

[www.bsc.es](http://www.bsc.es)



**Barcelona  
Supercomputing  
Center**  
*Centro Nacional de Supercomputación*

# From OmpSs to the OpenMP Standard

**Xavier Martorell, Sergi Mateo, Xavier Teruel, Josep M. Pérez,  
Vicenç Beltran, Eduard Ayguadé and Jesús Labarta**



OpenMPCon '17  
Stony Brook, NY, USA, September 18th-19th, 2017

# History



UPC – Universitat Politècnica de Catalunya

1971



FIB (Computer Science Faculty) started in October

1977

– Now, 40 years anniversary!!

DAC (Computer Architecture Department) started in

1978

– Tomas Lang, Mateo Valero

CEPBA – European Center for Parallelism in Barcelona

1991

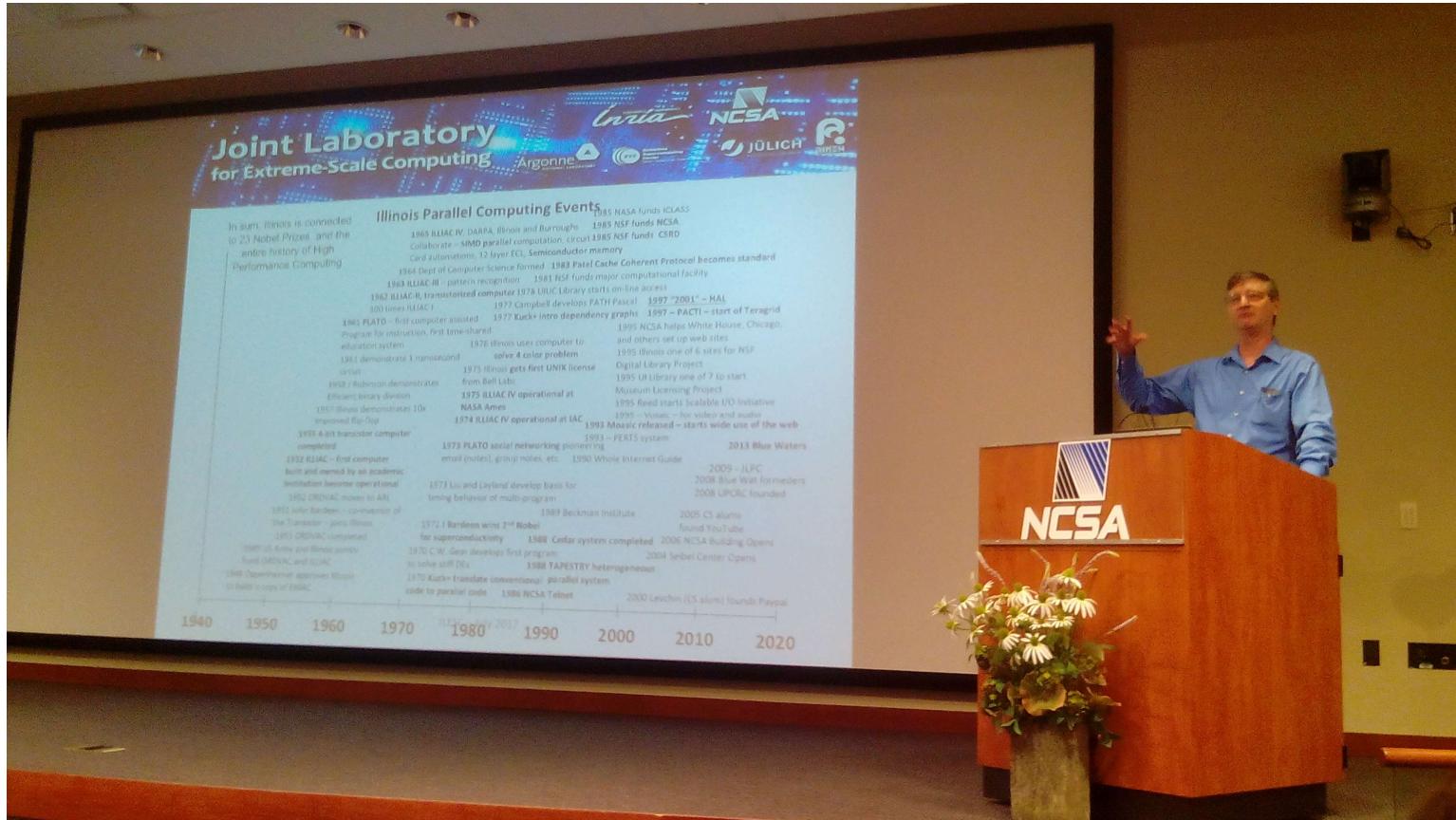
BSC – Barcelona Supercomputing Center

2005

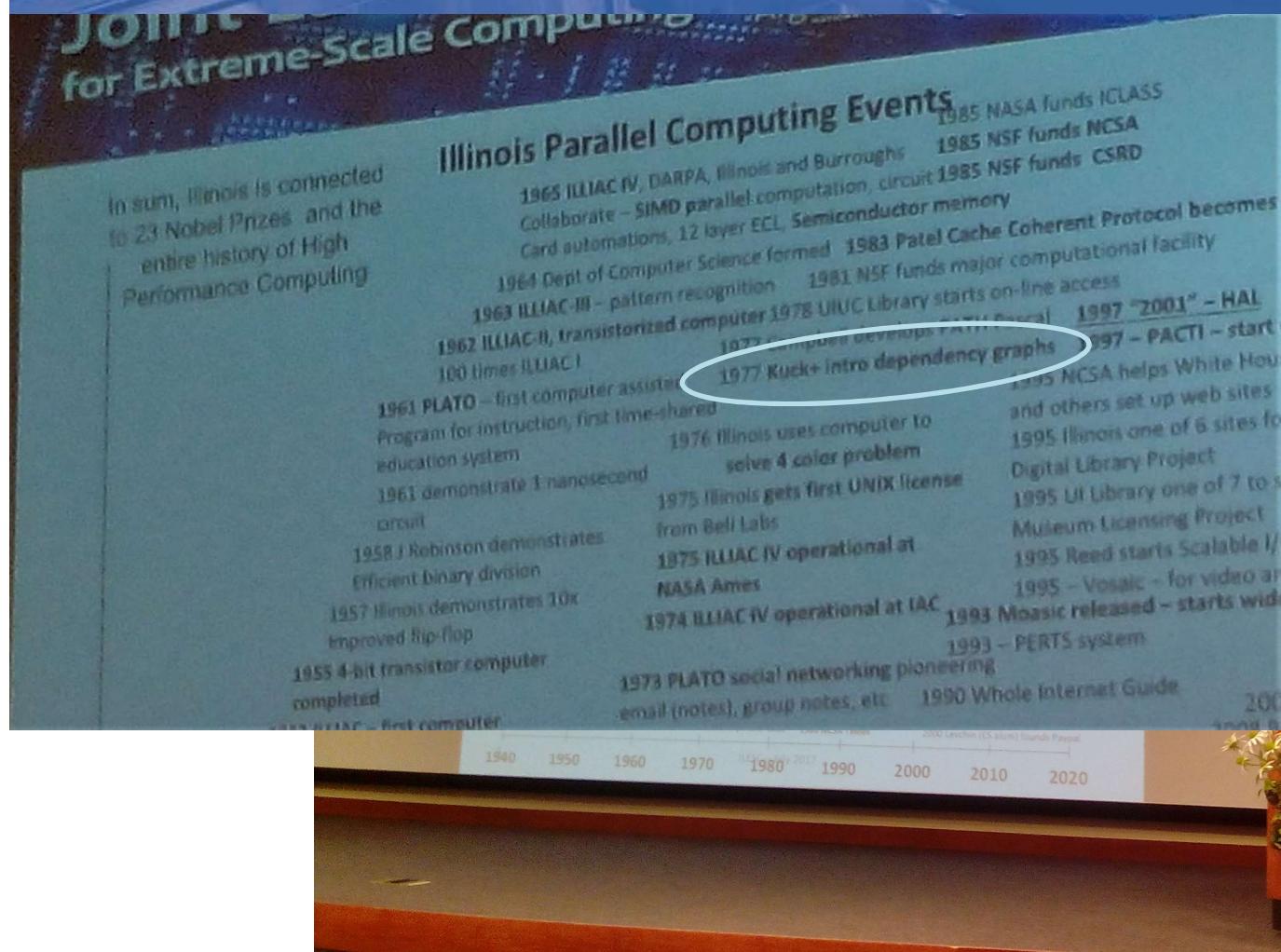
# Origins of task dependences



My take from this past July JLESC meeting at UIUC



# Origins of task dependences



1977 Kuck+  
Intro dependency graphs



# Tradition on Parallel Programming



The group started publishing on parallel programming (1989)

Jesús Labarta, Eduard Ayguadé: GTS: Extracting Full Parallelism Out of DO Loops.  
PARLE (2) 1989: 43-54

## GTS: Extracting Full Parallelism Out of DO Loops

Jesús Labarta and Eduard Ayguadé.

Departament d'Arquitectura de Computadors  
Universitat Politècnica de Catalunya (U.P.C.)

### Abstract.

In this paper we present a new method for extracting the maximum parallelism out of DO loops with tight recurrences in a sequential programming language. We have named the method **Graph Traverse Scheduling** (GTS). It is devised for producing code for shared memory multiprocessors. Hardware support for fast synchronization is assumed.

Based on the dependence graph of a loop we first show how its parallelism can be evaluated. Then we apply GTS to distribute iterations of a recurrence between tasks. With

# Motivation



Let's help programmers with higher level abstractions

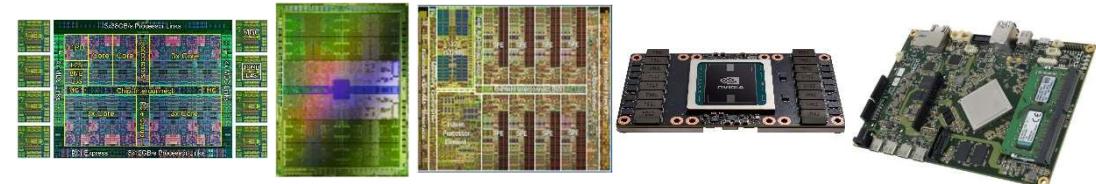
- Tasking
  - » basics
  - » dependences
  - » priorities
  - » task loops
- SIMD code generation
- Heterogeneity support
- Better decomposition of applications targetting parallelism exploitation
  - » Coarse grain
  - » Distant parallelism

Applications

PM: High-level, clean, abstract interface

Power to the runtime

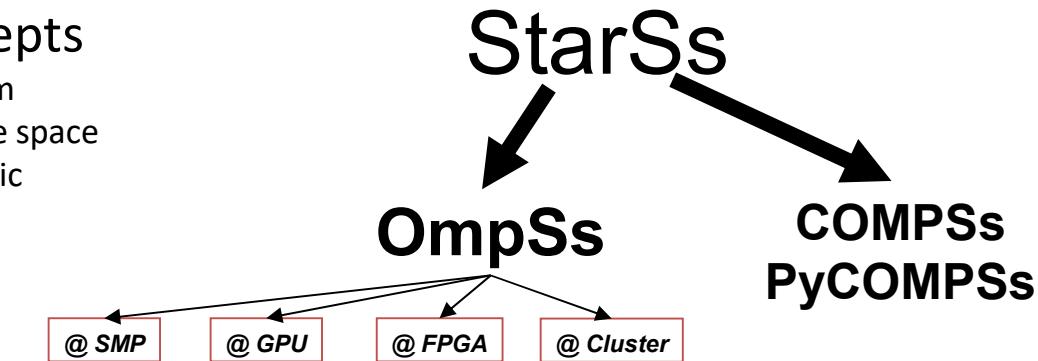
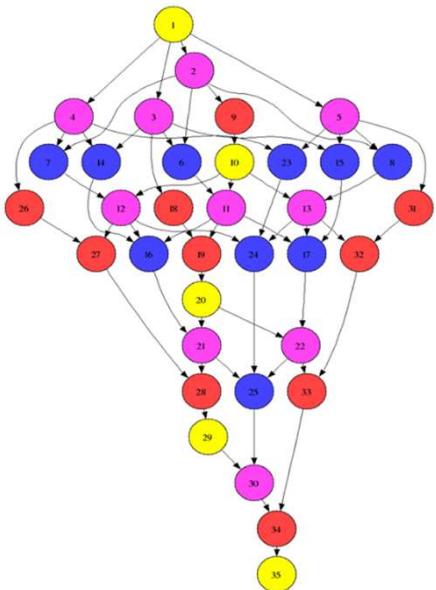
ISA / API



# Task-based programming



- StarSs family key concepts
  - Sequential task-based program
  - View of a single address/name space
  - Execution in parallel: automatic
  - runtime computation of dependencies
- Productivity and portability



- OmpSs is used for prototyping extensions to OpenMP
  - Tasking
  - data-dependences
  - Heterogeneity
  - Multiple address spaces
  - ...
- Mercurium compiler
- Nanos runtime system



[www.bsc.es](http://www.bsc.es)



**Barcelona  
Supercomputing  
Center**

*Centro Nacional de Supercomputación*

# Influence of OmpSs on OpenMP

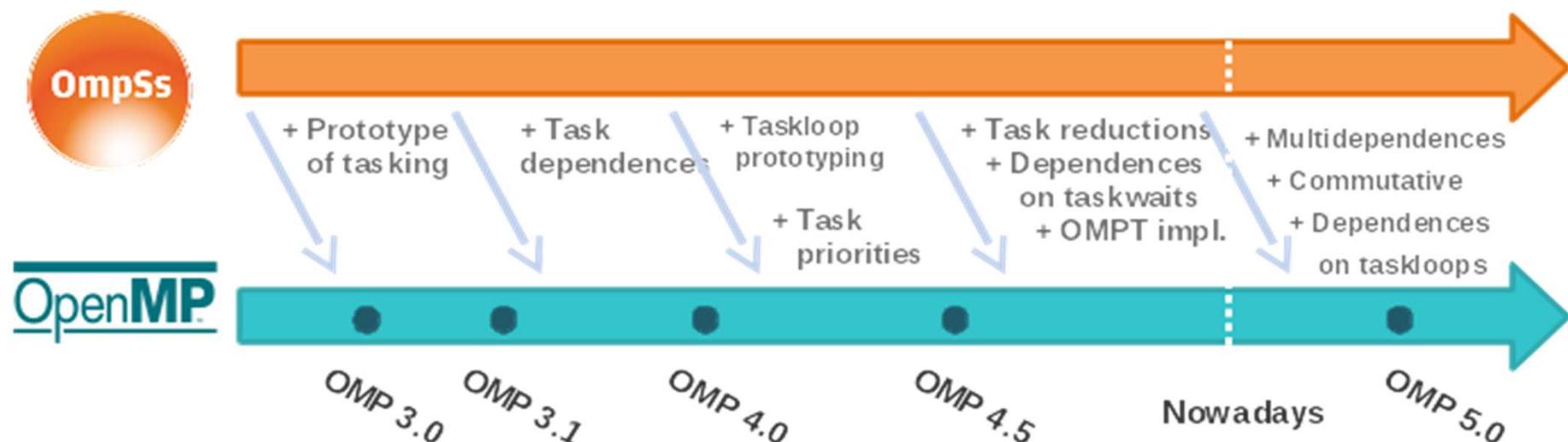
OpenMPCon

Stony Brook, NY, USA September 18th-19th, 2017

# Time line

## Collaboration with OpenMP initiated in the 90's

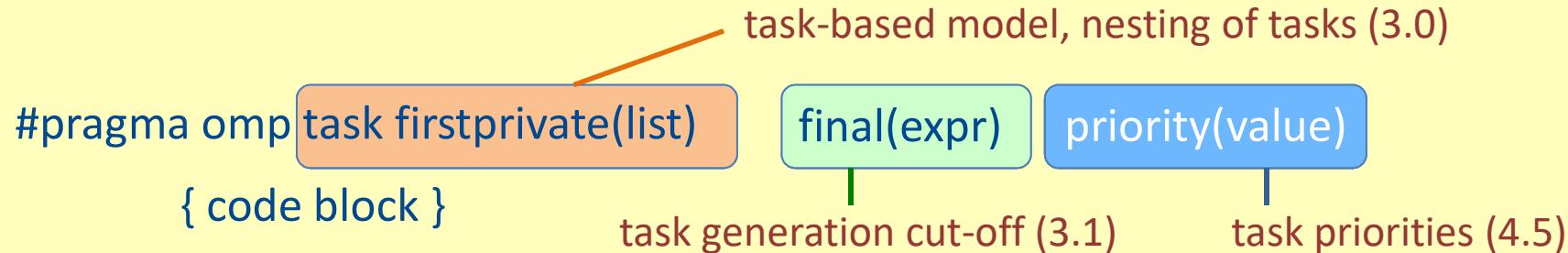
- Around the Parallel Computing Forum, within FORTRAN
- The Nanos EU project, and the nano-threads library, EWOMP 1999
- Support for multiple levels of parallelism, EWOMP 2001
- NUMA support and page migrations, WOMPAT 2001
- Pipelined executions, EWOMP 2000, WOMPAT 2001
- `auto` scheduling clause for loops, WOMPAT 2003



# Tasking model

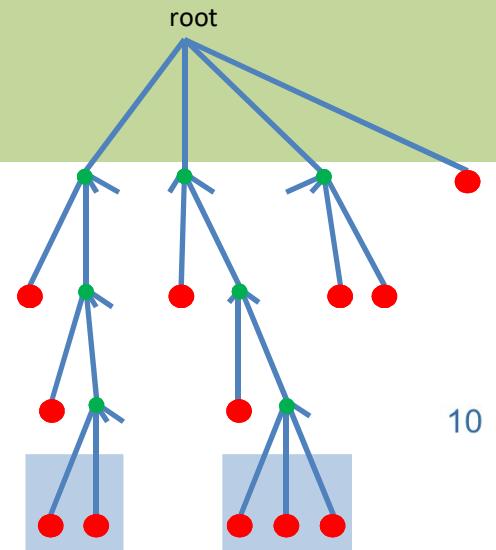


The big change...



**#pragma omp taskwait** (orange box) is associated with **task generation waits for child tasks to finish (3.0)**.

- From a loop-based model to a task-based model
  - » Dynamically allocated data structures
    - *lists, trees, graphs...*
  - » Loop-based irregular parallelism
  - » Recursion

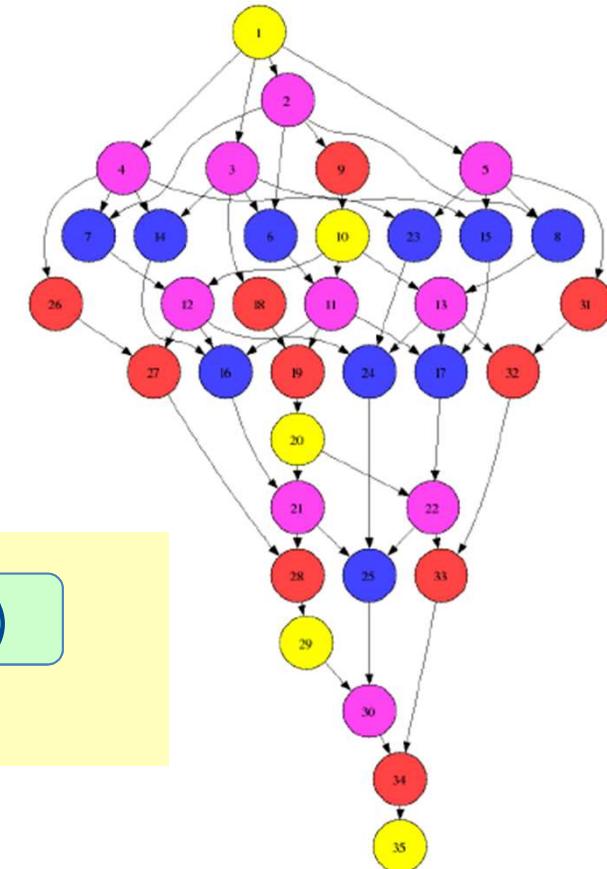


# Task dependences

- Programmer expresses directionality of task variables
- Actual dependences between tasks computed at runtime
- Tasks are executed as soon as all their dependences are satisfied
  - » Provided there are resources available

directionality for task arguments (4.0)

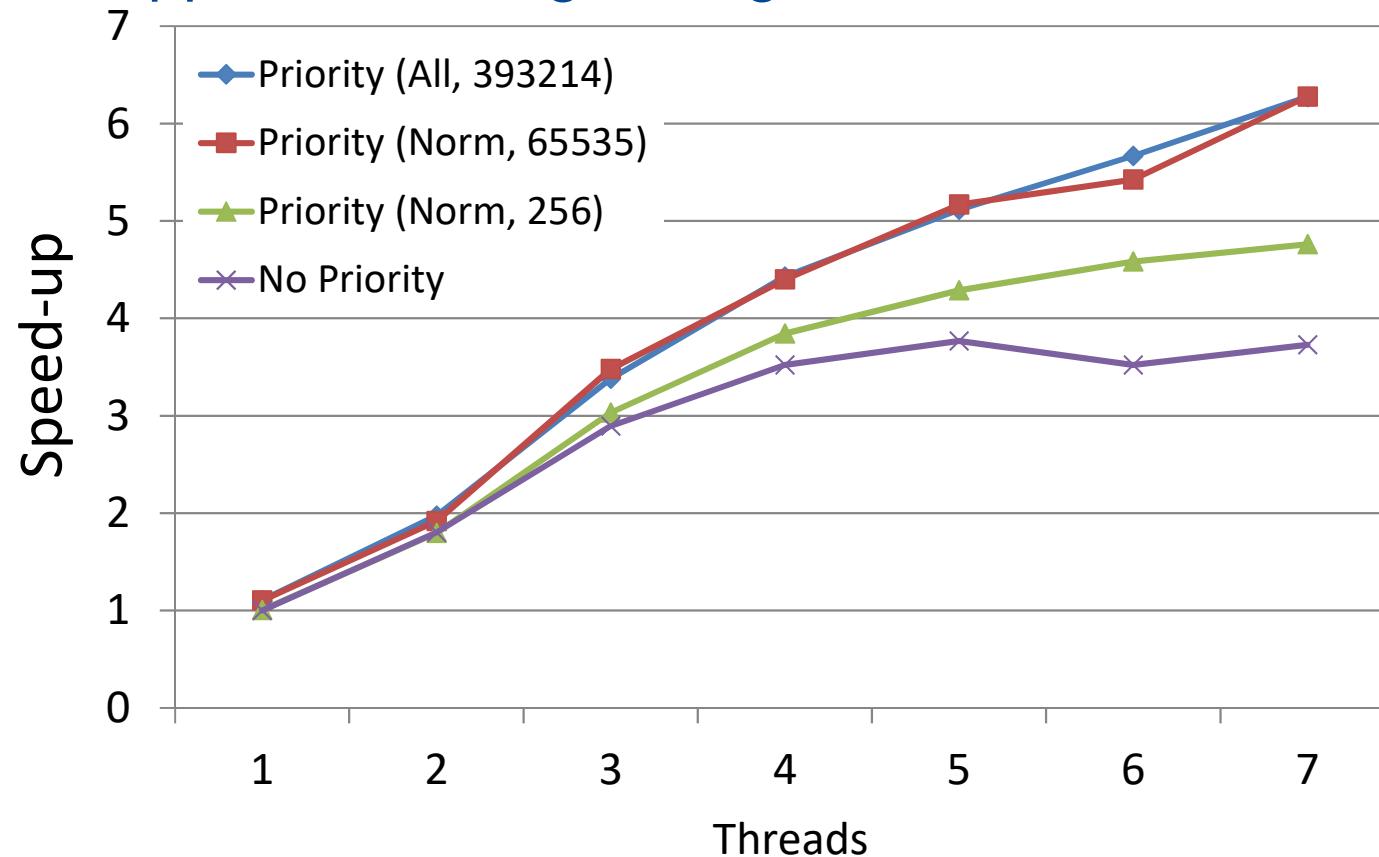
```
#pragma omp task depend(in: list, out: list, inout: list)  
{ code block }
```



# Prototyping task priorities



Determined that reducing the priority range may be important for some applications. E.g., merge-sort



# Taskloops



Chunks of iterations of one or more loops become tasks

- Do code conversion that is tedious and error-prone for the programmer

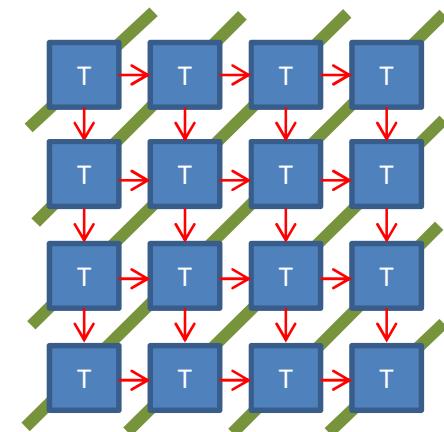
```
#pragma omp taskloop [num_tasks(value) | grainsize(value)]
```

for-loops

tasks out of loops and granularity control (4.5)

- Dependences between tasks generated in the same or different taskloop under discussion for 5.0

```
#pragma omp taskloop block(2) grainsize(B, B)
    depend(in : block[i-1][j], block[i][j-1])
    depend(in : block[i+1][j], block[i][j+1])
    depend(out: block[i][j])
for (i=1; i<n i++)
    for (j=1; j<n; j++)
        foo(i, j);
```



# Taskgroups & task reduction



We used the taskgroup functionality to propose the task reductions implementation

```
#pragma omp taskgroup reduction(+: res)
{
    while (node) {
        #pragma omp task in_reduction(+: res)
        res += node->value;

        node = node->next;
    }
}
```

— Plans for taskloop reductions also in 5.0

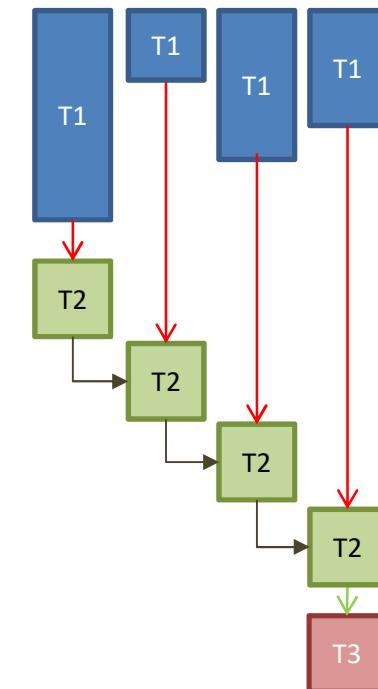
```
#pragma omp taskloop grainsize(BS) reduction(+: res)
for (i = 0; i < N; ++i) {
    res += foo(v[i]);
}
```

# inout dependence chain



Strict in-order execution (in order of creation),  
with mutual exclusion

```
for (i = 0; i < l.size(); ++i) {  
    #pragma omp task depend(out: l[i])  
    // T1  
}  
  
for (i = 0; i < l.size(); ++i) {  
    #pragma omp task shared(x) depend(in: l[i])  
        depend(inout: x)  
    // T2  
}  
  
#pragma omp task shared(x) depend(in: x)    // T3  
for (i = 0; i < l.size(); ++i) {  
    ...  
}
```



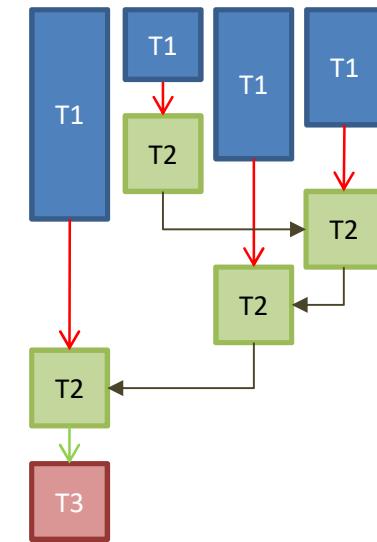
# Commutative dependences



## Relaxing inout dependence type

- Allows any order of execution, mutual exclusion is kept

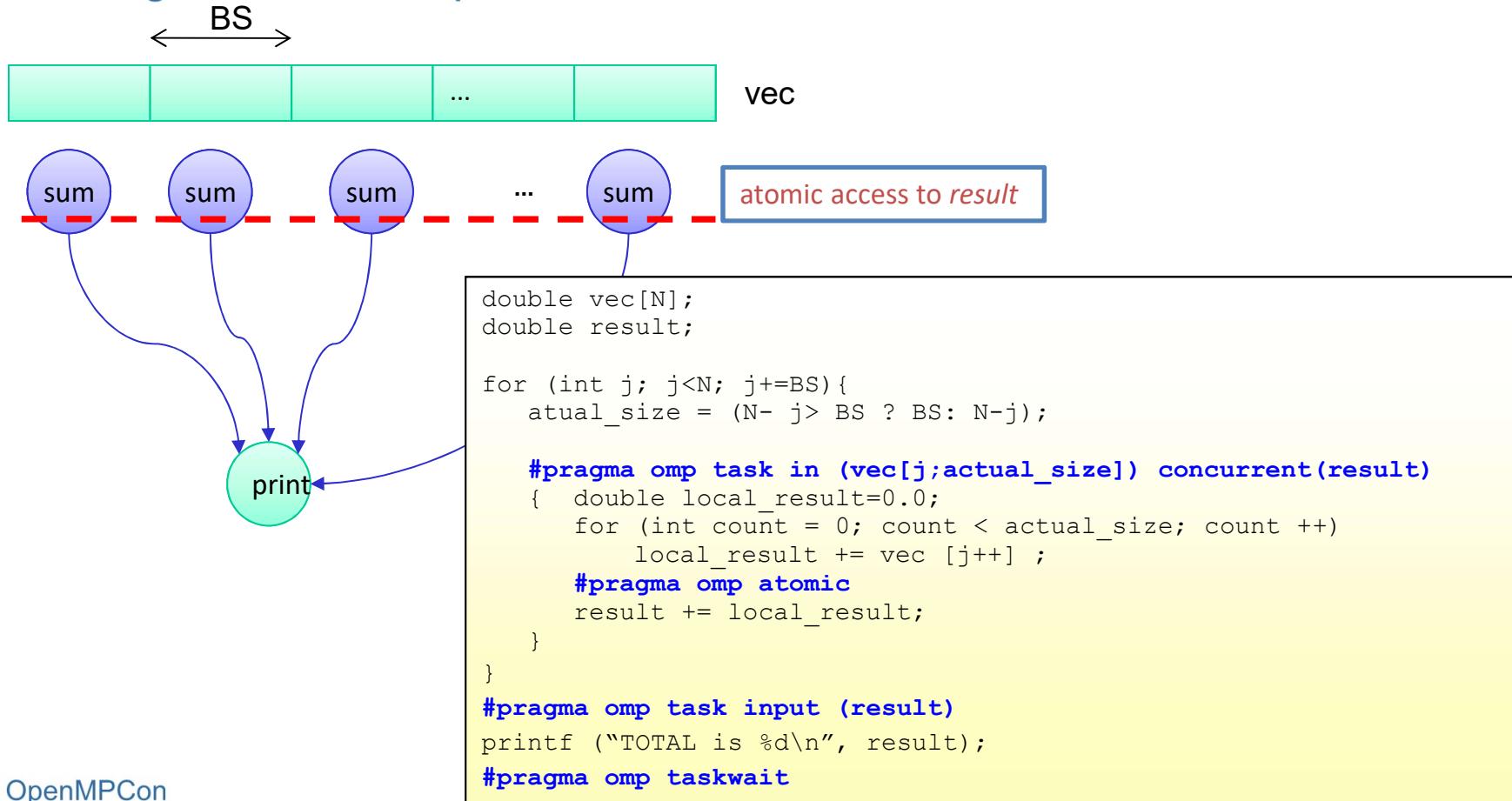
```
for (i = 0; i < l.size(); ++i) {  
    #pragma omp task depend(out: l[i])  
    // T1  
}  
  
for (i = 0; i < l.size(); ++i) {  
    #pragma omp task shared(x) depend(in: l[i])  
        depend(commutative: x)  
    // T2  
}  
  
#pragma omp task shared(x) depend(in: x)    // T3  
for (i = 0; i < l.size(); ++i) {  
    ...  
}
```



# An alternative: concurrent



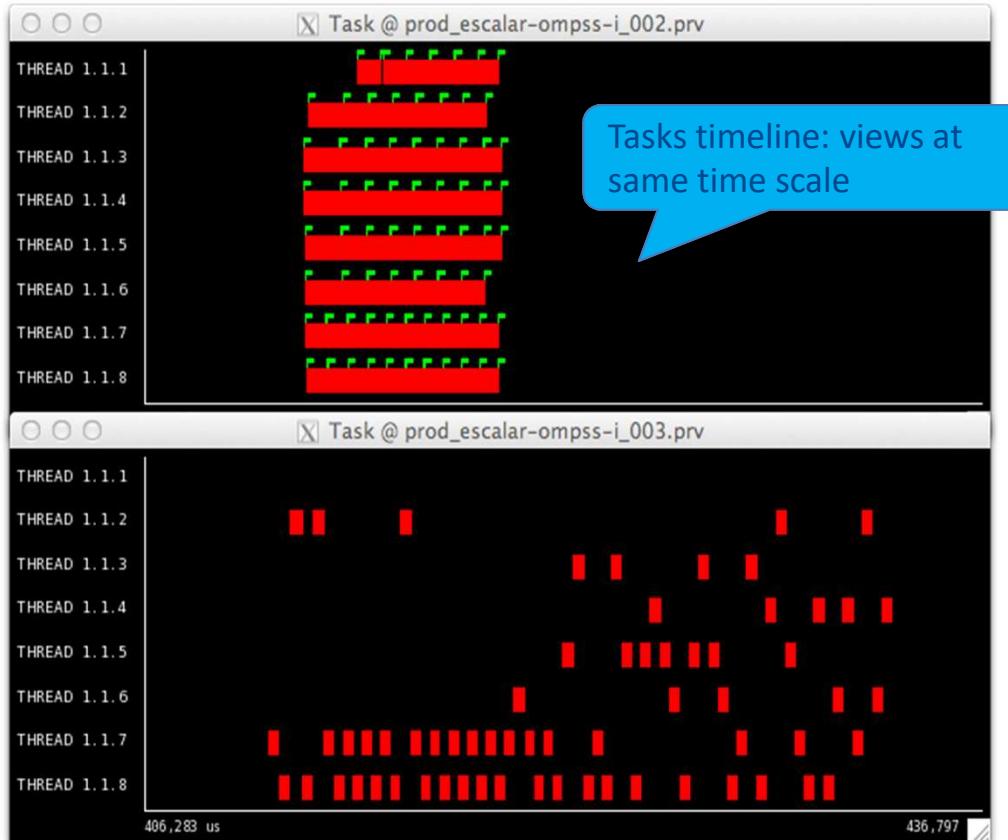
Further relax execution, now they may execute in parallel  
– Programmer is responsible of mutual exclusion to access shared data



# Differences between concurrent and commutative

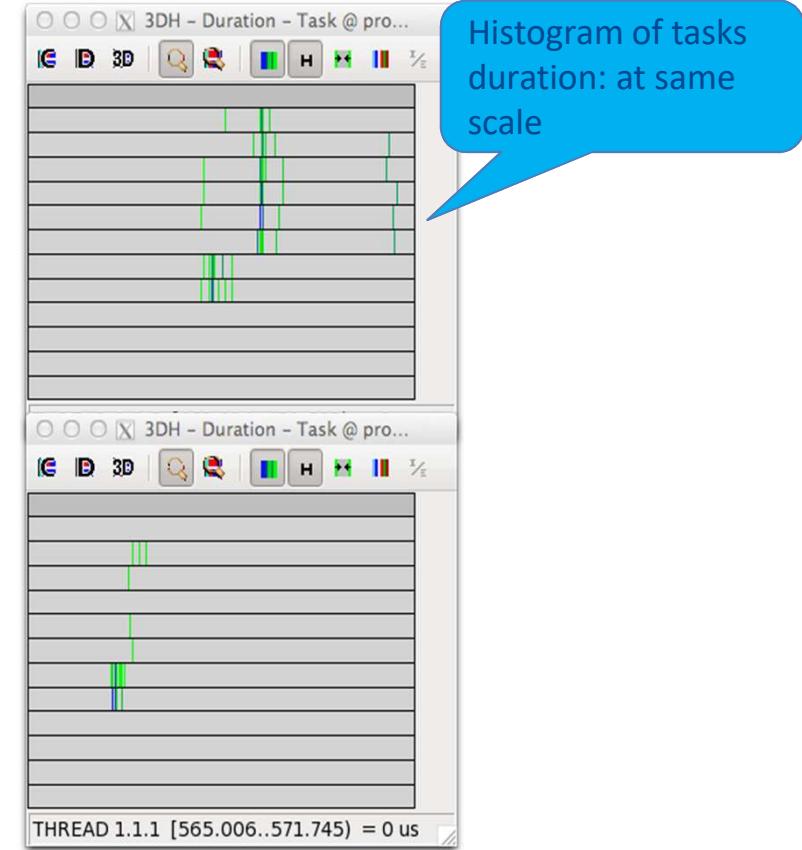


## Execution and time variability of tasks



In this case, concurrent is more efficient

... but tasks have more duration and variability

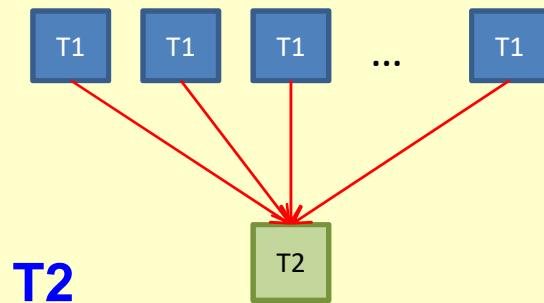


# Multidependences



Sometimes applications have a variable number of data dependences

```
for (i = 0; i < l.size(); ++i) {  
    #pragma omp task depend(inout: l[i])      // T1  
    { ... }  
}  
  
#pragma omp task depend(in: {l[j], j=0:l.size()}) // T2  
for (i = 0; i < l.size(); ++i) {  
    ...  
}
```



Defines an iterator with a range of values that only exists in the context of the directive. Equivalent to:

```
depend(in: l[0], l[1], ..., l[l.size()-1])
```

# Multidependences



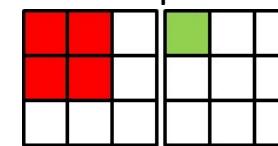
Sometimes applications have a variable number of data dependences

- Due to conditions determined at runtime

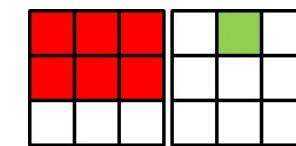
```
// Each iteration execute 8 different kernels. For the sake of simplicity we only show one

for (int i = 0; i < threadnum; ++i) {
    #pragma omp task out(y[i]) \
    in({z[neighborhood[i][j]], \
        j=0:neighborhood[i].size()})
}
RebuildGridMT(i);
}
```

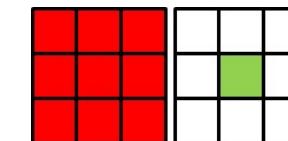
The number of dependences depend on the position in the mesh



Element in the corner:  
4 dependences



Element in the halo:  
6 dependences



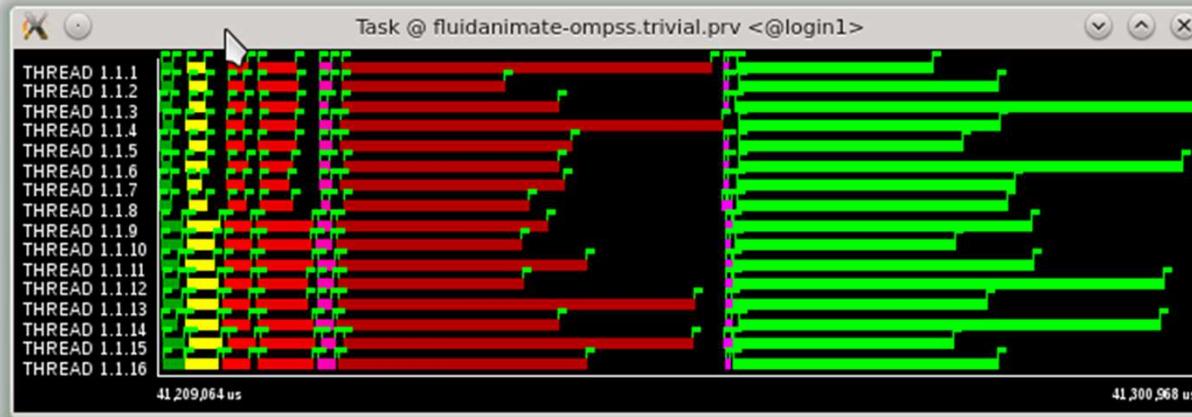
Element in the center: 9 dependences

**Multidependences increase programmability!**

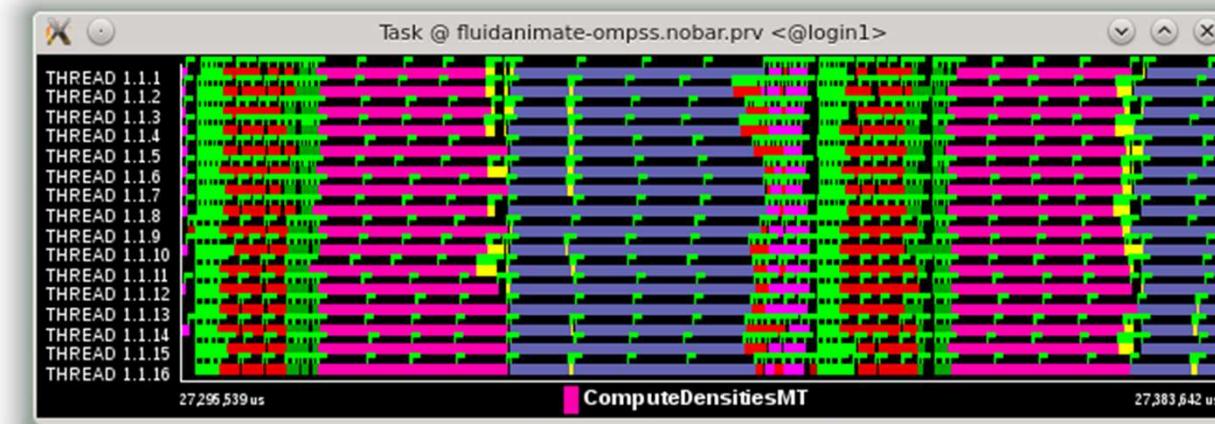
# Multidependences



## Comparing naive parallelization



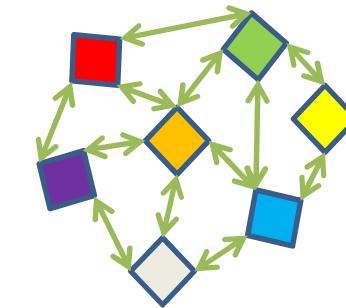
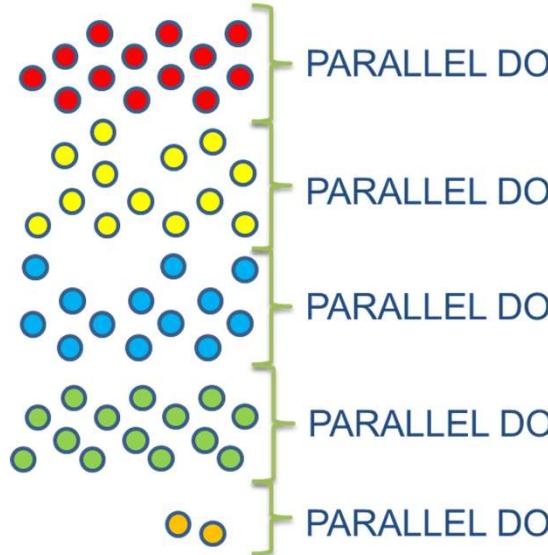
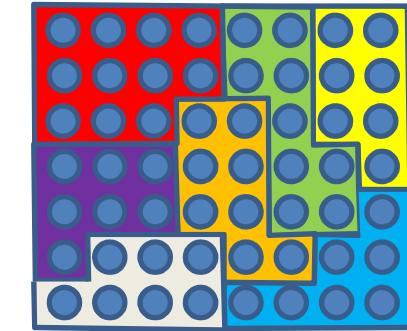
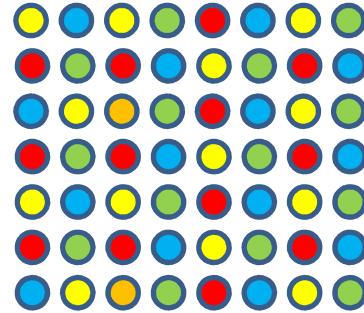
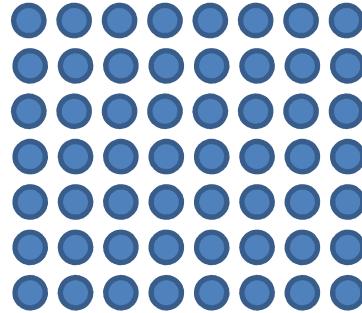
Using parallelization with multidependences



# Multidependences



FEM app: matrix assembly phase  
– Reduction on large matrix with indirection



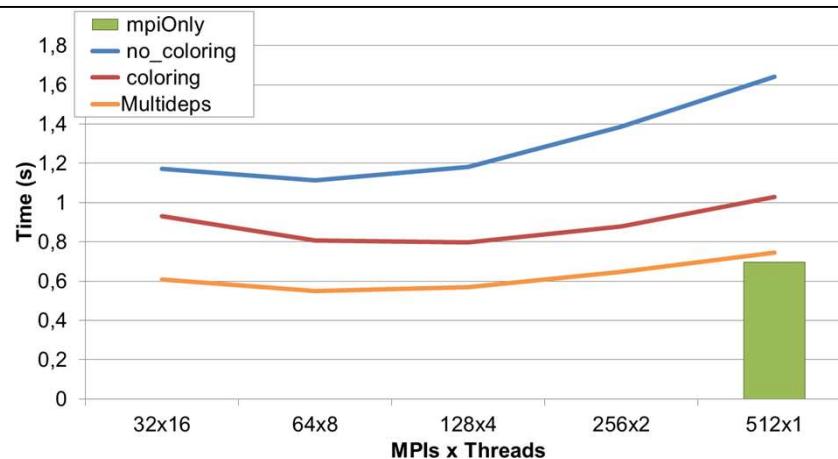
Specify **incompatibilities!!**  
Commutative  
+  
multidependences

# Multidependences

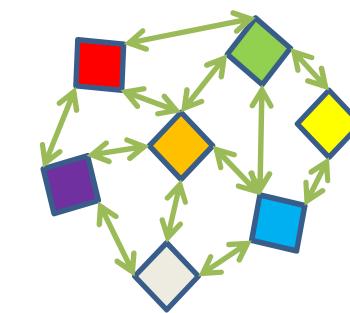
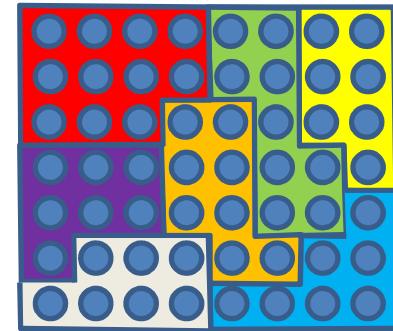


## FEM app: matrix assembly phase

```
NUM_DOMAINS = SIZE(domains)
DO I = 1, NUM_DOMAINS
    NUM_NEIGHBOURS = SIZE(domains(I) % neighbours)
    !$OMP TASK ...
        COMMUTATIVE( [domains(domains(I)%neighbours(J)) , &
                      J = 1, NUM_NEIGHBOURS])
    do kelem = 1, SIZE(domains(I)%elements)
        assembly_element(domains(I)%elements(kelem))
    end do
    !$OMP END TASK
END DO
```



Hybrid execution (MPI + OmpSs) gets  
better performance than pure MPI!!



Specify **incompatibilities!!**  
Commutative

+

multidependences

# Introduction to weak dependences



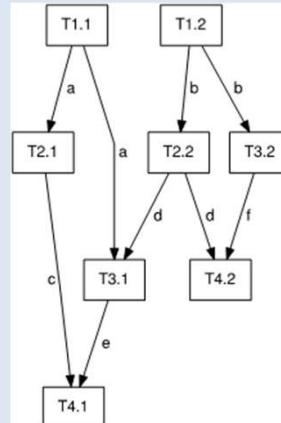
## Single-level tasks

```
a++; b++;
#pragma omp task depend(inout: a) // T1.1
a += ...;
#pragma omp task depend(inout: b) // T1.2
b += ...;
```

```
z = ...;
#pragma omp task depend(in: a) depend(out: c) // T2.1
c = ... + a + ...;
#pragma omp task depend(in: b) depend(out: d) // T2.2
d = ... + b + ...;
```

```
#pragma omp task depend(in:a, d) depend(out:e) // T3.1
e = ... + a + d + ...;
#pragma omp task depend(in:b) depend(out:f) // T3.2
f = ... + b + ...;
```

```
#pragma omp task depend(in: c, e) // T4.1
... = ... + c + e + ...;
#pragma omp task depend(in: d, f) // T4.2
... = ... + d + f + ...;
```



```

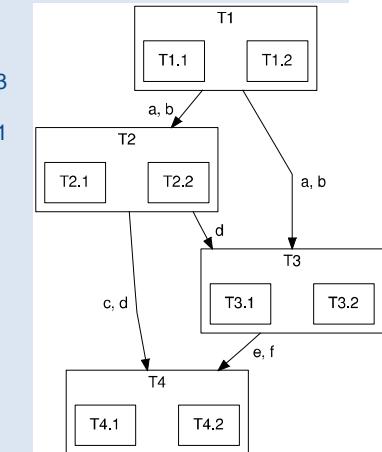
#pragma omp task depend(inout: a, b) // T1
{
    a++;
    b++;
    #pragma omp task depend(inout: a) // T1.1
    a += ...;
    #pragma omp task depend(inout: b) // T1.2
    b += ...;
    #pragma omp taskwait
}

#pragma omp task depend(in:a,b) depend(out:z,c,d) // T2
{
    z = ...;
    #pragma omp task depend(in: a) depend(out: c) // T2.1
    c = ... + a + ...;
    #pragma omp task depend(in: b) depend(out: d) // T2.2
    d = ... + b + ...;
    #pragma omp taskwait
}

#pragma omp task depend(in:a, b, d) depend(out:e, f) // T3
{
    #pragma omp task depend(in:a, d) depend(out:e) // T3.1
    e = ... + a + d + ...;
    #pragma omp task depend(in:b) depend(out:f) // T3.2
    f = ... + b + ...;
    #pragma omp taskwait
}

#pragma omp task depend(in: c, d, e, f) // T4
{
    #pragma omp task depend(in: c, e) // T4.1
    ... = ... + c + e + ...;
    #pragma omp task depend(in: d, f) // T4.2
    ... = ... + d + f + ...;
    #pragma omp taskwait
}
```

## Nested tasks



# Introduction to weak dependences

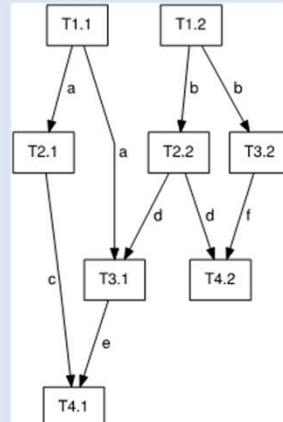


## Single-level tasks

- Fine-grain synchronization
  - » Task T1.2 does not block T2.1
- Less runtime overhead
  - » Less tasks created
  - » No additional taskwaits

```
a++; b++;
#pragma omp task depend(inout: a) // T1.1
a += ...;
#pragma omp task depend(inout: b) // T1.2
b += ...;

z = ...;
#pragma omp task depend(in: a) depend(out: c) // T2.1
c = ... + a + ...;
#pragma omp task depend(in: b) depend(out: d) // T2.2
d = ... + b + ...;
```

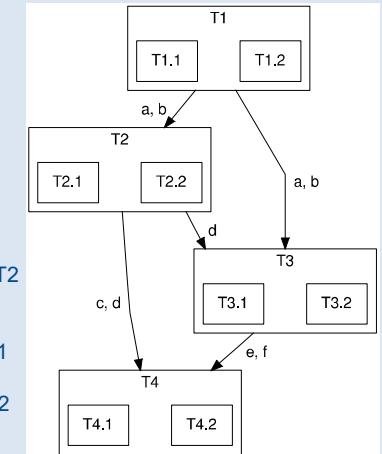


## Nested tasks

- Top-down programming model
- Parallel task creation
  - » Task T2 can be created in parallel with tasks T1.1 and T1.2
- Distant coarse-grain parallelism

```
#pragma omp task depend(inout: a, b) // T1
{
    a++; b++;
    #pragma omp task depend(inout: a) // T1.1
    a += ...;
    #pragma omp task depend(inout: b) // T1.2
    b += ...;
    #pragma omp taskwait
}

#pragma omp task depend(in:a,b) depend(out:z,c,d) // T2
{
    z = ...;
    #pragma omp task depend(in: a) depend(out: c) // T2.1
    c = ... + a + ...;
    #pragma omp task depend(in: b) depend(out: d) // T2.2
    d = ... + b + ...;
    #pragma omp taskwait
}
```



# Weak dependences

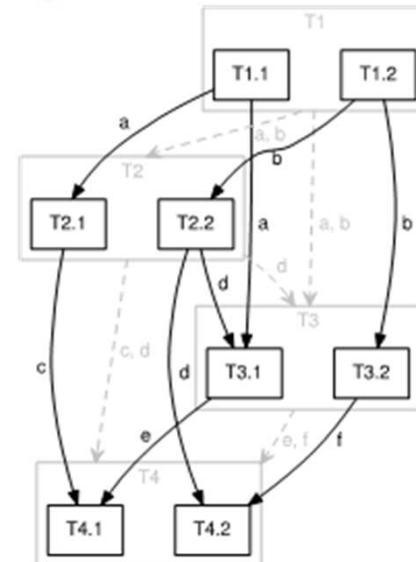


```
#pragma omp task depend(inout: a, b) wait(weak) // T1
{
    a++; b++;
    #pragma omp task depend(inout: a) // T1.1
    a += ...;
    #pragma omp task depend(inout: b) // T1.2
    b += ...;
}
#pragma omp task depend(out: z) depend(weak,in: a, b) depend(weak,out: c, d) wait(weak) // T2
{
    z = ...;
    #pragma omp task depend(in: a) depend(out: c) // T2.1
    c = ... + a + ...;
    #pragma omp task depend(in: b) depend(out: d) // T2.2
    d = ... + b + ...;
}
#pragma omp task depend(weak,in: a, b, d) depend(weak,out: e, f) wait(weak) // T3
{
    #pragma omp task depend(in: a, d) depend(out: e) // T3.1
    e = ... + a + d + ...;
    #pragma omp task depend(in: b) depend(out: f) // T3.2
    f = ... + b + ...;
}
#pragma omp task depend(weak,in: c, d, e, f) wait(weak) // T4
{
    #pragma omp task depend(in: c, e) // T4.1
    ... = ... + c + e + ...;
    #pragma omp task depend(in: d, f) // T4.2
    ... = ... + d + f + ...;
}
```

Supported by Nanos6

Can we get the benefits of single level tasks with nested tasks?

- having dependences crossing to the children dependence domain





[www.bsc.es](http://www.bsc.es)



**Barcelona  
Supercomputing  
Center**

*Centro Nacional de Supercomputación*

# Current trends within OmpSs

OpenMPCon

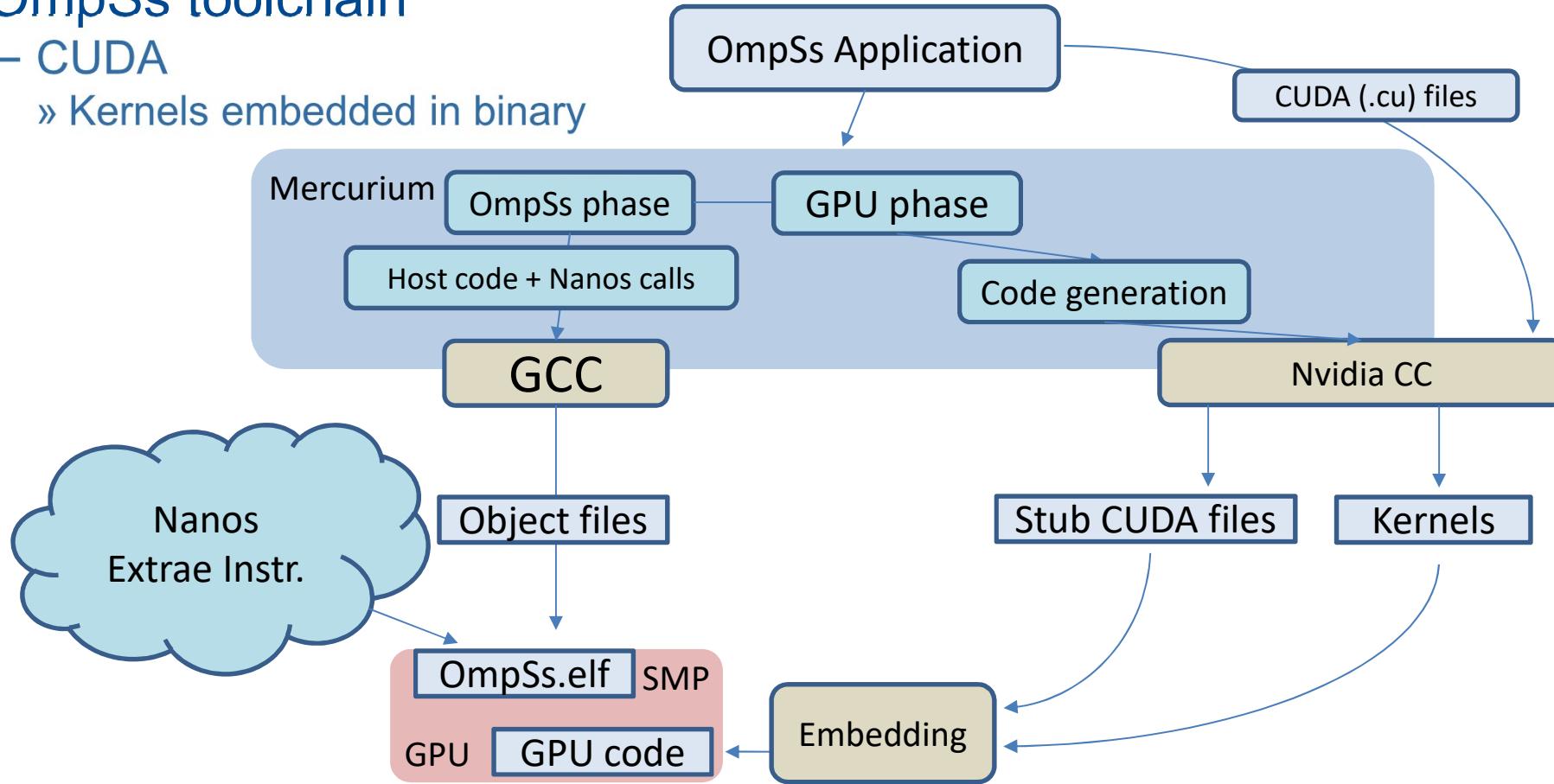
Stony Brook, NY, USA September 18th-19th, 2017

# OmpSs infrastructure

## OmpSs toolchain

### - CUDA

» Kernels embedded in binary

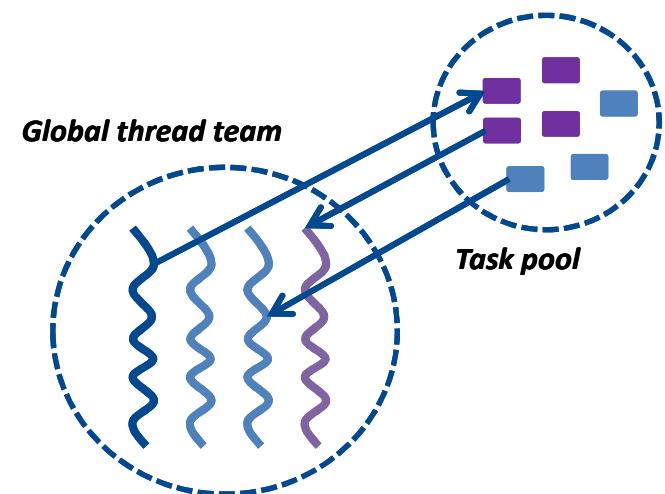


# Execution Model (OmpSs + accelerators)



## Global thread team created on startup

- One worker starts main task (also executes)
- N-1 workers execute tasks
- One representative per device (accelerator)



## All threads get work from a task pool

- Accelerator kernels become tasks
- Tasks are labeled with (at least) one target device
  - » smp, **cuda**, opencl, fpga
- Scheduler decides which task to execute
- Tasks may have several targets (implements)

# Implements & versioning



One single task → two different implementations

- Scheduler decides (at runtime) which version to execute
  - » On resource availability (first thread requesting work)
  - » Smart scheduling (shortest execution time)

```
#pragma omp target device (smp)
#pragma omp task in([N] c) out([N] b)
void scale_task(double *b, double *c, double a, int N);
```

scale.h

```
void scale_task(double *b, double *c, double a, int N){
    for (int j=0; j < N; j++) b[j] = a*c[j];
}
```

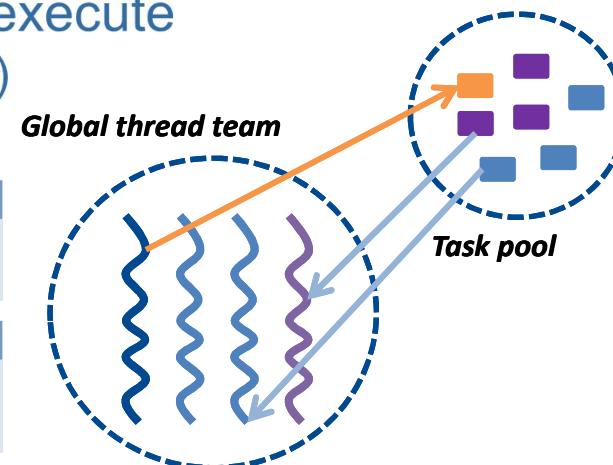
scale.c

```
#pragma omp target device(cuda) implements(scale_task) ndrange(1,N,128)
#pragma omp task in([N] c) out([N] b)
__global__ void scale_task_cu(double *b, double *c, double a, int size);
```

scale.cuh

```
__global__ void scale_task_cu(double *b, double *c, double a, int N) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j < N) b[j] = a * c[j];
}
```

scale.cu



```
#include <scale.h>
#include <scale.cuh>
```

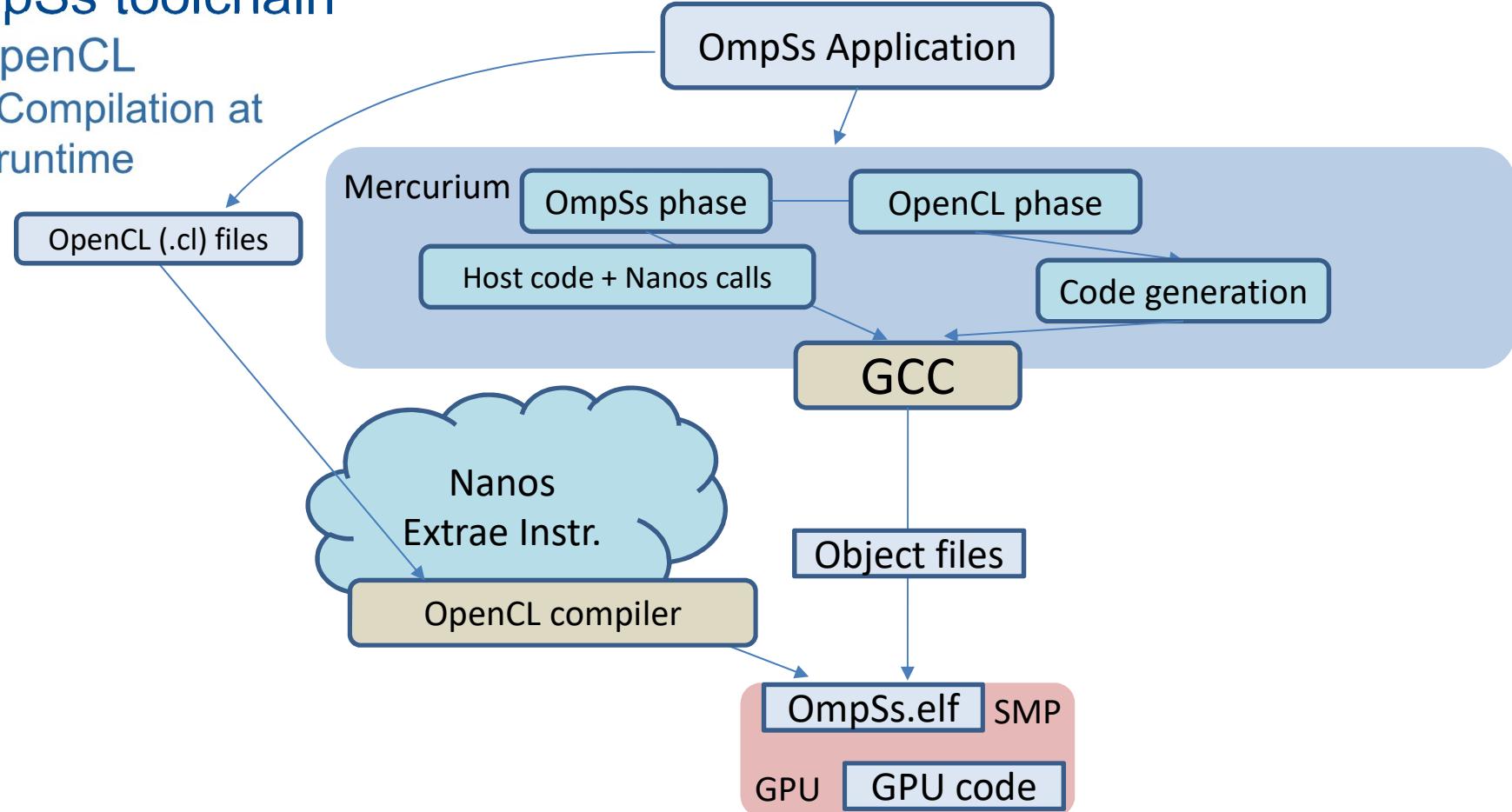
main.c

```
#define SIZE 100
int main (int argc, char *argv[])
{
    . .
    scale_task(B,C,alpha,SIZE);
    . .
}
```

# OmpSs infrastructure

## OmpSs toolchain

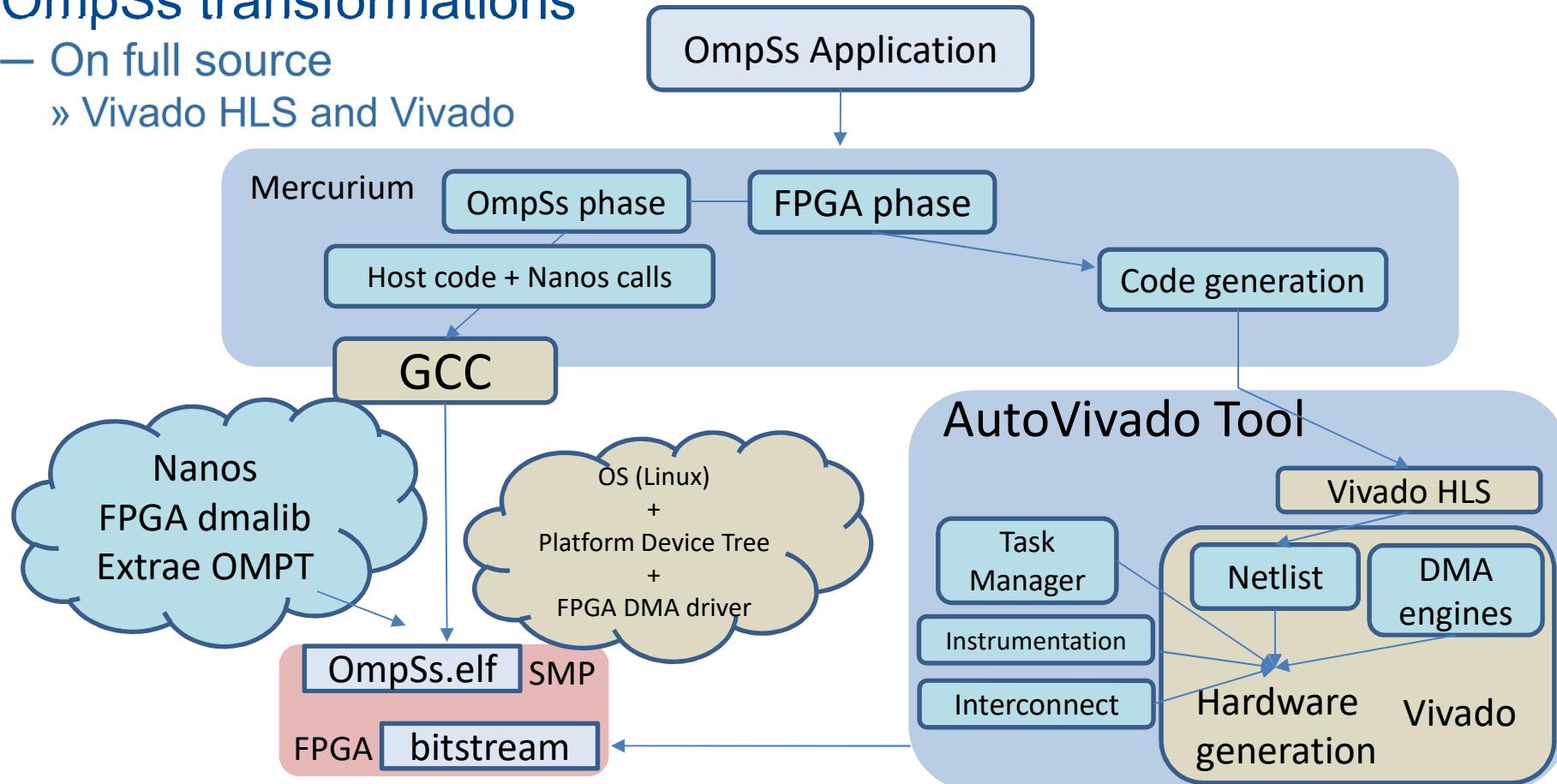
- OpenCL
  - » Compilation at runtime



# OmpSs infrastructure

## OmpSs transformations

- On full source
  - » Vivado HLS and Vivado



# OmpSs autoVivado



## Same kind of transformation for FPGAs

```
#define BS 128

void matrix_multiply(float a[BS][BS], float b[BS][BS],float c[BS][BS])
{
#pragma HLS inline
    int const FACTOR = BS/2;
#pragma HLS array_partition variable=a block factor=FACTOR dim=2
#pragma HLS array_partition variable=b block factor=FACTOR dim=1
    // matrix multiplication of a A*B matrix
    for (int ia = 0; ia < BS; ++ia)
        for (int ib = 0; ib < BS; ++ib) {
#pragma HLS PIPELINE II=1
            float sum = 0;
            for (int id = 0; id < BS; ++id)
                sum += a[ia][id] * b[id][ib];
            c[ia][ib] += sum;
        }
}
```



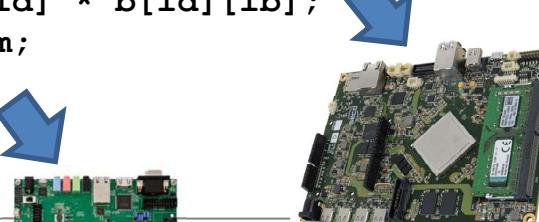
Zynq-7000 Family

2x Cortex-A9 cores + FPGA (32-bit platforms)

SECO AXIOM Board

Zynq U+ XCZU9EG-ES2

4x Cortex-A53 cores + FPGA (64-bit platforms)



Trenz Electronics Zynq U+  
TE0808 XCZU9EG-ES1

Towards  
Discrete FPGAs

# OmpSs autoVivado

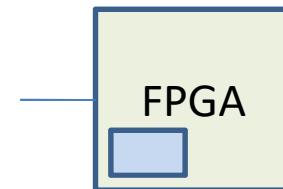
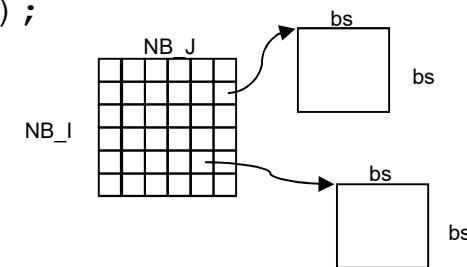


## Mercurium compiler with autoVivado tool

- Triggers the bitstream generation automatically
- Few extensions (onto clause)

```
#pragma omp target device(fpga) copy_deps onto(0,1)  
#pragma omp task in(a,b) inout(c)  
  
void matrix_multiply(float a[BS][BS], float b[BS][BS],  
                     float c[BS][BS]);  
  
...  
  
for (i_b=0; i_b<NB_I; i_b++)  
    for (j_b=0; j_b<NB_J; j_b++)  
        for (k_b=0; k_b<NB_K; k_b++)  
            matrix_multiply(AA[i_b][k_b], BB[k_b][j_b], CC[i_b][j_b]);  
  
...
```

- Compiler also generates a stub function
  - » Data transfers and IP invocation



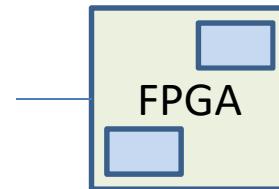
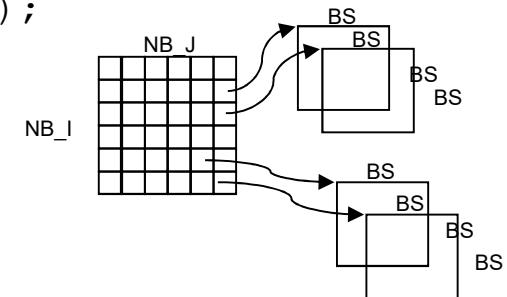
# OmpSs autoVivado



## Mercurium compiler with autoVivado tool

- Triggers the bitstream generation automatically
- Few extensions (onto clause)

```
#pragma omp target device(fpga) copy_deps onto(0,2)  
#pragma omp task in(a,b) inout(c)  
  
void matrix_multiply(float a[BS][BS], float b[BS][BS],  
                     float c[BS][BS]);  
  
...  
  
for (i_b=0; i_b<NB_I; i_b++)  
    for (j_b=0; j_b<NB_J; j_b++)  
        for (k_b=0; k_b<NB_K; k_b++)  
            matrix_multiply(AA[i_b][k_b], BB[k_b][j_b], CC[i_b][j_b]);  
  
...
```



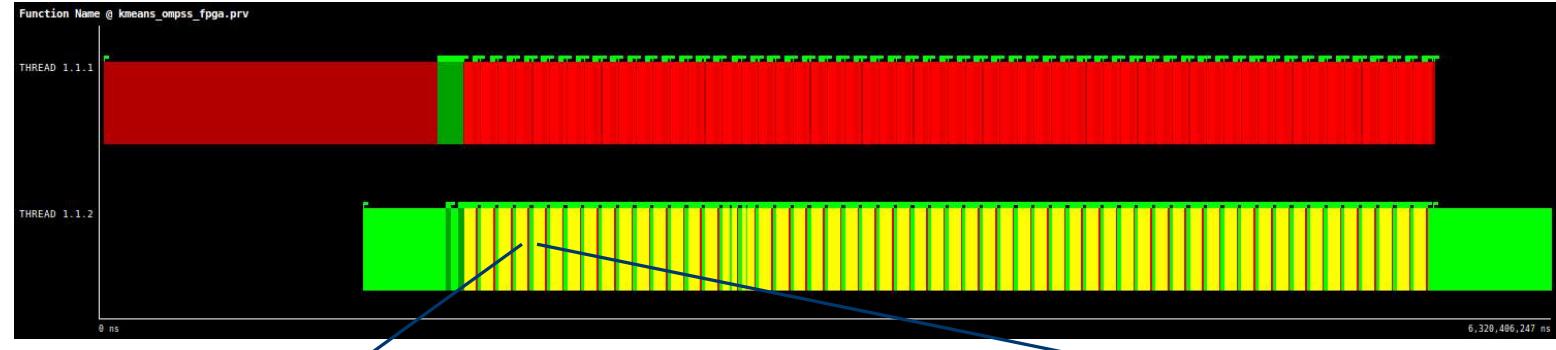
# Tracing facilities on the FPGAs



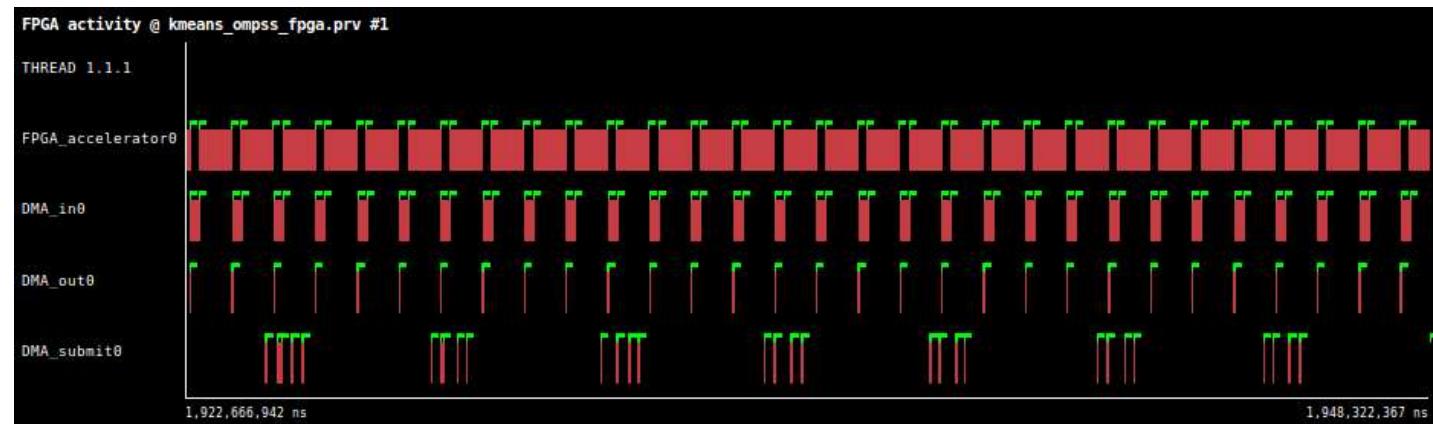
Tracing the internals... of the FPGA!!

Master thread

FPGA  
representative



IP is active  
Input channel  
Output channel  
Submission



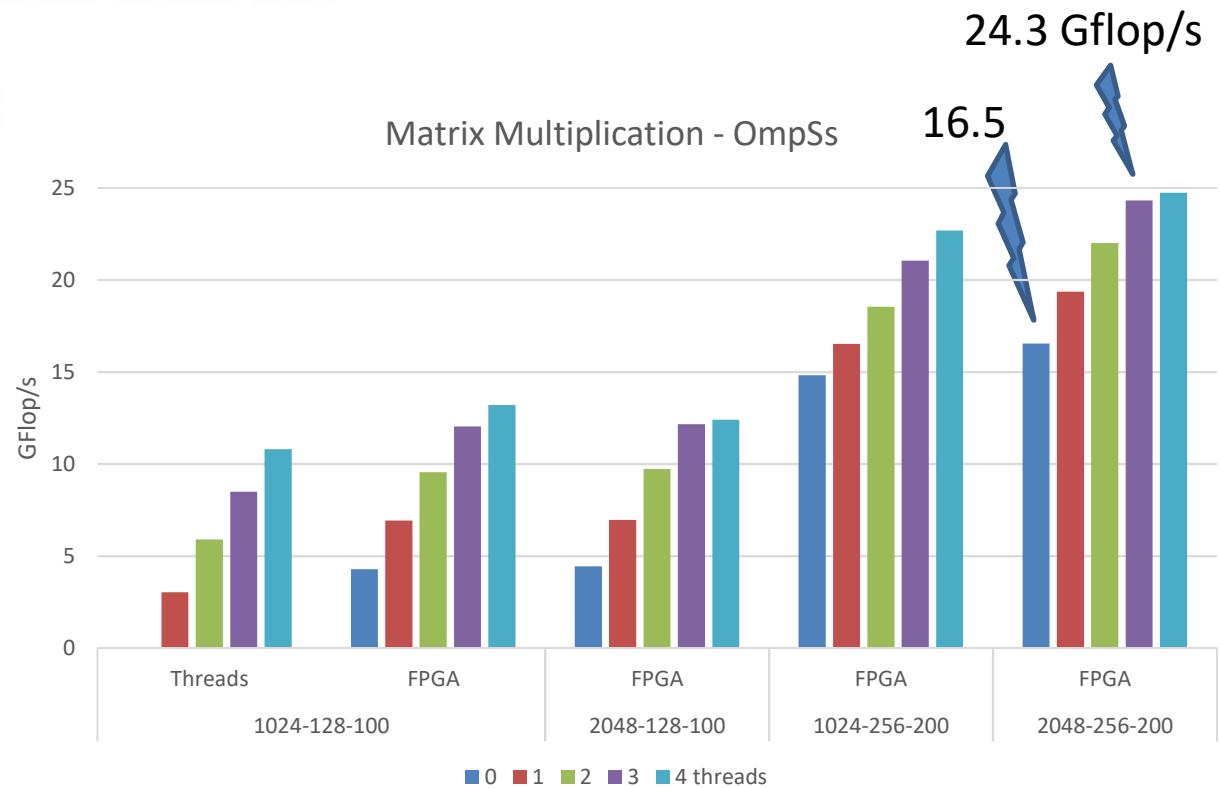
# Results on matrix multiplication



## On Xilinx Zynq Ultrascale+

- BS 128 (100MHz) & 256 (200Mhz)
- With the OmpSs Implements extension

- FPGA @200MHz clearly outperforms the 4 ARM cores
- “Implements” is **greatly** successful
  - » Adds up to 30% performance increase



# Conclusions and future work



We have long experience on improvements to parallel programming

- Enabling productivity and portability
- Supporting heterogeneous/hierarchical architectures
- Asynchrony → global synchronization is not an answer anymore
- Data locality

OmpSs is publicly available at <http://pm.bsc.es>

We will keep working with the OpenMP standard

- Multidependences
- Weak dependences
- Directives on function prototypes
- Unshackable (task-only) threads
- ...

# Thank you!

For further information, please contact

xavier.martorell@bsc.es



**Barcelona  
Supercomputing  
Center**  
*Centro Nacional de Supercomputación*

## ***Intellectual Property Rights Notice***

*The User may only download, make and retain a copy of the materials for his/her use for non-commercial and research purposes.*

*The User may not commercially use the material, unless has been granted prior written consent by the Licensor to do so; and cannot remove, obscure or modify copyright notices, text acknowledging or other means of identification or disclaimers as they appear.*

*For further details, please contact BSC-CNS.*

OpenMPCon

Stony Brook, NY, USA September 18th-19th, 2017