www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introduction to the OmpSs Programming Model

Rosa M. Badia, Xavier Teruel, Xavier Martorell

PACT2018 tutorial
Limassol, Nov 4th, 2018

Agenda

- Introduction to StarSs
  - Motivation and goals
  - OmpSs example: matrix multiplication
    - tasks, task dependences
    - target extension, “implements” approach
    - Leveraging CUDA and OpenCL kernels
  - Experiments
- Basic OmpSs
  - Outlined and inlined directives
  - Task dependences, concurrent, commutative clauses
- Development environment, installation, visualizing timelines…
- Hands-on Lab: OmpSs@heterogeneous Platforms
  - Parallelizing applications on heterogeneous architectures

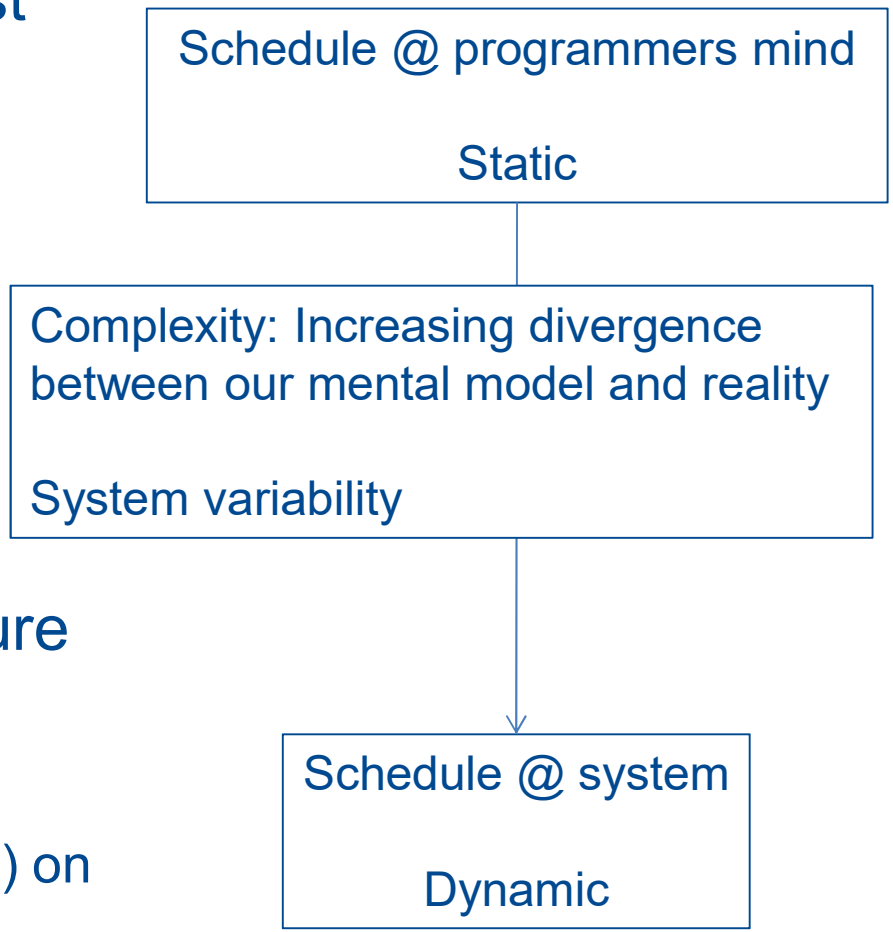
- Contact: pm-tools@bsc.es
- Source code available from http://pm.bsc.es/ompss/

Introduction to StarSs

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

The parallel programming revolution

- Parallel programming in the past
  - Where to place data
  - What to run where
  - How to communicate



- Parallel programming in the future
  - What do I need to compute
  - What data do I need to use
  - Hints (not necessarily very precise) on potential concurrency, locality,…

BSC Vision in the programming revolution

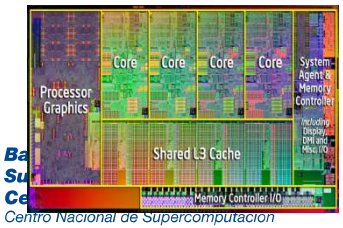
Applications

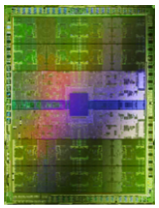
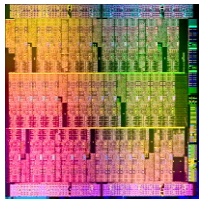
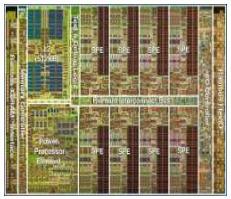
DSL1 DSL2 DSL3

PM: High-level, clean, abstract interface

Power to the runtime

ISA / API



Fast prototyping

Special purpose

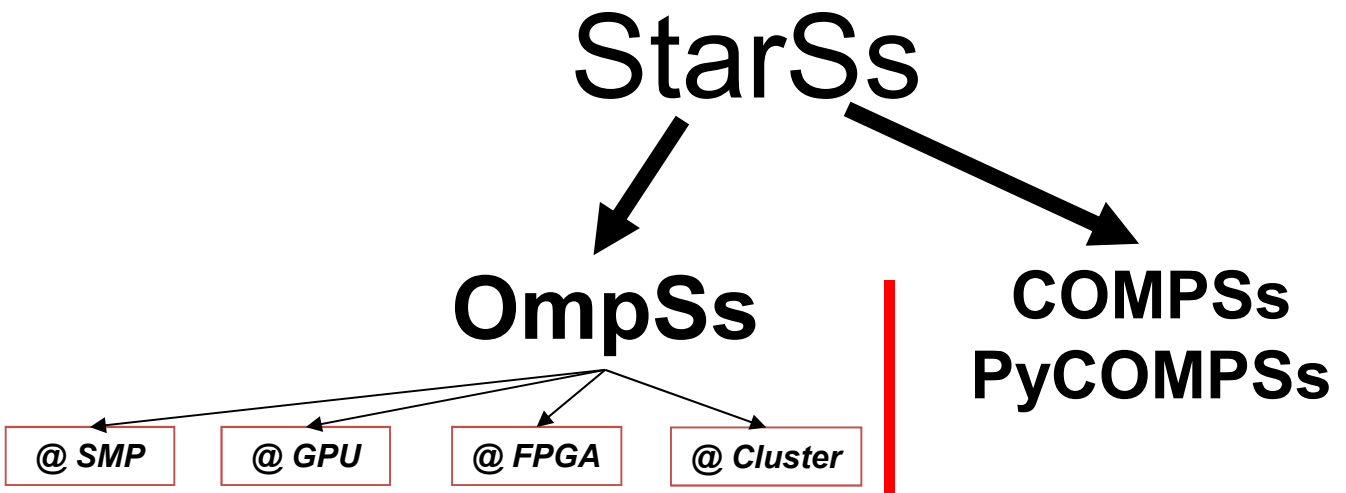
Must be easy to develop/maintain

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

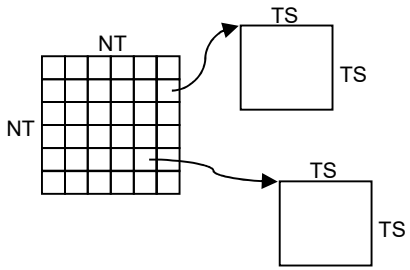
- StarSs main characteristics
  - Dependences: Tasks instantiated but not ready
    - Order IS defined by task creation times
    - Lookahead
      - Avoid stalling the main control flow when a computation depending on previous tasks is reached
      - Possibility to “see” the future searching for further potential concurrency
  - Locality aware
  - Homogenizing heterogeneity

The StarSs "Granularities"



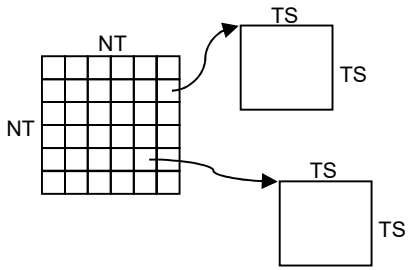
Average task Granularity: 100 microseconds – 10 milliseconds	1second - 1 day
Address space to compute dependences: Memory	Files, Objects (SCM)
Language binding: C, C++, FORTRAN	Java, Python, C/C++

StarSs: a sequential program …



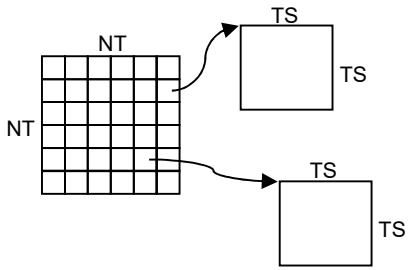

```
void Cholesky( float *A[NT][NT] ) {
int i, j, k;
for (k=0; k<NT; k++) {

   spotrf (A[k*NT+k]) ;
   for (i=k+1; i<NT; i++) {

      strsm (A[k][k], A[k][i]);
   }
   for (i=k+1; i<NT; i++) {
      for (j=k+1; j<i; j++) {

         sgemm( A[k][i], A[k][j], A[j][i]);
      }

      ssyrk (A[k][i], A[i][i]);
   }
}
```

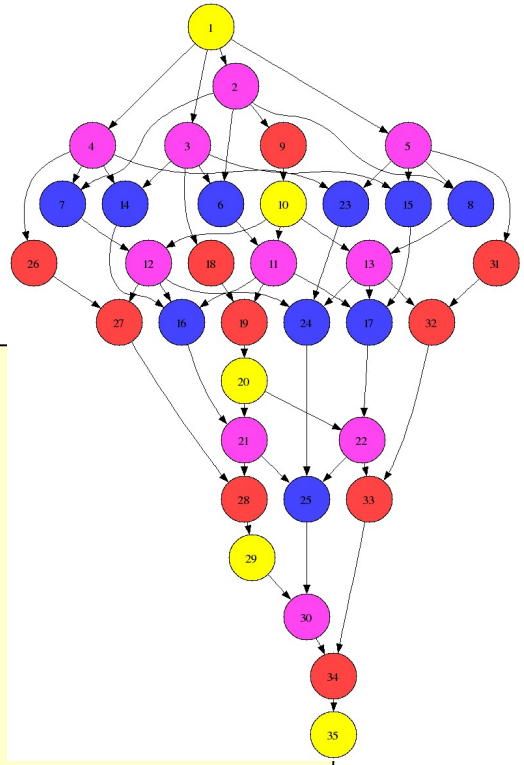
StarSs: … with directionality annotations …




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

StarSs: … that happens to execute in parallel

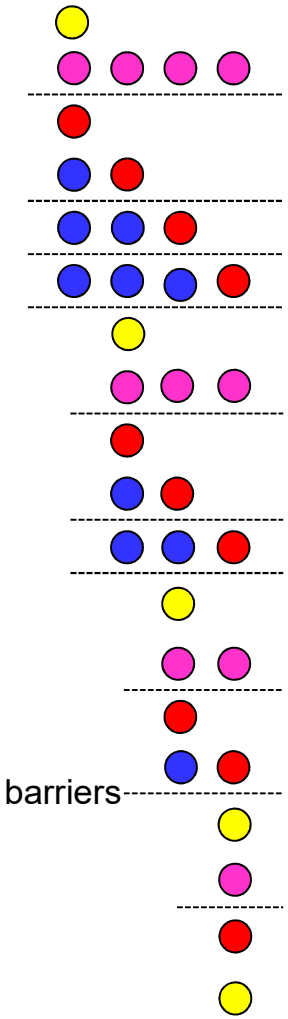




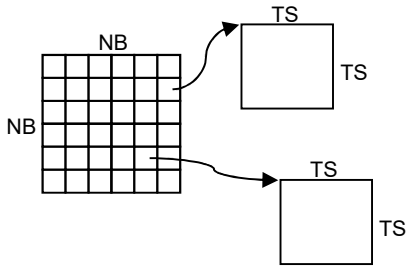

```
void Cholesky( float *A[NT][NT] ) {
int i, j, k;
for (k=0; k<NT; k++) {
   #pragma omp task inout (A[k][k])
   spotrf (A[k][k]) ;
   for (i=k+1; i<NT; i++) {
      #pragma omp task in (A[k][k]) inout (A[k][i])
      strsm (A[k][k], A[k][i]);
   }
   for (i=k+1; i<NT; i++) {
      for (j=k+1; j<i;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

Decouple how we write/think (sequential) from how it is executed

StarSs vs OpenMP 3.0 …




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



OmpSs-like dependences also in OpenMP since release 4.0 !!!

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Related Models and environments

- Programming models
  - OpenMP 4.5
  - Cilk++
  - TBB
  - OpenACC
  - CUDA
  - OpenCL
  - PGAS models
    - UPC
    - X10
    - Chapel
- Runtimes
  - ParalleX, HPX @ LSU
  - Swarm
  - StarPU @ U. Bordeaux
  - PLASMA, … @ UTK
  - Argobots @ ANL
  - Open Community Runtime
  - …

Goals

Heterogeneous architectures have come to stay

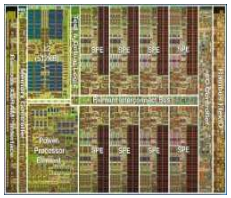
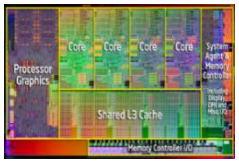
Applications

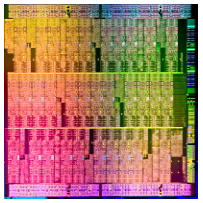
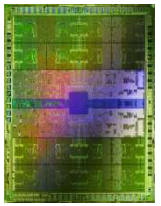
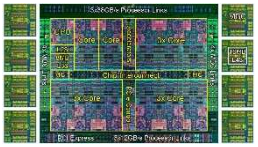
ISA / API

Direct Accesses to Architectural Details

- SIMD
- Local memories / data transfers
- Code execution

OmpSs: can we provide a consistent view, including these heterogeneous architectures?

OmpSs example

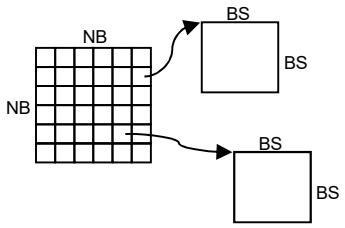
Matrix multiplication

```
#define BS 128

void matrix_multiply(float a[BS][BS], float b[BS][BS],float c[BS][BS])
{
// matrix multiplication of two A, B matrices, to accumulate the result on C
  for (int ia = 0; ia < BS; ++ia)
    for (int ib = 0; ib < BS; ++ib) {
      float sum = 0;
      for (int ic = 0; ic < BS; ++ic)
        sum += a[ia][ic] * b[ic][ib];
      c[ia][ib] += sum;
    }
}
```

Kernel

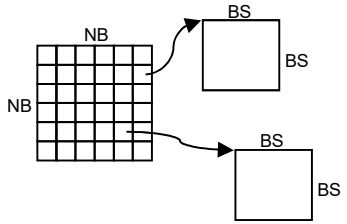
```
...
for (i=0; i<NB; i++)
  for (j=0; j<NB; j++)
    for (k=0; k<NB; k++)
      matrix_multiply(A[i][k], B[k][j], C[i][j]);
...
```

Main program

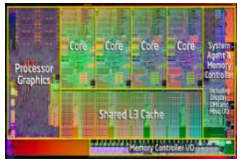


Matrix multiplication

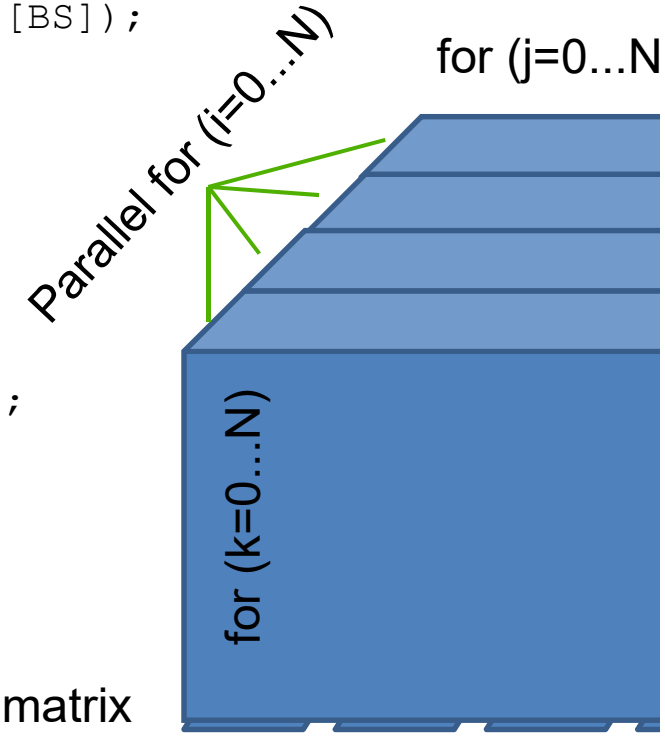
Homogeneous programming




```
void matrix_multiply(float a[BS][BS], float b[BS][BS],
                                      float c[BS][BS]);
…
#pragma omp parallel for private(j,k)
for (i=0; i<NB; i++)
  for (j=0; j<NB; j++)
    for (k=0; k<NB; k++)
      matrix_multiply(A[i][k], B[k][j], C[i][j]);
…
```

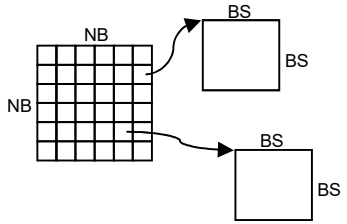


SMP

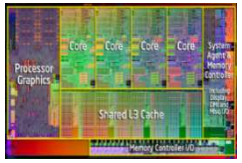


Matrix multiplication

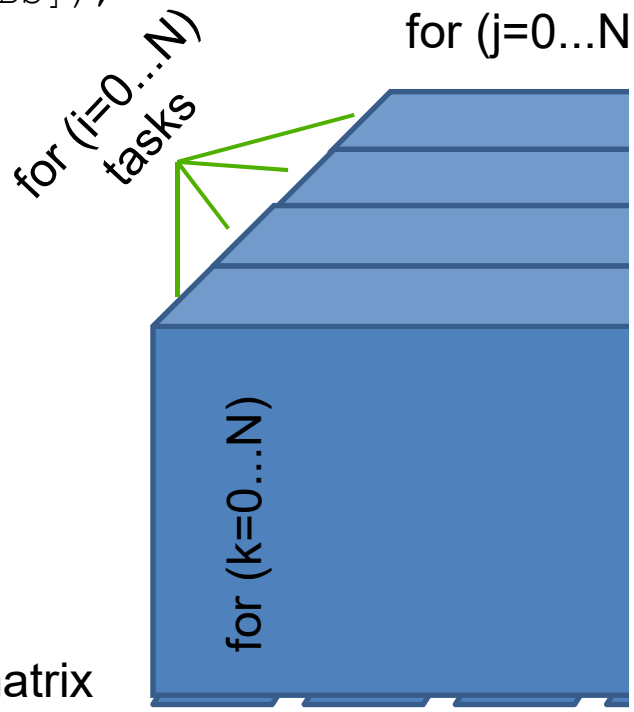
Homogeneous programming




```
void matrix_multiply(float a[BS][BS], float b[BS][BS],
                                      float c[BS][BS]);
…
for (i=0; i<NB; i++)
  #pragma omp task private(j,k)
  for (j=0; j<NB; j++)
    for (k=0; k<NB; k++)
      matrix_multiply(A[i][k], B[k][j], C[i][j]);
…
```



SMP

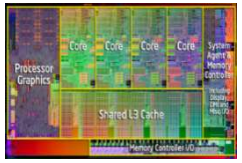


C matrix

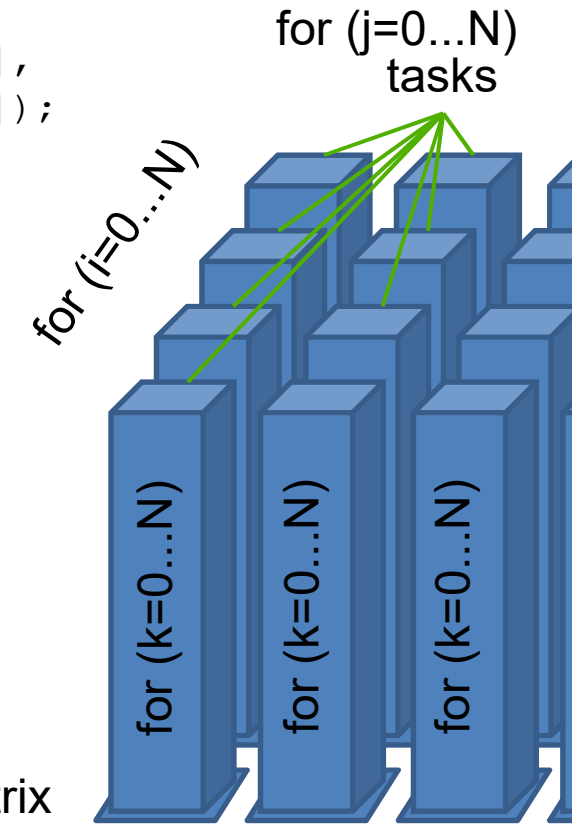
Matrix multiplication

Homogeneous programming

```
void matrix_multiply(float a[BS][BS], float b[BS][BS],
                                      float c[BS][BS]);

…
for (i=0; i<NB; i++)
  for (j=0; j<NB; j++)
    #pragma omp task private(k)
    for (k=0; k<NB; k++)
       matrix_multiply(A[i][k], B[k][j], C[i][j]);
…
```



SMP

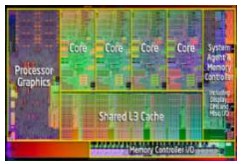


C matrix

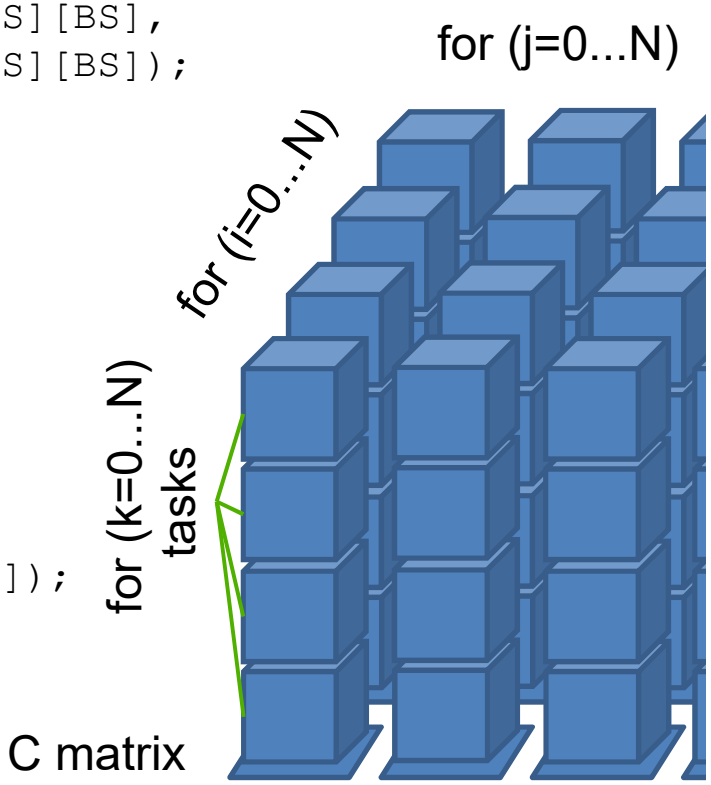
Matrix multiplication

Homogeneous programming

```
void matrix_multiply(float a[BS][BS], float b[BS][BS],
                                      float c[BS][BS]);
…
for (i=0; i<NB; i++)
  for (j=0; j<NB; j++)
    for (k=0; k<NB; k++)
      #pragma omp task \
                  depend(in:A[i][k],B[k][j]) \
                  depend(inout:C[i][j])
      matrix_multiply(A[i][k], B[k][j], C[i][j]);
…
```



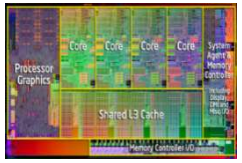
SMP



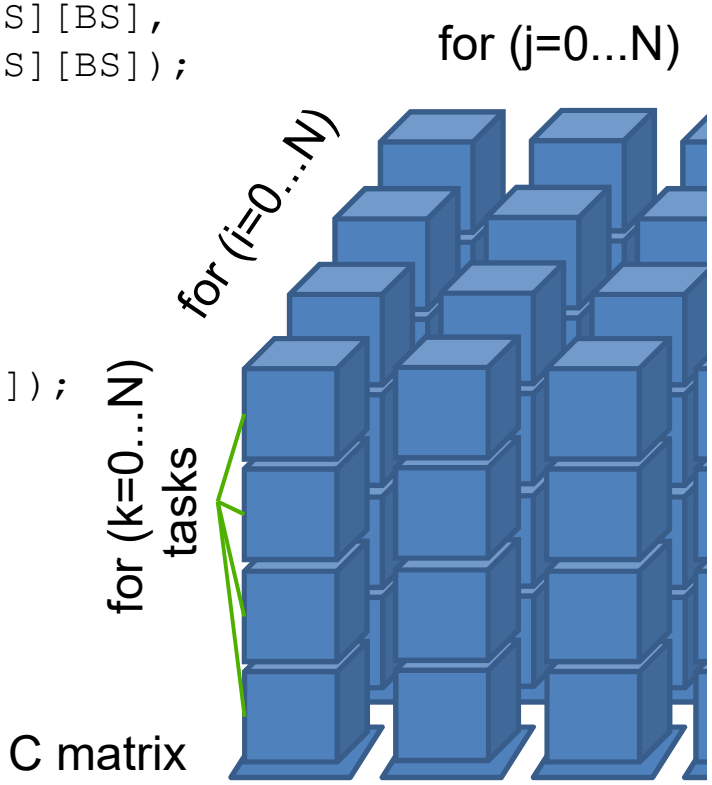
Matrix multiplication

Homogeneous programming

```
#pragma omp task depend(in:a,b) depend (inout:c)
void matrix_multiply(float a[BS][BS], float b[BS][BS],
                                      float c[BS][BS]);
…
for (i=0; i<NB; i++)
  for (j=0; j<NB; j++)
    for (k=0; k<NB; k++)
      matrix_multiply(A[i][k], B[k][j], C[i][j]);
…
```



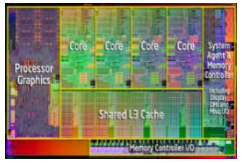
SMP



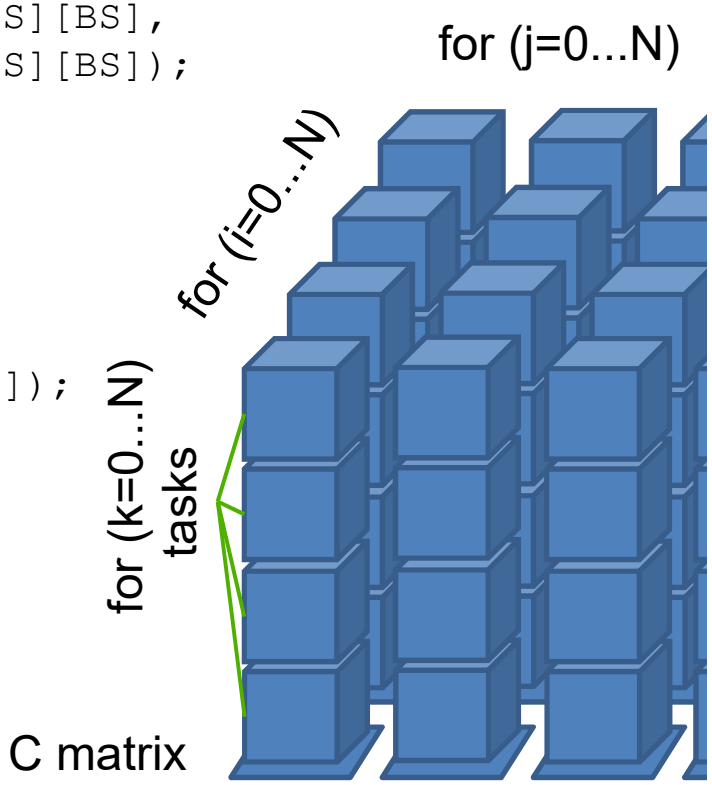
Heterogeneous programming

Homogenizing the support

```
#pragma omp target device(smp) copy_deps
#pragma omp task depend(in:a,b) depend (inout:c)
void matrix_multiply(float a[BS][BS], float b[BS][BS],
                                      float c[BS][BS]);

…
for (i=0; i<NB; i++)
  for (j=0; j<NB; j++)
    for (k=0; k<NB; k++)
      matrix_multiply(A[i][k], B[k][j], C[i][j]);
…
```



SMP



It is all about interfaces…

int ol_main_par (**int low_i, int_high_i**, float *A, float *B, float *C)

– With private j, k

Slide 16

int ol_main_task (**int i**, float * A, float * B, float * C)

– With private j, k

Slide 17

int ol_main_task (**int i, int j**, float * A, float * B, float * C)

– With private k

Slide 18

int ol_main_task (**int i, int j, int k**, float * A, float * B, float * C,
dependence_descriptor_t * deps)

– No private variables

Slides 19-20-21

It is all about interfaces...

In previous slide, variables move from private to function args

- As programmers, we do not want to keep doing these transformations on our applications!!

The compiler is the proper place to do these transformations

- Algorithm has been used in OpenMP over the past 20 years
- There is no magic with it

Avoid having to take into account these details from high level programming

- Let's remind that **in heterogeneous systems...**
  - » the call to these interfaces is done on the host code
  - » and the target is the accelerator code

Changes are so much ERROR prone!!!

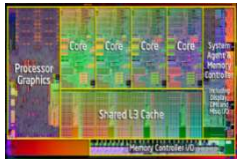
Heterogeneous programming

The “implements” approach

- Kernel provided in CUDA
- Kernel compiled “offline”
- Data transfers automatically generated by OmpSs

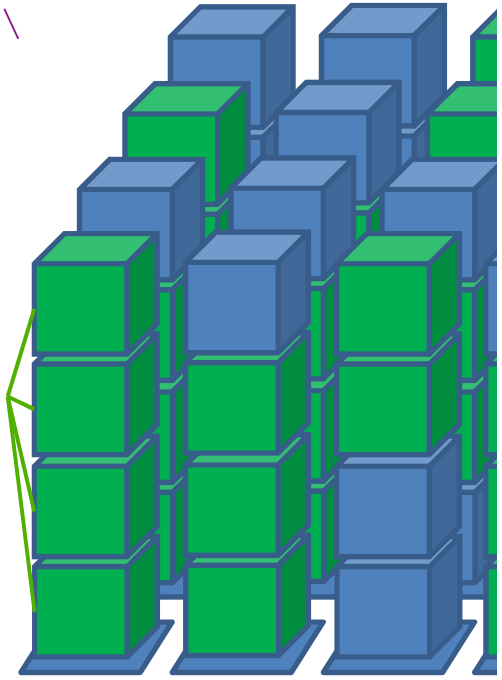
Single point of change

```
#pragma omp target device(cuda) ndrange(2,NB,NB,16,16) \
                 implements(matrix_multiply)
#pragma omp task depend(in:a,b) depend (inout:c)
__global__ void matrix_multiply_cuda(float a[BS][BS],
                                     float b[BS][BS],
                                     float c[BS][BS]);
```



SMP

GPGPU



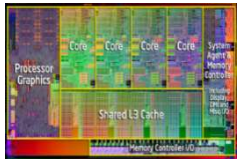
Heterogeneous programming

The “implements” approach

- Kernel provided in OpenCL
- Kernel compiled “online”
- Data transfers automatically generated by OmpSs

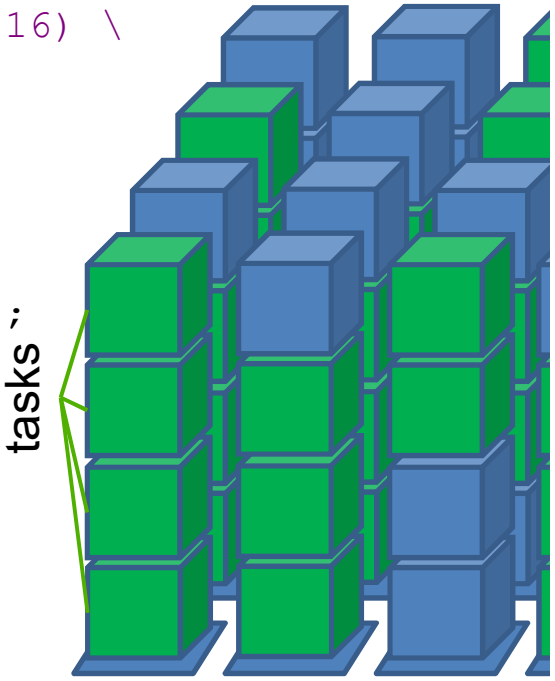
Single point of change

```
#pragma omp target device(opencl) ndrange(2,NB,NB,16,16) \
                  implements(matrix_multiply)
#pragma omp task depend(in:a,b) depend (inout:c)
__kernel void matrix_multiply_opencl(float a[BS][BS],
                                     float b[BS][BS],
                                     float c[BS][BS]);
```



SMP

GPGPU



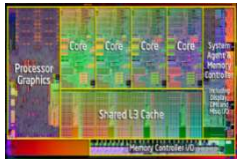
Heterogeneous programming

The “implements” approach

- Native C/C++ kernel!!! => Vivado HLS
- Kernel compiled “offline”... takes some time
- Data transfers automatically generated by OmpSs

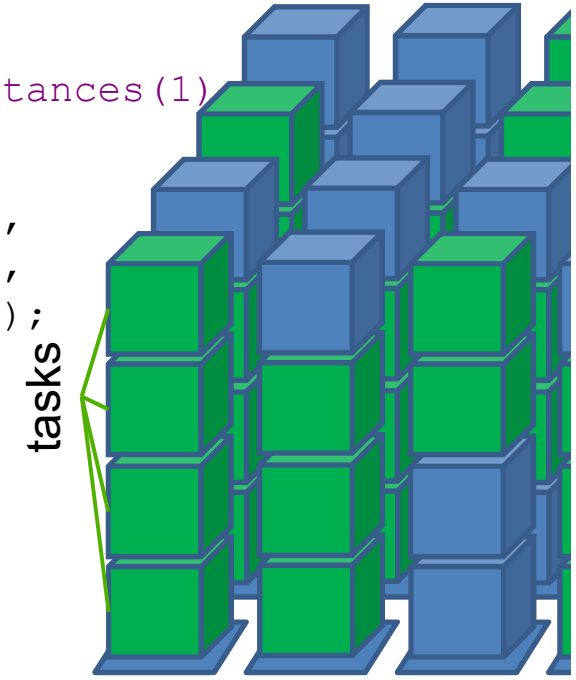
Single point of change

```
#pragma omp target device(fpga) \
                 implements(matrix_multiply) //num_instances(1)
#pragma omp task in(a,b) inout(c)
         void   matrix_multiply_fpga(float a[BS][BS],
                                     float b[BS][BS],
                                     float c[BS][BS]);
```



SMP

FPGA

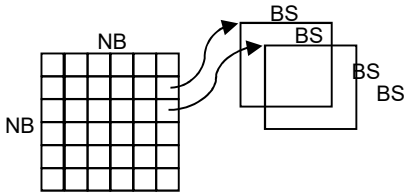


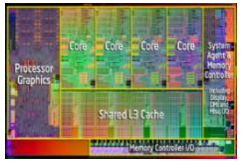
Heterogeneous programming

The “implements” approach

- Native C/C++ kernel!!! => Vivado HLS
- Kernel compiled “offline”... takes some time
- Data transfers automatically generated by OmpSs

```
#pragma omp target device(fpga) \
                 implements(matrix_multiply) //num_instances(2)
#pragma omp task in(a,b) inout(c)
         void   matrix_multiply_fpga(float a[BS][BS],
                                     float b[BS][BS],
                                     float c[BS][BS]);
```

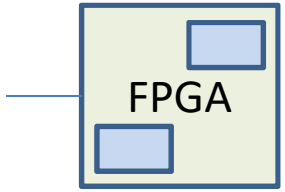




SMP

FPGA

```
# config file
ID   NUM  FUNCTION
0    2        matmul
```

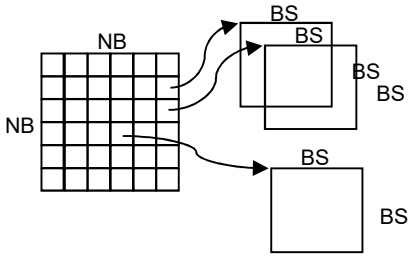


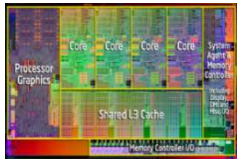
Heterogeneous programming

The “implements” approach

- Native C/C++ kernel!!! => Vivado HLS
- Kernel compiled “offline”... takes some time
- Data transfers automatically generated by OmpSs

```
#pragma omp target device(fpga) \
                 implements(matrix_multiply) //num_instances(3)
#pragma omp task in(a,b) inout(c)
         void   matrix_multiply_fpga(float a[BS][BS],
                                     float b[BS][BS],
                                     float c[BS][BS]);
```

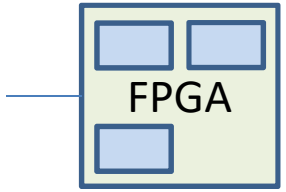




SMP

FPGA

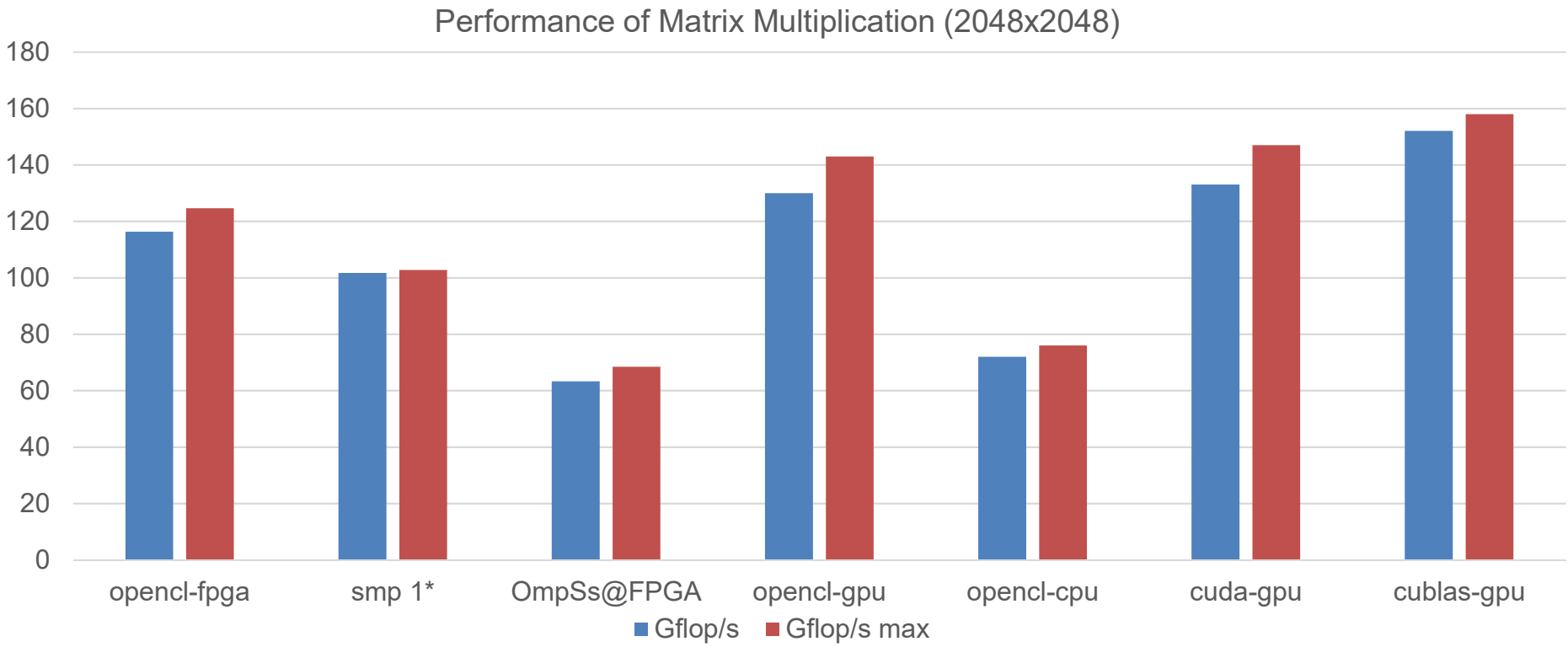
```
# config file
ID   NUM  FUNCTION
0    3    matmul
```



Experiments

Matrix multiplication on Grendel

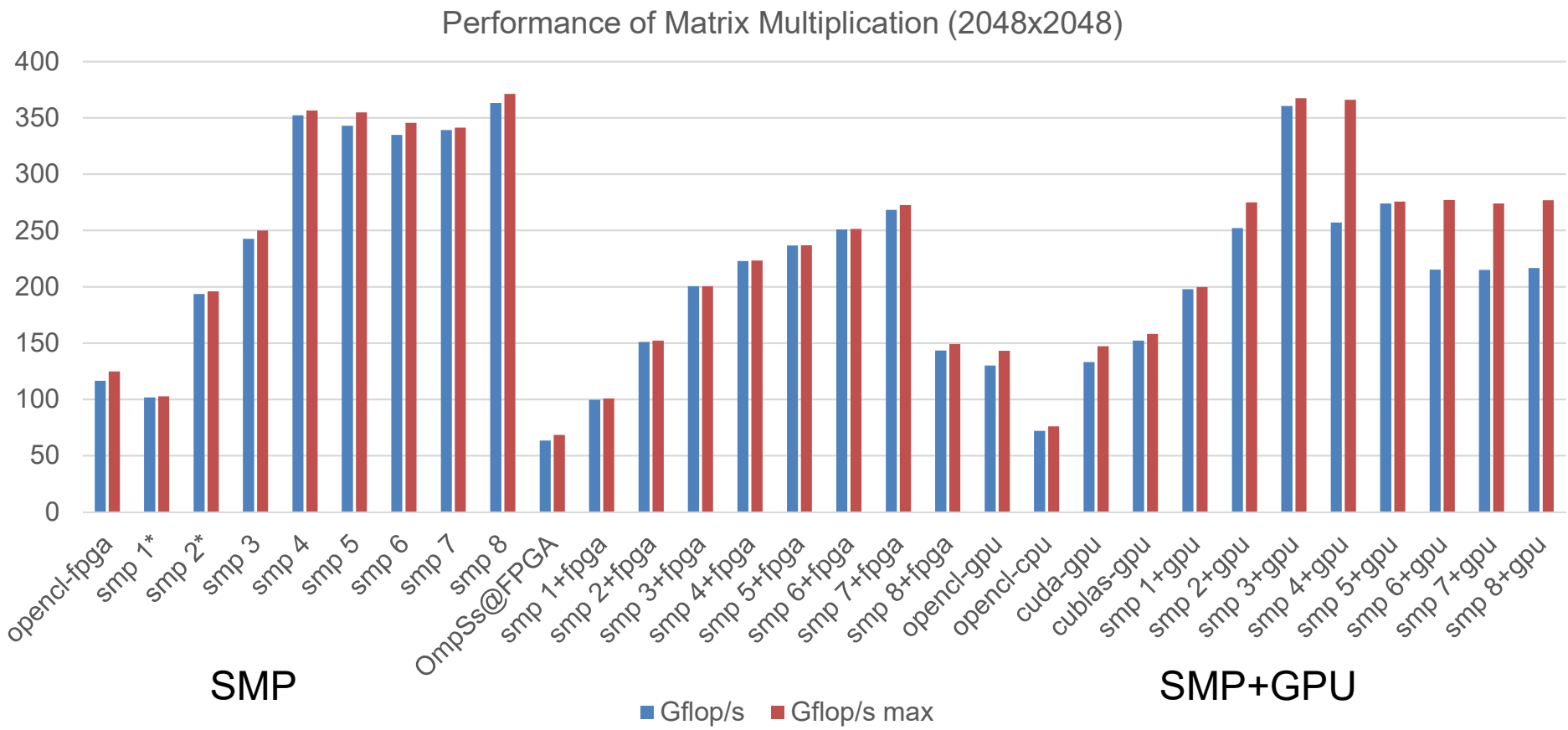
- OmpSs@FPGA reduced performance compared to OpenCL
- CUBLAS, CUDA, OpenCL, OpenCL-FPGA



Experiments

Matrix multiplication on Grendel

— + including MKL on Intel CPUs (☹ not so good for the FPGA...)

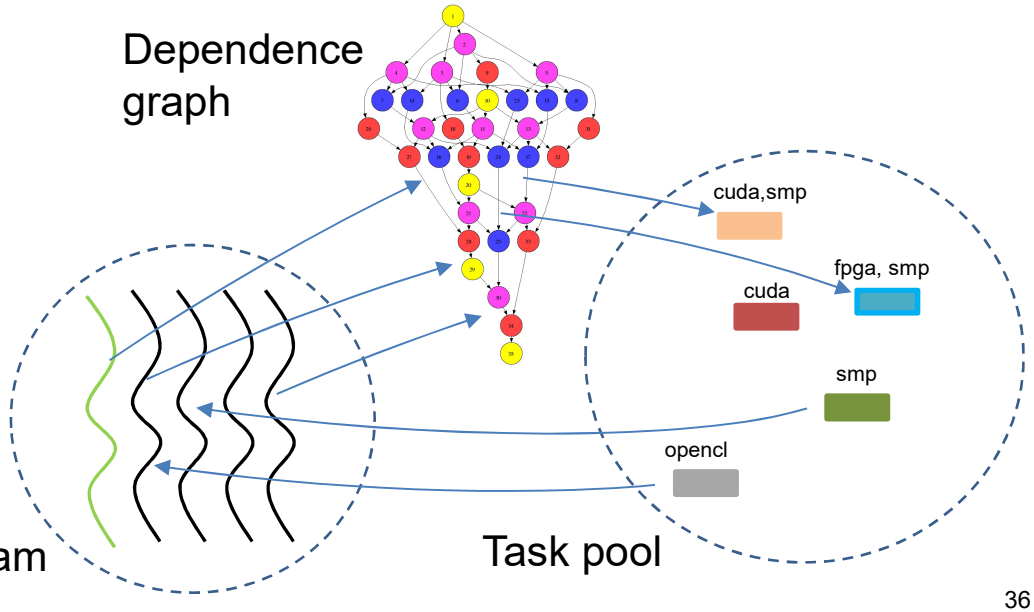


Basic OmpSs

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

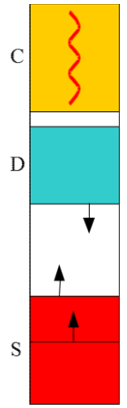
Execution Model

- Thread-pool model
  - OpenMP parallel “ignored”
- All threads created on startup
  - One of them starts executing main (on SMP device)
  - P-1 workers execute SMP tasks
    - OMP_NUM_THREADS
    - --smp-workers
  - One representative (OpenCL/CUDA/FPGA) per device/accelerator
    - Experimenting with a single representative for N devices
    - NX_GPUS
- All get work from a task pool
  - And can generate new work
  - Work is labeled with possible “targets”
  - Single or multiple work-queues



Memory Model

- From the point of view of the programmer a single naming space exists
  - The standard sequential/shared memory address space



- From the point of view of the runtime and target platform, different possible scenarios
  - Pure SMP:
    - Single address space
  - Distributed/heterogeneous (cluster, gpus, ...):
    - Multiple address spaces exist
      - Versions of same data may exist in multiple of these address spaces
    - Data consistency ensured by the implementation

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.

OmpSs main initial contribution: Dependences

```
#pragma omp task in(lvalue_expr-list) \
                 out(lvalue_expr-list) \
                 inout(lvalue_expr-list)
```

- Specify data that has to be available before activating task
- Specify data this task produces that might activate other tasks
- Contiguous / array regions
- You do NOT specify dependences but information for the runtime to compute them
- Contributed to OpenMP 4.0

- Relaxations:
  - No need to specify for ALL data touched
  - Order in inout chains
    - Programmer responsible of possible races

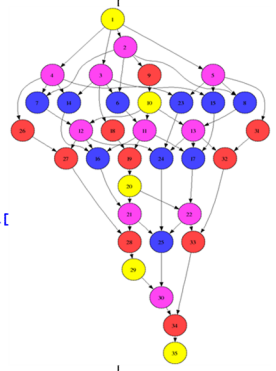
```
#pragma omp concurrent (…)
#pragma omp critical
```

```
#pragma omp commutative (…)
```

Inlined and outlined tasks

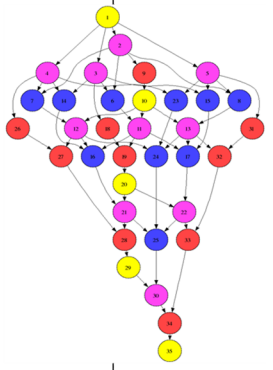
- Directives inlined
  - Pragma applies to immediately following statement
  - The compiler outlines the statement (as in OpenMP)

```
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]),\
                           in([TS][TS]A[k][j])  inout ([TS][TS]A[
          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);
      }
   }
}
```



- Directives outlined
  - Attached to function declaration
    - All function invocations become a task
    - The programmer gives a name, this enables to later provide several implementations

```
#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);
      }
   }
}
```



Outlined directives

Outlined tasks instantiation

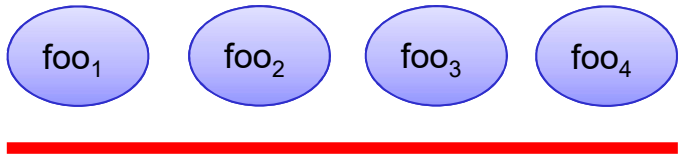
- Task pragma attached to function definition
  - All function invocations become a task
  - The programmer gives a name, this enables later to provide several implementations

```
#pragma  omp   task
void   foo (int Y[size], int size) {
   for (int j=0; j<size; j++) Y[j]= j;
}

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

int main()
{
   int X[100];

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

Invocation instantiates task

Outlined tasks data scoping

- The semantic is capture value at task instantiation time (function invocation)
  - For scalars, the value is captured. Equivalent to firstprivate
  - For pointers, the value of the pointer is captured
  - For arrays, the address is captured

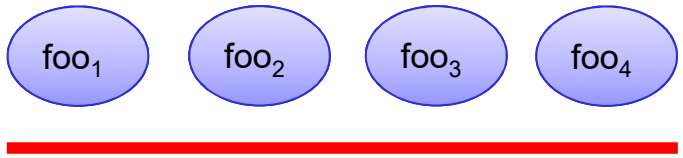



```
#pragma  omp   task
void   foo (int Y[size], int size, int a)
{
   for (int j=0; j<size; j++) Y[j]= a++;
}
```

```
int Y[6]={1,2,3,4,5,6};

int main()
{
int v=0;
int X[4]={5,6,7,8};

   int *Z;
   Z=(int *)malloc(4*sizeof(int));

   foo (X, 4, v) ;
   foo (Y, 2, v+2);
   foo (&Y[3], 3, v++);
   foo (Z,4,v);
   #pragma omp taskwait

// value of v here ?
   // value of X[1] here ?
   // value of Y[4] here ?
   // value of Y[3] here ?
   // value of Z[1] here ?
}
```

Outlined tasks data scoping

- The semantic is capture value at task instantiation time (function invocation)
  - For scalars, the value is captured. Equivalent to firstprivate
  - For pointers, the value of the pointer is captured
  - For arrays, the address is captured




```
#pragma  omp   task
void   foo (int Y[size], int size, int a)
{
   for (int j=0; j<size; j++) Y[j]= a++;
}
```

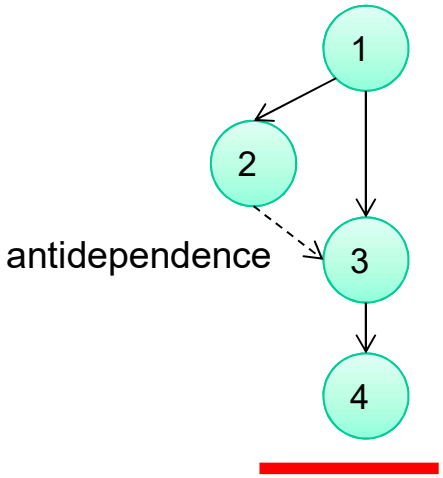
```
int Y[6]={1,2,3,4,5,6};

int main()
{
int v=0;
int X[4]={5,6,7,8};

   int *Z;
   Z=(int *)malloc(4*sizeof(int));

   foo (X, 4, v) ;
   foo (Y, 2, v+2);
   foo (&Y[3], 3, v++);
   foo (Z,4,v);
   #pragma omp taskwait

// value of v here is 1
   // value of X[1] here is 1
   // value of Y[4] here is 1
   // value of Y[3] here is 0
   // value of Z[1] here is 2
}
```

Defining dependences for outlined tasks

- Clauses that express data direction:
  - Input, output, inout
  - The argument is an lvalue expression based on data visible at the point of declaration (global variables and arguments)
  - The object pointed by the lvalue expression will be used to compute dependences.

```
#pragma omp task out(*px)
void set (int *px, int v) {*px = v;}

#pragma omp task inout(*px)
void incr (int *px) {(*px)++;}

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

```
set(&x,5);       //1
do_print(x);    //2
incr(&x);        //3
do_print(x);    //4
#pragma omp taskwait
```

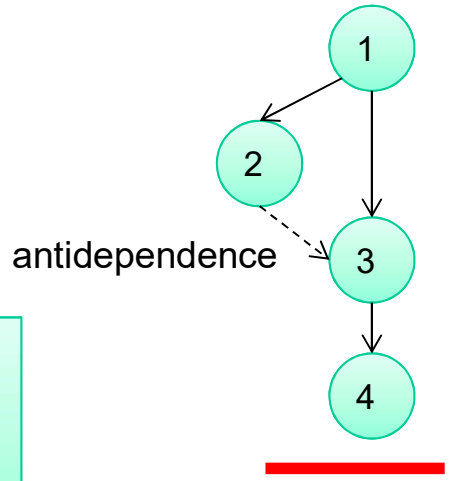


Defining dependences for outlined tasks

- Clauses that express data direction:
  - Input, output, inout
  - The argument is an lvalue expression based on data visible at the point of declaration (global variables and arguments)
  - The object pointed by the lvalue expression will be used to compute dependences.

```
#pragma omp task out(*px)
void set (int *px, int v) {*px = v;}

#pragma omp task in(x)
int incr (int x) {int y; y= x+1; return(y);}

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

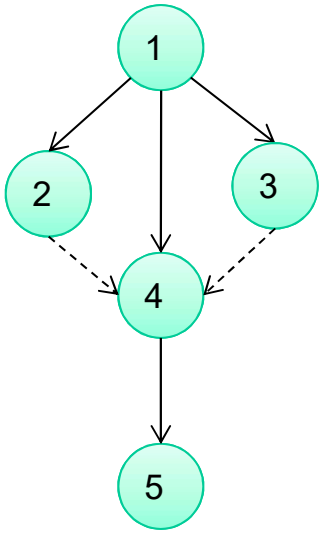
```
set(&x,5);        //1
do_print(x);     //2
x = incr(x);          //3
do_print(x);     //4
#pragma omp taskwait
```



Mixing inlined and outlined tasks

non-taskified: executed sequentially

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

int main()
{
int x;

   x=3;

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

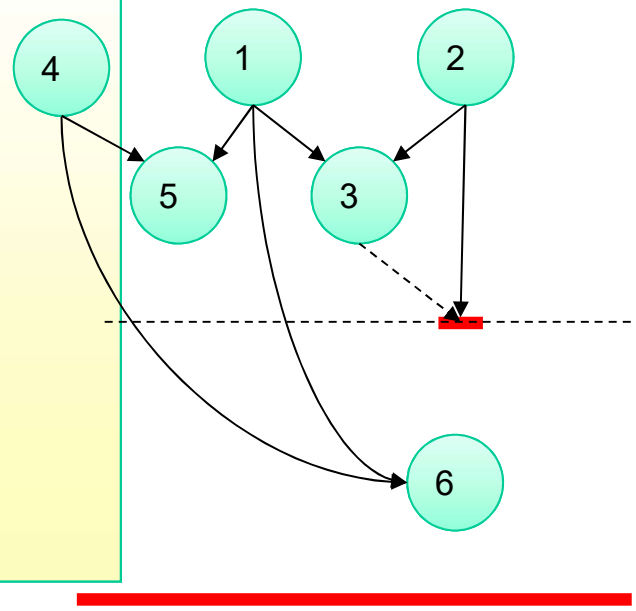


Partial control flow synchronization

#pragma taskwait on (lvalue_expr-list)

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

```
#pragma omp task in([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,C,J); //6

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



Array sections

- Indicating as in/out/inout subregions of a larger structure:

  in (A[i])
  → the input argument is element *i* of A

- Indicating an array section:

  in ([BS]A)
  → the input argument is a block of size BS from address A

  in (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)

  in (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]
  → the upper bound can be omitted if size is known to the compiler (default is N-1, being N the size)

Array sections

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

=

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

=

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

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

```
int a[N];
#pragma omp task in(a[2:3])
//elements 2 and 3 of the array used to compute dependences
```

=

```
int a[N];
#pragma omp task in(a[2;2])
//elements 2 and 3 of the array used to compute dependences
```

Array sections

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

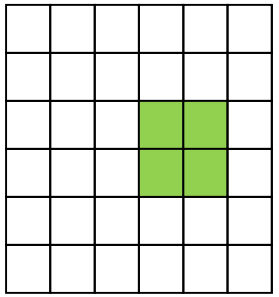
=

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

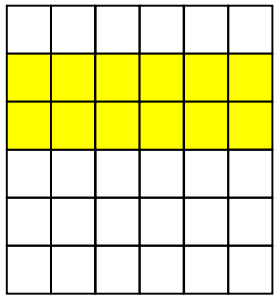
```
int a[N][M];
#pragma omp task in(a[2:3][3:4])
// 2 x 2 subblock of a at a[2][3]
```

=

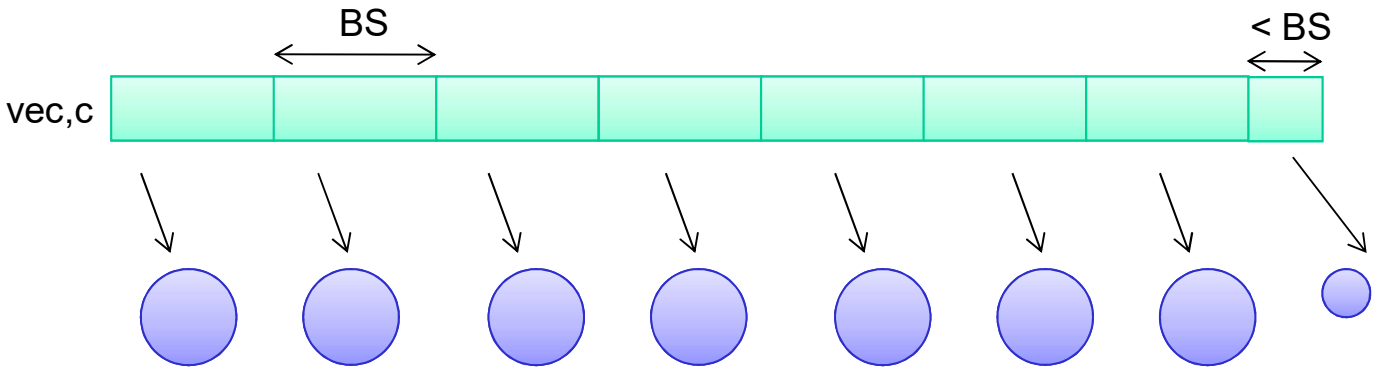
```
int a[N][M];
#pragma omp task in(a[2;2][3;2])
// 2 x 2 subblock of a at a[2][3]
```



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



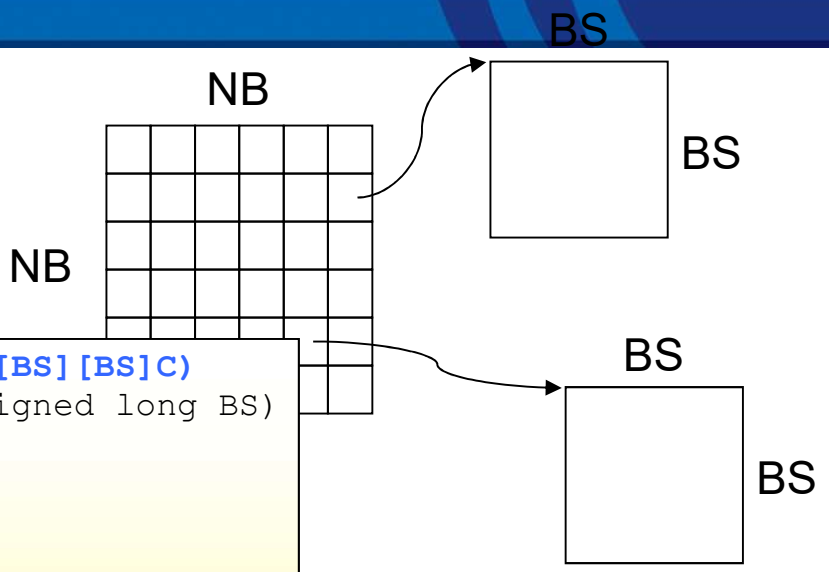
Array sections in outlined tasks




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

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, &c[j]);
   }
}
```

dynamic size of argument

Array sections in outlined tasks

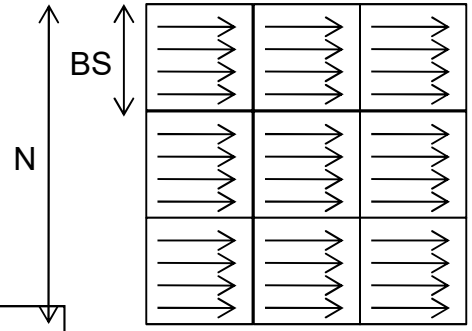



```
#pragma omp task in([BS][BS]A, [BS][BS]B) inout([BS][BS]C)
void matmul(double *A, double *B, double *C, unsigned long BS)
{
  int i, j, k;

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

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

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

Array sections in outlined tasks

N = total size of matrix
BS = block size
NB = number of blocks

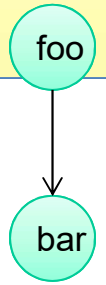



```
#pragma omp task in([BS][BS]A, [BS][BS]B) inout([BS][BS]C)
void matmul(double  *A, double *B, double *C, unsigned long BS)
{
  int i, j, k;

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

```
void compute(unsigned long BS, unsigned long N, double *A, double *B, double *C)
{
  unsigned i, j, k;

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

Specification of incomplete data-directionality

```
#pragma omp task out (*sentinel)
void foo ( .... , int *sentinel){
   ...
}

#pragma omp task in (*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

Specification of incomplete data-directionality

- Directionality not required for all arguments
- May even be used with variables not accessed in that way or even used
  - used to force dependences under complex structures (graphs, ...)

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

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

Using element C[0][0] as representative/sentinel for the whole block.

Will build proper dependences between tasks.

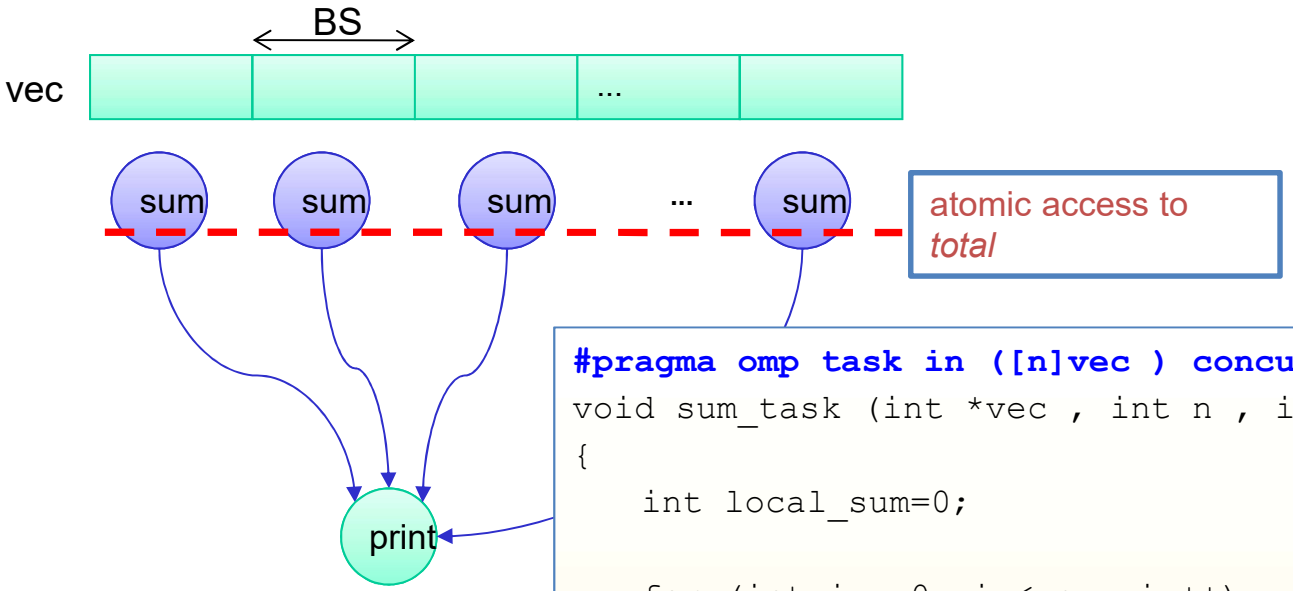
Does NOT provide actual information of data access pattern. (see copy clauses)

Fortran

- Pragmas outlined

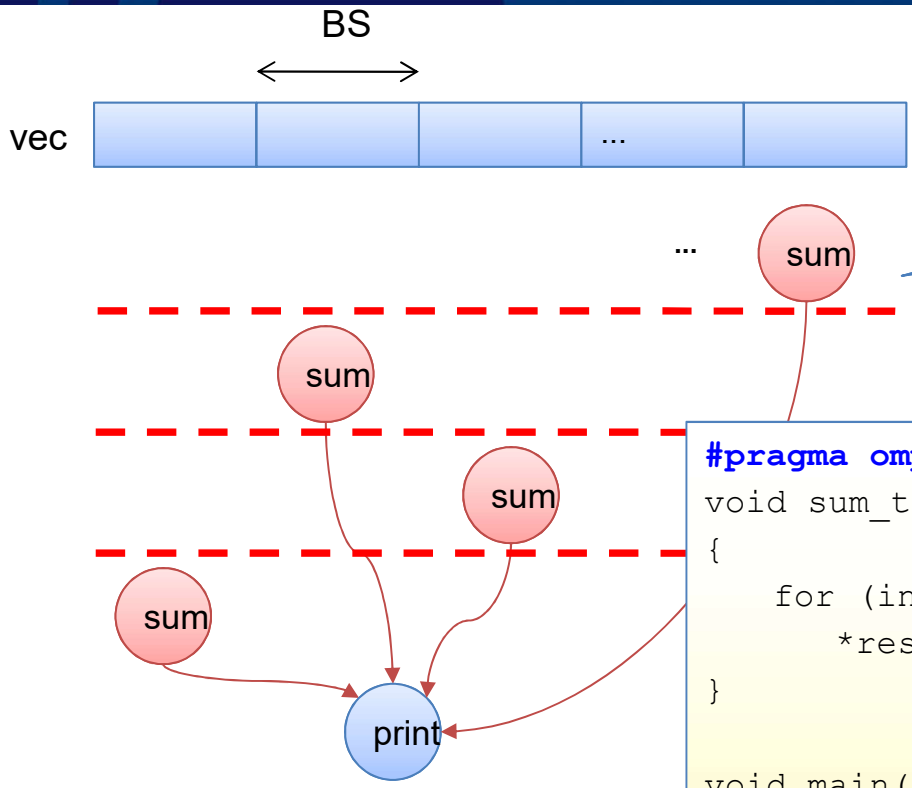
```
program example
parameter(N=2048)
integer, parameter :: BSIZE = 64
real v(N), vx(N),vy(N),vz(N)
integer :: jj
...
interface
    !$omp task out (v(1:BSIZE)) in (vx(1:BSIZE), vy(1:BSIZE), vz(1:BSIZE))\
    label(vmod)
    subroutine v_mod(BSIZE, v, vx, vy, vz)
            implicit none
            integer, intent(in) :: BSIZE
            real, intent(out) :: v(BSIZE)
            real, intent(in), dimension(BSIZE) :: vx, vy, vz
    end subroutine
end interface
...
do jj=1, N, BSIZE
call v_mod(BSIZE, v(jj), vx(jj), vy(jj), vz(jj))
enddo
…
!$omp taskwait
```

Concurrent in outlined tasks




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

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

#pragma omp atomic
    *result += local_sum;
}

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

Commutative in outlined tasks



Tasks executed out of order but not concurrently

No mutual access required

```
#pragma omp task in ([n]vec ) commutative(*result)
void sum_task (int *vec , int n , int  *result)
{
    for (int i = 0; i < n ; i ++)
       *result += vec [i] ;
}

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

Our team

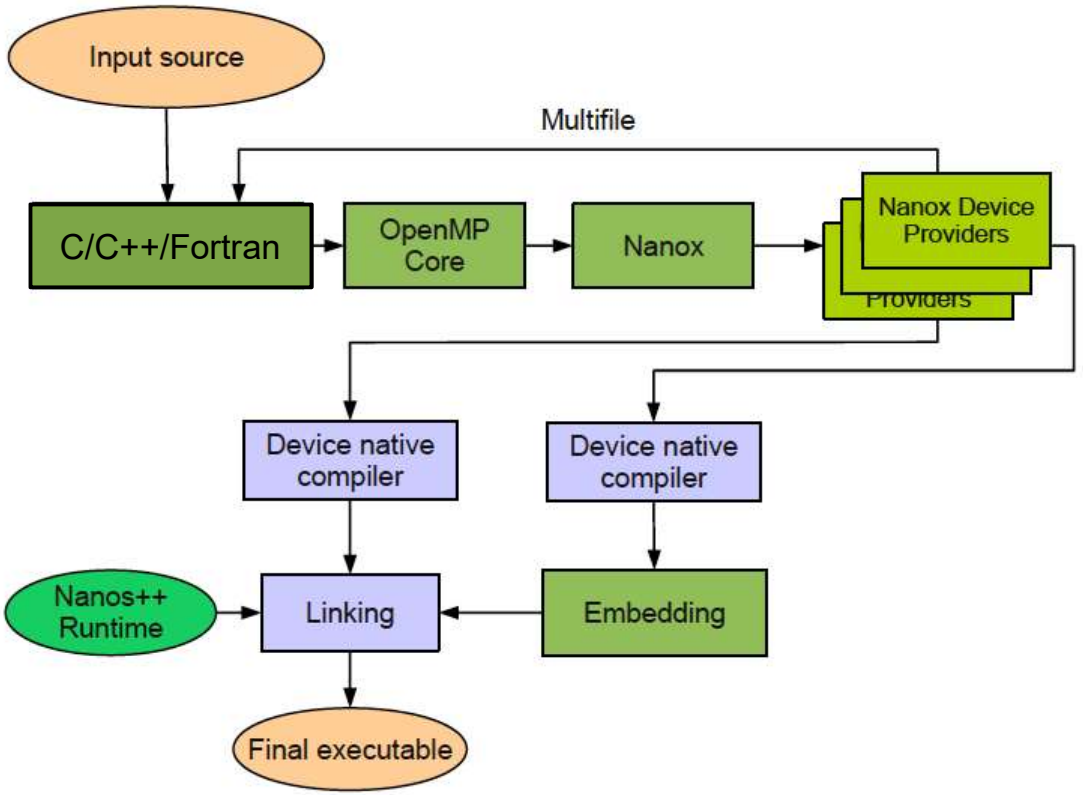
- Jesus Labarta
- Eduard Ayguade
- Rosa M. Badia
- Xavier Martorell
- Vicenç Bertran
- Antonio Peña
- Daniel Jiménez
- Carlos Álvarez
- Xavier Teruel
- Sergi Mateo
- Josep M. Perez
- Marta Garcia
- Victor Lopez
- Guray Ozen
- Antonio Filgueras
- Aimar Rodriguez
- Daniel Peyrolon
- Jaume Bosch
- Miquel Vidal

- Artem Cherkashin
- Marc Josep
- Julian Morillo
- Kevin Sala
- Marc Marí
- Ferran Pallarès
- Antoni Navarro
- Albert Navarro
- Alex Duran (Intel)
- Roger Ferrer (ARM)
- Javier Bueno (Metempsy)
- Judit Planas (Lausanne)
- Guillermo Miranda (UPC)
- Florentino Sainz (BBVA)
- Omer Subasi
- Javier Arias
- Judit Gimenez
- German Llort
- Harald Servat (Intel)
- …

OmpSs environment

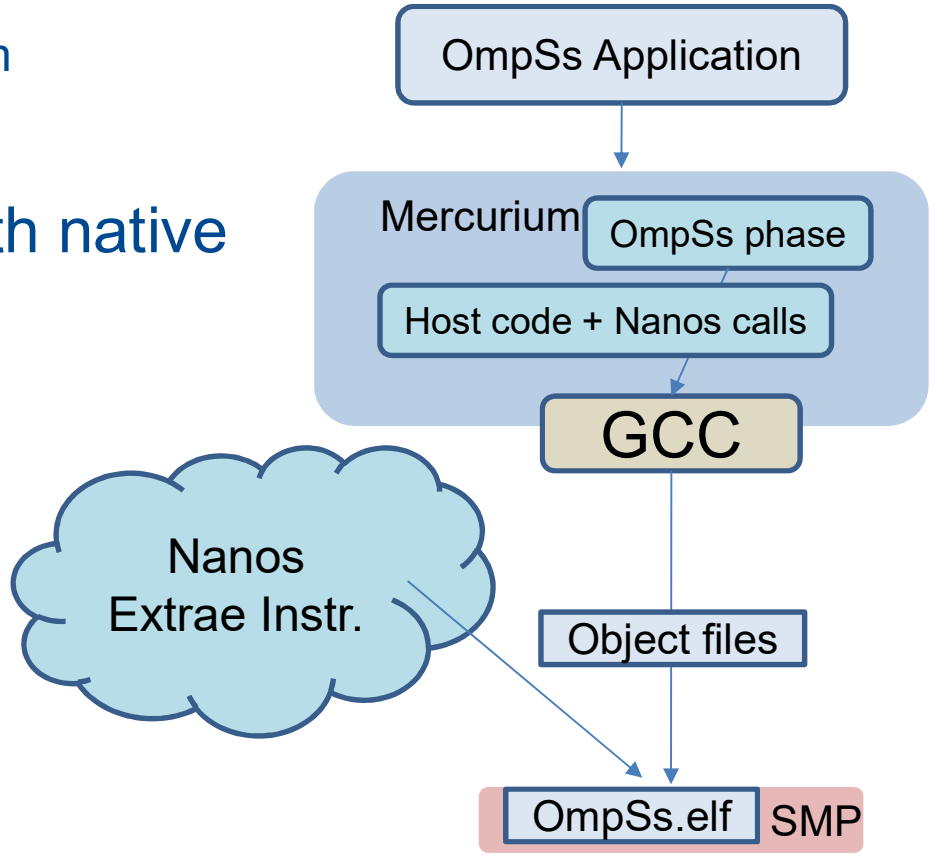
Mercurium Compiler

- 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
    - gcc, icc, xlc… for regular code
    - nvcc for NVIDIA



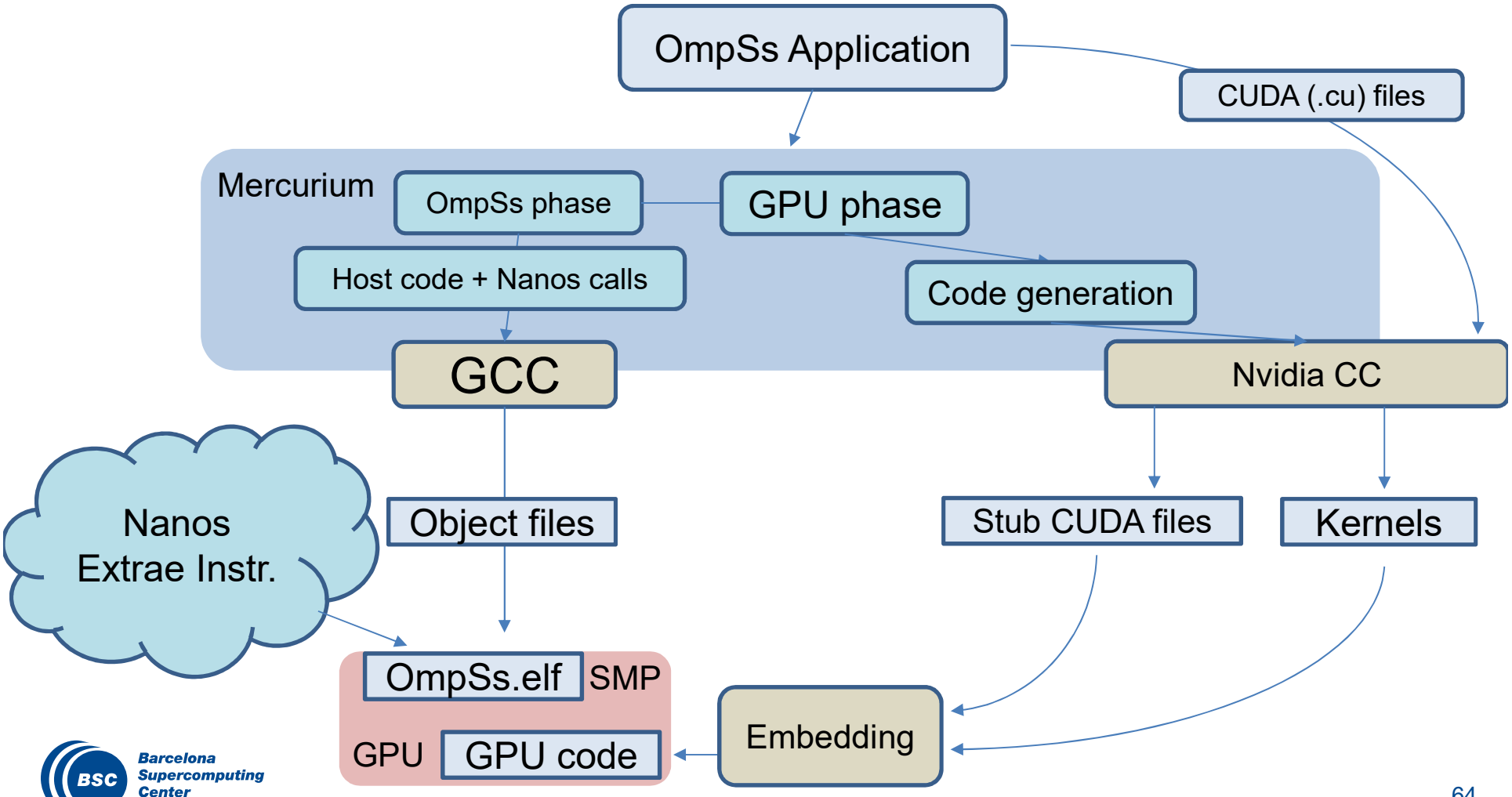
Mercurium compiler for SMP

- Takes application annotated with OmpSs directives
  - Inlined tasks
    - Code offloading + invocation from runtime system
  - Outlined tasks
    - Invocation from runtime system
- Compiles generated code with native compiler (gcc, llvm, icc...)
- Links against Nanos and Extrae



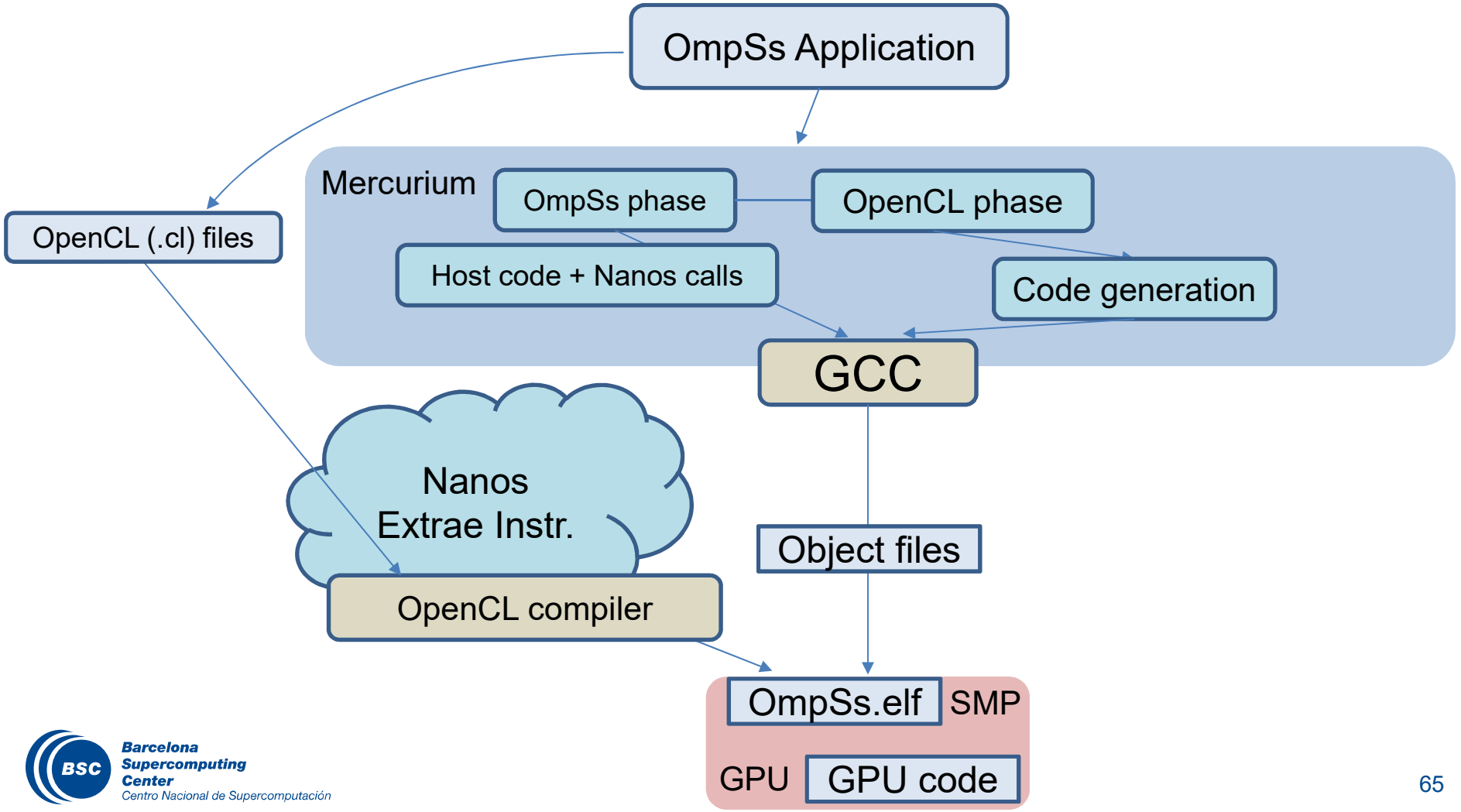
Mercurium for OmpSs@CUDA

- OmpSs transformations +
  - CUDA kernels compiled and embedded in binary format



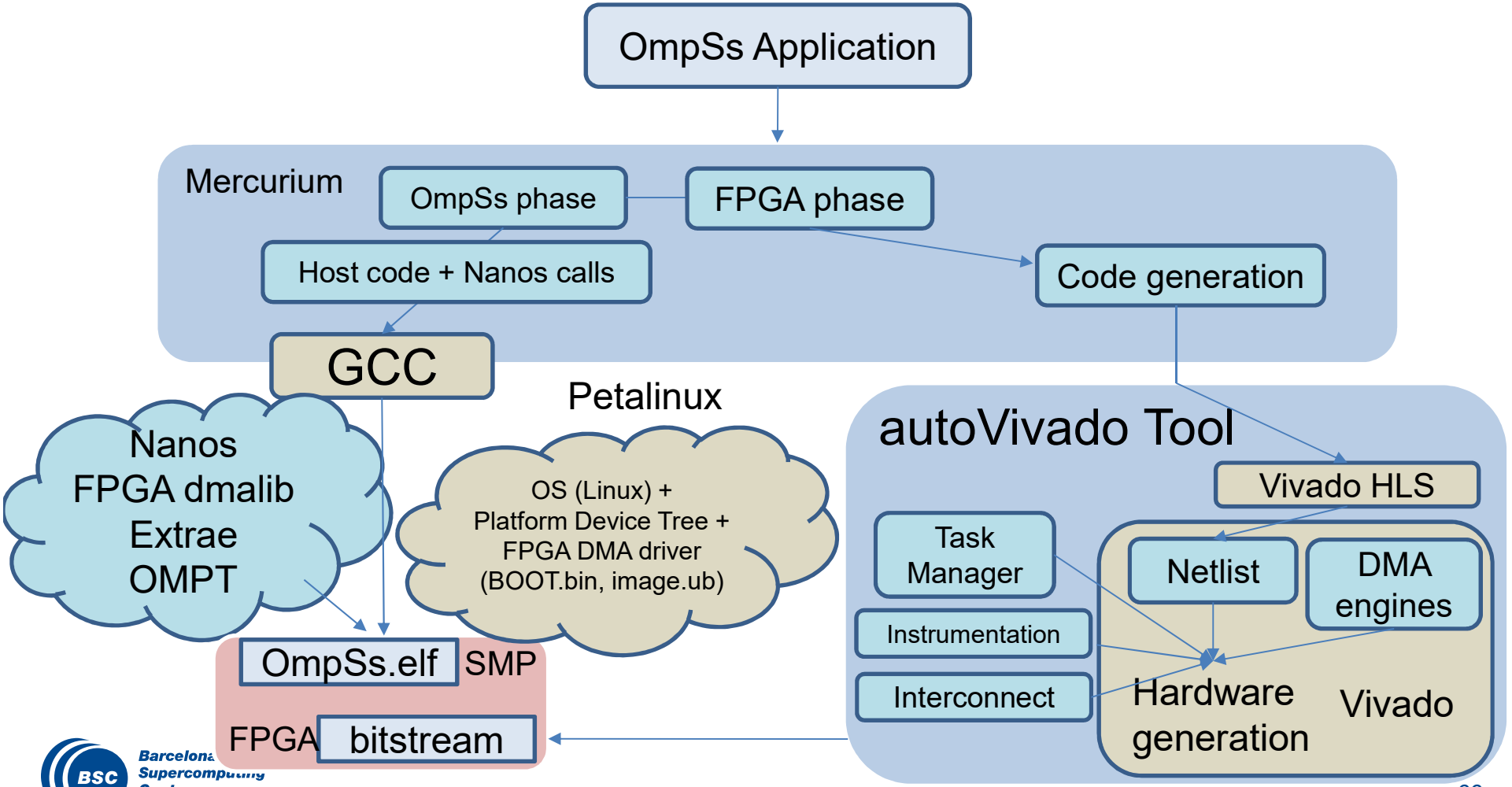
Mercurium for OmpSs@OpenCL

- OmpSs transformations +
  - OpenCL compilation at runtime



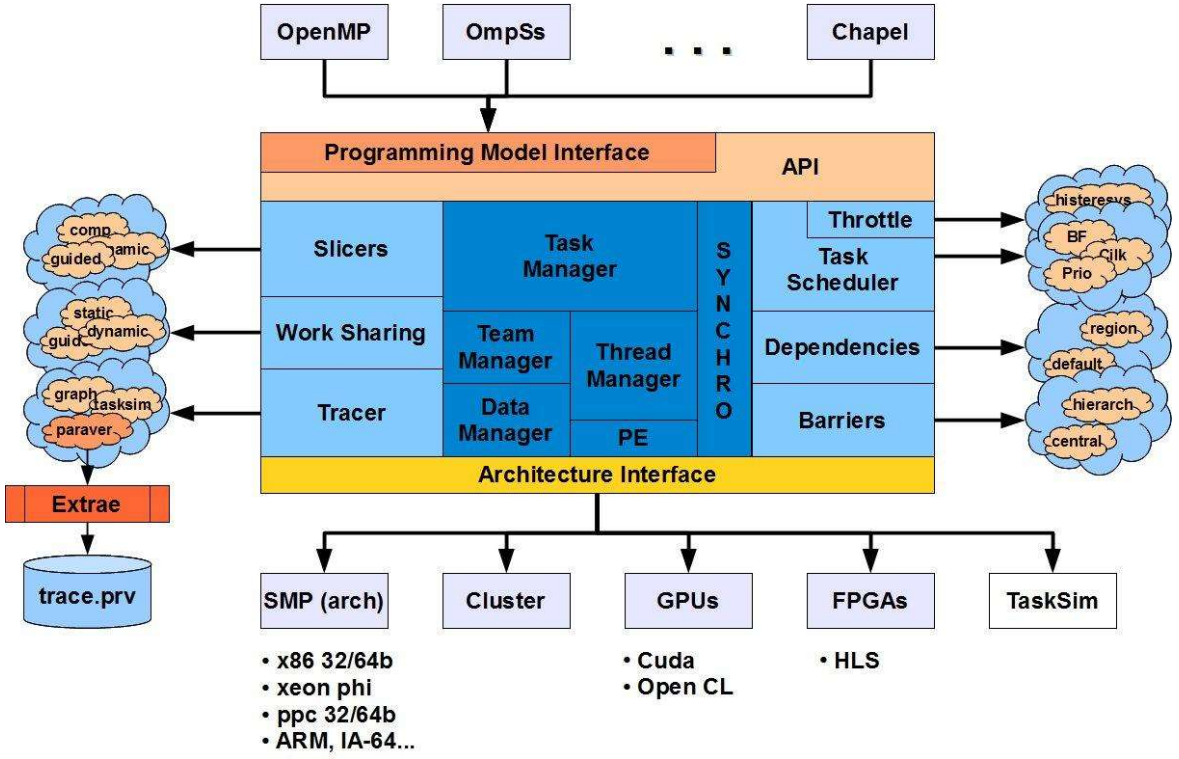
Mercurium, new FPGA toolchain: autoVivado

- OmpSs transformations on full source code +
  - Vivado HLS and Vivado invocations to generate the bitstream



The NANOS++ Runtime

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



Compiling

- Compiling

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

- Linking

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

- where *frontend* can be:

mcc	C (gcc backend)
mcxx	C++
imcc	C (icc backend)
mnvcc	CUDA and C
mnvcxx	CUDA and C++
mfc	Fortran (gfortran backend)
imfc	Fortran (ifort backend)
oclmcc	C and OpenCL
oclmcxx	C++ and OpenCL

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

- Basic execution:

```
> ./bin
```

  - No LD_LIBRARY_PATH or LD_PRELOAD needed

- Number of threads can be adjusted with OMP_NUM_THREADS

```
> OMP_NUM_THREADS=4 ./bin
> NX_ARGS="--smp-workers 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 named 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 runtime functionalities:

```
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

Nanos++ scheduling policies

- The scheduling policy defines how ready tasks are executed
  - The scheduling policy decides the order of execution of tasks and the resource where each task will be executed
- Alternatives:
  - Breadth first, default (bf)
  - Distributed breadth first (dbf)
  - Work First (wf)
  - Priority (priority)
  - Smart Priority (smartpriority)
  - Socket-aware (socket)
  - Affinity (affinity)
  - Affinity Smart Priority (affinity-smartpriority)
  - Versioning (versioning)
- Each scheduling policy has additional options:
  - See https://pm.bsc.es/projects/nanox/wiki/UserManual/Schedule
  - nanox --help

Nanos++ scheduling policies

- Use of schedulers:
  - export NX_SCHEDULE = <string>
  - export NX_ARGS="--schedule[=]<string>"

- Example:
  - export NX_SCHEDULE=dbf
  - export NX_ARGS="--enable-priorities"
  - ./bin

Task dependence analysis

- Nanos++ provides several plugins to handle task dependencies, with different performance and features

- Three different implementations:
  - Plain (plain, is the default)
  - Regions (regions)
  - Perfect regions (perfect-regions)
- Use:
  - export NX_DEPS=<string>
  - export NX_ARGS="--deps[=]<string>"
- Example:

```
> export NX_DEPS=regions
> ./bin
```

Task graph generation

- Compile and link with --instrument

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

- When executing specify graph as instrumentation package:

```
NX_INSTRUMENTATION=graph
./bin
```

- Will generate a pdf with the task graph
  - bin_xxxx.pdf

System installation

- Install Extrae library
  - Optional, to enable tracefile generation
  - Downloadable from
    - http://www.bsc.es/computer-sciences/performance-tools/downloads
  - Binary downloads available

- OmpSs components downloadable from
    - pm.bsc.es/ompss-downloads
  - Nanos++ runtime
    - Installation: configure & make
  - Mercurium compiler
    - Installation: configure & make

- Installation guide:
  - http://pm.bsc.es/ompss-docs/user-guide/installation.html

Nanos++ requirements

- The following software is needed in order to build Nanos++
  - A supported platform running Linux (i386, x86-64, PowerPC, IA64)
  - GNU gcc/g++ 4.3 or better
- In order to enable GPU support the following is also needed
  - CUDA 3.0 or better
- OpenCL support
  - 1.1 or better
- In order to enable Cluster support the following is also needed
  - GASNet 1.14.2 or better
- In order to enable Instrumentation support the following is also needed
  - Extrae 2.1.1 or better

Mercurium requirements

- You must fulfill the following requirements before building Mercurium:
  - A supported platform: A Linux platform supporting dynamic shared objects (i386, IA64, PowerPC, etc)
    - Compilation in Win32 under Cygwin is possible but it is not supported. Use at your own risk!
  - GNU gcc 4.1 (or better) compiler with C++ support
  - GNU Bison 2.3 or better (Note: bison-rofi is not needed anymore)
    - bison 2.4 is known to fail, use bison 2.4.1 instead
  - GNU gperf 3.0.0 or better
  - GNU flex 2.5.4 or 2.5.33 or better.
    - flex 2.5.31 is known to fail (it might work in some distributions), use 2.5.33 instead
    - flex 2.5.34 is known to fail, use 2.5.35 instead
  - Python 2.4 or better

Reporting problems

- Support mail
  - pm-tools@bsc.es
- Please include snapshot of the problem
  - Description
  - [Mercurium] preprocessed input file (gcc –E), when possible

Visualizing timelines

Visualizing Paraver tracefiles

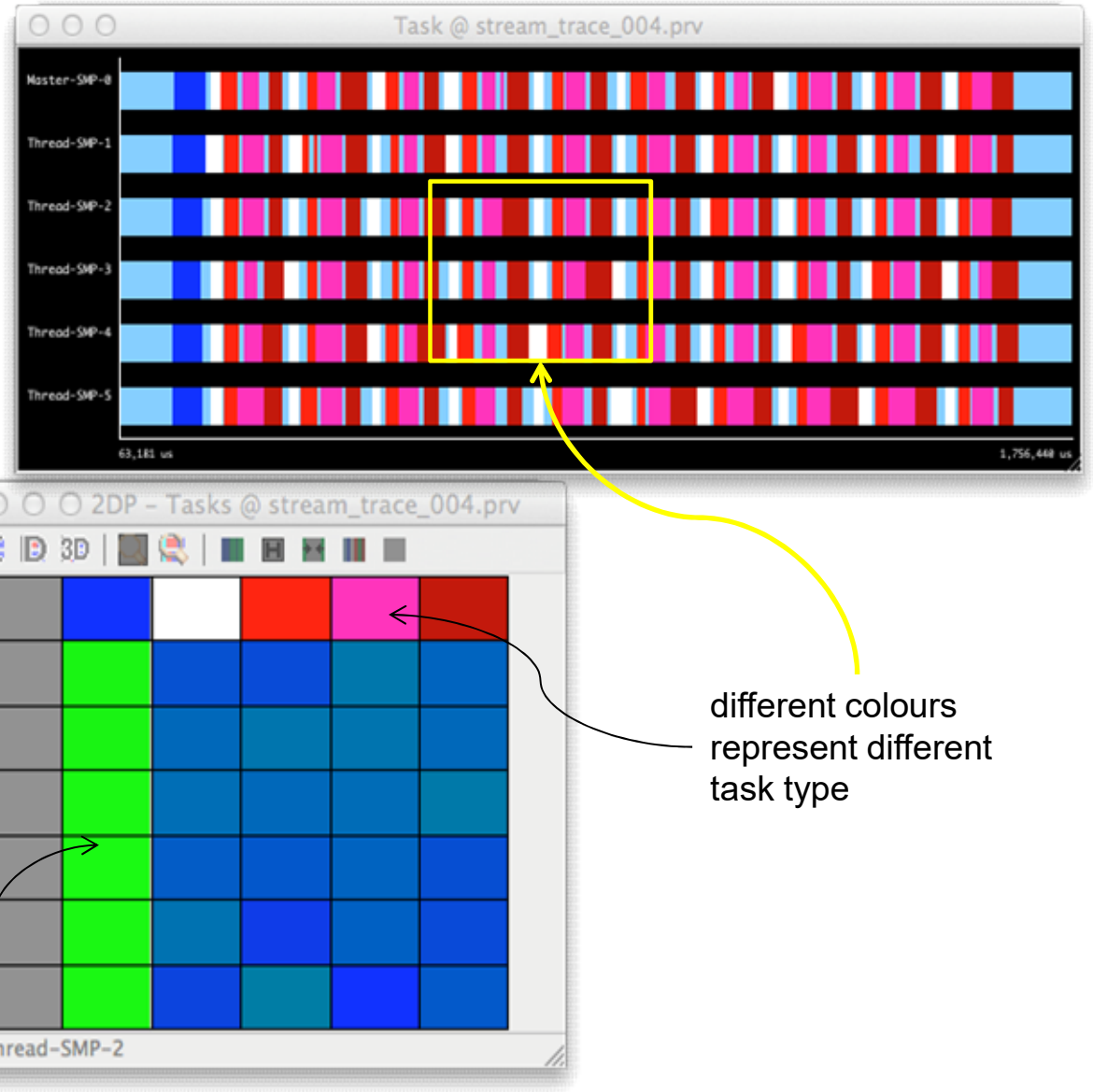
```
$ mcc --ompss --instrument -c prog.c
$ export EXTRAE_CONFIG_FILE=extrae.xml
$ NX_ARGS="--smp-workers 8 --instrumentation extrae"  ./a.out
```

- Will generate a paraver trace

- 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

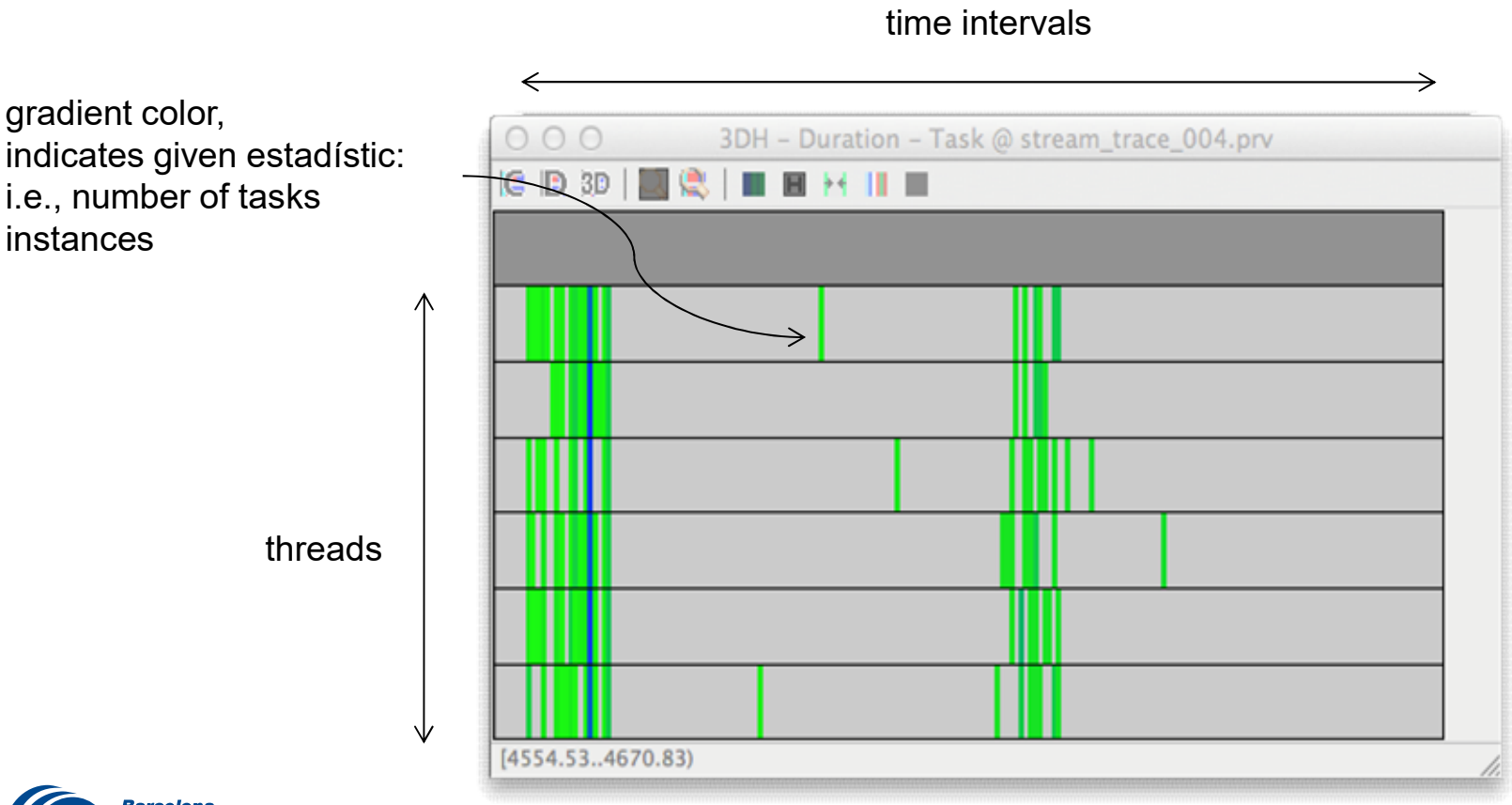
different colours represent different task type

threads

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

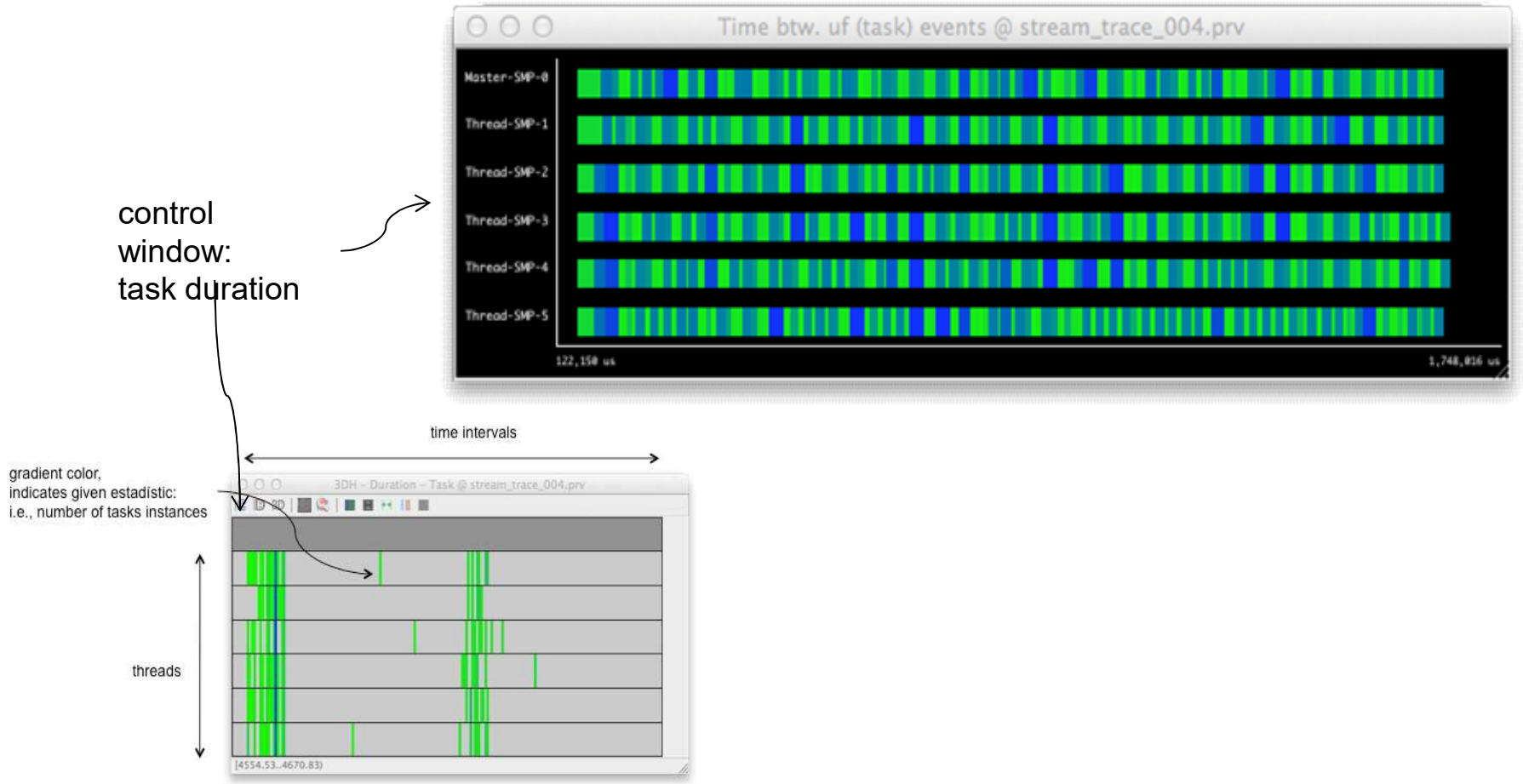
Tasks duration histogram

- 3dh_duration_task.cfg



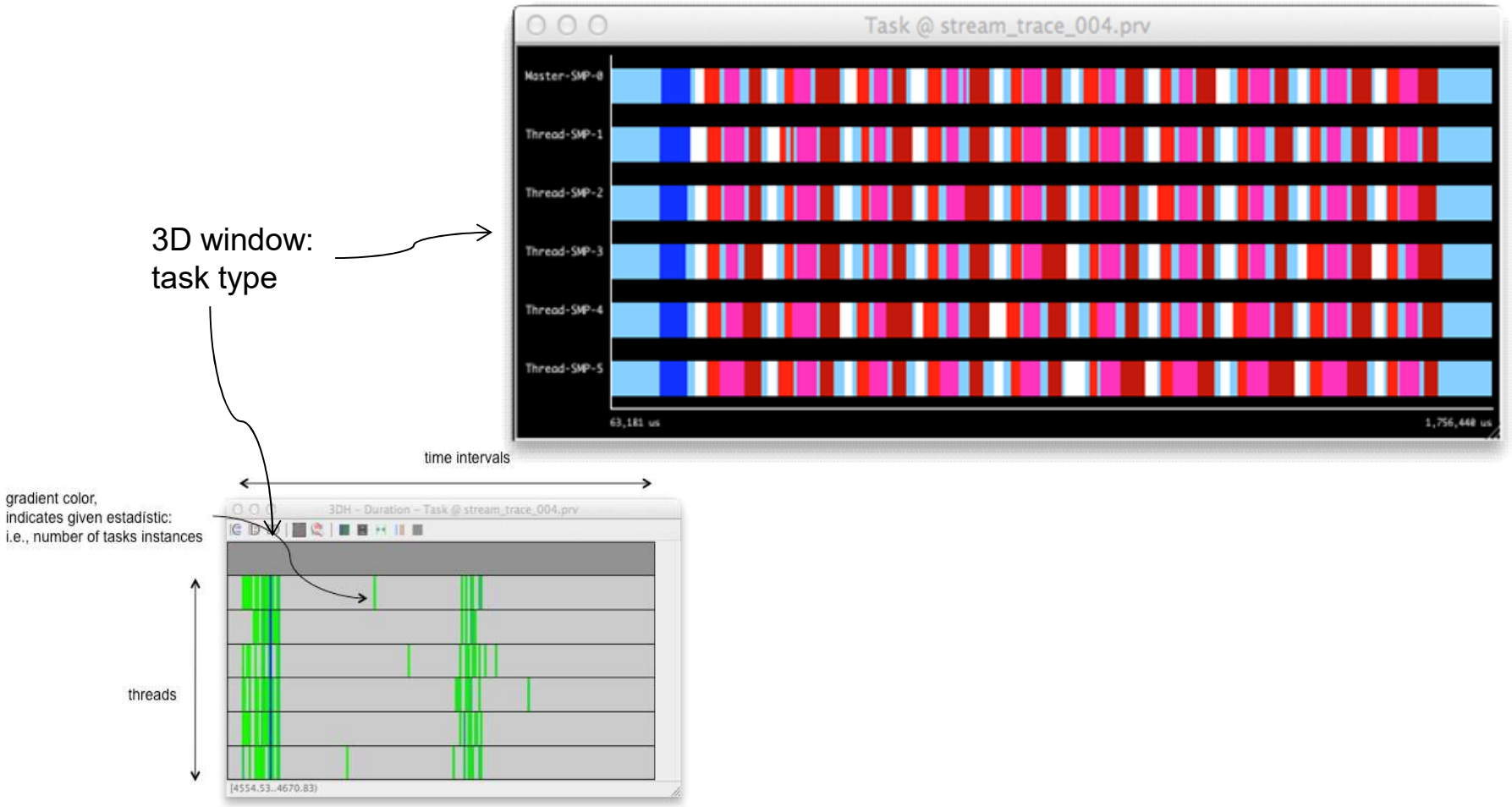
Tasks duration histogram

- 3dh_duration_task.cfg



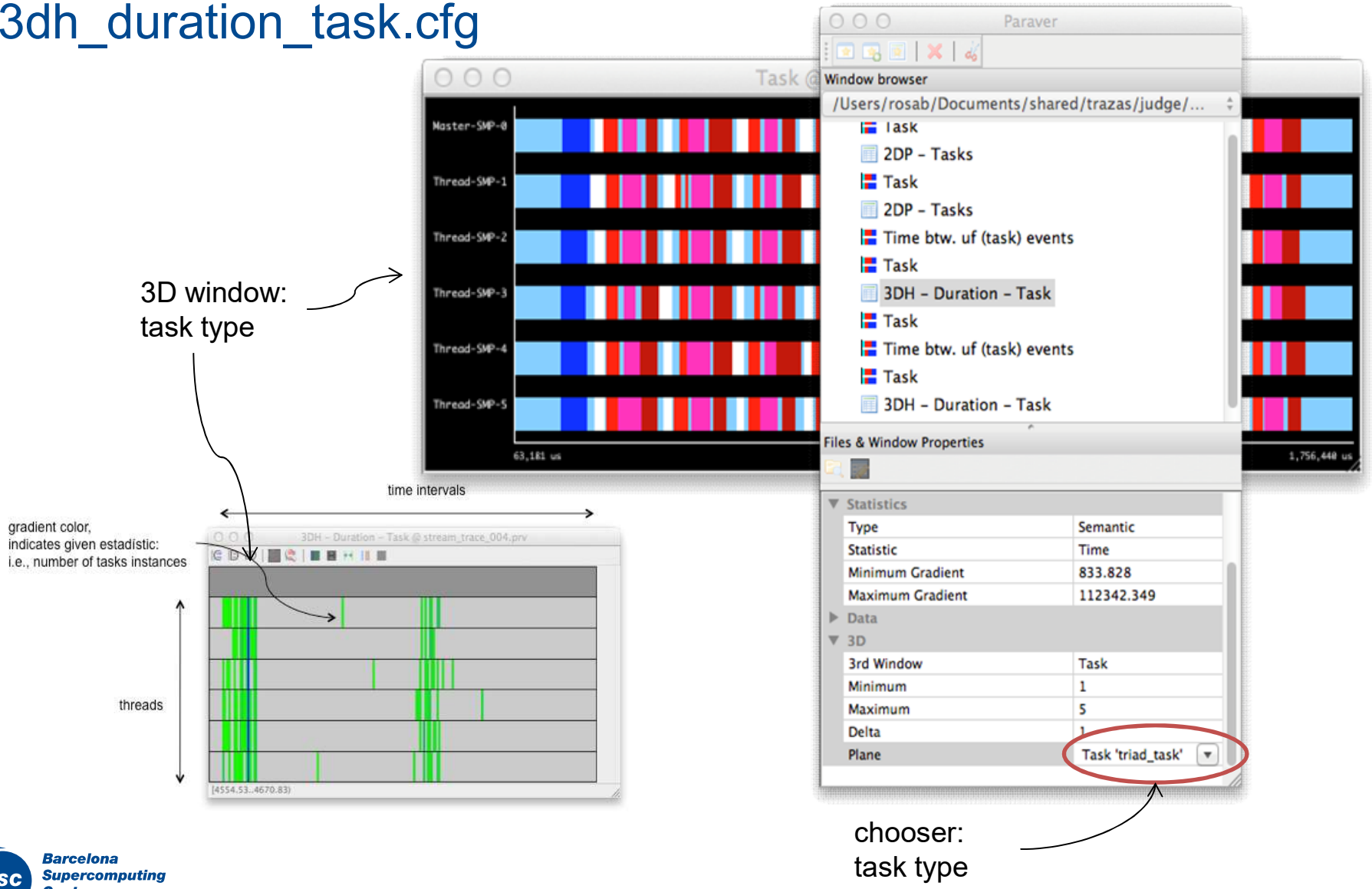
Tasks duration histogram

- 3dh_duration_task.cfg



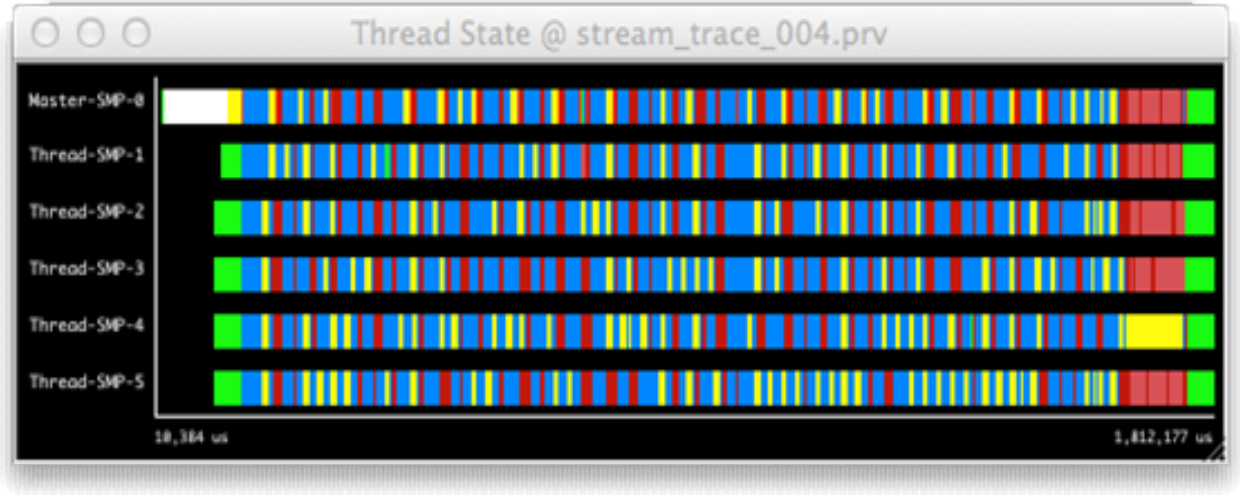
Tasks duration histogram

- 3dh_duration_task.cfg



Threads state profile

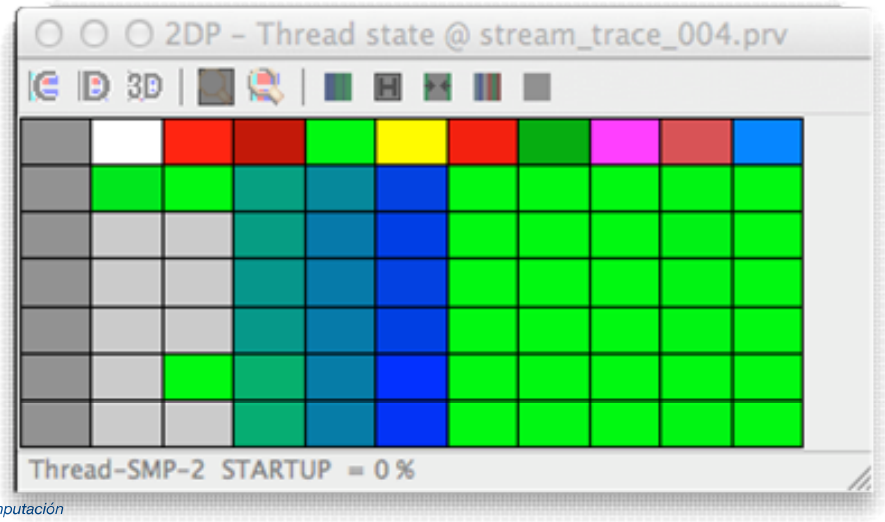
2dp_threads_state.cfg



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

runtime state

threads



THANKS

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Annex A: Inlined directives

Inlined task instantiation and wait

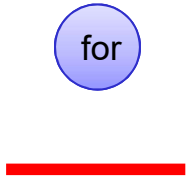
#pragma omp task

- Instantiates a task and control flow continues

#pragma omp taskwait

- Suspends the current control flow until all children tasks are completed

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

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

   ...
}
```



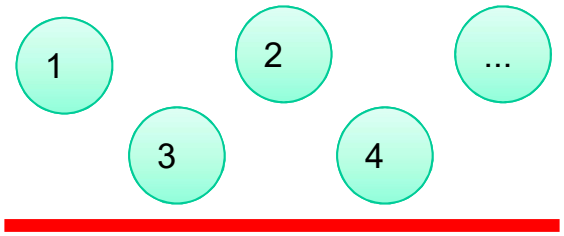
Inlined task instantiation and wait

#pragma omp task

- Instantiates a task and control flow continues

#pragma omp taskwait

- Suspends the current control flow 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

Data scope

- The data accessed by a task may be (as in OpenMP):
  - **Shared**: the task uses the original variable in the context where it is instantiated
  - **Private**: the task uses an uninitialized private copy of the data allocated at task activation time
  - **Firstprivate**: the task uses a private copy of the data initialized to the value of the variable in the instantiating context at task instantiation time
  - **NO thread private scope**.
- OmpSs scoping rules == OpenMP scoping rules

Inlined tasks data scope

- Variables for which no clause appears on the pragma:
  - Arrays declared global are shared by default
  - For the rest a capture value at task instantiation time semantic is followed
    - Scalars and local arrays the value is captured to initialize a private copy of the task. (== firstprivate)
    - For pointers, the pointer is captured so the task will refer to the original global object.

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

int main( )
{
   int X[4]={5,6,7,8};
   int v=0;

   #pragma omp task
   for (int i=0 ; i<4; i++) Y[i]=X[i]=v++;
   #pragma omp taskwait
      // value of v here ?
      // value of X[1] here ?
      // value of Y[1] here ?
}
```

Inlined tasks data scope

- Variables for which no clause appears on the pragma:
  - Arrays declared global are shared by default
  - For the rest a capture value at task instantiation time semantic is followed
    - Scalars and local arrays the value is captured to initialize a private copy of the task. (== firstprivate)
    - For pointers, the pointer is captured so the task will refer to the original global object.

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

int main( )
{
   int X[4]={5,6,7,8};
   int v=0;

   #pragma omp task
   for (int i=0 ; i<4; i++) Y[i]=X[i]=v++;
   #pragma omp taskwait
      // value of v here is 0
      // value of X[1] here is 6
      // value of Y[1] here is 1
}
```

Inlined tasks data scope

- Variables for which no clause appears on the pragma:
  - Arrays declared global are shared by default
  - For the rest a capture value at task instantiation time semantic is followed
    - Scalars and local arrays the value is captured to initialize a private copy of the task. (== firstprivate)
    - For pointers, the pointer is captured so the task will refer to the original global object.

```
int main( )
{
   int *X;
   int v=0;

      X = (int *) malloc(4*sizeof(int));

   #pragma omp task
   for (int i=0 ; i< 4; i++) X[i]=v++;
   #pragma omp taskwait

      //value of X[1] here ?
}
```

Inlined tasks data scope

- Variables for which no clause appears on the pragma:
  - Arrays declared global are shared by default
  - For the rest a capture value at task instantiation time semantic is followed
    - Scalars and local arrays the value is captured to initialize a private copy of the task. (== firstprivate)
    - For pointers, the pointer is captured so the task will refer to the original global object.

```
int main( )
{
   int *X;
   int v=0;

      X = (int *) malloc(4*sizeof(int));

   #pragma omp task
   for (int i=0 ; i< 4; i++) X[i]=v++;
   #pragma omp taskwait

      //value of X[1] here is 1
}
```

Inlined tasks data scope

- Data scope (… as in OpenMP)
  - Variables can be explicitly declared as :
    - Shared
    - Private
    - Firstprivate
- In case of doubt
  - Default (none)
  - Explicitly declare

```
int Y[4]={1,2,3,4};
int main()
{
   int X[4]={5,6,7,8};
   int v=0;

   #pragma omp task shared (v, X)
   for (int i=0 ; i<4; i++) Y[i]=X[i]=v++;
   #pragma omp taskwait

      // value of v here ?
      // value of X[1] here ?
      // value of Y[1] here ?
}
```

Inlined tasks data scope

- Data scope (… as in OpenMP)
  - Variables can be explicitly declared as :
    - Shared
    - Private
    - Firstprivate
- In case of doubt
  - Default (none)
  - Explicitly declare

```
int Y[4]={1,2,3,4};
int main()
{
   int X[4]={5,6,7,8};
   int v=0;

   #pragma omp task shared (v, X)
   for (int i=0 ; i<4; i++) Y[i]=X[i]=v++;
   #pragma omp taskwait

      // value of v here is 4
      // value of X[1] here is 1
      // value of Y[1] here is 1
}
```

Inlined tasks data scope

- Explicit scope declaration
  - Firstprivate: a private instance of the data initialized to the value of the original variable at task instantiation time.
  - Private: an uninitialized instance is allocated and used by the task
  - In both cases, the private data instance is deleted on task completion.

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

int main( )
{
int i=2;

#pragma omp task firstprivate (i)
for ( ; i< 4; i++) X[i]=i;
#pragma omp taskwait

   // value of i here ?
   // value of X[0] here ?
   // value of X[3] here ?
}
```

```
int main( )
{
int X[4]={1,2,3,4};

int i=2;

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

   // value of i here ?
   // value of X[0] here ?
   // value of X[3] here ?
}
```

Inlined tasks data scope

- Explicit scope declaration
  - Firstprivate: a private instance of the data initialized to the value of the original variable at task instantiation time.
  - Private: an uninitialized instance is allocated and used by the task
  - In both cases, the private data instance is deleted on task completion.

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

int main( )
{
int i=2;

#pragma omp task firstprivate (i)
for ( ; i< 4; i++) X[i]=i;
#pragma omp taskwait

   // value of i here is 2
   // value of X[0] here is 1
   // value of X[3] here is 3
}
```

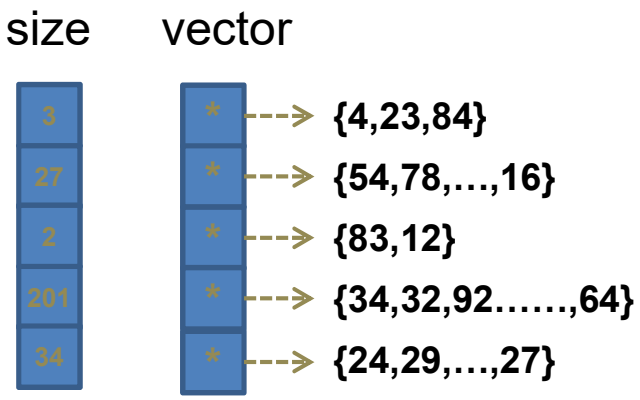
```
int main( )
{
int X[4]={1,2,3,4};

int i=2;

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

   // value of i here is 2
   // value of X[0] here is 0
   // value of X[3] here is 3
}
```

Tasks: the if clause

```
#pragma omp task [ if(…) ]
```

- if (expr): If expression evaluates to false, task will be created but will be executed immediately (not deferred)
  - Scheduling rather than overhead
  - OpenMP issues if taskwait inside the structured block and it is executed inline as part of the calling region

```
void foo(int *size, int **vector, int N) {
 for (int i=0; i<N; i++) {
   #pragma omp task if ( size[i] > MIN_SIZE )
   compute(vector[i], size[i]) ;
 }
}
```



Dependences

```
#pragma omp task in(lvalue_expr-list) \
                 out(lvalue_expr-list) \
                 inout(lvalue_expr-list)
```

- Specify data that has to be **available** before activating task
- Specify data this task produces that might activate other tasks
- Contiguous / array sections
- You do NOT specify dependencies but information for the runtime to compute them

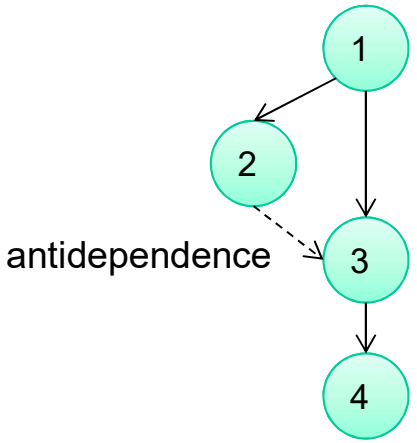
- Relaxations:
  - No need to specify for ALL data touched
  - Order of inout chains
    - Programmer responsible of possible races

```
#pragma omp concurrent (…)
    #pragma omp critical

#pragma omp commutative (…)
```

Defining dependences for inlined tasks

- Clauses that express data direction:
  - In, out, inout
  - These clauses **imply shared scope** for the variable referenced in them
  - The argument is an **lvalue** expression referring to the object that will be used to compute dependences
  - Uses of the lvalue expression in the task body reference the global object, whose value is guaranteed to have been produced by the “latest” task to “modify” it (latest out/inout task in sequential program order) if any.
- Dependences computed at runtime taking into account these clauses
  - Flow, anti, output

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



OpenMP: Dependencies

- Dependences are now supported in OpenMP 4.0
  - Different syntax but same semantics

#pragma omp task [depend (in: var_list)] [depend(out: var_list)] [depend(inout: var_list)]

{ code block }

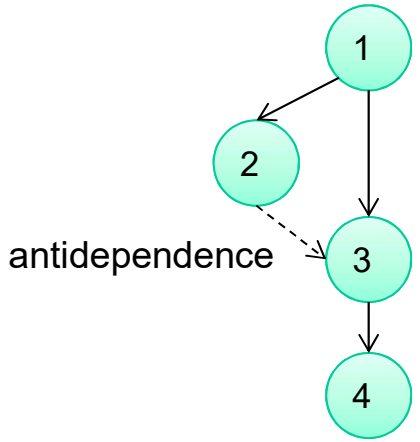
"OpenMP Application Program Interface. Version 4.0 " July 2013

and

"OpenMP Application Program Interface. Version 4.5" November 2015

Defining dependences for inlined tasks in OpenMP 4.0

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

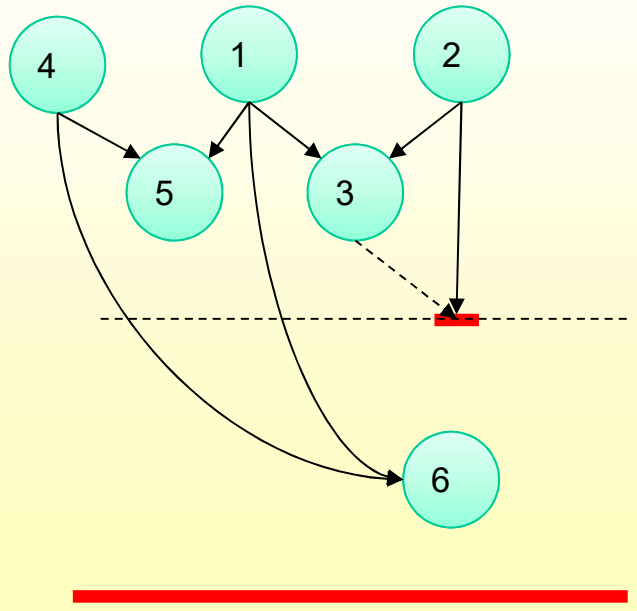


Partial control flow synchronization (OmpSs)

#pragma taskwait on (lvalue_expr-list)

- Expressions allowed are the same as for the directionality clauses
- Stalls the encountering control flow until the referenced data is available

```
double A[N][N], B[N][N], C[N][N], D[N][N], E[N][N],
       F[N][N], G[N][N], H[N][N], I[N][N], J[N][N];

main() {
  #pragma omp task in (A, B) out(C)
  dgemm(A,B,C);     //1
  #pragma omp task in (D, E) out(F)
  dgemm(D,E,F);     //2
  #pragma omp task in (C, F) out(G)
  dgemm(C,F,G);     //3
  #pragma omp task in (A, D) out(H)
  dgemm(A,D,H);     //4
  #pragma omp task in (C, H) out(I)
  dgemm(C,H,I);     //5

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

  #pragma omp task in (C, H) out(J)
  dgemm(H,C,J);     //6

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

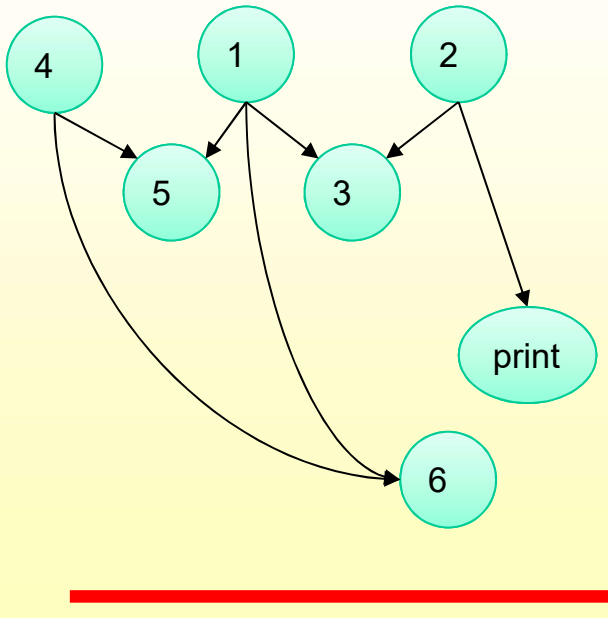


Partial control flow synchronization (OmpSs & OpenMP)

#pragma taskwait on (lvalue_expr-list)

- Expressions allowed are the same as for the directionality clauses
- Stalls the encountering control flow until the referenced data is available

```
double A[N][N], B[N][N], C[N][N], D[N][N], E[N][N],
       F[N][N], G[N][N], H[N][N], I[N][N], J[N][N];

main() {
  #pragma omp task in (A, B) out(C)
  dgemm(A,B,C);     //1
  #pragma omp task in (D, E) out(F)
  dgemm(D,E,F);     //2
  #pragma omp task in (C, F) out(G)
  dgemm(C,F,G);     //3
  #pragma omp task in (A, D) out(H)
  dgemm(A,D,H);     //4
  #pragma omp task in (C, H) out(I)
  dgemm(C,H,I);     //5

  #pragma omp task in(F)
  prinft ("result F = %f\n", F[0][0]);

  #pragma omp task in (C, H) out(J)
  dgemm(H,C,J);     //6

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



Array sections

- Indicating as in/out/inout subregions of a larger structure:

  in (A[i])
  → the input argument is element *i* of A

- Indicating an array section:

  in ([BS]A)
  → the input argument is a block of size BS from address A

  in (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)

  in (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]
  → the upper bound can be omitted if size is known to the compiler
  (default is N-1, being N the size)

Array sections

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

=

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

=

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

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

```
int a[N];
#pragma omp task in(a[2:3])
//elements 2 and 3 of the array used to compute dependences
```

=

```
int a[N];
#pragma omp task in(a[2;2])
//elements 2 and 3 of the array used to compute dependences
```

Array sections

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

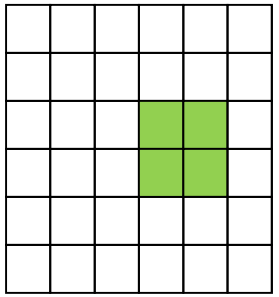
=

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

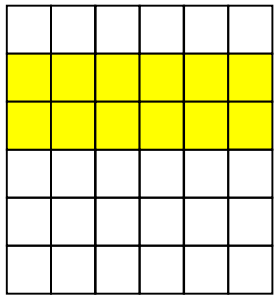
```
int a[N][M];
#pragma omp task in(a[2:3][3:4])
// 2 x 2 subblock of a at a[2][3]
```

=

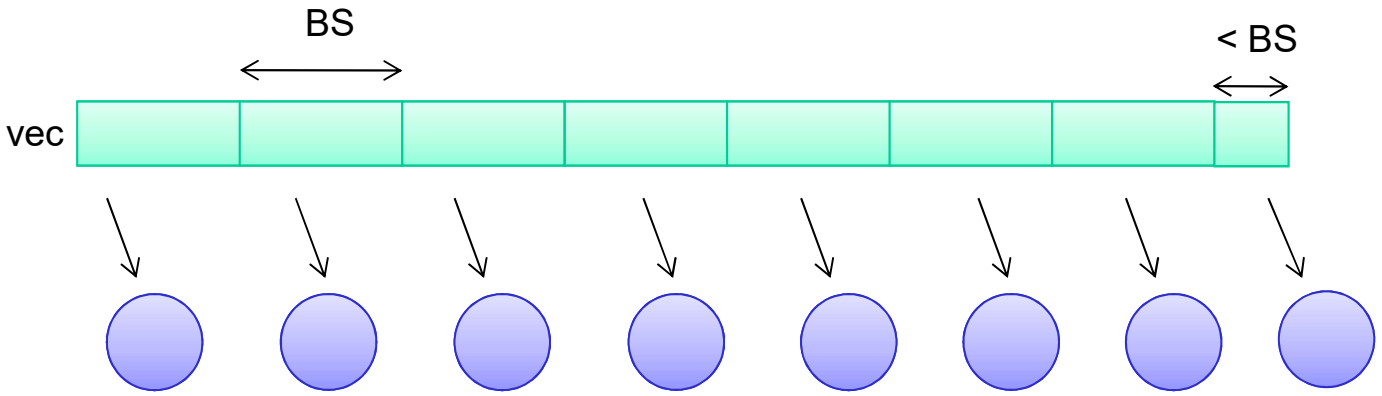
```
int a[N][M];
#pragma omp task in(a[2;2][3;2])
// 2 x 2 subblock of a at a[2][3]
```



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



Array sections

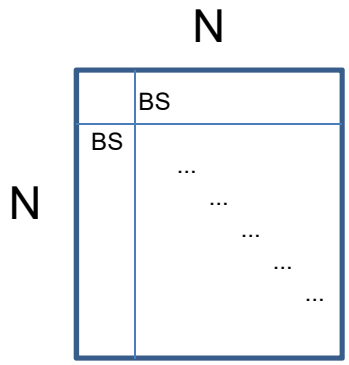



```
for (int j; j<N; j+=BS){

   actual_size = (N- j> BS ? BS: N-j);

   #pragma omp task in (vec[j;actual_size]) inout(c[j;actual_size])
   for (int count = 0; count < actual_size; count ++, j++)
        c[j] += vec [j] ;

}
```

dynamic size of argument

Array sections

```
void matmul(double A[N][N], double B[N][N],
            double C[N][N], unsigned long BS)
{
  int i, j, k;

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



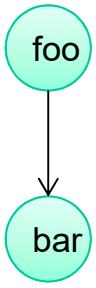
N = NB * BS

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

  for (i = 0; i < N; i+=BS)
    for (j = 0; j < N; j+=BS)
      for (k = 0; k < N; k+=BS) {
        #pragma omp task in(A[i;BS][k;BS], B[k;BS][j;BS]) inout(C[i;BS][j;BS])
        matmul (&A[i][k], &B[k][j], &C[i][j], BS);
      }
}
```

Incomplete directionalities specification

- Directionality not required for all arguments
- May even refer to variables not used in the body of the task
  - Used to force dependences under complex structures (graphs, ...)

```
main () {
   int sentinel;

#pragma omp task out (sentinel)
   foo (...);

#pragma omp task in (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 make code non-portable to heterogeneous platforms if copy in/out clauses cannot properly specify the address space that should be accessible in the devices

Incomplete directionalities specification

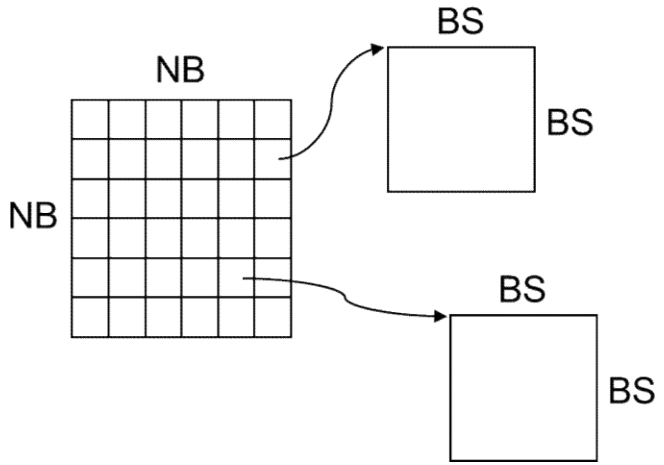
- Directionality not required for all arguments
- May use sentinel as representative of a data object

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

  for (i=0; i<NB; i++)
    for (j=0; j<NB; j++)
      for (k=0; k<NB; k++) {
        #pragma omp task in(A[i][k], B[k][j]) inout(C[i][j])
        matmul (A[i][k], B[k][j], C[i][j], BS);
      }
}
```

Using entry in C matrix of pointers as representative/sentinel for the whole block it points to.

Will build proper dependences between tasks.

Does NOT provide actual information of data access pattern. (see copy clauses)



Incomplete directionalities specification

- Directionality not required for all arguments
- May use sentinel as representative of a data object

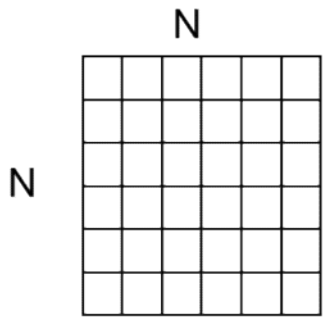
```
void compute(unsigned long N, unsigned long BS,
             double A[N][N], double B[N][N], double C[N][N])
{
  unsigned i, j, k;

  for (i=0; i<N; i+=BS)
    for (j=0; j<N; j+=BS)
      for (k=0; k<N; k+=BS) {
        #pragma omp task in(A[i][k]) inout(C[i][j])
        matmul (&A[i][k], &B[k][j], &C[i][j], BS);
      }
}
```

NO B block specified as in.
Still correct results if B is available before calling compute.

First element of C block used as representative/sentinel for the whole block

Will build proper dependences between tasks serializing updates of C blocks

Does NOT provide actual information of data access pattern. (see copy clauses)



N = NB * BS

OmpSs Cholesky with OpenMP 4.0 syntax

```
void cholesky_blocked(const int ts, const int nt, double* Ah[nt][nt])
{
   for (int k = 0; k < nt; k++) {
      // Diagonal Block factorization
#pragma omp task depend( inout: Ah[k][k] ) firstprivate(k,ts) label (potrf)
      omp_potrf (Ah[k][k], ts, ts);
      // Triangular systems
      for (int i = k + 1; i < nt; i++) {
#pragma omp task depend(in: Ah[k][k] ) depend(inout: Ah[k][i]) firstprivate(k,i,ts) \
                                                                    label (trsm)
         omp_trsm (Ah[k][k], Ah[k][i], ts, ts);
      }
      // Update trailing matrix
      for (int i = k + 1; i < nt; i++) {
         for (int j = k + 1; j < i; j++) {
#pragma omp task depend(in: Ah[k][i], Ah[k][j] ) depend(inout: Ah[j][i] ) \
                                                 firstprivate(k,i,j,ts) label (gemm)
            omp_gemm (Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);
         }
#pragma omp task depend(in: Ah[k][i]) depend(inout: Ah[i][i]) firstprivate(k,i,ts) \
                                                                    label (syrk)
         omp_syrk (Ah[k][i], Ah[i][i], ts, ts);
      }
   }
#pragma omp taskwait
}
```

OmpSs: syntax in Fortran

```
!$OMP TARGET DEVICE({SMP|CUDA|OPENCL|MPI})&
          [NDRANGE (…)]&
          [IMPLEMENTS (function_name)]&
          {[COPY_DEPS][NO_COPY_DEPS |[COPY_IN (…)][COPY_OUT (…)][COPY_INOUT(..)]}&
          [FILE (…)] [SHMEM (…)][NAME (…)]


!$OMP TASK [IN (…)] [OUT(...)] [INOUT (…)] &
         [CONCURRENT (…)][COMMUTATIVE (…)] &
         [PRIORITY (…)][LABEL (…)]&
         [SHARED(…)][PRIVATE(…)][FIRSTPRIVATE(…)][DEFAULT (…)]&
         [UNTIED][IF(…)][FINAL(…)]
{ code or function }


  !$OMP TASKWAIT [ON(…)][NOFLUSH]
```

Fortran

- Pragmas inlined

```
program example
parameter(N=2048)
integer, parameter :: BSIZE = 64
real v(N), vx(N),vy(N),vz(N)
integer :: jj
...
interface
    subroutine v_mod(BSIZE, v, vx, vy, vz)
           implicit none
           integer, intent(in) :: BSIZE
           real, intent(out) :: v(BSIZE)
           real, intent(in), dimension(BSIZE) :: vx, vy, vz
    end subroutine
end interface
...
do jj=1, N, BSIZE
!$omp task out (v(jj)) in (vx(jj), vy(jj), vz(jj)) firstprivate (jj) \
                                                          label(vmod)
    call v_mod(BSIZE, v(jj), vx(jj), vy(jj), vz(jj))
!$omp end task
enddo
...
!$omp taskwait
```

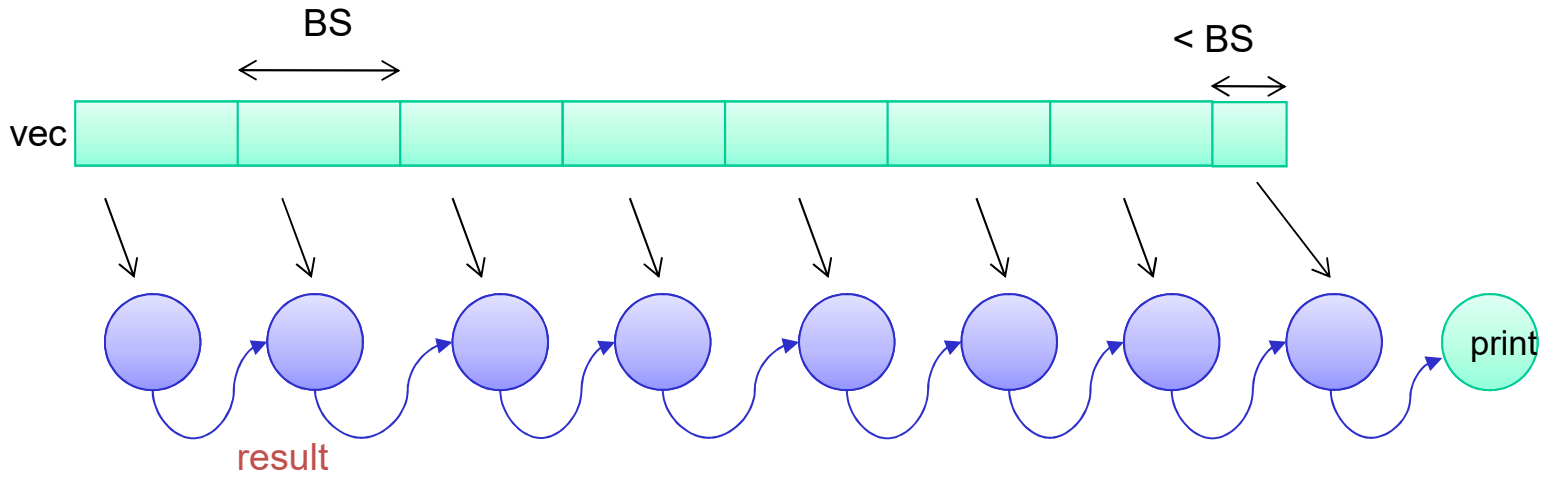
Concurrent

```
#pragma omp task concurrent (lvalue_expr)
```

- **Relaxed inout** directionality clause
  - Enables the scheduler to change the order (or **even execute concurrently**) the tasks within the inout chains built by the concurrent clause.
    - Dependences resulting form the regular in, out and inout clauses towards and from the concurrent chain are honoured for **all** tasks in the concurrent chain.
  - The programmer assumes responsibility to handle potential races within the concurrent chain using if needed OpenMP pragmas

```
#pragma omp atomic
#pragma omp critical
```

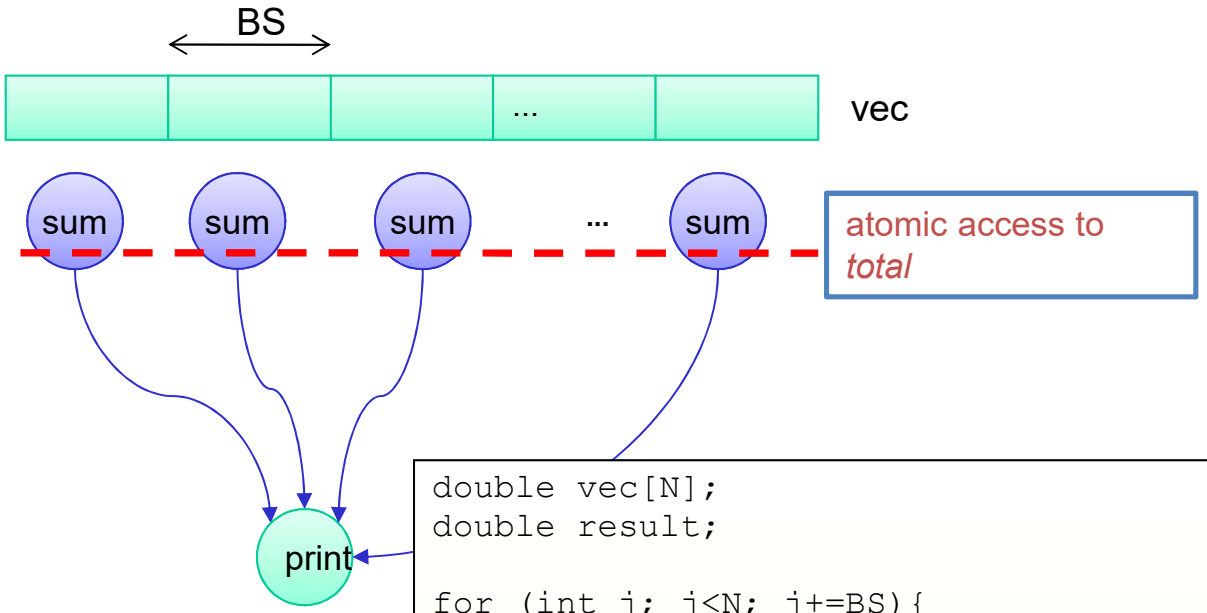
Serialized reduction pattern




```
for (int j=0; j<N; j+=BS){

   actual_size = (N- j> BS ? BS: N-j);

   #pragma omp task in (vec[j;actual_size]) inout(result)
   for (int count = 0; count < actual_size; count ++,j++)
      result += vec [j] ;

}
#pragma omp task input (result)
  printf ("TOTAL is %d\n", result);
#pragma omp taskwait
```

Serialization

Concurrent



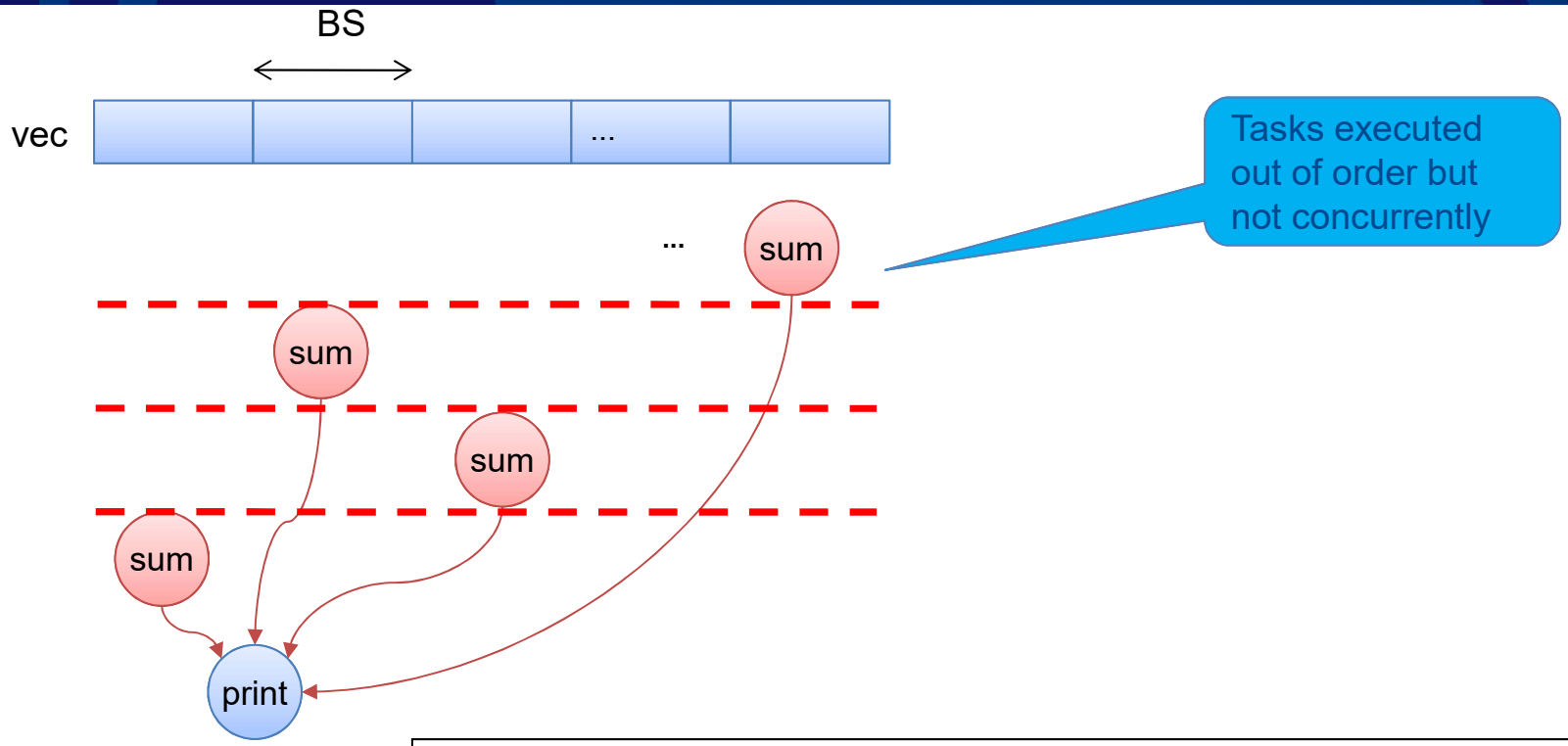

```
double vec[N];
double result;

for (int j; j<N; j+=BS){
   atual_size = (N- j> BS ? BS: N-j);

   #pragma omp task in (vec[j;actual_size]) concurrent(result)
   {  double local_result=0.0;
      for (int count = 0; count < actual_size; count ++)
         local_result += vec [j++] ;
      #pragma omp atomic
      result += local_result;
   }
}
#pragma omp task input (result)
  printf ("TOTAL is %d\n", result);
#pragma omp taskwait
```

Commutative

```
#pragma omp task commutative (lvalue_expr)
```

- **Relaxed inout** directionality clause
  - Enables the scheduler to change the order, but **not to execute concurrently** the tasks within the inout chains built by the commutative clause.
    - Dependences resulting form the regular in, out and inout clauses towards and from the commutative chain are honoured for all tasks in the commutative chain.

Commutative

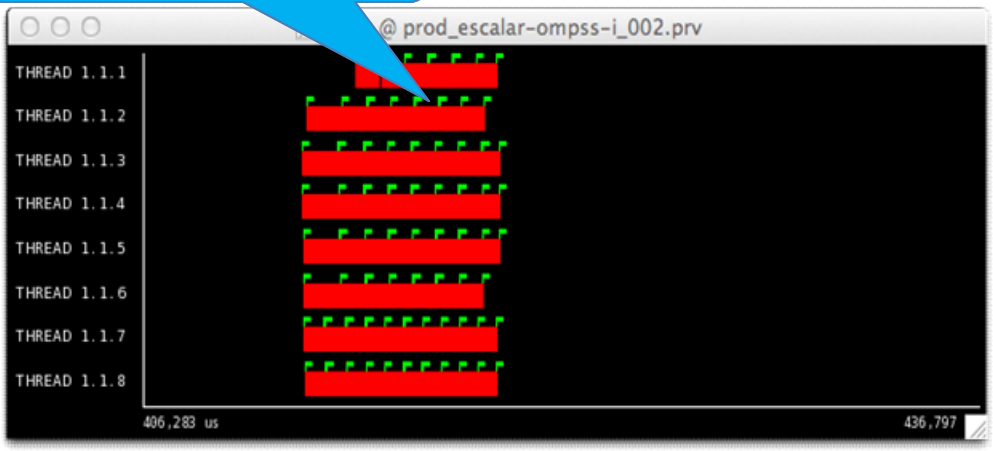


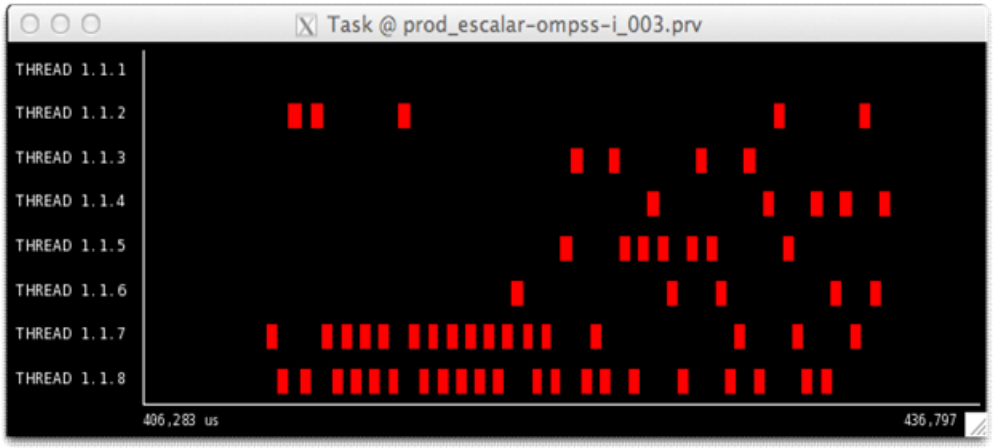

```
for (int j; j<N; j+=BS){

    actual_size = (N- j> BS ? BS: N-j);

    #pragma omp task in (vec[j;actual_size]) commutative(result)
    for (int count = 0; count < actual_size; count ++, j++)
       result += vec [j] ;
}
#pragma omp task input (result)
  printf ("TOTAL is %d\n", result);
#pragma omp taskwait
```

No mutual exclusion required

Differences between concurrent and commutative

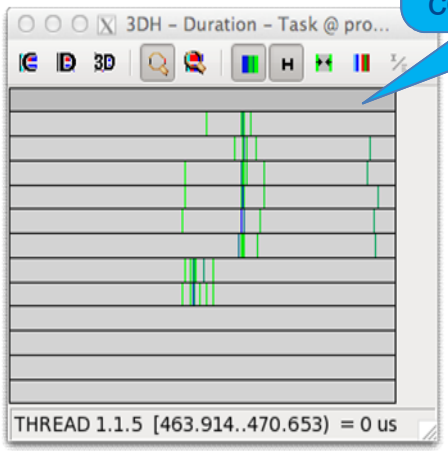
Tasks timeline: views at same time scale

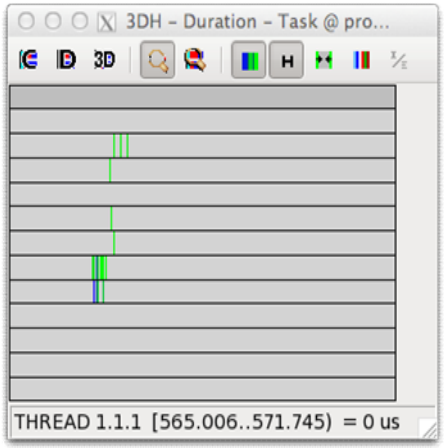




In this case, concurrent is more efficient

Histogram of tasks duration: at same control scale





… but tasks have more duration and variability

Hierarchical task graph

- Nesting
  - As in OpenMP
  - Tasks can generate tasks themselves
  - Everything said so far now applies within the context of a task whose sequential control flow now becomes the main in the local context
- Hierarchical task dependences
  - The generation of children only takes place once father is activated
  - Dependences only checked between siblings
  - Every father has a separate task graph (Hierarchy)
  - There is no implicit taskwait at the end of a task instantiating children
- Impact
  - Allows for each level to contribute to express potential parallelism
  - Effectively parallelizes task creation and dependence handling

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
}
```