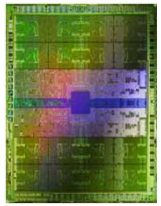
We are in the middle of a "revolution"

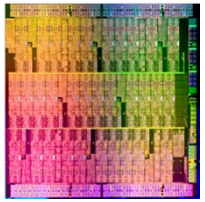
Living in the programming revolution

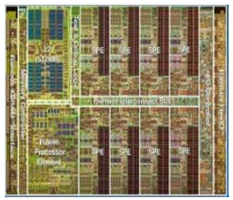
- The power wall made us go multicore and the ISA interface to leak → our world is shaking











Application logic
+
Platform specificites

Address spaces (hierarchy, transfer), control flows,…

… complexity !!!!

The programming revolution

- An age changing revolution
  - From the latency age …
    - Specify what to compute, where and when
    - Performance dominated by latency in a broad sense
      - Memory, communication, pipeline depths, fork-join, …
      - I need something … I need it now!!!
  - …to the throughput age
    - Ability to instantiate “lots” of work and avoiding stalling for specific requests
      - I need this and this and that … and as long as it keeps coming I am ok
      - (Broader interpretation than just GPU computing !!)
    - Performance dominated by overall availability/balance of resources

From the latency age to the throughput age

- It will require a programming effort !!!
  - Must make the transition as easy/smooth as possible
  - Must make it as long lived as possible

- Need
  - Simple mechanisms at the programming model level to express potential concurrency, letting exploitation responsibility to the runtime
    - Dynamic task based, asynchrony, look-ahead, malleability, …
  - A change in programmers mentality/attitude
    - Top down programming methodology
    - Think global, of potentials rather than how-to's
    - Specify local, real needs and outcomes of the functionality being written

Vision in the programming revolution

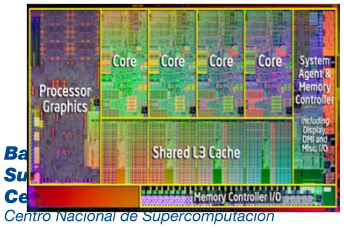
- Need to decouple again

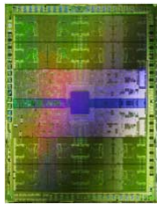
Applications

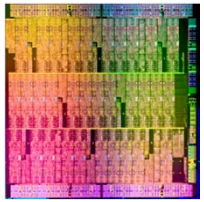
PM: High-level, clean, abstract interface

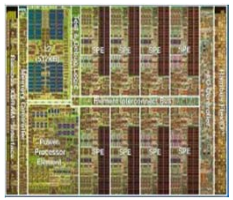
Power to the runtime

ISA / API









Application logic

Arch. independent

General purpose
Task based
Single address space

"Reuse"
architectural ideas
under
new constraints

Vision in the programming revolution

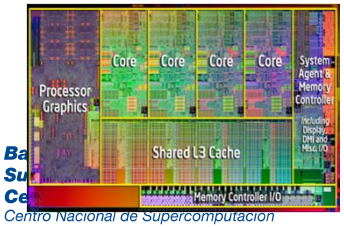
Applications

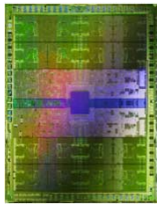
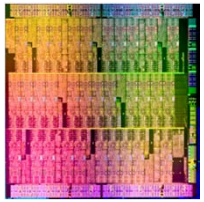
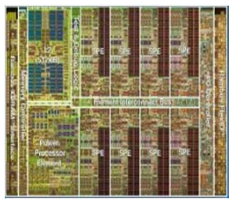
DSL1 DSL2 DSL3

PM: High-level, clean, abstract interface

Power to the runtime

ISA / API



Fast prototyping

Special purpose

Must be easy to develop/maintain

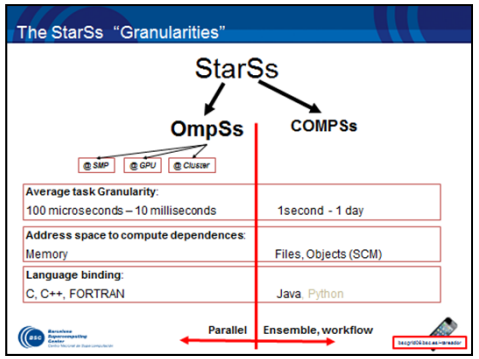
General purpose
Task based
Single address space

"Reuse" architectural ideas under new constraints

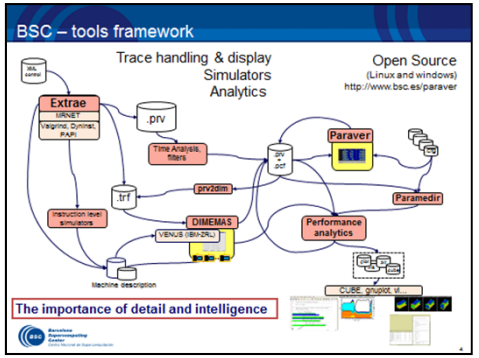

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
WHAT WE DO?

BSC technologies

- Programming model
  - The StarSs concept (*Superscalar) :
    - sequential programming + directionality annotations → Out of order execution
  - The OmpSs implementation → OpenMP Standard



- Performance tools
  - Trace visualization and analysis:
    - extreme flexibility and detail
  - Performance analytics



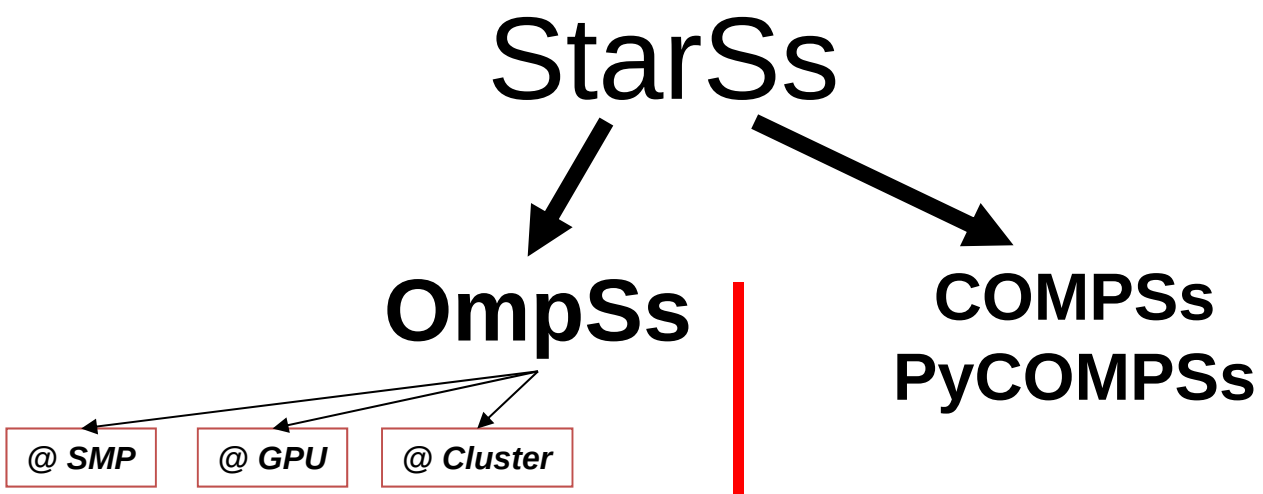

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
PROGRAMMING MODELS

The StarSs family of programming models

- Key concept
  - **Sequential task based** program on **single address/name space + directionality annotations**
  - Happens to execute parallel: Automatic run time computation of dependencies between tasks

- Differentiation of StarSs
  - Dependences: Tasks instantiated but not ready. Order IS defined
    - Lookahead
      - Avoid stalling the main control flow when a computation depending on previous tasks is reached
      - Possibility to “see” the future searching for further potential concurrency
    - Dependences built from data access specification
  - Locality aware
    - Without defining new concepts
  - Homogenizing heterogeneity
    - Device specific tasks but homogeneous program logic

The StarSs "Granularities"



Average task Granularity: 100 microseconds – 10 milliseconds	1second - 1 day
Address space to compute dependences: Memory	Files, Objects (SCM)
Language binding: C, C++, FORTRAN	Java, Python

Parallel **Ensemble, workflow**

OmpSs in one slide

- Minimalist set of concepts …
  - … ”extending” OpenMP
  - … relaxing StarSs funtional model

```
#pragma omp task [ in (array_spec...)] [ out (...)] [ inout (...)] \
                 [ concurrent (…)] [commutative(...)] [ priority(P) ] [ label(...) ] \
                 [ shared(...)][private(...)][firstprivate(...)][default(...)][untied][final][if (expression)]
                 [reduction(identfier : list)]
 {code block or function}
```

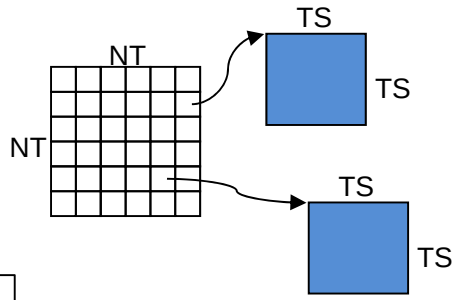
```
#pragma omp taskwait [ on (...) ][noflush]
```

```
#pragma omp for [ shared(...)][private(...)][firstprivate(...)][schedule_clause]
 {for_loop}
```

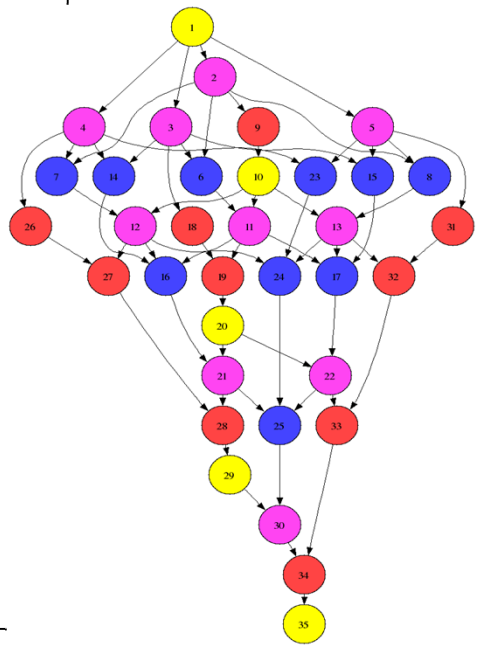
```
#pragma omp target device ({ smp | opencl | cuda })     \
        [ implements ( function_name )]                 \
        [ copy_deps | no_copy_deps ] [ copy_in ( array_spec ,...)] [ copy_out (...)] [ copy_inout (...)] } \
        [ndrange (dim, …)] [shmem(...) ]
```

Pragmas

Inlined

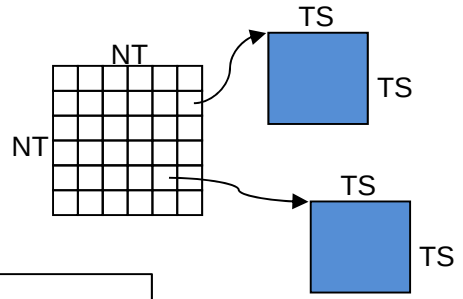



```
void Cholesky(int NT, float *A[NT][NT] ) {
   for (int k=0; k<NT; k++) {
      #pragma omp task inout ([TS][TS](A[k][k]))
      spotrf (A[k][k], TS) ;
      for (int i=k+1; i<NT; i++) {
        #pragma omp task in([TS][TS](A[k][k])) inout ([TS][TS](A[k][i]))
        strsm (A[k][k], A[k][i], TS);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

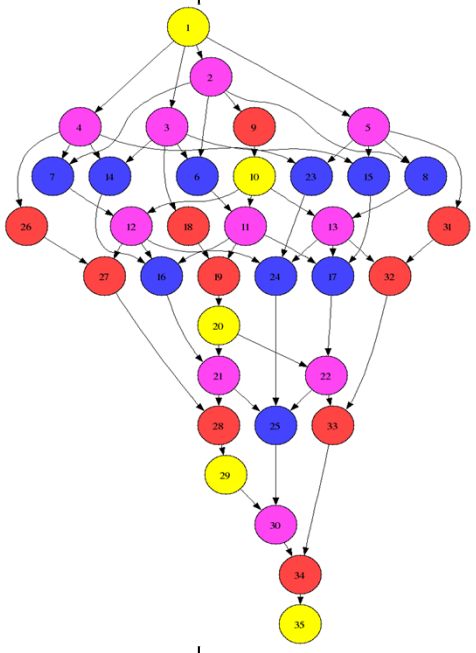


Pragmas

- …or outlined

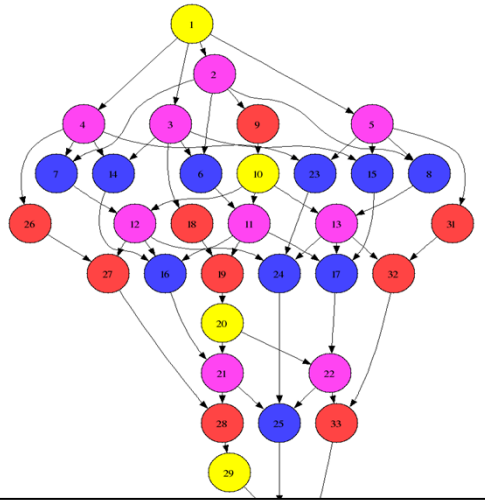


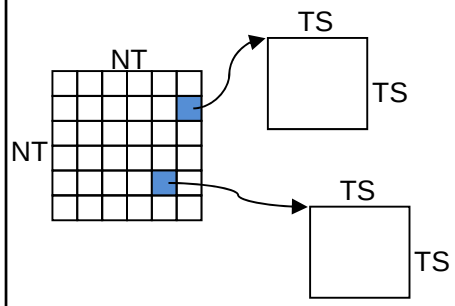

```
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A, int TS);
#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
void strsm (float *T, float *B, int TS);
#pragma omp task input ([TS][TS]A,[TS][TS]B) inout ([TS][TS]C)
void sgemm (float *A, float *B, float *C, int TS);
#pragma omp task input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

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

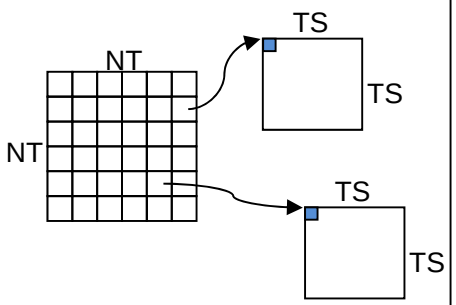


Incomplete directionalities specification: sentinels

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








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

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

Homogenizing Heterogeneity

- ISA heterogeneity
- Single address space program … executes in several non coherent address spaces
  - Copy clauses:
    - ensure sequentially consistent copy accessible in address space where task is going to be executed
    - Requires precise specification of data accessed (e.g. array sections)
  - Runtime offloads data and computation

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

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

CUDA tasks @ OmpSs

- Compiler splits code and sends codelet to nvcc
- Data transfers to/from device are performed by runtime
- Constrains for “codelet”
  - Can not access copied data ☹. Pointers translated when activating “codelet” task.
  - Can access firstprivate data

```
void Calc_forces_cuda( int npart, Particle *particles, Particle *result, float dtime) {

  const int bs = npart/8;
  int first, last, nblocks;

  for ( int i = 0; i < npart; i += bs ) {
    first = i;
    last = (i+bs-1 > npart) ? npart : i+bs-1;
    nblocks = (last - first + MAX_THREADS ) / MAX_THREADS;

    #pragma omp target device(cuda) copy_deps
    #pragma omp task in( particles[0:npart-1] ) out( result[first:(first+bs)-1])
    {
      calculate_forces <<< nblocks, MAX_THREADS >>> (dtime, particles, npart,
                    &result[first], first, last);
    }
  }
}
```

MACC (Mercurium ACcelerator Compiler)

- "OpenMP 4.0 accelerator directives" compiler
  - Generates OmpSs code + CUDA kernels (for Intel & Power8 + GPUs)
  - Propose clauses that improve kernel performance
- Extended semantics
  - Change in mentality … minor details make a difference

Type of device

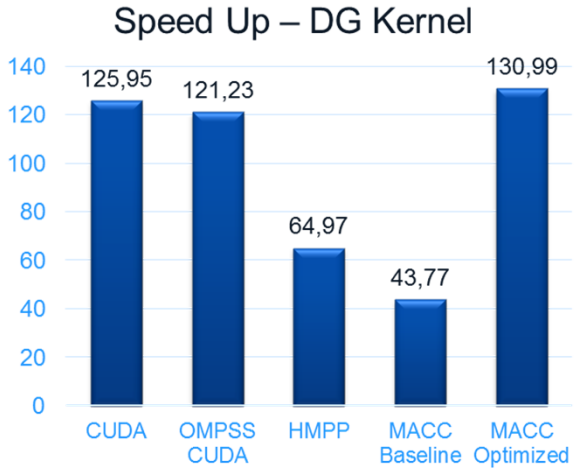
Ensure availability

OmpSs	`for (...)`
	`{`
	`  #pragma omp target device(acc) copy_deps`
	`  #pragma omp task inout(x[beg:end])`
	`  #pragma omp teams distribute parallel for`
	`  << ..Computation.. >>`
	`}`

OpenMP 4.0	`for (...)`
	`{`
	`  int dev_id = i % omp_get_num_devices();`
	`  #pragma omp task`
	`  #pragma omp target device(dev_id) map(tofrom: x[beg:end])`
	`  #pragma omp teams distribute parallel for`
	`  << ..Computation.. >>`
	`}`

Specific device

DO transfer



Barcelona Supercomputing

G. Ozen et al, "On the roles of the programmer, the compiler and the runtime system when facing accelerators in OpenMP 4.0" IWOMP 2014

Managing separate address spaces

- OmpSs @ Cluster runtime
  - Directory @ master
  - A software cache @ device manages its individual address space:
    - Manages local space at device (logical and physical)
    - Translate address @ main address space → device address
  - Implements transfers
    - Packing if needed
    - Device/network specific transfer APIs (i.e. GASNet, CUDA copies, MPI, …)
  - Constraints
    - No pointers in offloaded data, no deep copy, …
    - Same layout at host and device

J. Bueno et al, "Productive Programming of GPU Clusters with OmpSs", IPDPS2012

J. Bueno et al, "Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces", ICS 2013

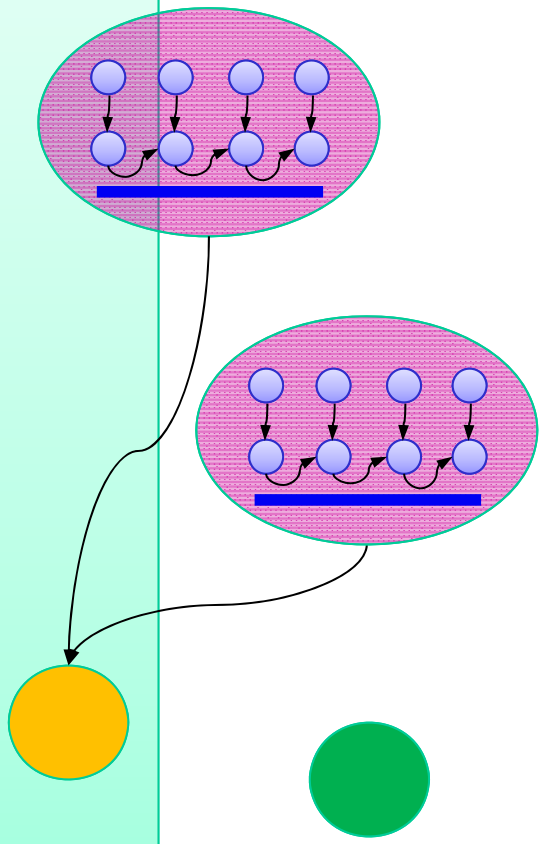
Multiple implementations

```
#pragma omp target device(opencl) ndrange(1,size,128) copy_deps implements (calculate_forces)
#pragma omp task out([size] out) in([npart] part)
__kernel void calculate_force_opencl(int size, float time,  int npart, __global Part* part,
                                                                __global Part* out, int gid);

#pragma omp target device(cuda) ndrange(1,size,128) copy_deps implements (calculate_forces)
#pragma omp task out([size] out) in([npart] part)
__global__ void calculate_force_cuda(int size, float time,  int npar, Part* part, Particle *out, int gid);

#pragma omp target device(smp) copy_deps
#pragma omp task out([size] out) in([npart] part)
void calculate_forces(int size, float time,  int npart, Part* part, Particle *out, int gid);
```

```
void Particle_array_calculate_forces(Particle* input, Particle *output, int npart, float time) {
   for (int i = 0; i < npart; i += BS )
        calculate_forces(BS, time, npart, input, &output[i], i);
}
```

Nesting

```
int Y[4]={1,2,3,4}

int main( )
{
   int X[4]={5,6,7,8};

   for (int i=0; i<2; i++) {
      #pragma omp task out(Y[i]) firstprivate(i,X)
      {
         for (int j=0 ; j<3; j++) {
            #pragma omp task inout(X[j])
            X[j]=f(X[j], j);
            #pragma omp task in (X[j]) inout (Y[i])
            Y[i] +=g(X[j]);
         }
         #pragma omp taskwait
      }
   }
   #pragma omp task inout(Y[0;2])
    for (int i=0; i<2; i++) Y[i] += h(Y[i]);
   #pragma omp task inout (v, Y[3])
   for (int i=1; i<N; i++) Y[3]=h(Y[3]);

   #pragma omp taskwait
}
```

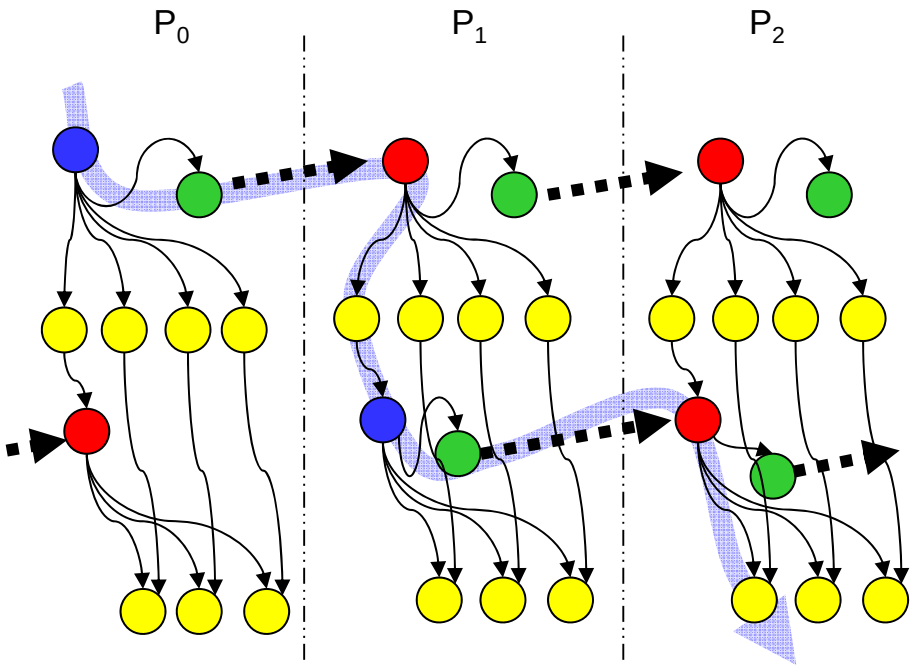


Hybrid MPI/ompSs: Linpack example

- Linpack example
- Overlap communication/computation
- Extend asynchronous data-flow execution to outer level
- Automatic lookahead

```
…
for (k=0; k<N; k++) {
   if (mine) {
      Factor_panel(A[k]);
      send (A[k])
   } else {
      receive (A[k]);
      if (necessary) resend (A[k]);
   }
   for (j=k+1; j<N; j++)
      update (A[k], A[j]);
…
```

```
#pragma omp task inout([SIZE]A)
void Factor_panel(float *A);
#pragma omp task in([SIZE]A) inout([SIZE]B)
void update(float *A, float *B);
```

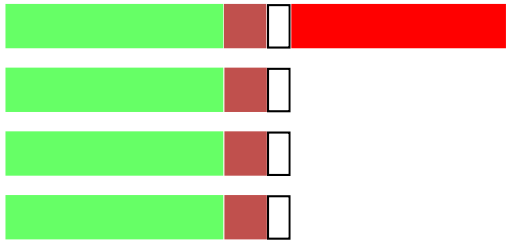



```
#pragma omp task in([SIZE]A)
void send(float *A);
#pragma omp task out([SIZE]A)
void receive(float *A);
#pragma omp task in([SIZE]A)
void resend(float *A);
```

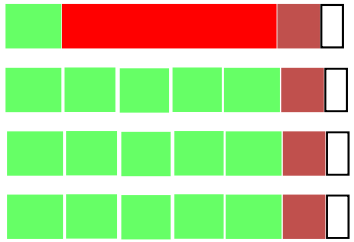
Fighting Amdahl's law: A chance for lazy programmers

Four loops/routines
Sequential program order

OpenMP 2.5
not parallelizing one loop



OmpSs/OpenMP4.0
not parallelizing one loop

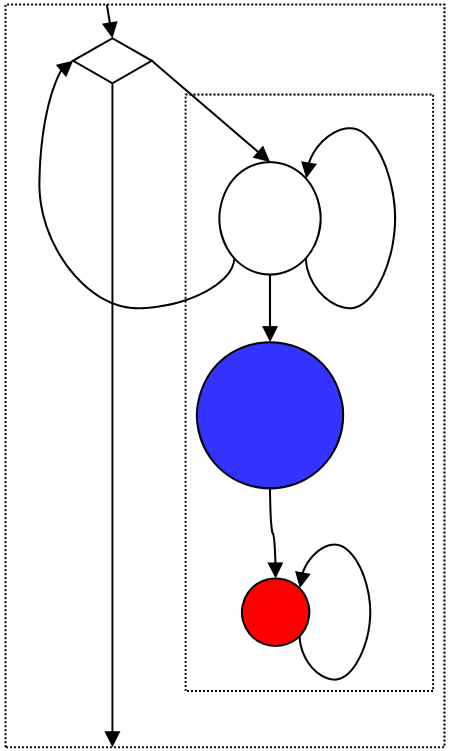


GROMACS@SMPSs


BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
OMPSS EXAMPLES

Streamed file processing

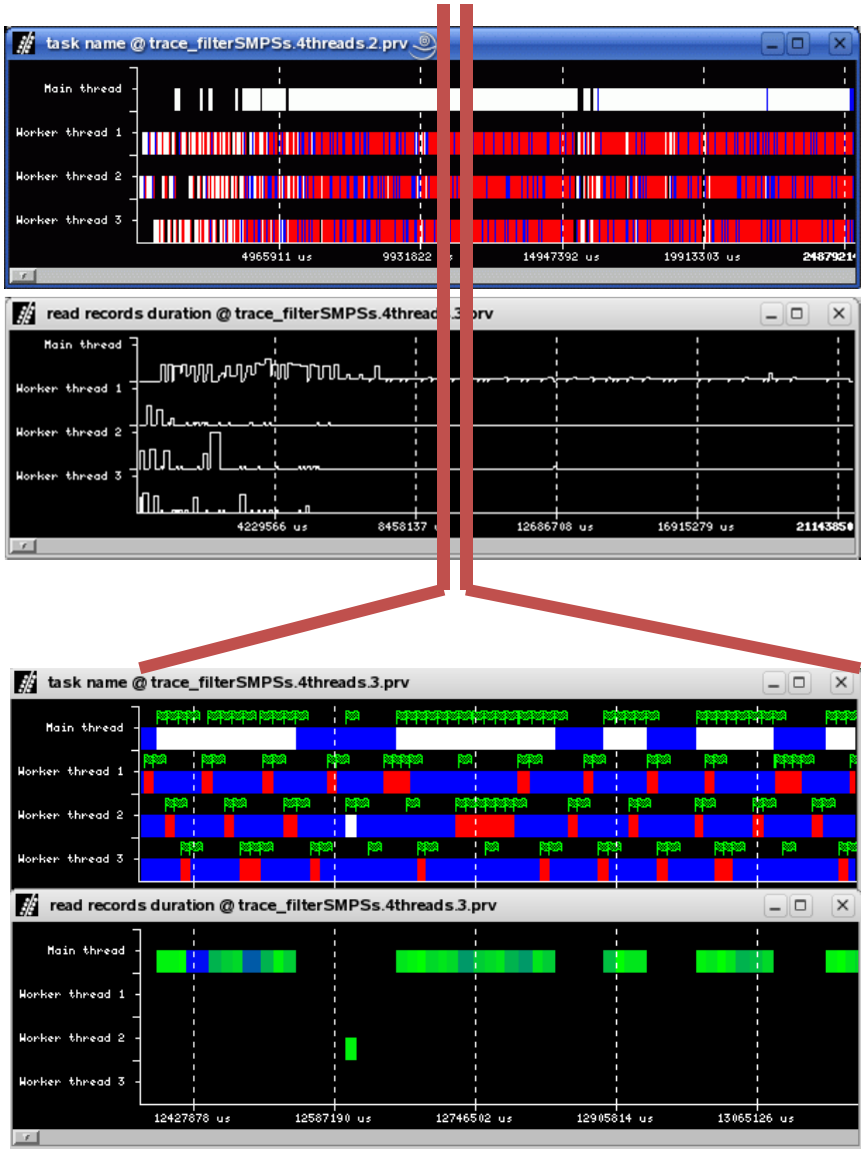
- A typical pattern
- Sequential file processing
- Automatically achieve asynchronous I/O

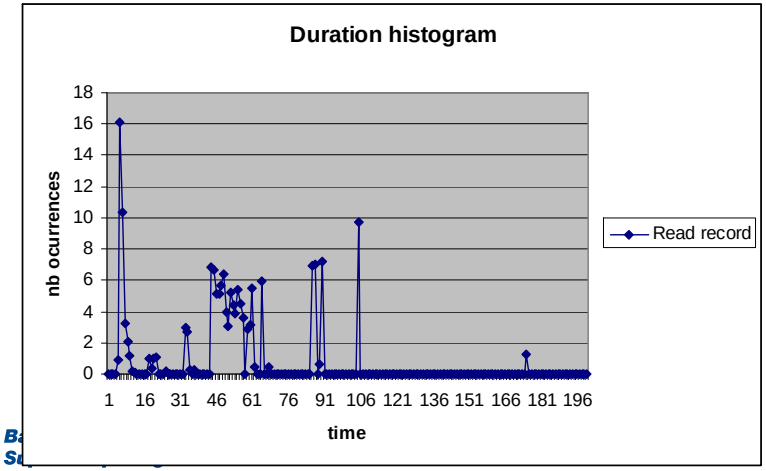


```
typedef struct {int size, char buff[MAXSIZE]} buf_t;

but_t *p[NBUF];
int j=0, total_records=0;

int main()
{  …
  while(!end_trace) {
     buf_t **pb=&p[j%NBUF]; j++;
     #pragma omp task inout(infile) out(*pb, end_trace) \
             priority(10)
     { *pb= malloc(sizeof(buf_t));
       Read (infile, *pb, &end_trace);
     }
     #pragma omp task inout(*pb)
     Process (*pb);
     #pragma omp task inout (outfile, *pb, total_records)\
                priority(10)
     {  int records;
        Write (outfile, *pb, &records);
        total_records += records;
        free (*pb);
     }
     #pragma omp taskwait on (&end_trace)
  }
}
```

Asynchrony: I/O

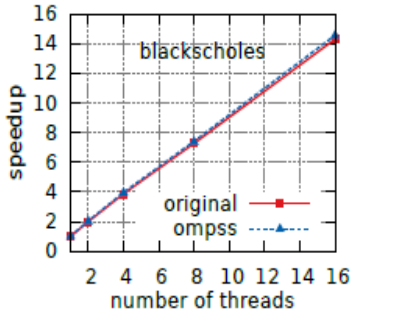
- Asynchrony
  - Decoupling/overlapping I/O and processing
  - Serialization of I/O
  - Resource constraints
    - Request for specific thread,…
  - Task duration variance
    - Dynamic schedule

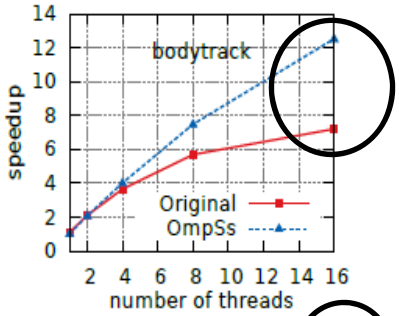


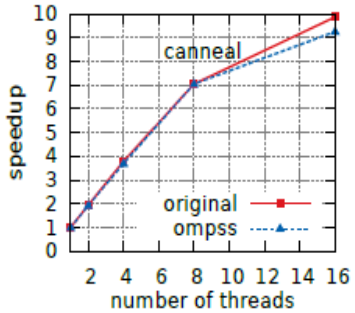


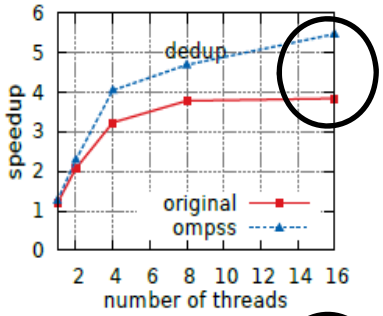
PARSEC benchmark ported to OmpSs

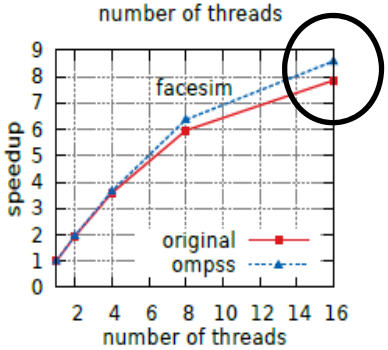
- Improved scalability … and LOC

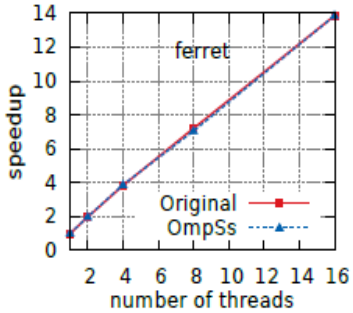


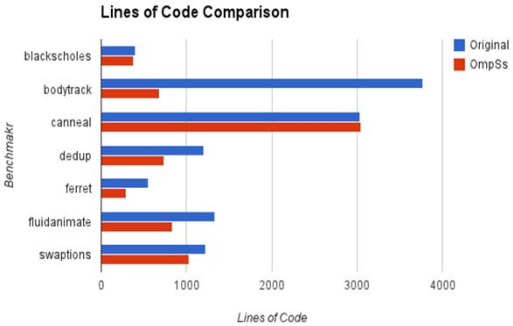


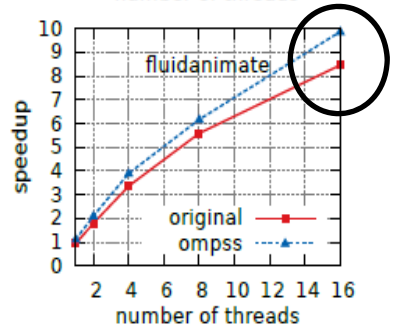


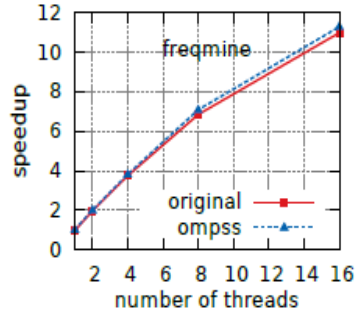


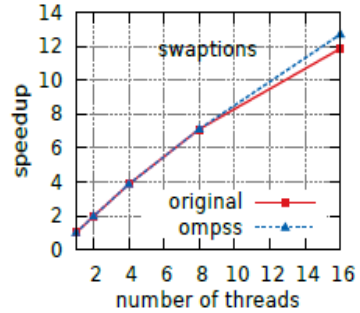








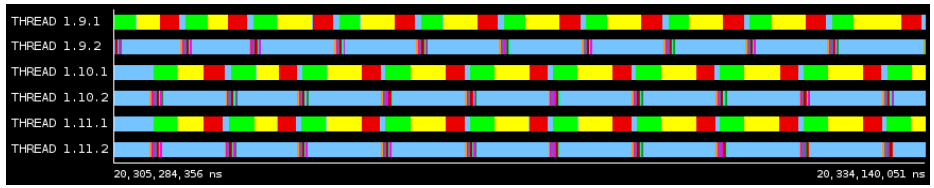


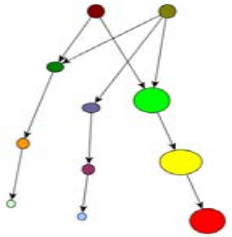


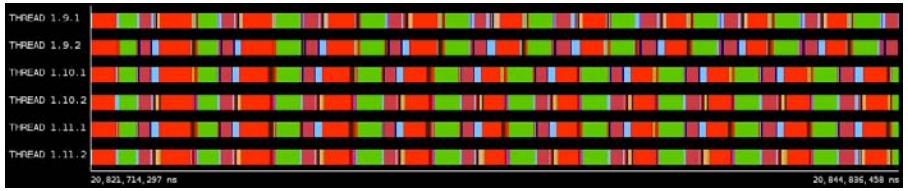
D. Chasapis et al., "Exploring the Impact of Task Parallelism Beyond the HPC Domain", Submitted

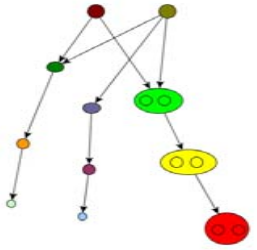
NMMB: Weather code + Chemical transport

- Eliminating latency sensitivity through nesting



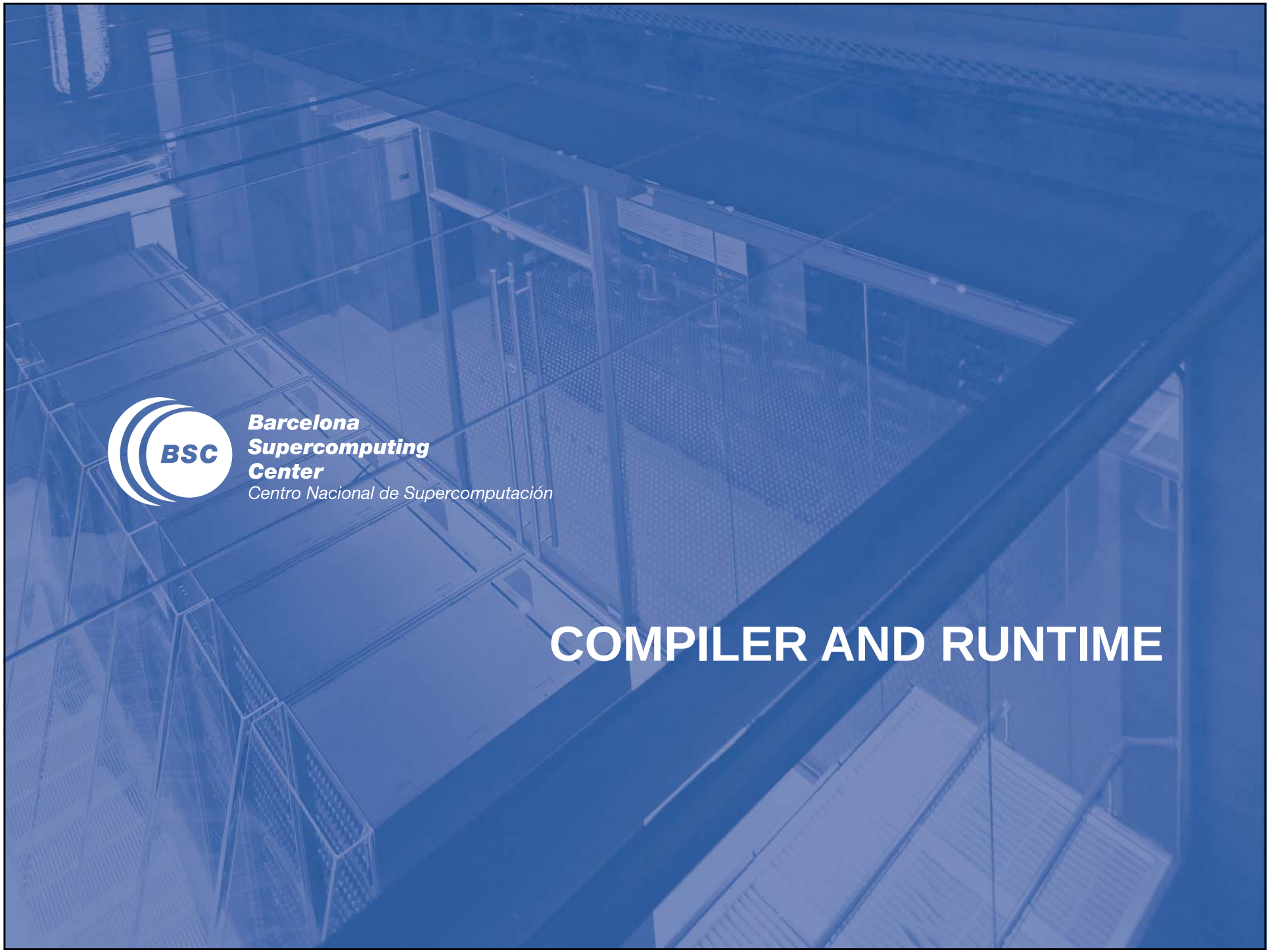




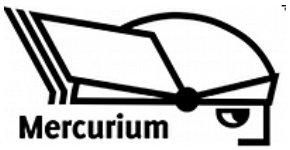


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

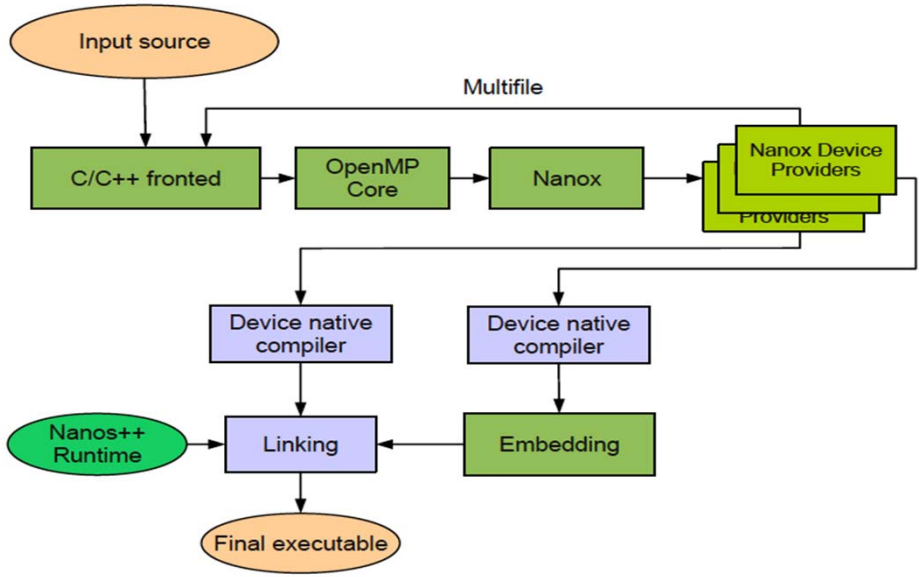
G. Markomanolis, "Optimizing an Earth Science Atmospheric Application with the OmpSs Programming Model" . PRACE days 2014


BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
COMPILER AND RUNTIME

The Mercurium Compiler



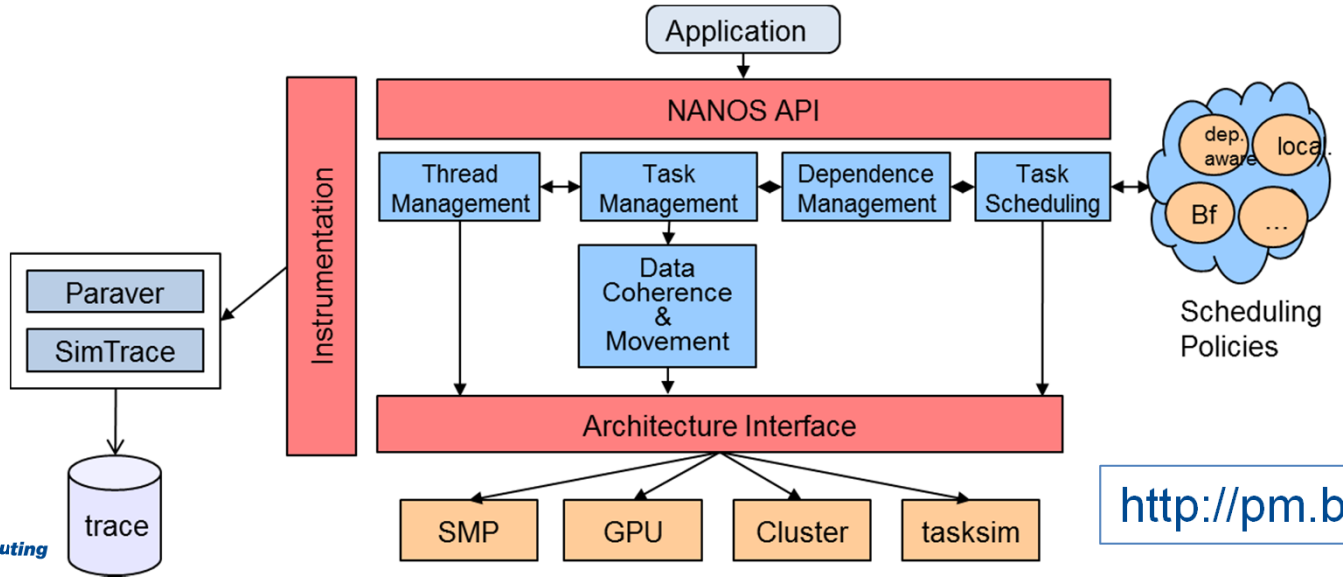
- Mercurium
  - Source-to-source compiler (supports OpenMP and OmpSs extensions)
  - Recognize pragmas and transforms original program to call Nanox++
  - Supports Fortran, C and C++ languages (backends: gcc, icc, nvcc, …)
  - Supports complex scenarios
    - Ex: Single program with MPI, OpenMP, CUDA and OpenCL kernels



http://pm.bsc.es

The NANOS++ Runtime

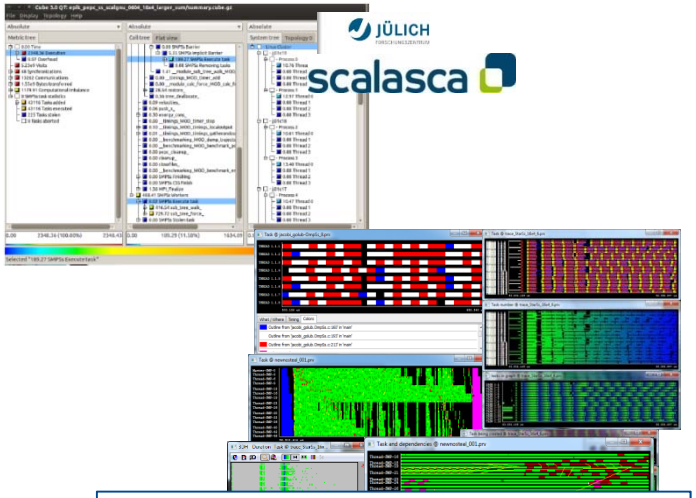
- Nanos++
  - Common execution runtime (C, C++ and Fortran)
  - Task creation, dependence management, resilience, …
  - Task scheduling (FIFO, DF, BF, Cilk, Priority, Socket, affinity, …)
  - Data management: Unified directory/cache architecture
    - Transparently manages separate address spaces (host, device, cluster)…
    - … and data transfer between them
  - Target specific features



http://pm.bsc.es

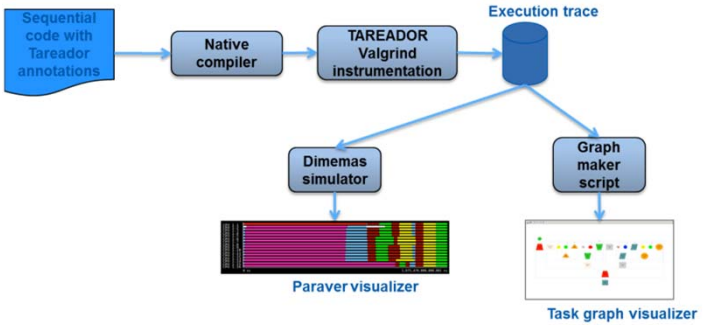
Support environment for dynamic task based systems

- Performance analysis Tools
  - Profiles
    - Scalasca @ SMPSs, OmpSs
    - Metrics, first order moments
  - Traces
    - Analysis of snapshots
    - Paraver instrumentation in all our developments

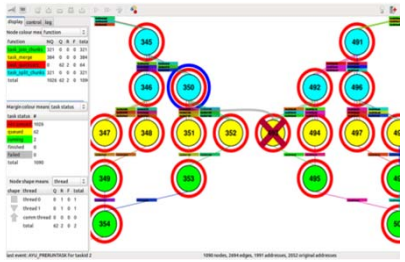


http://www.bsc.es/paraver

- Potential concurrency detection
  - Tareador



- Debugging
  - Temanejo



TEMANEJO

HLRS

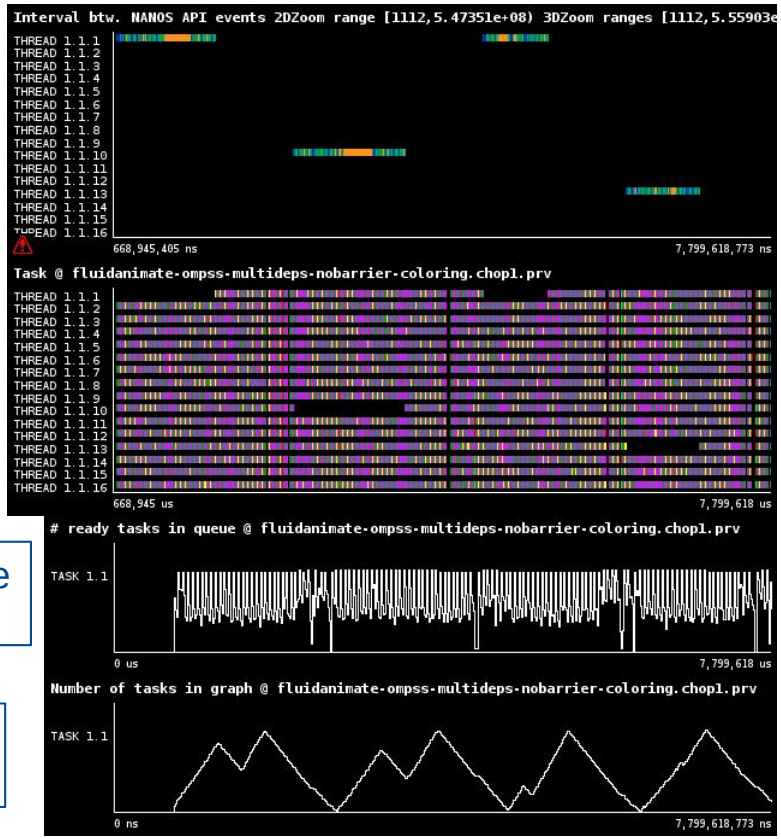
OmpSs instrumentation → Paraver

creation

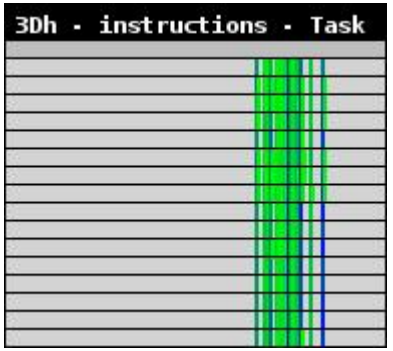
tasks

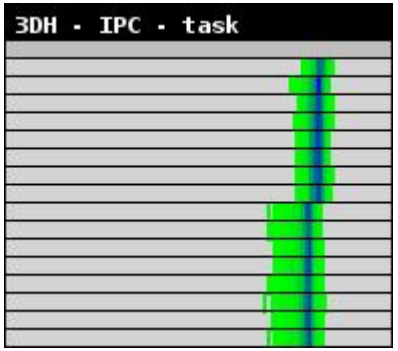
Ready queue
(0..65)

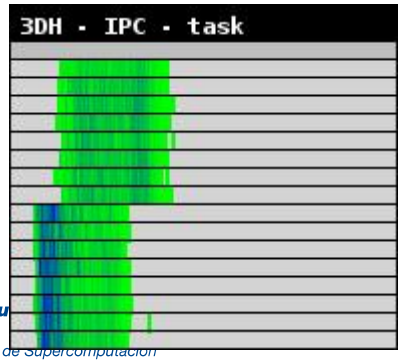
Graph size
(0..8000)

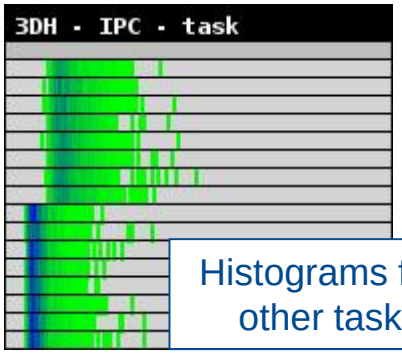


Histograms for task
“ComputeForcesMT”



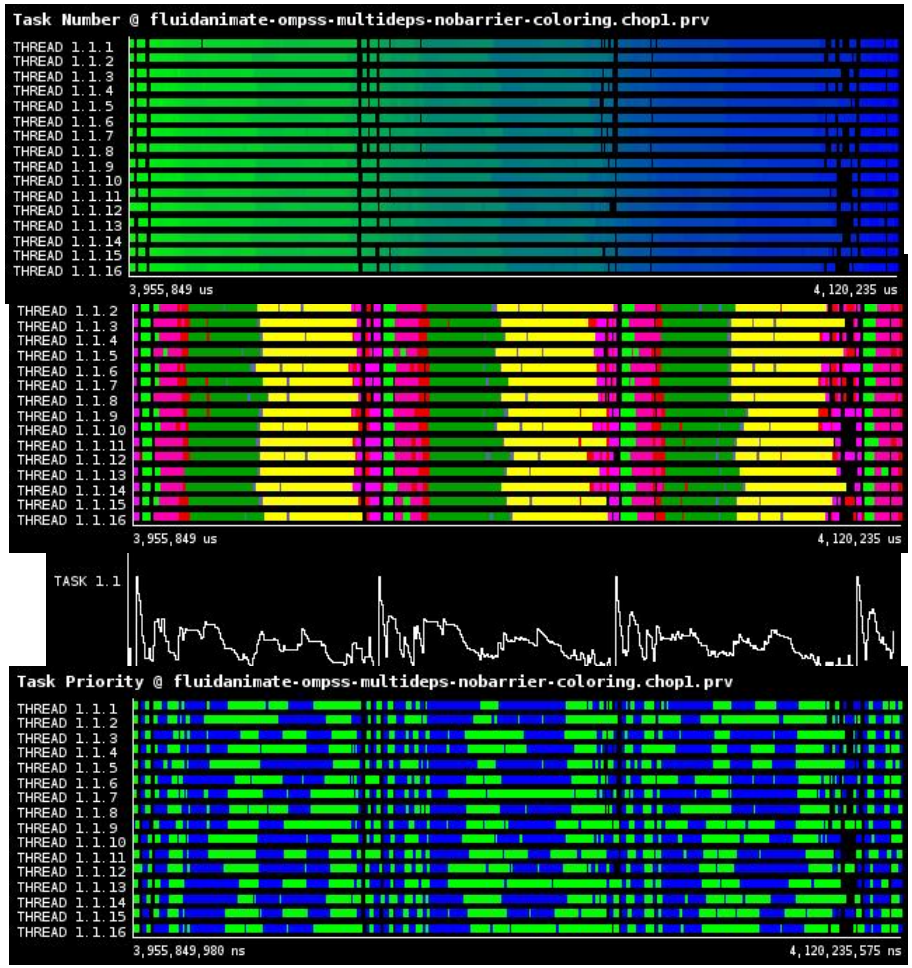


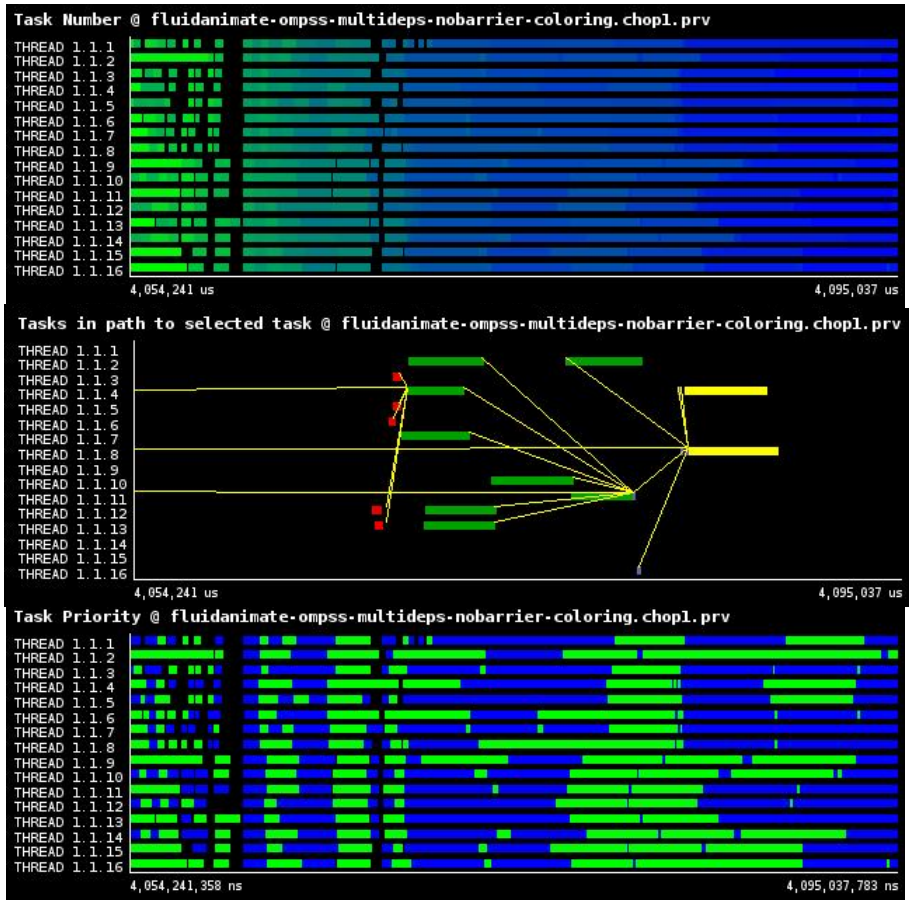




Histograms for
other task

OmpSs instrumentation → Paraver

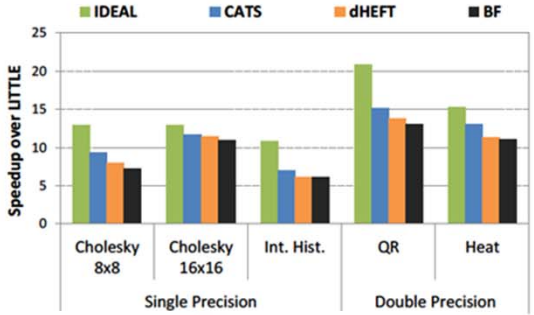


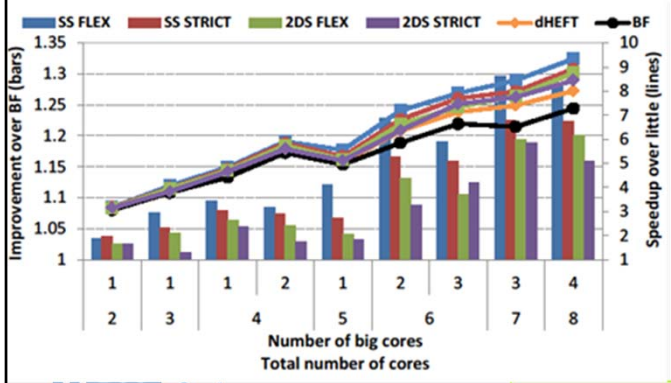


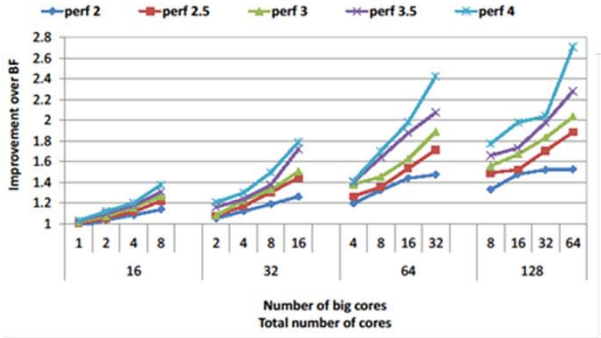
Criticality-awareness in heterogeneous architectures

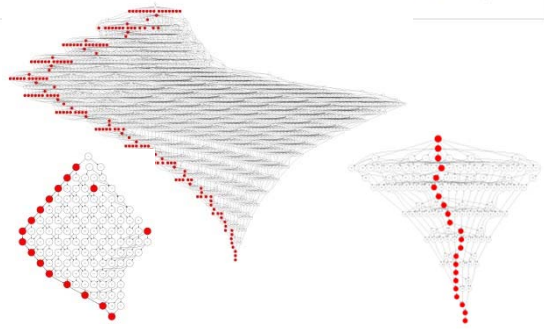
- Heterogeneous multicores
  - ARM biLITTLE 4 A-15@2GHz; 4A-7@1.4GHz
  - Tasksim simulator: 16-256 cores; 2-4x
- Runtime approximation of critical path
  - Implementable, small overhead that pay off
  - Approximation is enough
- Higher benefits the more cores, the more big cores, the higher performance ratio

Kernel	#Tasks	Measured Perf. Ratio
Cholesky 8x8	120	2.23
Cholesky 16x16	816	
Integral Histogram	2048	1.71
QR	1496	4.26
Heat difussion	5124	2.83





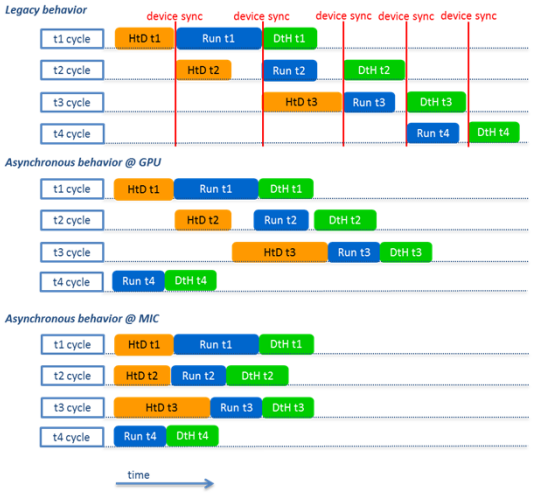




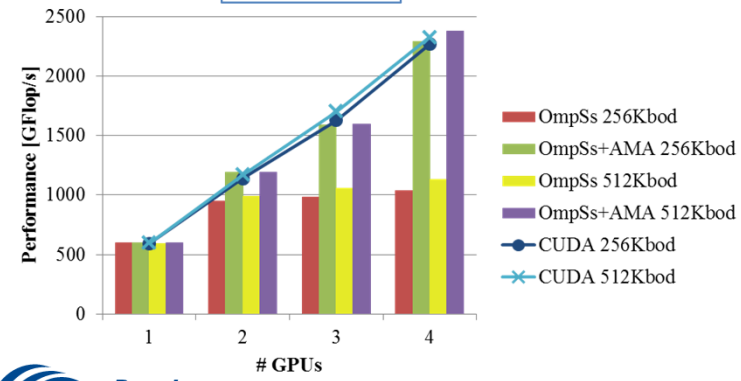
K. Chronaki et al, "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures." ICS 2015

OmpSs + CUDA runtime

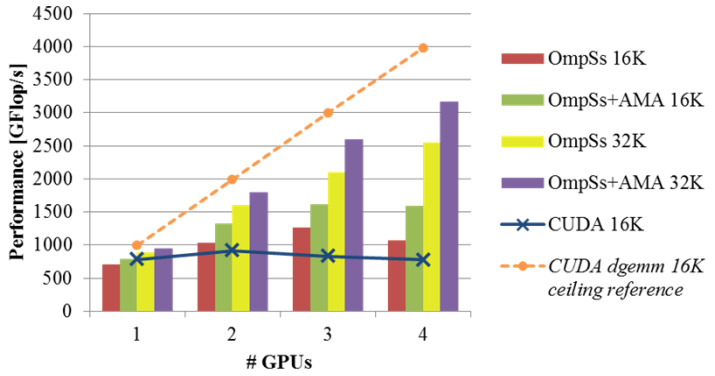
- Improvements in runtime mechanisms
  - Use of **multiple streams**
  - High asynchrony and overlap (transfers and kernels)
  - Overlap kernels
  - Take overheads out of the critical path
- Improvement in schedulers
  - Late binding of locality aware decisions
  - Propagate priorities



Nbody



Cholesky

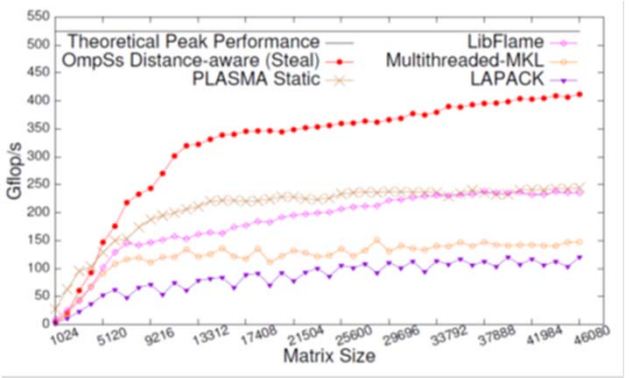


J. Planas et al, "AMA: Asynchronous Management of Accelerators for Task-based Programming Models." ICCS 2015

Scheduling

- Locality aware scheduling
  - Affinity to core/node/device can be computed based on pragmas and knowledge of where was data
  - Following dependences reduces data movement
  - Interaction between locality and load balance (work-stealing)

R. Al-Omairy et al, "Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing." Submitted



- Some "reasonable" criteria
  - Task instantiation order is typically a fair criteria
  - Honor previous scheduling decisions when using nesting
    - Ensure a minimum amount of resources
    - Prioritize continuation of a father task in a taskwait when synchronization fulfilled

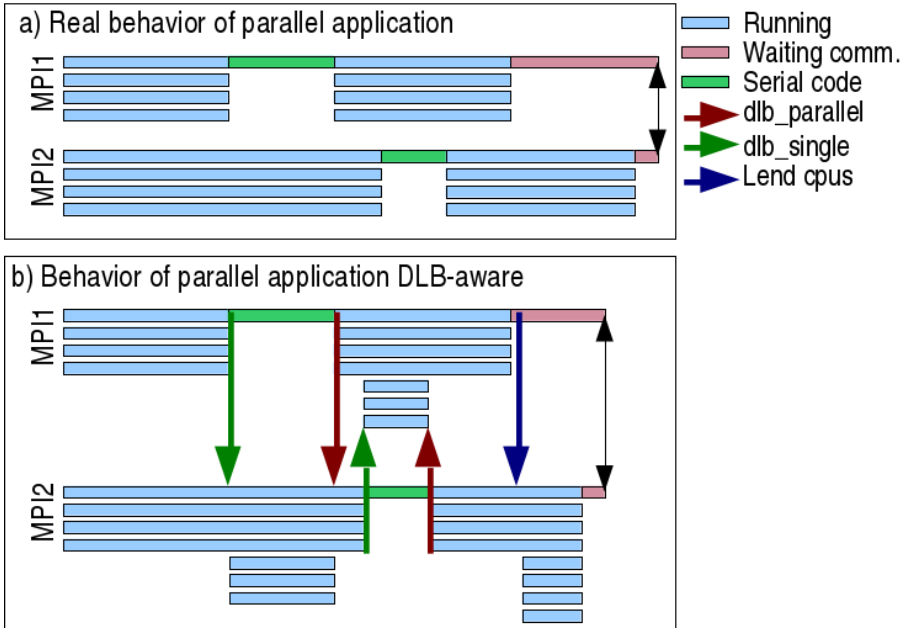

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
DYNAMIC LOAD BALANCING

Dynamic Load Balancing

- Automatically achieved by the runtime
  - Shifting cores **between MPI processes within node**
  - Fine grain
  - **Complementary to application level load balance**.
  - Leverage OmpSs malleability

dynamic load balancing

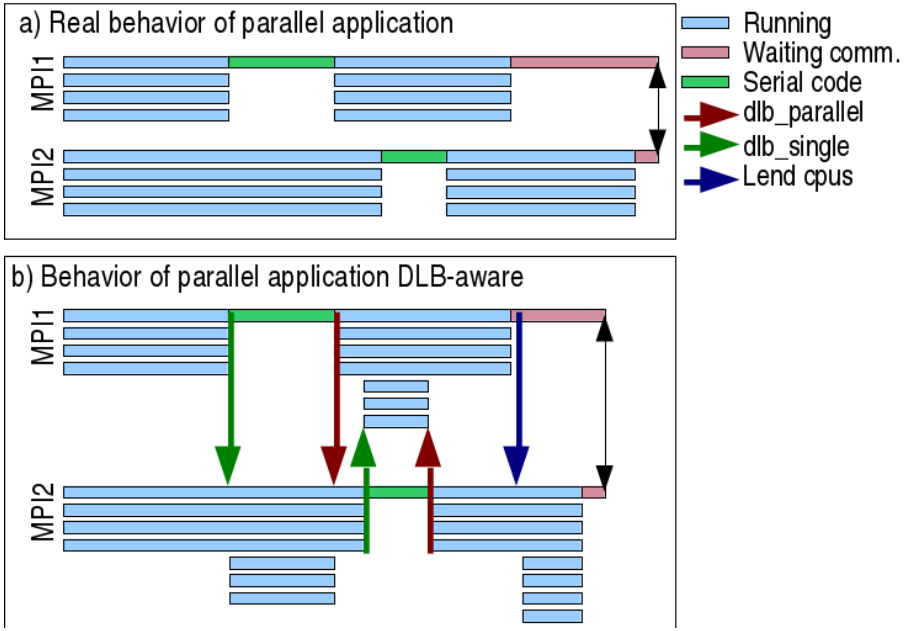
- DLB Mechanisms
  - User level Run time Library (DLB)
  - Detection of process needs
    - Intercepting runtime calls
      - Blocking
      - Detection of thread level concurrency
    - Request/release API
  - Coordinating processes within node
    - Through a shared memory region
    - Explicit pinning of threads and handoff scheduling (Fighting the Linux kernel)
  - Within and across apps



"LeWI: A Runtime Balancing Algorithm for Nested Parallelism". M.Garcia et al. ICPP09

Dynamic Load Balancing

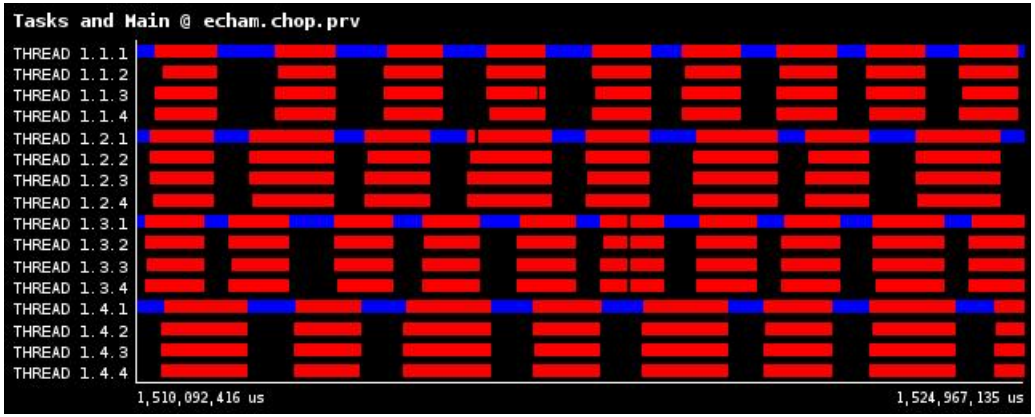
- DLB policies
  - LeWI: Lend core When Idle
  - …
- Support for “new” usage patterns
  - Interactive
  - System throughput
  - Response time

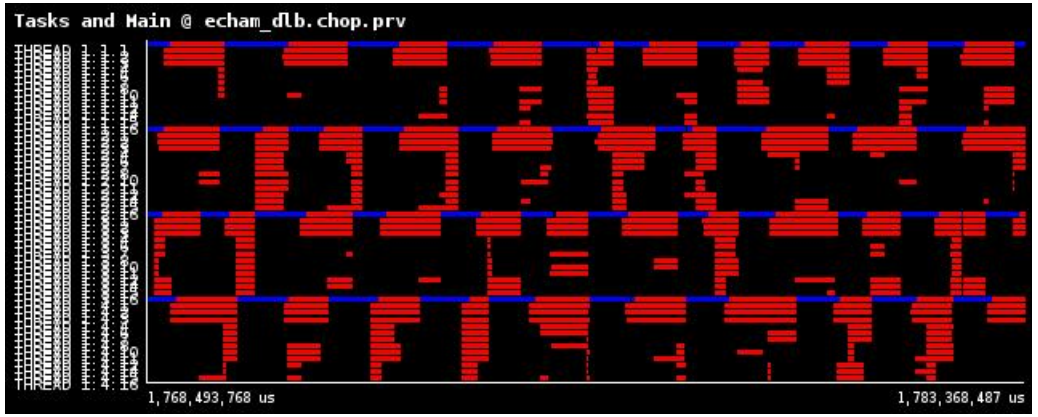


“LeWI: A Runtime Balancing Algorithm for Nested Parallelism”. M.Garcia et al. ICPP09

DLB @ ECHAM

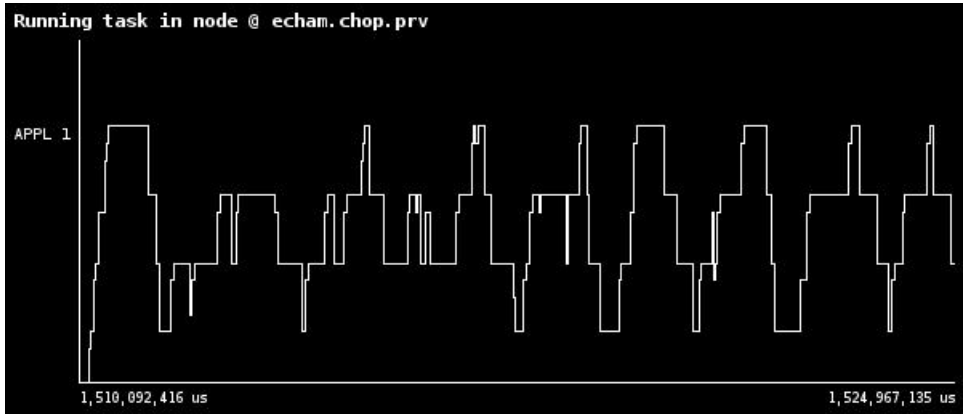
- Alternating parallelized and not parallelized
- Use API call to release/reclaim cores

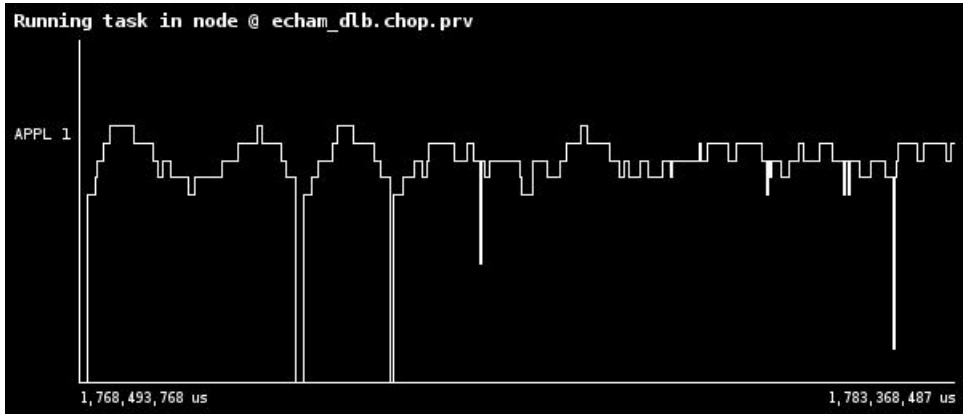




DLB @ ECHAM

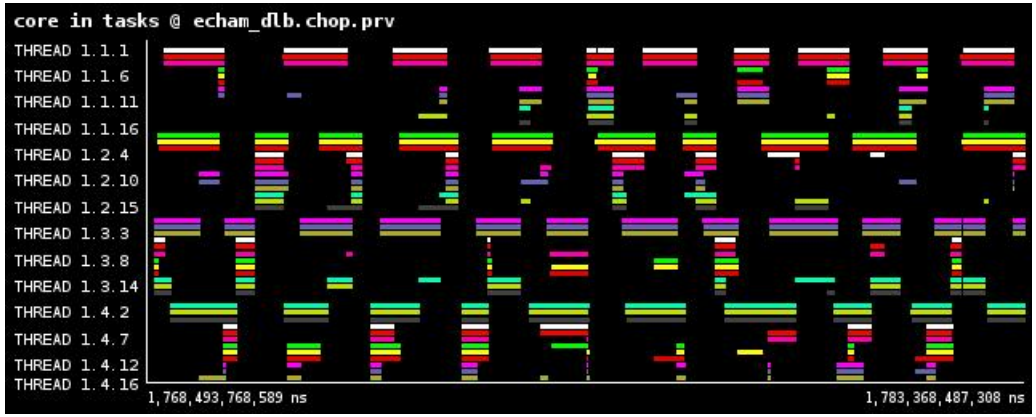
- Improved concurrency level
- Still some improvement possible





DLB @ ECHAM

- Tracking core migration

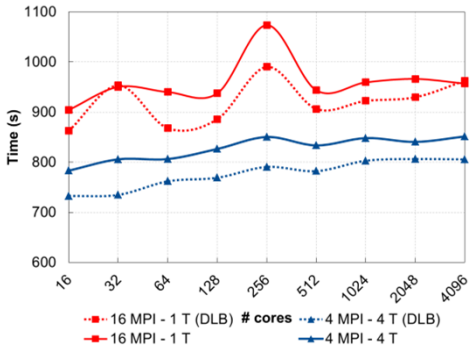


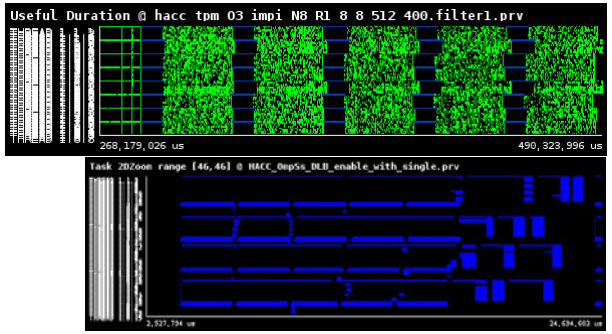
Dynamic Load Balancing

ECHAM

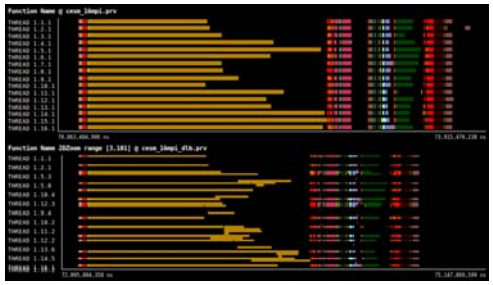
Problem size (NPY, MPX)	Mapping (nodes, ppns)	No DLB (s)	DLB (s)	Gain
2 x 2	1 x 4	2327.44	1541.47	~34%
4 x 2	2 x 4	1252.27	2915.92	~35%
4 x 4	4 x 4	811.27	1636.87	~44%

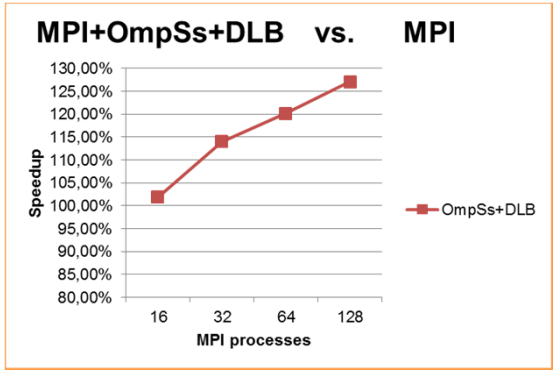
HACC





CESM



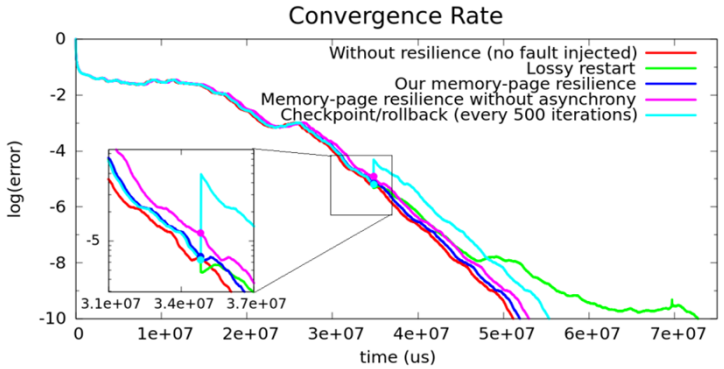


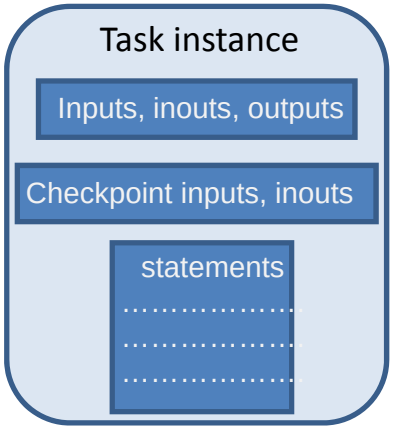

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
OTHER

OmpSs programming model

Resilience

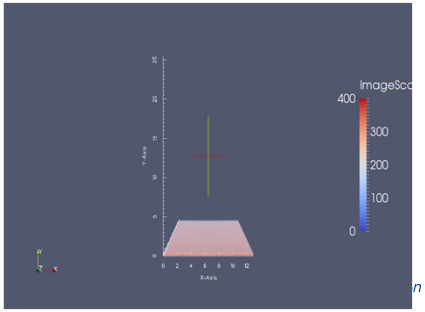
NANO-FT:: Task-level checkpoint/restart based FT
Algorithmic-based FT
Asynchronous task recovery





DSL and supporting DFL

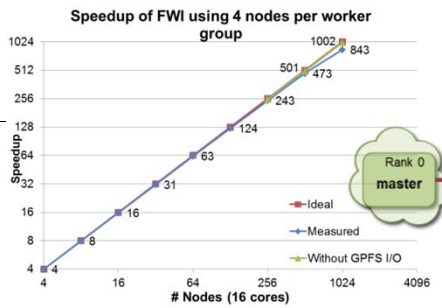
```
val c = Cartesian(12.5, 25.0, 37.5)
val temp = Unknown(c)
val cond = Dirichlet(lowXZ of c, temp, 400)
val hv = Vector(0.5, 0.5, 0.5)
val pre = PreProcess(nsteps = 10000, deltaT = 0.25, h = hv)(cond)
solve(pre) equation (0.15 * lapla(temp) - dt(temp)) to "diffusion"
```

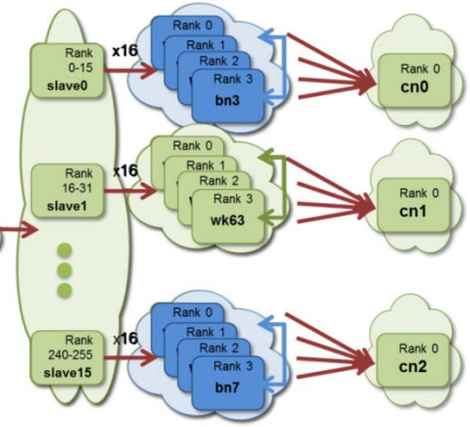


Support multicores, accelerators and distributed systems

CASE/REPSOL **Repsolver**

MPI offload





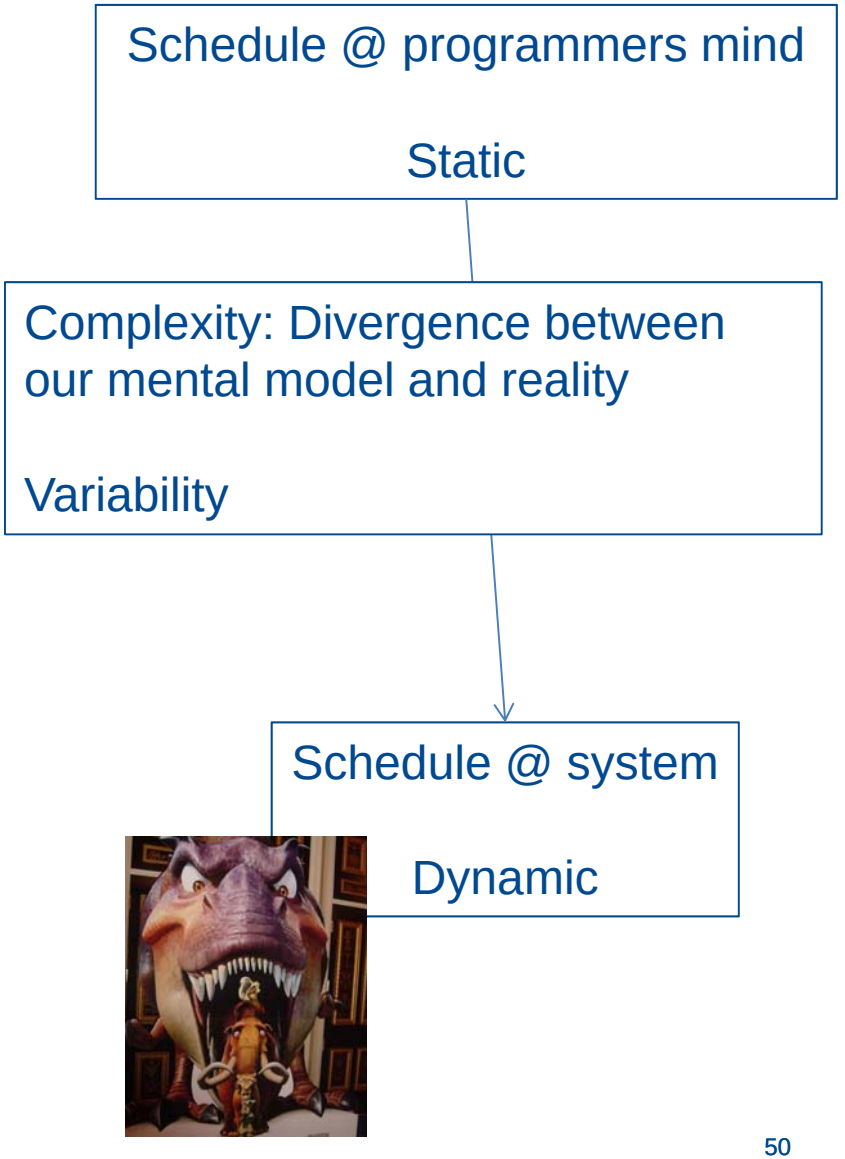
CASE/REPSOL FWI

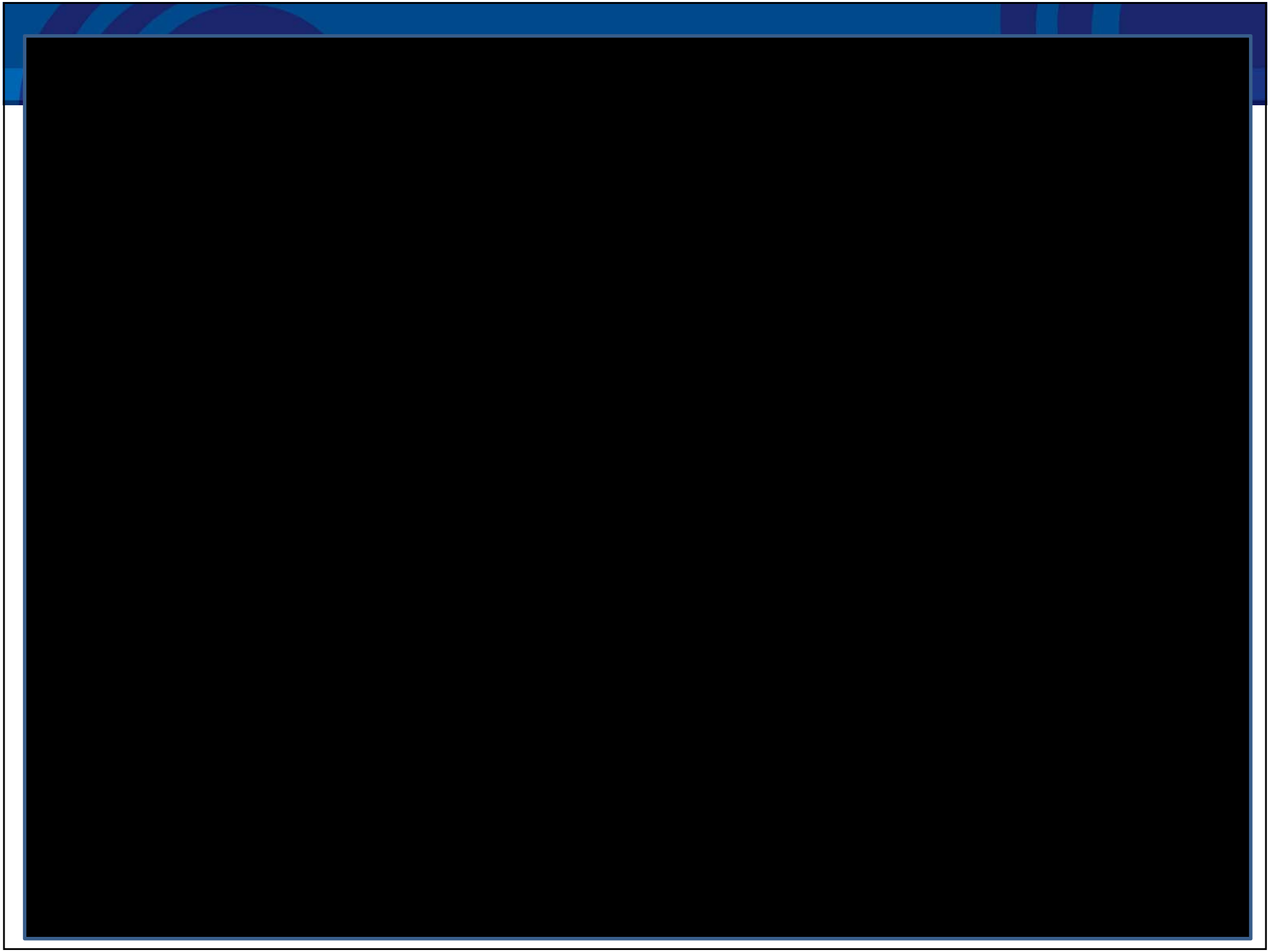

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
CONCLUSION

The parallel programming revolution

- Parallel programming in the past
  - Where to place data
  - What to run where
  - How to communicate
  - Talk to Machines
  - Dominated by Fears/Prides

- Parallel programming in the future
  - What data do I need to use
  - What do I need to compute
  - hints (not necessarily very precise) on potential concurrency, locality,…
  - Talk to Humans
  - Dominated by Semantics






BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
THANKS