www.bsc.es

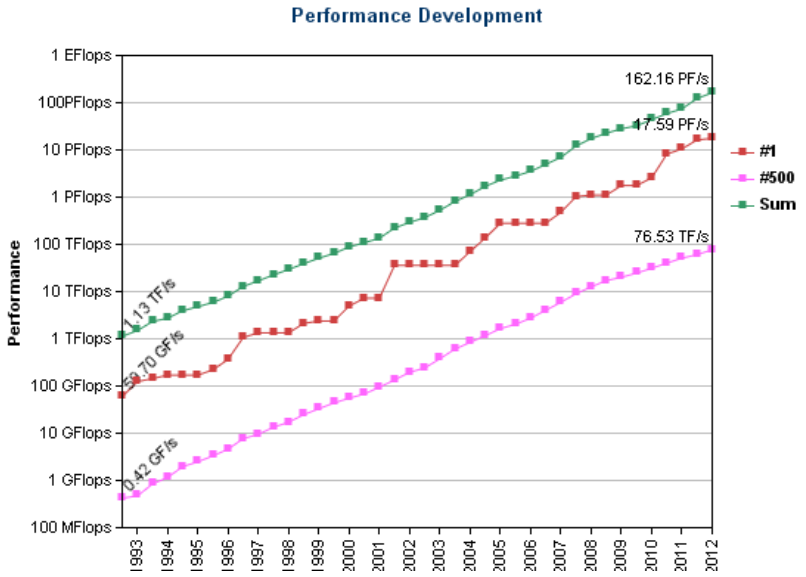
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

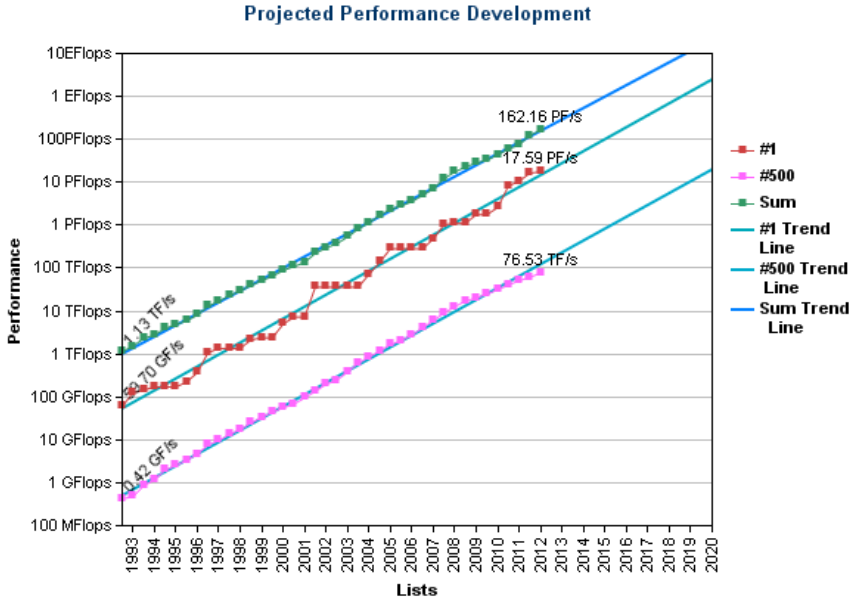
Programming heterogeneous platforms with OmpSs

Rosa M Badia

Lyon, 1 July 2013

Evolution of computers



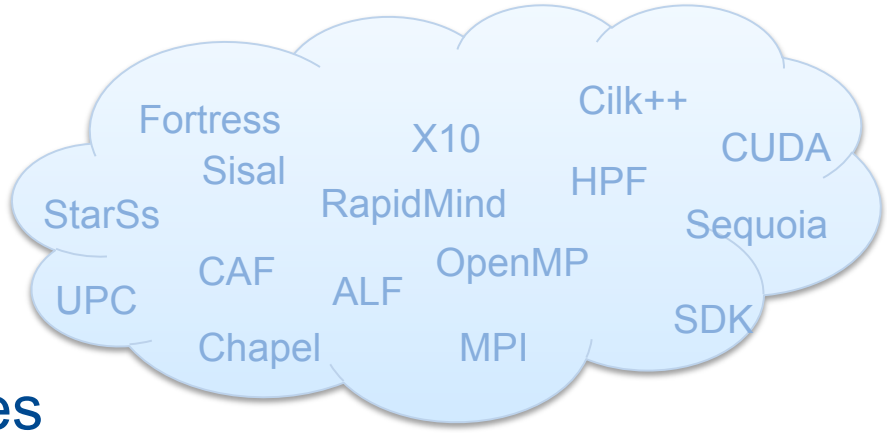


All include multicore or GPU/accelerators

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

Parallel programming models

- Traditional programming models
  - Message passing (MPI)
  - OpenMP
  - Hybrid MPI/OpenMP
- Heterogeneity
  - CUDA
  - OpenCL
  - ALF
  - RapidMind



- New approaches
  - Partitioned Global Address Space (PGAS) programming models
    - UPC, X10, C
- ...

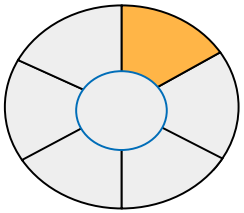
Simple programming paradigms that enable easy application development are required

Outline

- StarSs overview
- OmpSs syntax
- OmpSs examples
- OmpSs + heterogeneity
- OmpSs compiler & runtime
- Further examples
- MPI/OmpSs
- Conclusions

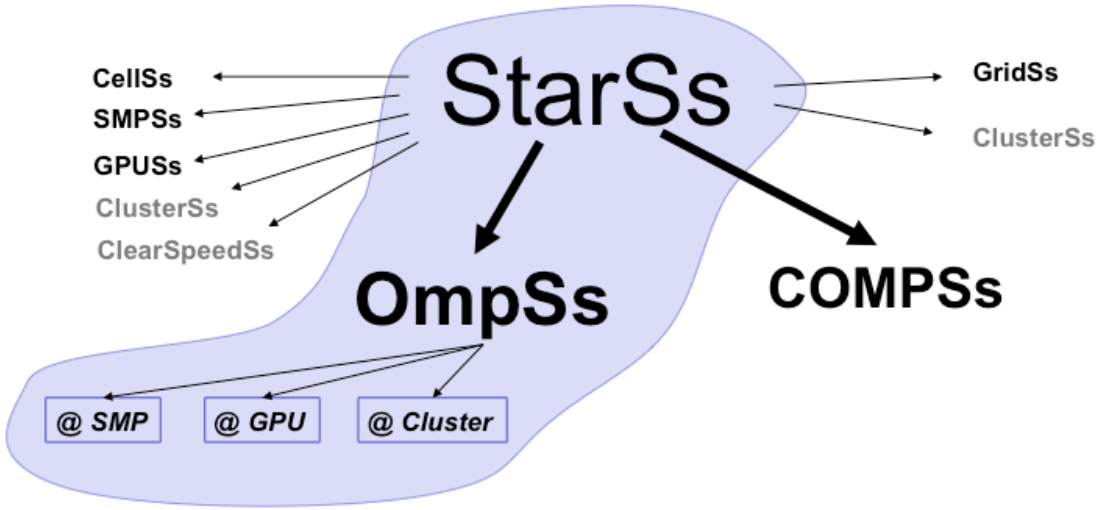
- Contact: pm-tools@bsc.es
- Source code available from http://pm.bsc.es/ompss/

StarSs overview



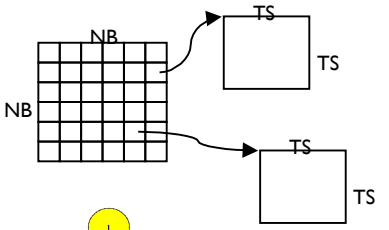
StarSs principles

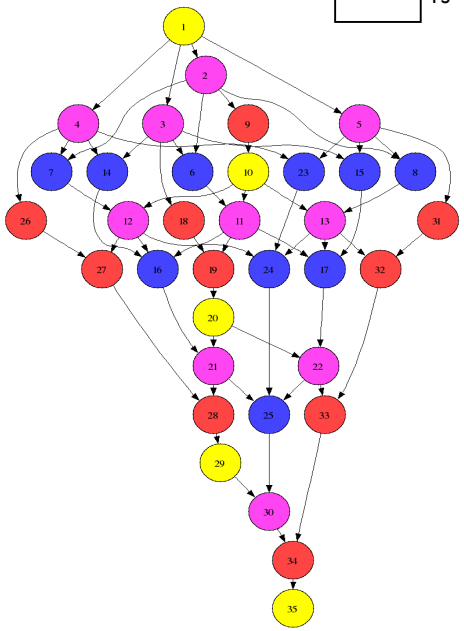
- StarSs: a family of **task based** programming models
  - Basic concept: **write sequential on a flat single address space + directionality annotations**
    - Dependence and data access information in a single mechanism
    - Runtime task-graph dependence generation
    - Intelligent runtime: scheduling, data transfer, support for heterogeneity, support for distributed address space



StarSs: data-flow execution of sequential programs

Decouple
how we write
form
how it is executed

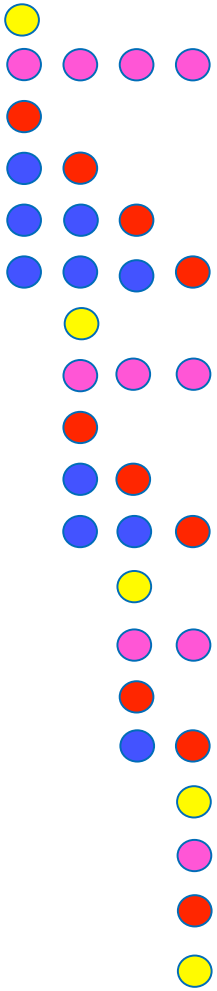




```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j],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

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

StarSs vs OpenMP

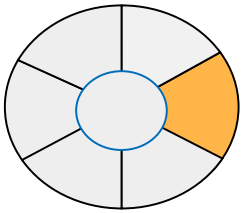


```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]);
      #pragma omp parallel for
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      for (i=k+1; i<NT; i++) {
         #pragma omp parallel for
         for (j=k+1; j<i; j++)
     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

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

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

OmpSs syntax



- OmpSs execution model and memory model
- Inlined pragmas
- Outlined pragmas
- Array sections
- Concurrent
- Commutative
- Nesting
- Sentinels

OmpSs = OpenMP + StarSs extensions

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

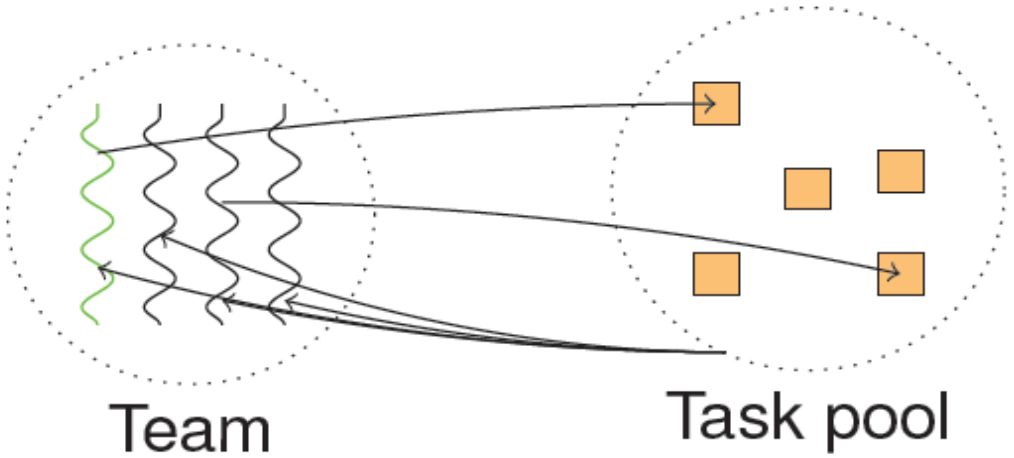
Main Program

- Sequential control flow
  - Defines a single address space
  - Executes sequential code that
    - Can spawn/instantiate tasks than will be executed sometime in the future
    - Can stall/wait for tasks

- Tasks annotated with directionality clauses
  - Input, output, inout
  - Used to build dependences between tasks and for main to wait for data to be produced
  - Can be used for memory management functionalities (replication, locality, movement,…)

Execution Model

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



Memory Model

- From the point of view of the programmer a single naming space exists

- 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
    - Data consistency ensured by the implementation

OmpSs: Directives

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

Provides configuration for CUDA/OpenCL kernel

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

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

Support for multiple implementations of a task

To compute dependences

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

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

 { function or code block }
```

To relax dependence order allowing concurrent execution of tasks

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

To set priorities to tasks

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

Wait for sons or specific data availability

Relax consistency to main program

OpenMP: Directives

OpenMP dependence specification

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

{ function or code block }

Direct contribution of BSC at OpenMP promoting dependences and heterogeneity clauses

Main element: tasks

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

Inlined and outlined tasks

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

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

Main element: inlined tasks

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

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

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

       ...
}
```

for

Main element: inlined tasks

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

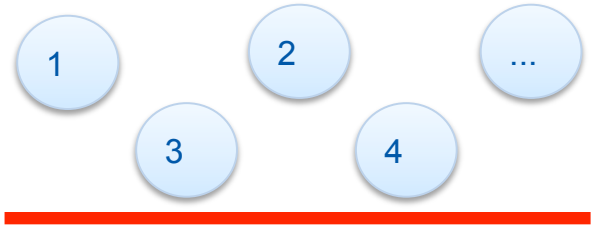
```
int    main    (     )
{
       int X[100];

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

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

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

Synchronization

#pragma omp taskwait

- Suspends the current task until all children tasks are completed



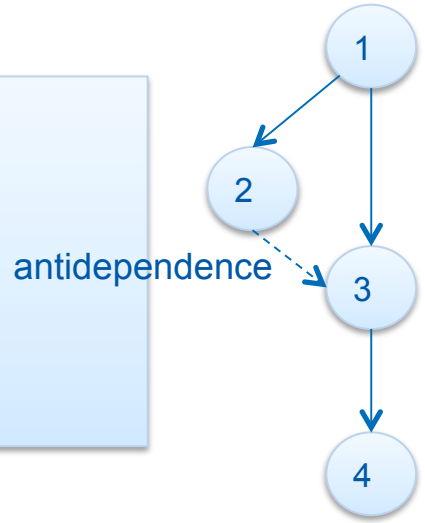

```
void traverse_list ( List l )
{
   Element e ;
   for ( e = l-> first; e ; e = e->next )
      #pragma omp task
       process ( e ) ;

   #pragma omp taskwait
}
```

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

Defining dependences

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

```
#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
```

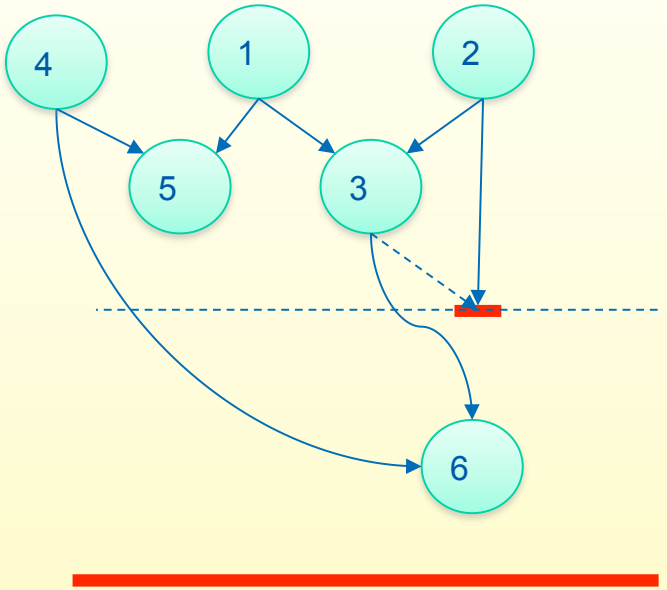


Partial control flow synchronization

#pragma taskwait on (expression)

- Expressions allowed are the same as for the directionality clauses
- Stalls the encountering control flow until the 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) inout(C)
  dgemm(A,B,C);     //1
  #pragma omp task in(D, E) inout(F)
  dgemm(D,E,F);     //2
  #pragma omp task in(C, F) inout(G)
  dgemm(C,F,G);     //3
  #pragma omp task in(A, D) inout(H)
  dgemm(A,D,H);     //4
  #pragma omp task in(C, H) inout(I)
  dgemm(C,H,I);     //5

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

  #pragma omp task in(G, H) inout(I)
  dgemm(H,G,J);     //6

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



Main element: outlined tasks

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

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

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

int main()
{
   int X[100];

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

Main element: outlined tasks

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

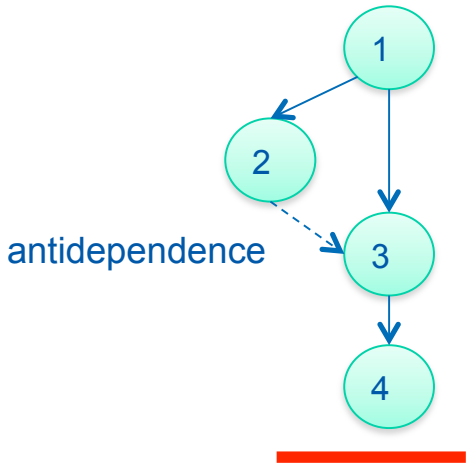
```
#pragma  omp task
void   foo (int Y[size], int size) {
int j;

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

int main()
{
   int X[100];

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

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(*px)
void do_print (int *px) {
   printf("from do_print %d\n" , *px );
}
```

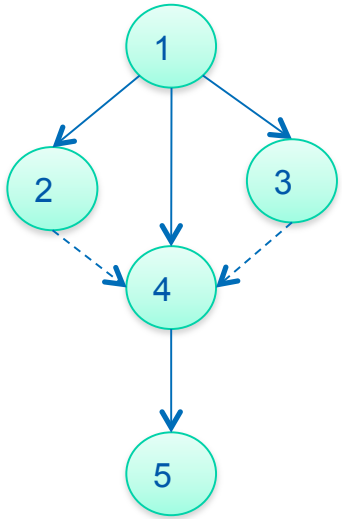
```
set(&x,5);       //1
do_print(&x);    //2
incr(&x);        //3
do_print(&x);    //4
#pragma omp taskwait
```



Mixing inlined and outlined tasks

non-taskified: executed sequentially

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

int main()
{
int x;

   x=3;

   #pragma omp task 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
}
```

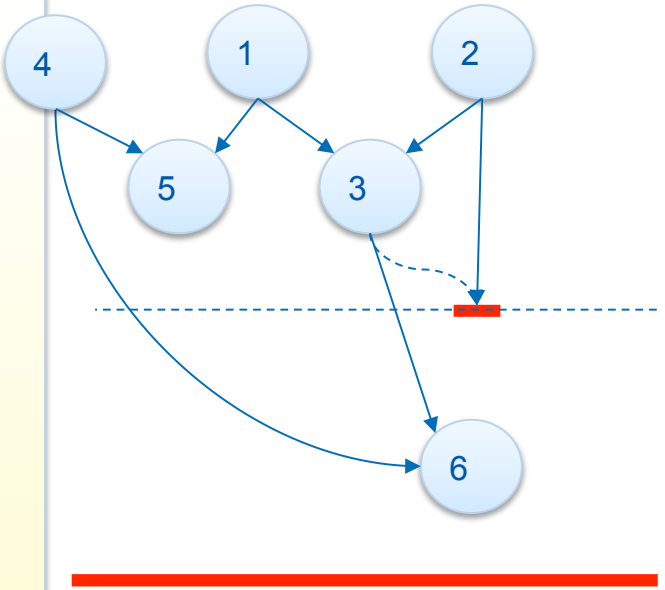


Partial control flow synchronization

#pragma taskwait on (expression)

- 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,G,C); //6


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



Task directive: array regions

- Indicating as input/output/inout subregions of a larger structure:
  - **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)
    - → the upper bound can be omitted if size is known (default is N-1, being N the size)
  - **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]

Examples dependency clauses, 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([N]a)
//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[0;4])
//first 4 elements of the array used to compute dependences
```

Examples dependency clauses, array sections (multidimensions)

```
int a[N][M];
#pragma omp task 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]
```

=

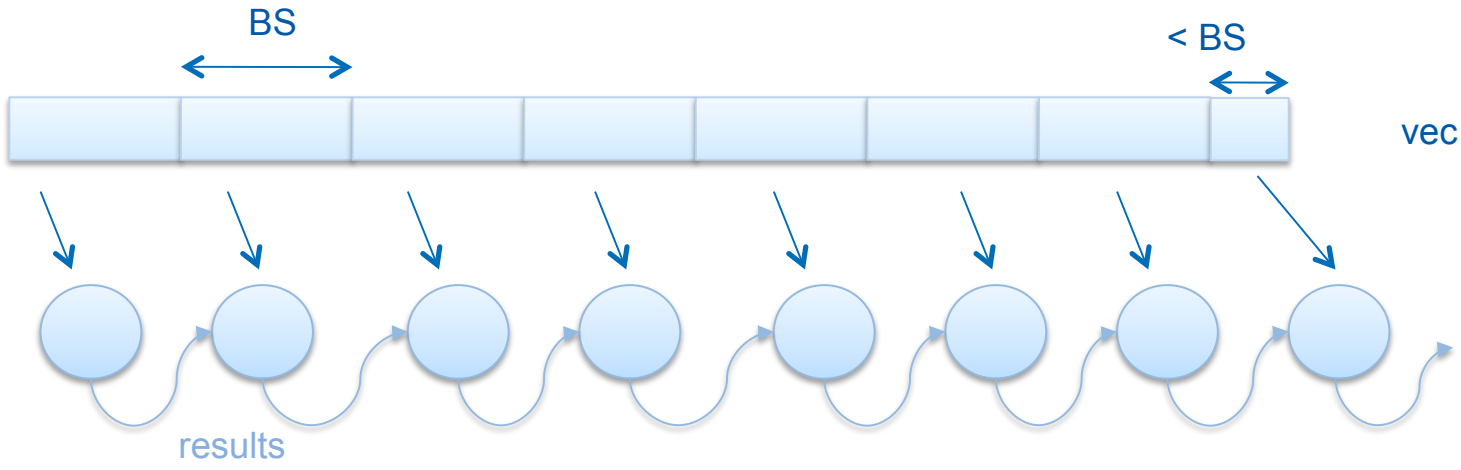
```
int a[N][M];
#pragma omp task in(a[2;2][3;2])
// 2 x 2 subblock of a at a[2][3]
```

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

=

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

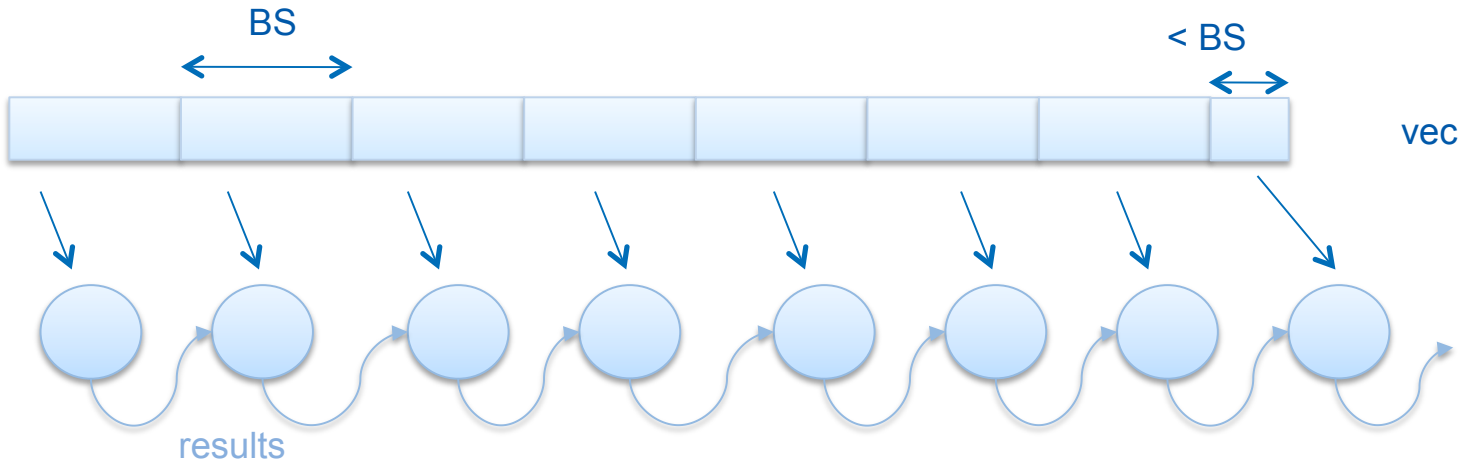
Examples dependency clauses, array sections




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

void main(){
int actual_size;
   for (int j=0; j<N; j+=BS){
       actual_size = (N- j> BS ? BS: N-j);
       sum_task (&vec[j], actual_size, &total);
   }
}
```

dynamic size of argument

Examples dependency clauses, array sections




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

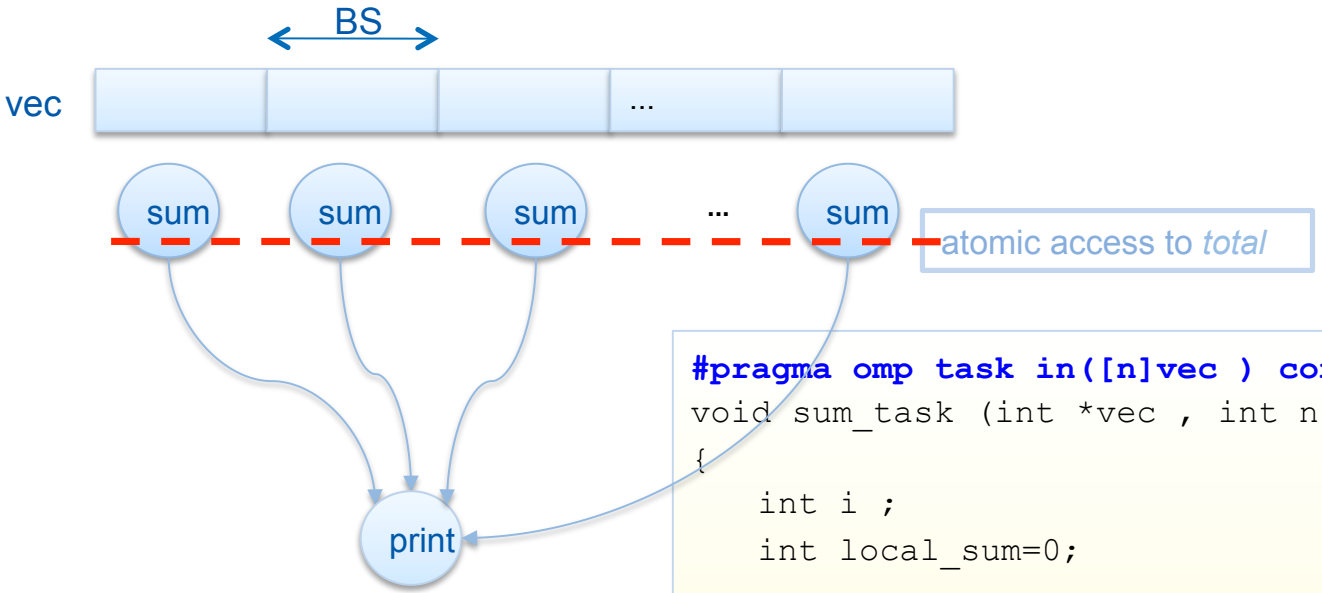
dynamic size of argument

Concurrent

```
#pragma omp task in ( ...) out (...) concurrent (var)
```

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

Concurrent




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

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

    #pragma omp atomic
    *results += local_sum;
}

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

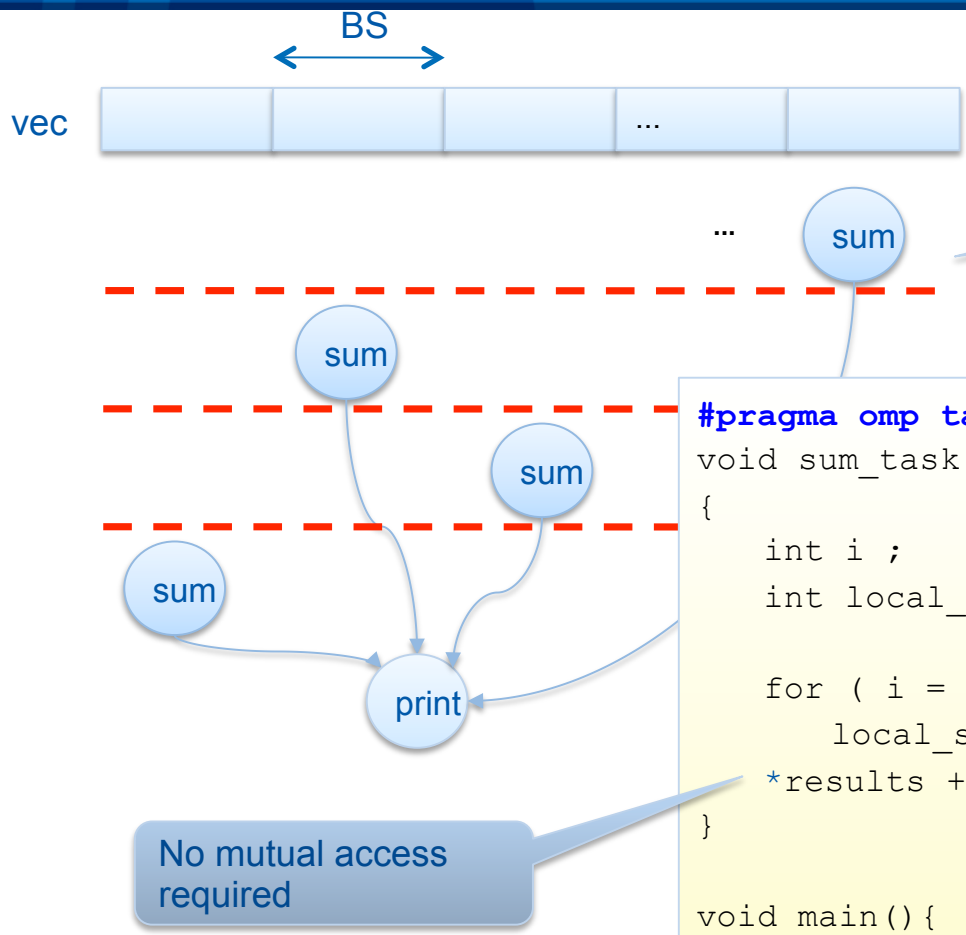
Commutative

```
#pragma omp task in ( ...) out (...) commutative(var)
```

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

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

Commutative



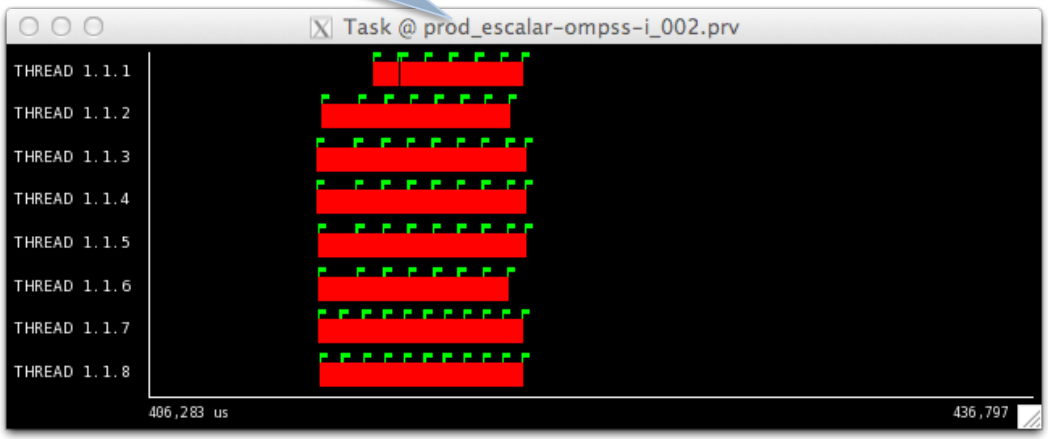
Tasks executed out of order but not concurrently

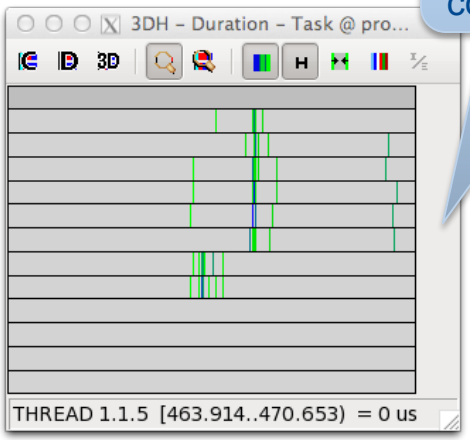
No mutual access required

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

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

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

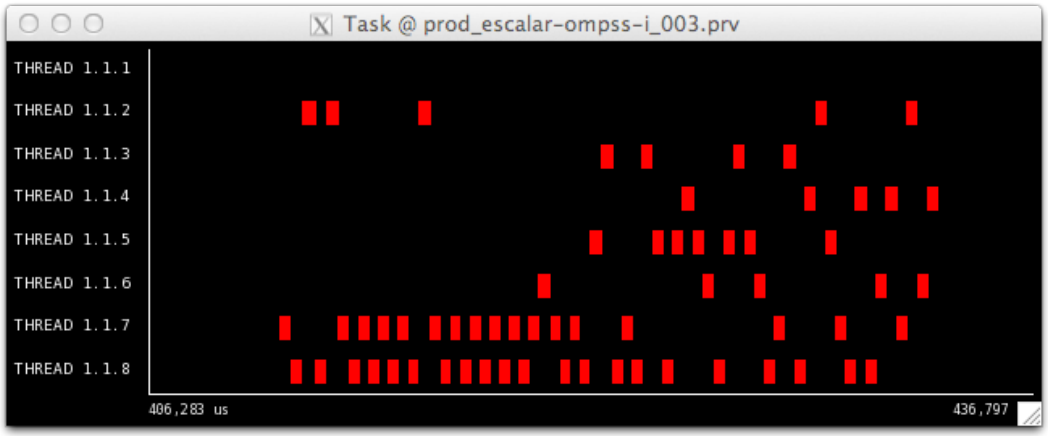
Differences between concurrent and commutative

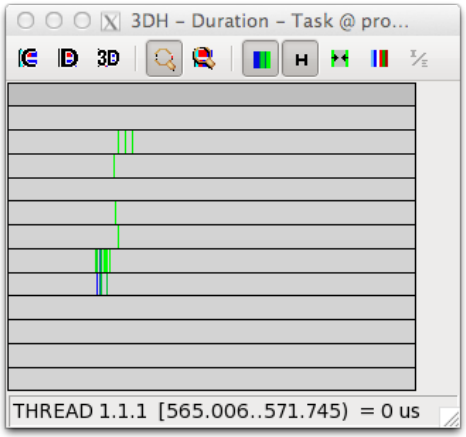
Tasks timeline: views at same time scale





Histogram of tasks duration: at same control scale





In this case, concurrent is more efficient

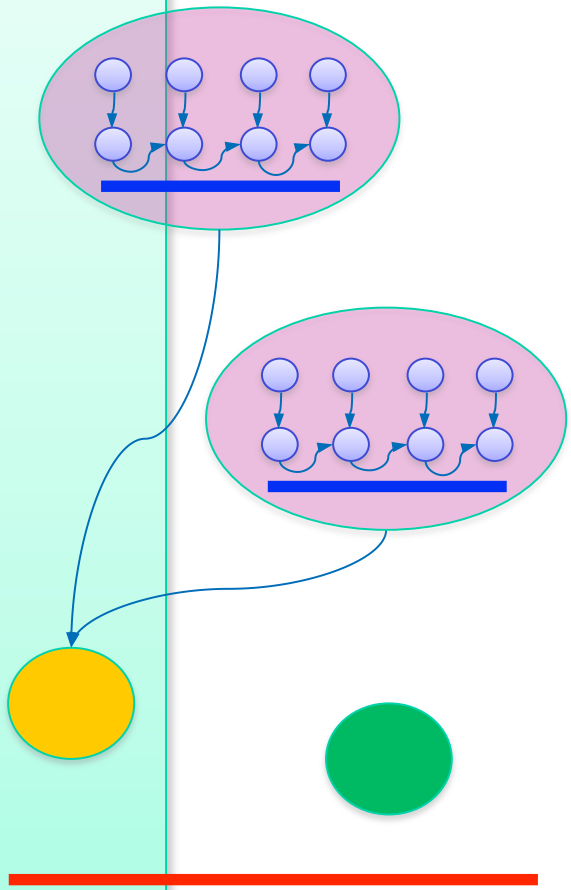
… but tasks have more duration and variability

Hierarchical task graph

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

Nesting inlined tasks

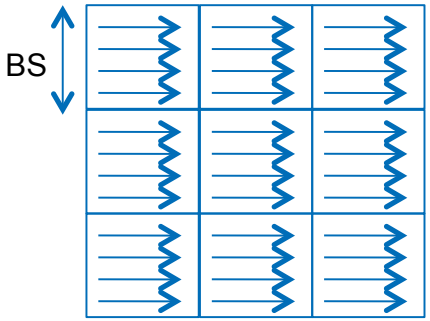
```
int Y[4]={1,2,3,4}

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) inout(Y[3])
   for (int i=1; i<N; i++) Y[3]=h(Y[3]);

   #pragma omp taskwait
   }
```

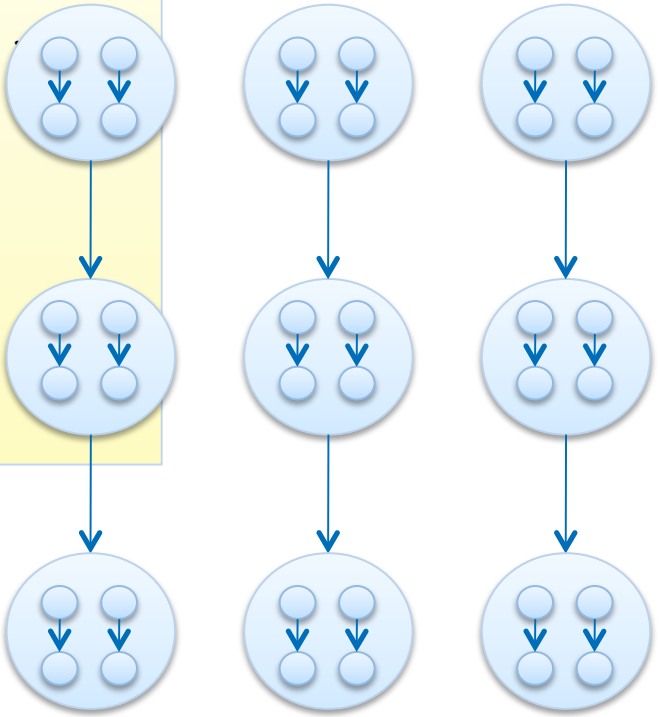


Nesting outlined tasks

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

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

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

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

}
```

Block data-layout





Incomplete directionalities specification

- 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, ...)

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

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

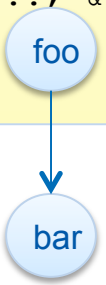
Using entry in C matrix of pointers as representative/sentinel for the whole block it points to.

Will build proper dependences between tasks.

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

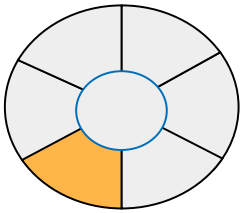
Example sentinels

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

...
}

#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

OmpSs + heterogeneity



Heterogeneity: the target directive

#pragma omp target [clauses]

- Specifies that the code after it is for a specific device (or devices)
- The compiler parses the specific syntax of that device and hands the code over to the appropriate back end compiler
- Currently supported devices:
  - smp: default device. Back end compiler to generate code can be gcc, icc, xlc,….
  - opencl: OpenCL code will be used from the indicated file, and handed over the runtime system at execution time for compilation and execution
  - cuda: CUDA code is separated to a temporary file and handed over to nvcc for code generation

Heterogeneity: the copy clauses

#pragma omp target [clauses]

- Some devices (opencl, cuda) have their private physical address space.
  - The copy_in, copy_out, an copy_inout clauses have to be used to specify what data has to be maintained consistent between the original address space of the program and the address space of the device.
  - The copy_deps is a shorthand to specify that for each input/output/inout declaration, an equivalent copy_in/out/inout is used.
- Tasks on the original program device (smp) also have to specify copy clauses to ensure consistency for those arguments referenced in some other device.
- The default taskwait semantic is to ensure consistency of all the data in the original program address space.

Heterogeneity: the OpenCL/CUDA information clauses

- ndrange: provides the configuration for the OpenCL/CUDA kernel

ndrange (ndim, {global/grid}_array, {local/block}_array)
ndrange (ndim, {global|grid}_dim1, … {local|block}_dim1, …)

  - 1 to 3 dimensions are valid
  - values can be provided through
  - 1-, 2-, 3-elements arrays (global, local)
  - Two lists of 1, 2, or 3 elements, matching the number of dimensions
  - Values can be function arguments or globally accessible variables

Example OmpSs@OpenCL

OmpSs C code

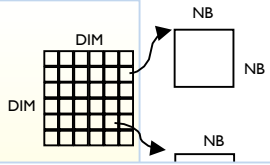
```
#pragma omp task in ([n]x) inout ([n]y)
void saxpy (int n, float a, float *x, float *y)
{
    for (int i=0; i<0; i++)
        y[i] = a * X[i] + y[i];
}

int main (int argc, char *argv[])
{
float a, x[1024], y[1024];
// initializa a, x and y

   saxpy (1024, a, x, y);

#pragma omp taskwait
   printf ("%f", y[0]);
   return 0;
}
```

OmpSs/OpenCL code

```
#pragma omp task in ([n]x) inout ([n]y)
#pragma omp target device (opencl) \
          ndrange (1, n, 128) copy_deps
__kernel void saxpy (int n, float a, __global
    float *x, __global float *y)
{
    int i = get_global_id(0);
    if (i<0)
       y[i] = a * X[i] + y[i];
}

int main (int argc, char *argv[])
{
float a, x[1024], y[1024];
// initializa a, x and y

   saxpy (1024, a, x, y);

#pragma omp taskwait
   printf ("%f", y[0]);
```

OmpSs@OpenCL matmul

```
#define BLOCK_SIZE 16
__constant int BL_SIZE= BLOCK_SIZE;

#pragma omp target device(opencl)
#pragma omp task input([NB*NB]A,[
__kernel void Muld( __global REAL
                    __global REAL
                    __global REAL
```




```
void matmul( int m, int l,
    REAL **tileB,REAL **tile
{
    int i, j, k;
    for(i = 0;i < mDIM; i++)
       for (k = 0; k < lDIM;
           for (j = 0; j < nD
              Muld(tileA[i*lD
}
```

```
#include "matmul_auxiliar_header.h"          // defines BLOCK_SIZE

// Device multiplication function
// Compute C = A * B
//       wA is the width of A
//       wB is the width of B
__kernel void Muld( __global REAL* A,
                    __global REAL* B, int wA, int wB,
                    __global REAL* C, int NB) {
  // Block index, Thread index
  int bx = get_group_id(0); int by = get_group_id(1);
  int tx = get_local_id(0); int ty = get_local_id(1);

  // Indexes of the first/last sub-matrix of A processed by the b
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;

  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;
...
```

OmpSs@CUDA matmul

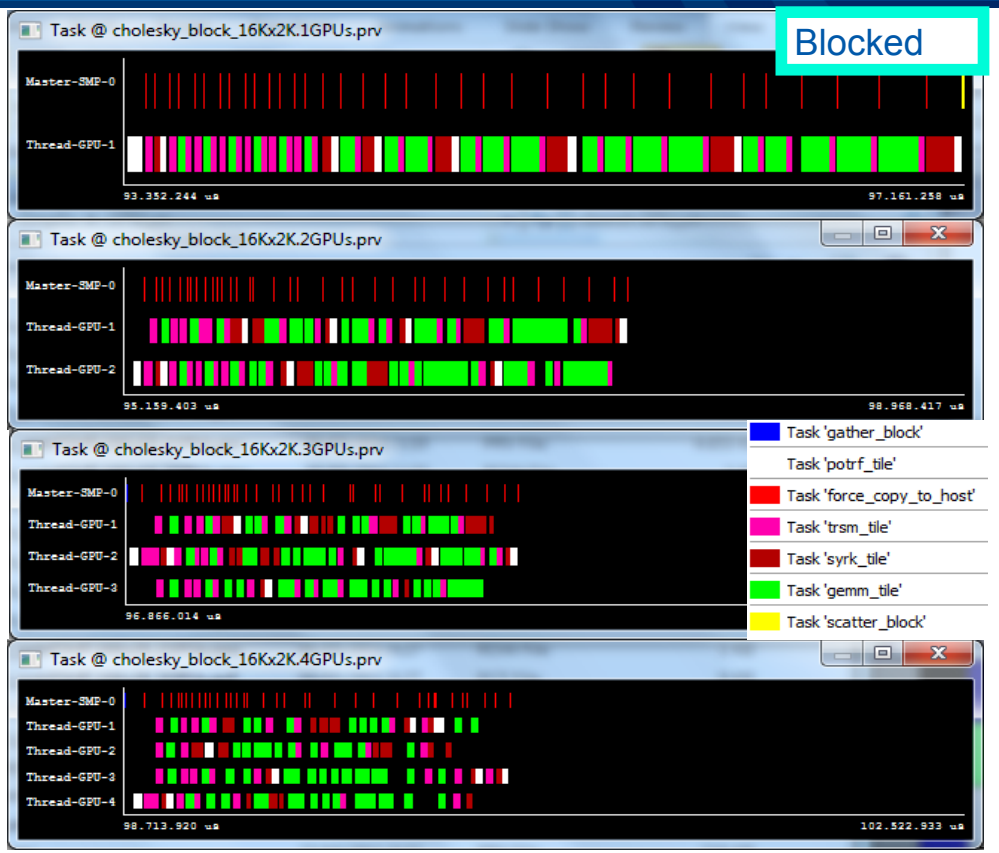
```
#pragma omp target d
#pragma omp task ino
__global__ void Muld
```

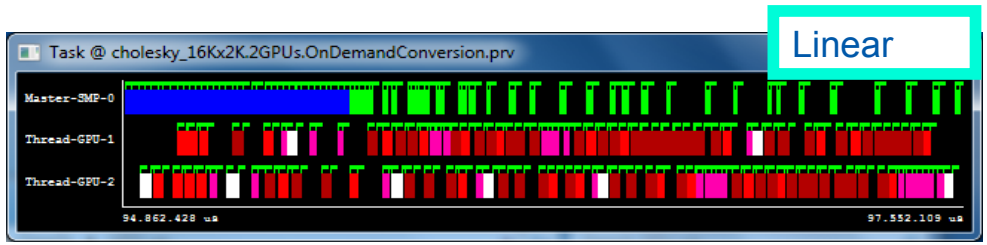
```
void matmul( int m,
    REAL **tileB, REA
{
   int i, j, k;
   for(i = 0;i < mDI
      for (k = 0; k <
         for (j = 0;
            Muld(til
}
```

```
#include "matmul_auxiliar_header.h"

// Thread block size
#define BLOCK_SIZE 16


// Device multiplication function called by Mul()
// Compute C = A * B
//      wA is the width of A
//      wB is the width of B
__global__ void Muld(REAL* A, REAL* B, int wA, int wB, REAL* C, int NB)
    {
  // Block index
  int bx = blockIdx.x; int by = blockIdx.y;
  // Thread index
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Index of the first sub-matrix of A processed by the block
  int aBegin = wA * BLOCK_SIZE * by;
  // Index of the last sub-matrix of A processed by the block
  int aEnd   = aBegin + wA - 1;

  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;
…
```

Runtime execution

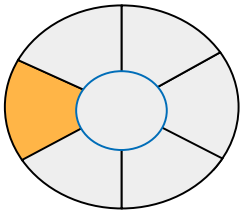
- Example: Cholesky
  - Matrix size: 16K x 16K
  - Block size: 2K x 2K
  - Storage: Blocked / contiguous
- GPU Tasks:
  - spotrf: Magma
  - trsm, syrk, gemm: CUBLAS
- Memory tasks
  - gather/scatter in CPU
- Same code runs in multiple GPUs





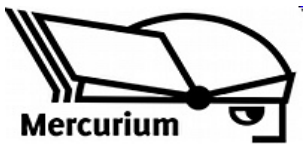
single precision

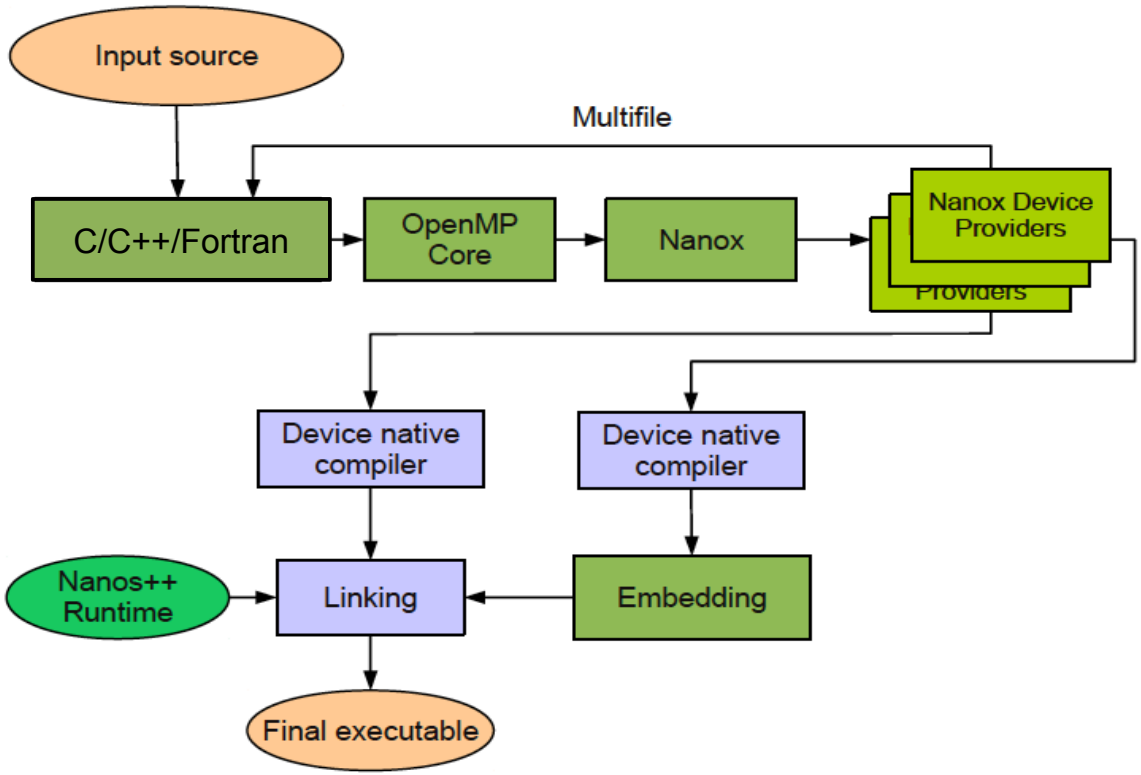
OmpSs compiler and runtime



Mercurium Compiler

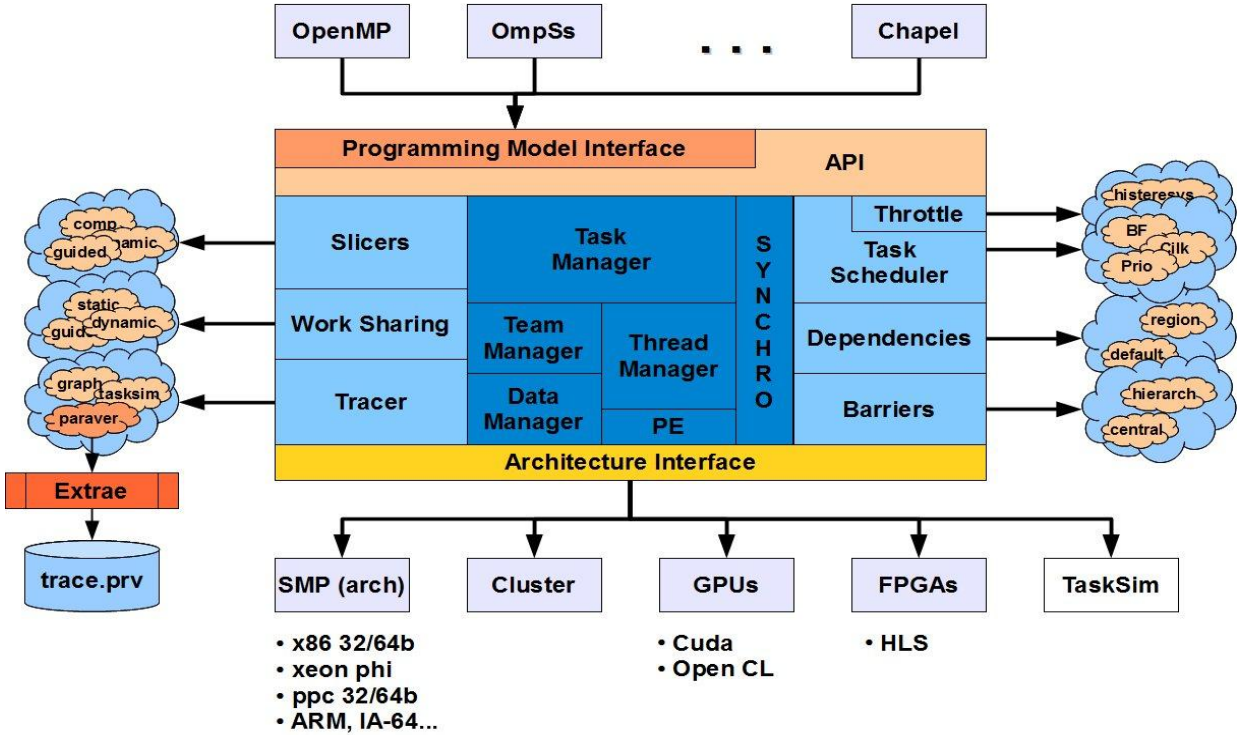
- Recognizes constructs and transforms them to calls to the runtime
- Manages code restructuring for different target devices
  - Device-specific handlers
  - May generate code in a separate file
  - Invokes different back-end compilers
    → nvcc for NVIDIA





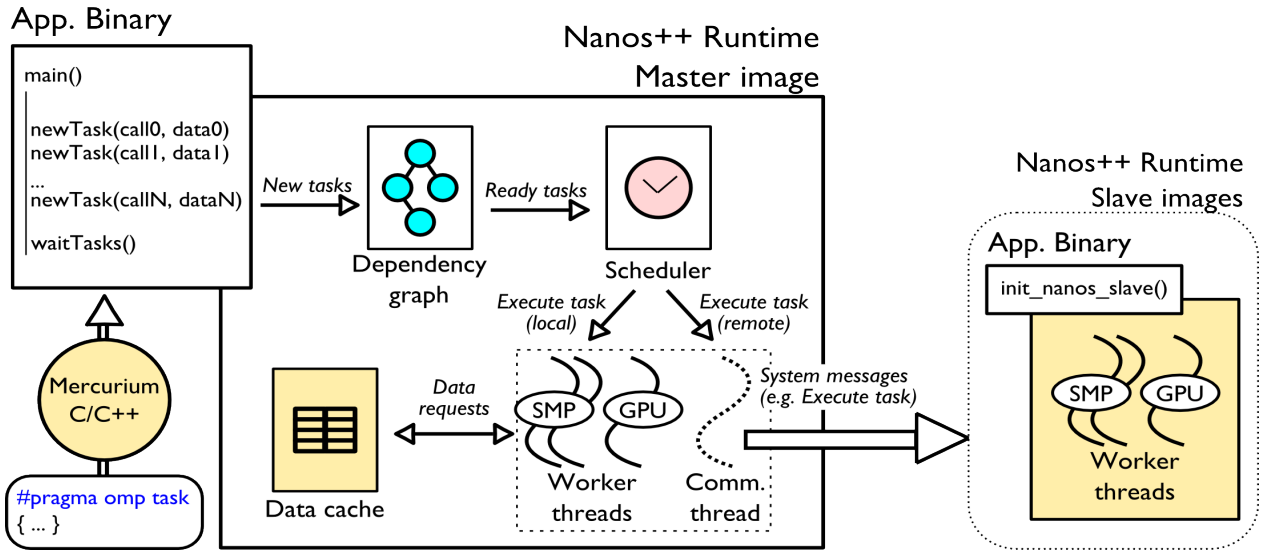
The NANOS++ Runtime

- 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



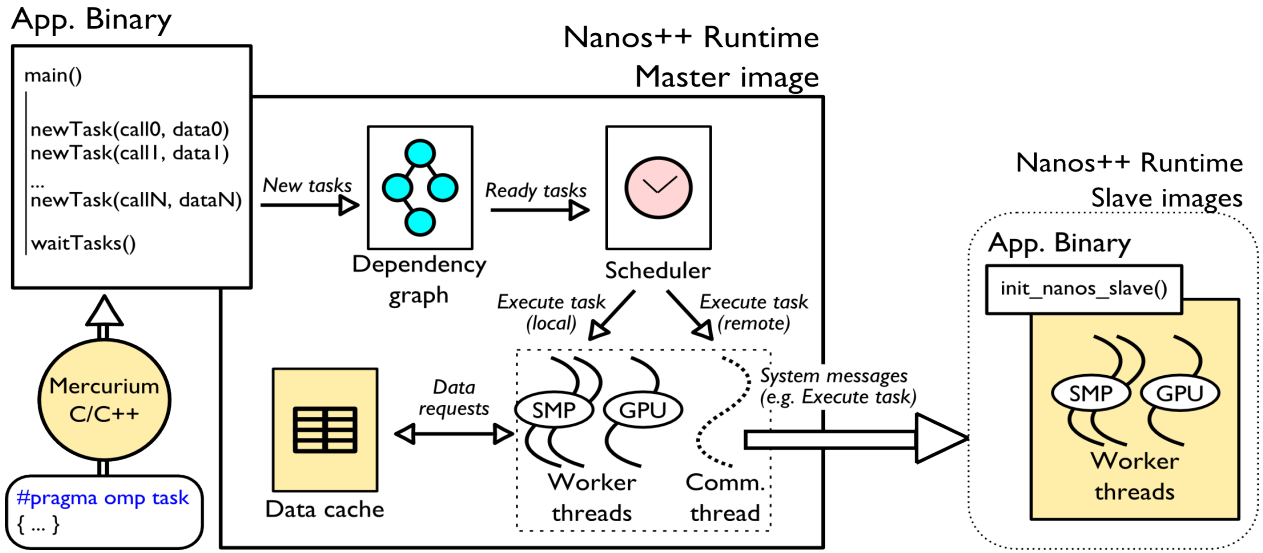
Runtime structure behaviour: task handling

- Task generation
- Data dependence analysis
- Task scheduling



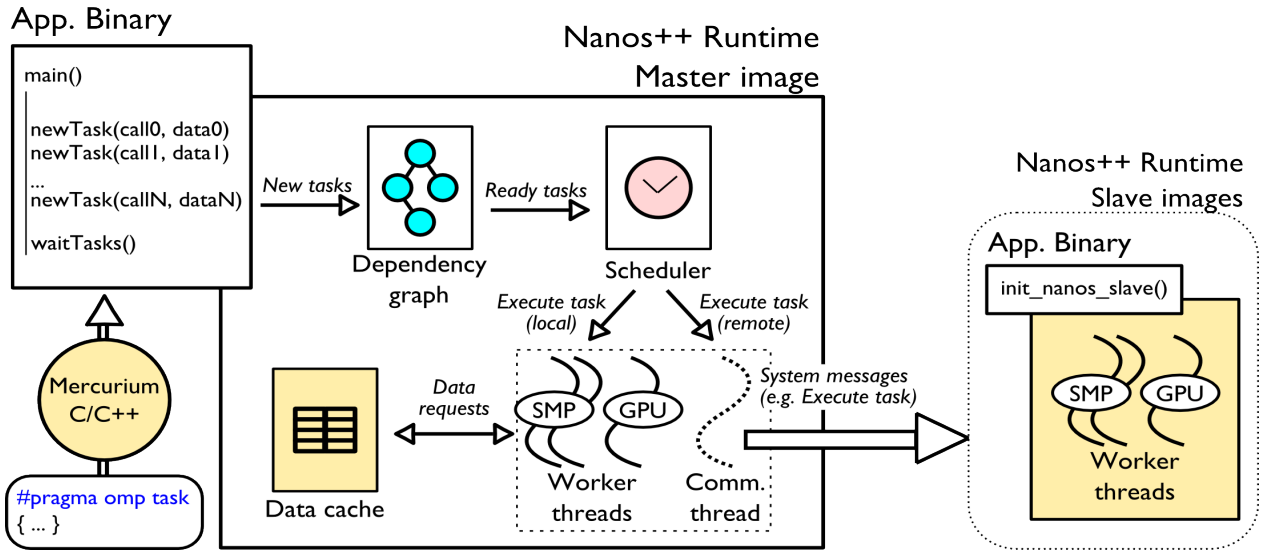
Runtime structure behaviour: coherence support

- Different address spaces managed with:
  - A hierarchical directory
  - A software cache per each:
    - Cluster node
    - GPU
- Data transfers between different memory spaces only when needed
  - Write-through
  - Write-back



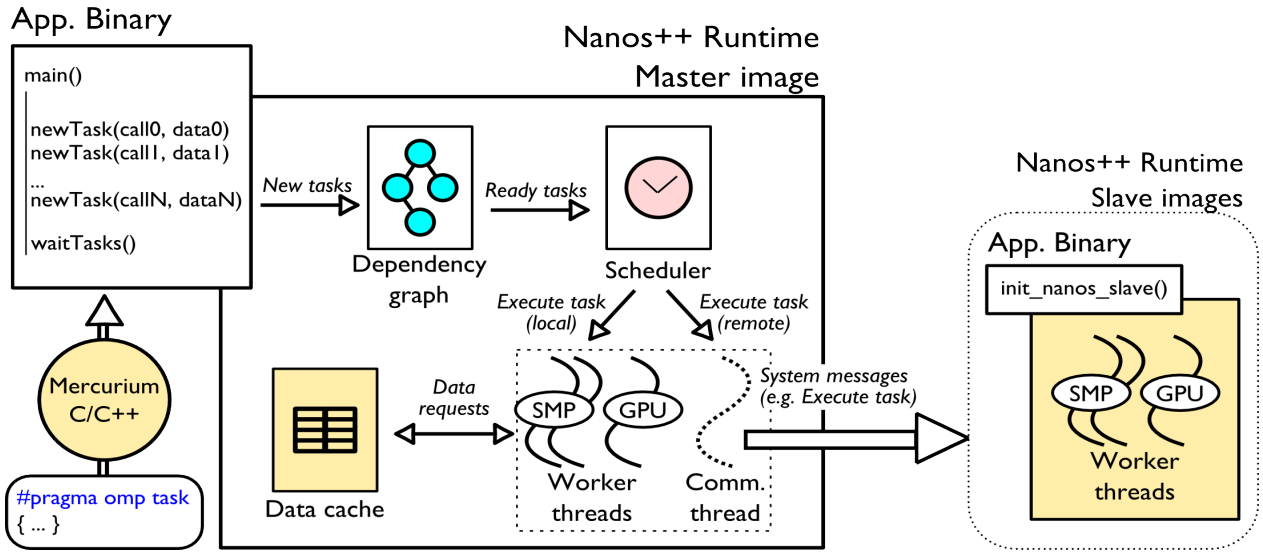
Runtime structure behaviour: GPUs

- Automatic handling of Multi-GPU execution
- Transparent data-management on GPU side (allocation, transfers, ...) and synchronization
- One manager thread in the host per GPU. Responsible for:
  - Transferring data from/to GPUs
  - Executing GPU tasks
  - Synchronization
- Overlap of computation and communication
- Data pre-fetch



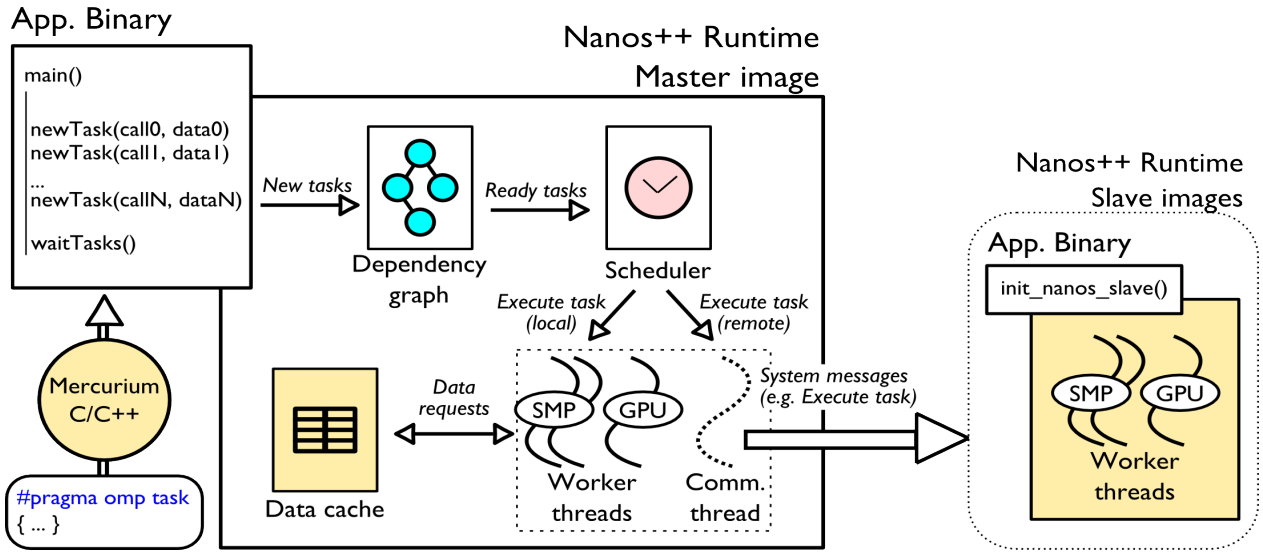
Runtime structure behaviour: clusters

- One runtime instance per node
  - One master image
  - N-1 slave images
- Low level communication through active messages
- Tasks generated by master
  - Tasks executed by worker threads in the master
  - Tasks delegated to slave nodes through the communication thread
- Remote task execution:
  - Data transfer (if necessary)
  - Overlap of computation with communication
  - Task execution
    - Local scheduler

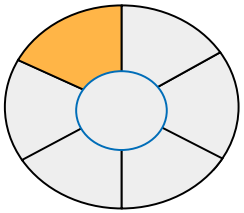


Runtime structure behavior: clusters of GPUs

- Composes previous approaches
- Supports for heterogeneity and hierarchy:
  - Application with homogeneous tasks: SMP or GPU
  - Applications with heterogeneous tasks: SMP and GPU
  - Applications with hierarchical and heterogeneous tasks:
    - I.e., coarser grain SMP tasks
    - Internally generating GPU tasks

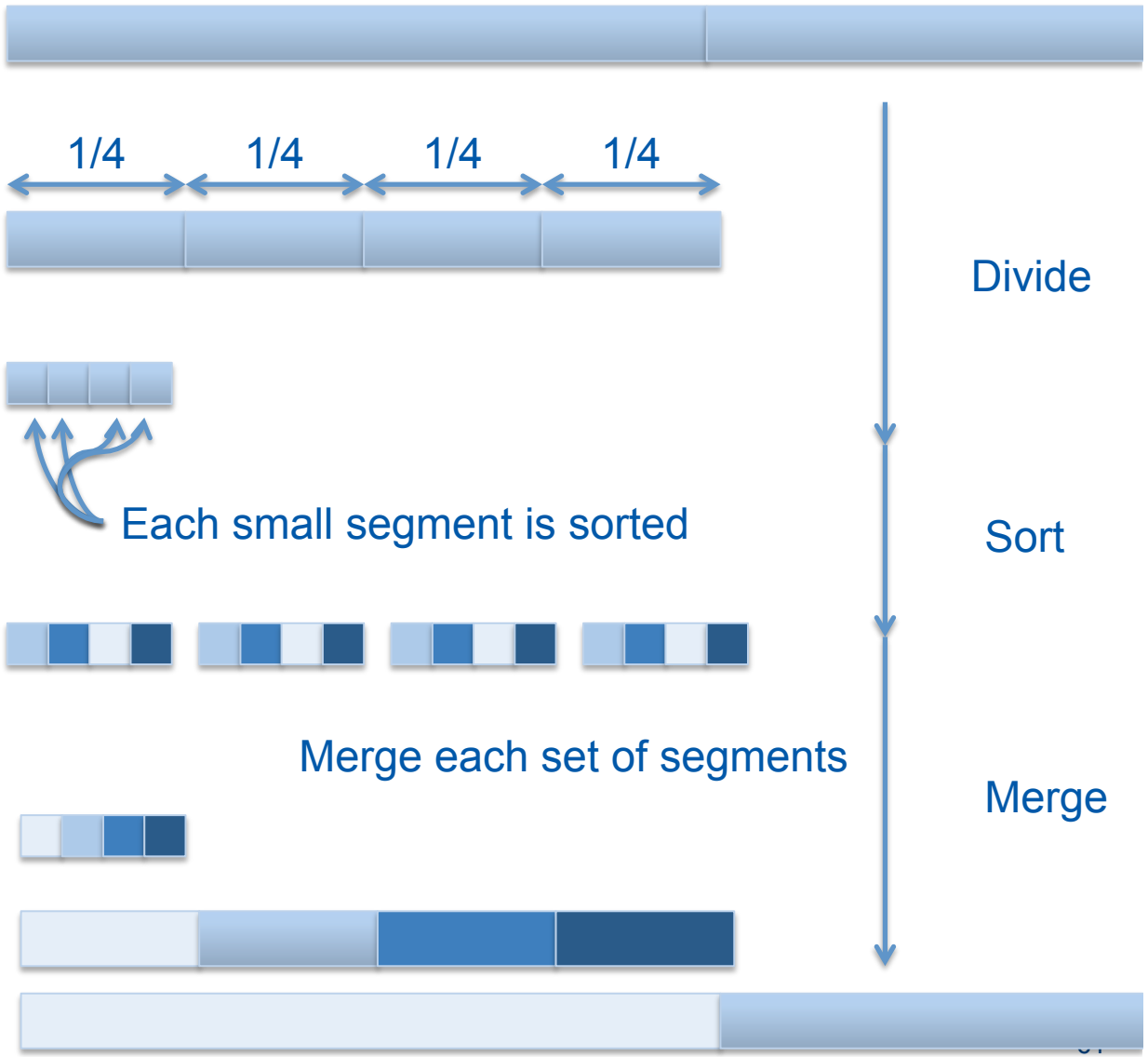


Further examples



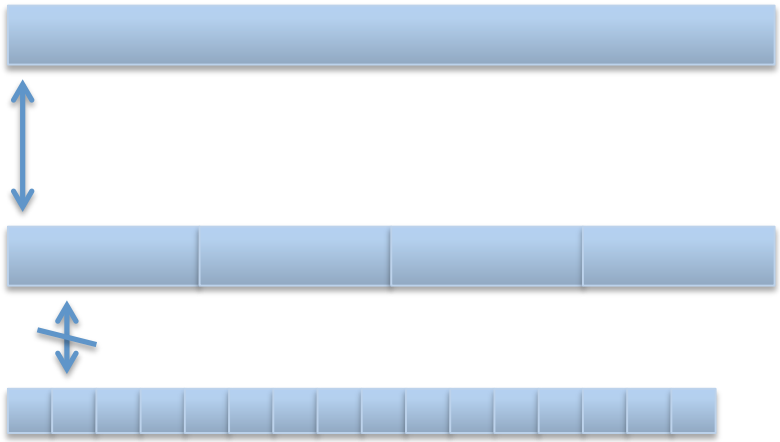
Accessing non-contiguous or partially overlapped regions

- Sorting arrays
  - Divide by ¼
  - Sort
  - Merge



Accessing non-contiguous or partially overlapped regions

- Why is the regions-aware dependences plug-in needed?
  - Regular dependence checking uses first element as representative (size is not considered)
  - Segment starting at address A[i] with length L/4 will be considered the same as A[i] with length L
  - Dependences between A[i] with lenght L and A[i+L/4] with length L/4 will not be detected
- All these is fixed with the regions plugin
- Two different implementations:
  - **NX_DEPS=** regions
  - **NX_DEPS=** perfect-regions



Accessing non-contiguous or partially overlapped regions

```
void multisort(long n, T data[n], T tmp[n]) {
   if (n >= MIN_SORT_SIZE*4L) {
   // Recursive decomposition
   #pragma omp task inout (data[0;n/4L]) firstprivate(n)
   multisort(n/4L, &data[0], &tmp[0]);
   #pragma omp task inout(data[n/4L;n/4L]) firstprivate(n)
   multisort(n/4L, &data[n/4L], &tmp[n/4L]);
   #pragma omp task inout (data[n/2L;n/4L]) firstprivate(n)
   multisort(n/4L, &data[n/2L], &tmp[n/2L]);
   #pragma omp task inout (data[3L*n/4L; n/4L]) firstprivate(n)
   multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

   #pragma omp task input (data[0;n/4L], data[n/4L;n/4L]) output (tmp[0; n/2L])\
   firstprivate(n)
   merge_rec(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
   #pragma omp task input (data[n/2L;n/4L], data[3L*n/4L; n/4L])\
   output (tmp[n/2L; n/2L]) firstprivate (n)
   merge_rec(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

   #pragma omp task input (tmp[0; n/2L], tmp[n/2L; n/2L]) output (data[0; n]) \
   firstprivate (n)
   merge_rec(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
   }
   else basicsort(n, data);
}
```

Accessing non-contiguous or partially overlapped regions

```
void merge_rec(long n, T left[n], T right[n], T result[n*2], long start, long le
ngth) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                #pragma omp task in (left[0;n], right[0;n])\
                 out (result[start;length/2]) firstprivate(n, start, length)
                merge_rec(n, left, right, result, start, length/2);
                #pragma omp task in (left[0;n], right[0;n])\
                 out (result[start+length/2;length/2]) firstprivate(n, start, length)
                merge_rec(n, left, right, result, start + length/2, length/2);
        }
}
```

Accessing non-contiguous or partially overlapped regions

```
T *data = malloc(N*sizeof(T));
T *tmp = malloc(N*sizeof(T));

posix_memalign ((void**)&data, N*sizeof(T), N*sizeof(T));
posix_memalign ((void**)&tmp, N*sizeof(T), N*sizeof(T));
. . .
multisort(N, data, tmp);
#pragma omp taskwait
```

Current implementation requires alignment of data for efficient data-dependence management

Using task versions

```
#pragma omp target device (smp) copy_deps
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void matmul(double  *A, double *B, double *C, unsigned long NB)
{
  int i, j, k, I;
  double tmp;
  for (i = 0; i < NB; i++) {
    I=i*NB;
    for (j = 0; j < NB; j++) {
      tmp=C[I+j];
      for (k = 0; k < NB; k++)
        tmp+=A[I+k]*B[k*NB+j];
      C[I+j]=tmp;
    }
  }
}
#pragma omp target device (smp) implements (matmul) copy_deps
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void matmul_mkl(double  *A, double *B, double *C, unsigned long NB)
{
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, NB, NB, NB, 1.0,
   (double *)A, NB, (double *)B, NB, 1.0, (double *)C, NB);
}
```

Using task versions

```
void compute(struct timeval *start, struct timeval *stop, unsigned long NB, unsi
gned long DIM, double *A[DIM][DIM], double *B[DIM][DIM], double *C[DIM][DIM])
{
  unsigned i, j, k;

  gettimeofday(start,NULL);

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

  #pragma omp taskwait
  gettimeofday(stop,NULL);
}
```

Using task versions

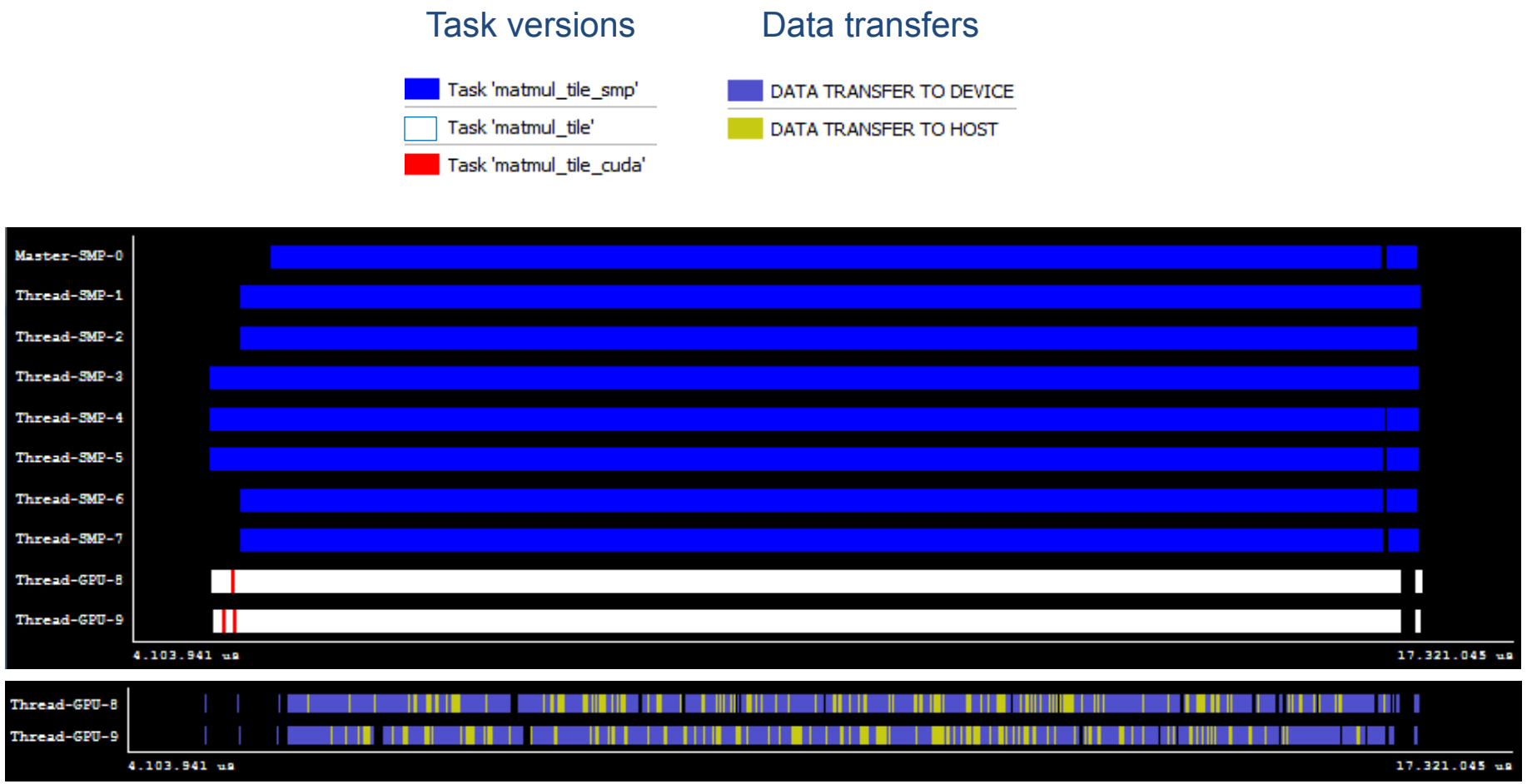
matmul.c

```
void matmul(int N, int TS, double *A, double *B, double *C) {
  for (int i = 0; i < N; j++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        dgemm_kernel(TS, &A[i][k], &B[k][j], &C[i][j]);
  #pragma omp taskwait
}
#pragma omp target device(cuda) ndrange(1, size, 256) copy_deps
#pragma omp task input([ts] A, [ts] B) inout([ts] C)
__global__ void dgemm_kernel(int ts, double *A, double *B, double *C);

#pragma omp target device(smp) implements(dgemm_kernel) copy_deps
#pragma omp task input([ts] A, [ts] B) inout([ts] C)
void cblas_dgemm(int ts, double *A, double *B, double *C) {
  cblas_gemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, ts, ts, ts, 1.0,
        A, ts, B, ts, 1.0, C, ts);
}

#pragma omp target device(cuda) implements(dgemm_kernel) copy_deps
#pragma omp task input([ts] A, [ts] B) inout([ts] C)
void dgemm_cublas(int ts, double *A, double *B, double *C) {
  double alpha = 1.0;
  cublasHandle_t handle = nanos_get_cublas_handle();
  cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, ts, ts, ts, &alpha,
        A, ts, B, ts, &alpha, C, ts);
}
```

Alternative implementation using CBLAS

Alternative implementation using CUBLAS kernel

Sample Execution Trace: Matrix Multiplication



Using task versions

- Use of especific scheduling:
  - export NX_SCHEDULE=versioning
- Tries each version a given number of times and automatically will choose the best version

Using socket aware scheduling

- Assign top level tasks (depth 1) to a NUMA node set by the user before task creation
  - nested tasks will run in the same node as their parent.
- nanos_current_socket API function must be called before instantiation of tasks to set the NUMA node the task will be assigned to.
- Queues sorted by priority with as many queues as NUMA nodes specified (see num-sockets parameter).

Using socket aware scheduling

Example: stream

```
#pragma omp task input ([bs]a, [bs]b) output ([bs]c)
void add_task (double *a, double *b, double *c, int bs)
{
    int j;
    for (j=0; j < BSIZE; j++)
       c[j] = a[j]+b[j];
}

void tuned_STREAM_Add()
{
    int j;
    for (j=0; j<N; j+=BSIZE){
        nanos_current_socket( ( j/((int)BSIZE) ) % 2  );
        add_task(&a[j], &b[j], &c[j], BSIZE);
    }
}
```

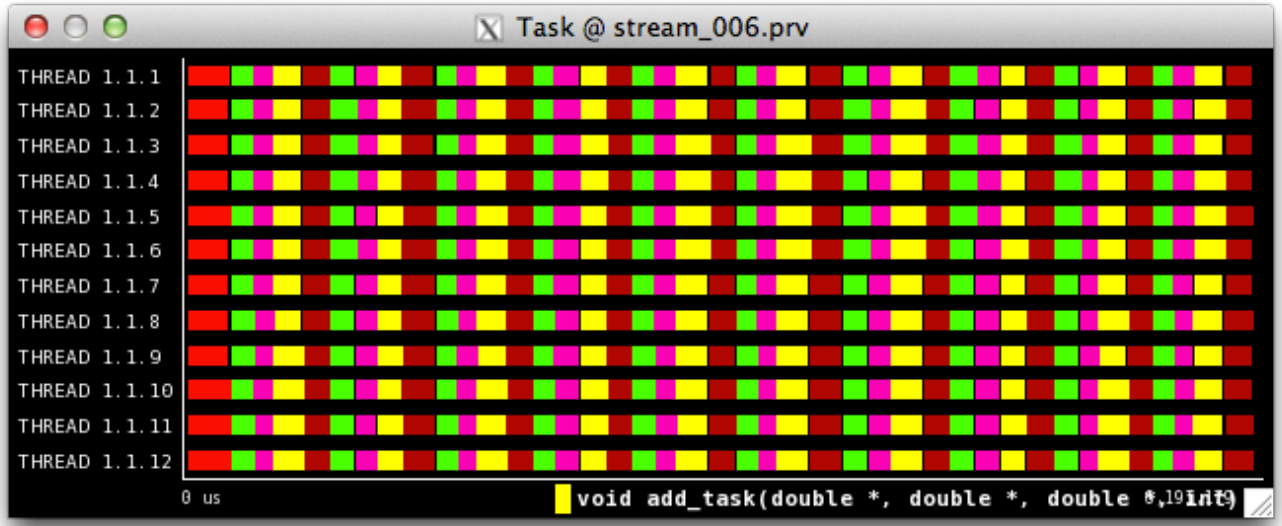
Using socket aware scheduling

- Usage:
  - export NX_SCHEDULE=socket
- If using less than N threads, being N the cores in a socket:
- I.E., for a socket of 6 cores:
  - export NX_ARGS="--binding-stride 6"

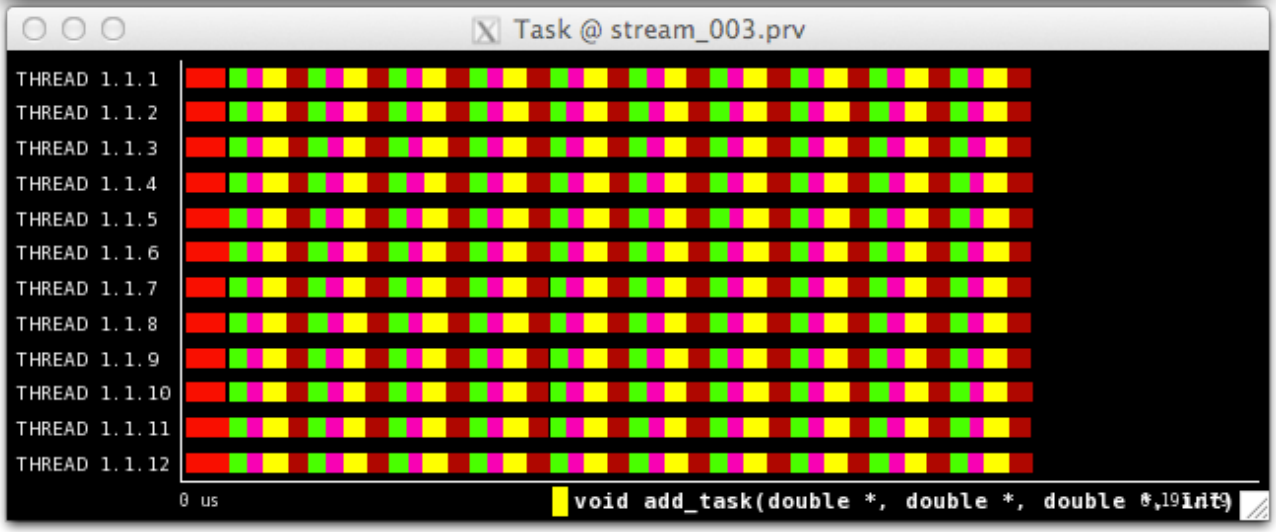
Using socket aware scheduling

Differences between the use of socket aware scheduling in the stream example:

Non Socket-aware



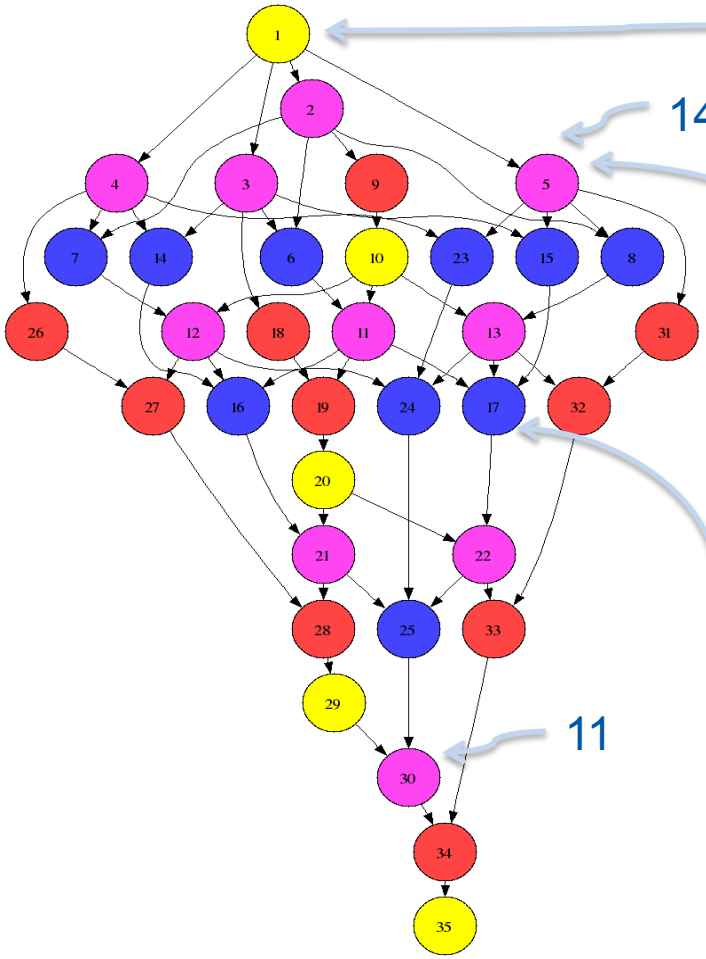
Socket-aware



Giving hints to the compiler: priorities

```
for (k = 0; k < nt; k++) {
        for (i = 0; i < k; i++) {
          #pragma omp task input([ts*ts]Ah[i*nt + k]) inout([ts*ts]Ah[k*nt + k]) \
          priority( (nt-i)+10 ) firstprivate (i, k, nt, ts)
            syrk_tile (Ah[i*nt + k], Ah[k*nt + k], ts, region)
        }
        // Diagonal Block factorization and panel permutations
        #pragma omp task inout([ts*ts]Ah[k*nt + k]) \
         priority( 100000 ) firstprivate (k, ts, nt)
        potr_tile(Ah[k*nt + k], ts, region)

        // update trailing matrix
        for (i = k + 1; i < nt; i++) {
            for (j = 0; j < k; j++) {
                #pragma omp task input ([ts*ts]Ah[j*nt+i], [ts*ts]Ah[j*nt+k]) \
                 inout ( [ts*ts]Ah[k*nt+i]) firstprivate (i, j, k, ts, nt)
                gemm_tile (Ah[j*nt + i], Ah[j*nt + k], Ah[k*nt + i], ts, region)
            }
            #pragma omp task input([ts*ts]Ah[k*nt + k]) inout([ts*ts]Ah[k*nt + i]) \
            priority( (nt-i)+10 ) firstprivate (i, k, ts, nt)
            trsm_tile (Ah[k*nt + k], Ah[k*nt + i], ts, region)
        }
    }
#pragma omp taskwait
```

Giving hints to the compiler: priorities



Potrf: Maximum priority

14

trsm: priority (nt – i) + 10

syrk: priority (nt – i) + 10

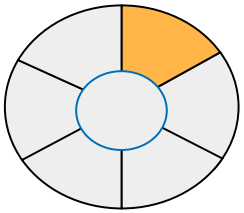
gemm: no priority

11

Giving hints to the compiler: priorities

- Two policies available:
  - Priority scheduler
    - Tasks are scheduled based on the assigned priority.
    - The priority is a number >= 0. Given two tasks with priority A and B, where A > B, the task with priority A will be executed earlier than the one with B
    - When a task T with priority A creates a task Tc that was given priority B by the user, the priority of Tc will be added to that of its parent. Thus, the priority of Tc will be A + B.
  - Smart Priority scheduler
    - Similar to the Priority scheduler, but also propagates the priority to the immediate preceding tasks.
- Using the schedulers:
  - export NX_SCHEDULE = priority
  - export NX_SCHEDULE = smartpriority

MPI/OmpSs



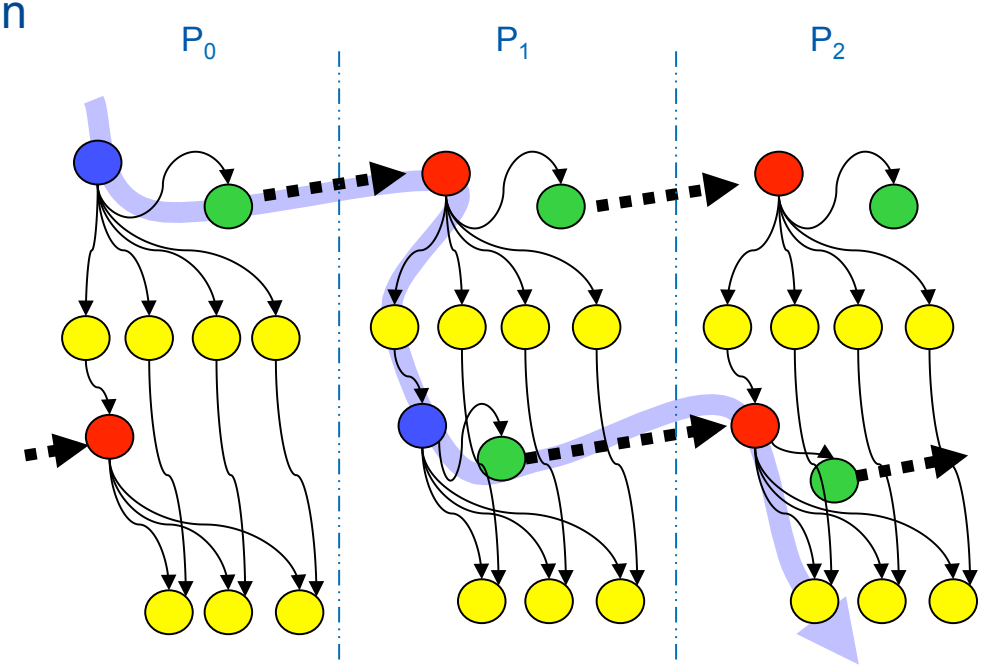
MPI/StarSs hybrid programming

- Why?
  - MPI is here to stay.
  - A lot of HPC applications already written in MPI.
  - MPI scales well to tens/hundreds of thousands of nodes.
  - MPI exploits intra-node parallelism while StarSs can exploit node parallelism.
  - Overlap of communication and computation through the inclusion of communication in the task graph
  - MPI/StarSs Overcomes the too synchronous structure of MPI/OpenMP, propagating the dataflow asynchronous behavior to the MPI level.

Hybrid MPI/StarSs

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

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




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

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


V. Marjanovic, et al, "Overlapping Communication and Computation by using a Hybrid MPI/SMPSs Approach" ICS 2010

Conclusions

- StarSs
  - Asynchronous Task-based programming model
  - Key aspect: data dependence detection which avoid global synchronization
  - Support for heterogeneity increasing portability
- Encompases a complete programming environment
  - StarSs programming model
  - Tareador: finding tasks
  - Paraver: Performance analysis
  - DLB: dynamic load balancing
  - Temanejo: debugger (under development at HLRS)
- Support for MPI
  - Overlap off computation and communication
- Fully open, available at: pm.bsc.es/ompss

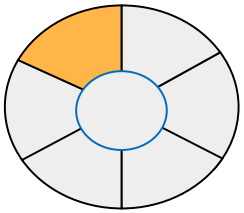
www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
rosa.m.badia@bsc.es

OmpSs environment



Compiling

- Compiling

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

- Linking

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

- where frontend is one of:

mcc	C
mcxx	C++
mnvcc	CUDA & C
mnvcxx	CUDA & C++
mfc	Fortran

Compiling

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

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

Executing

- No LD_LIBRARY_PATH or LD_PRELOAD needed

```
./bin
```

- Adjust number of threads with OMP_NUM_THREADS

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

Nanos++ options

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

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

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

Nanox helper

- Nanos++ utility to
  - list available modules:

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

  - list available options:

```
nanox --help
```

Tracing

- Compile and link with --instrument

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

- When executing specify which instrumentation module to use:

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

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

Reporting problems

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

Programming methodology

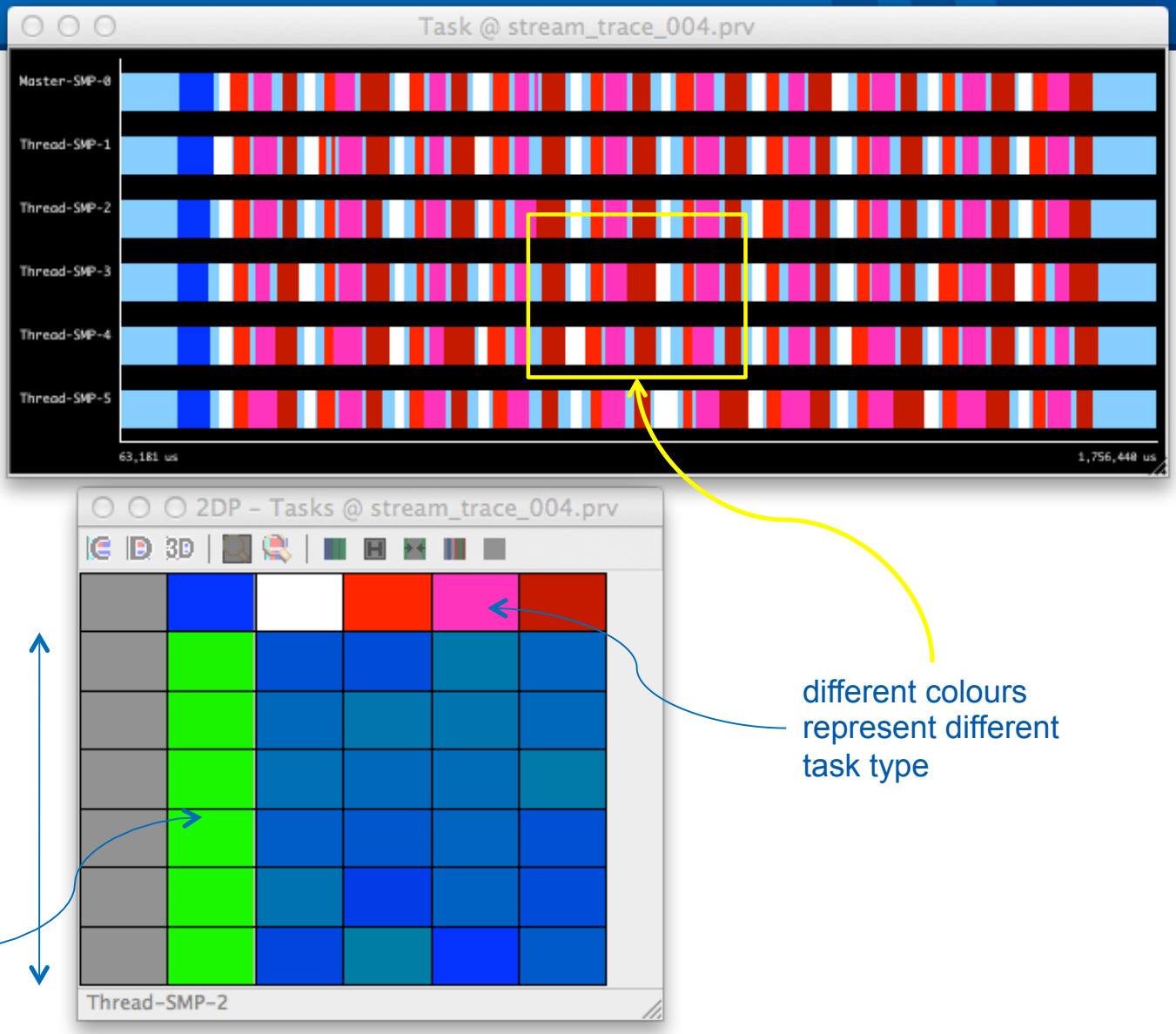
- Correct sequential program
- Incremental taskification
  - Test every individual task with forced sequential in-order execution
    - → 1 thread, scheduler = FIFO, throtle=1
- Single thread out-of-order execution
- Increment number of threads
  - Use taskwaits to force certain levels of serialization

Visualizing Paraver tracefiles

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

Tasks' profile

- 2dp_tasks.cfg
- Tasks' profile



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

light blue: not executing tasks

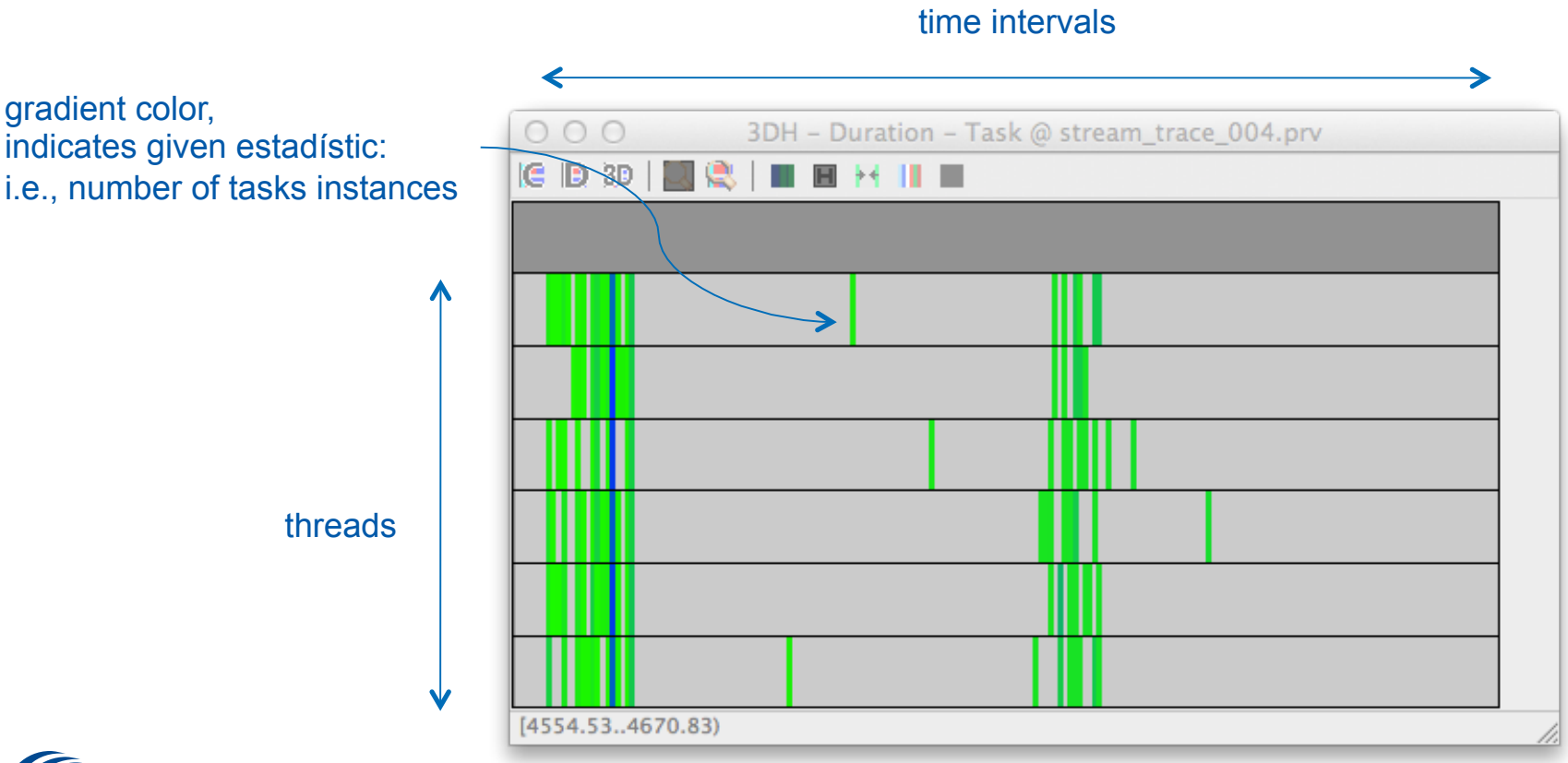
threads

different colours represent different task type

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

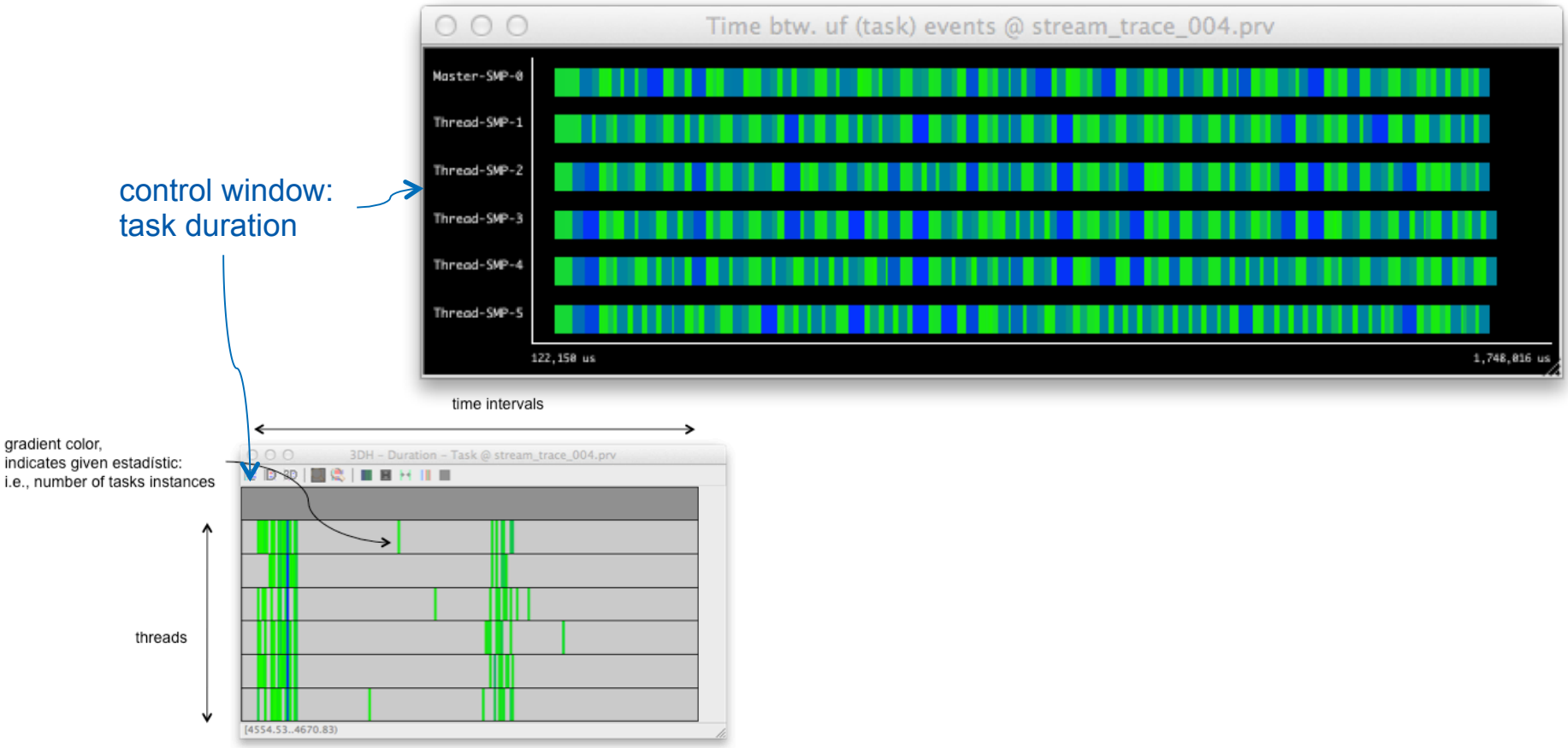
Tasks duration histogram

- 3dh_duration_task.cfg



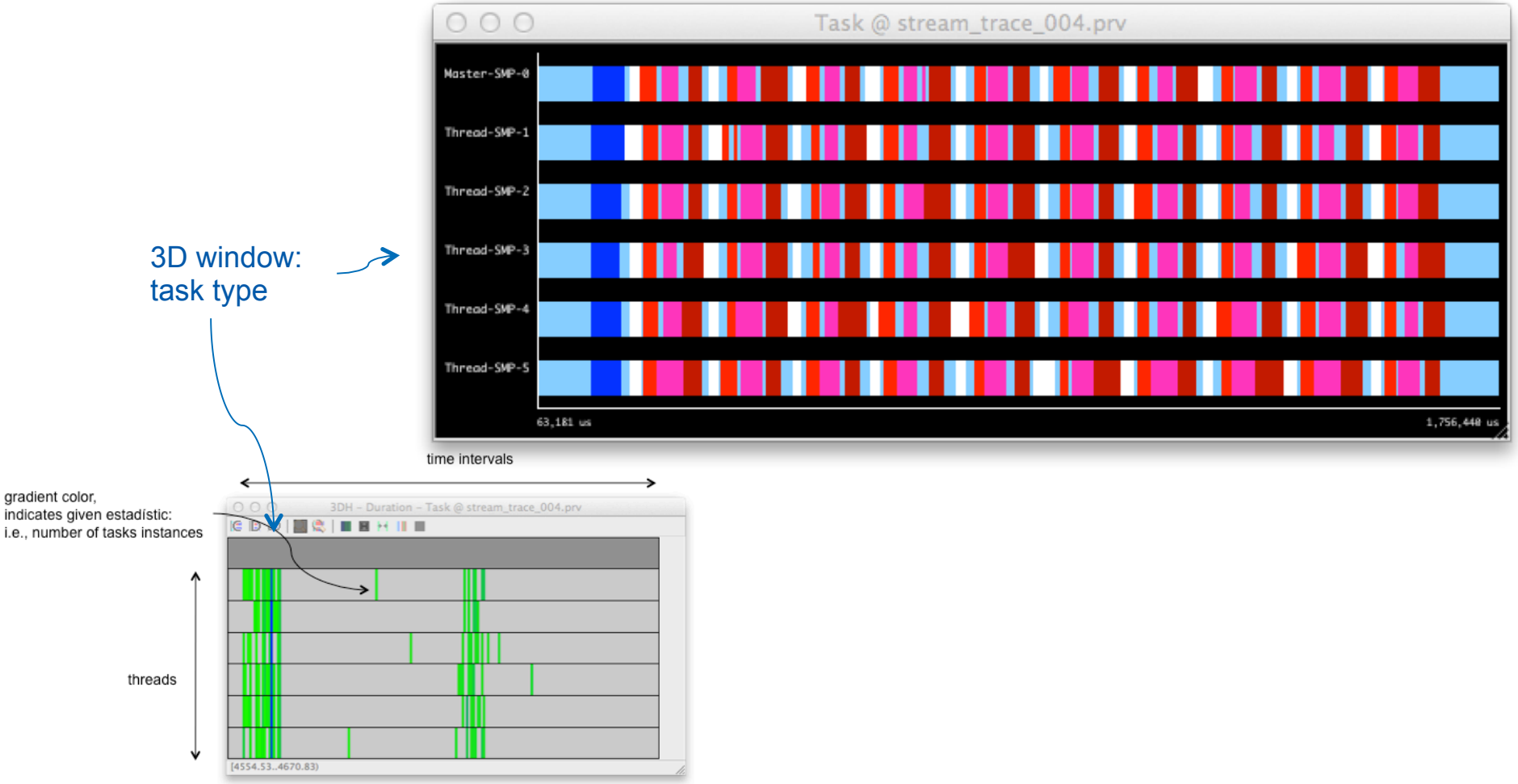
Tasks duration histogram

- 3dh_duration_task.cfg



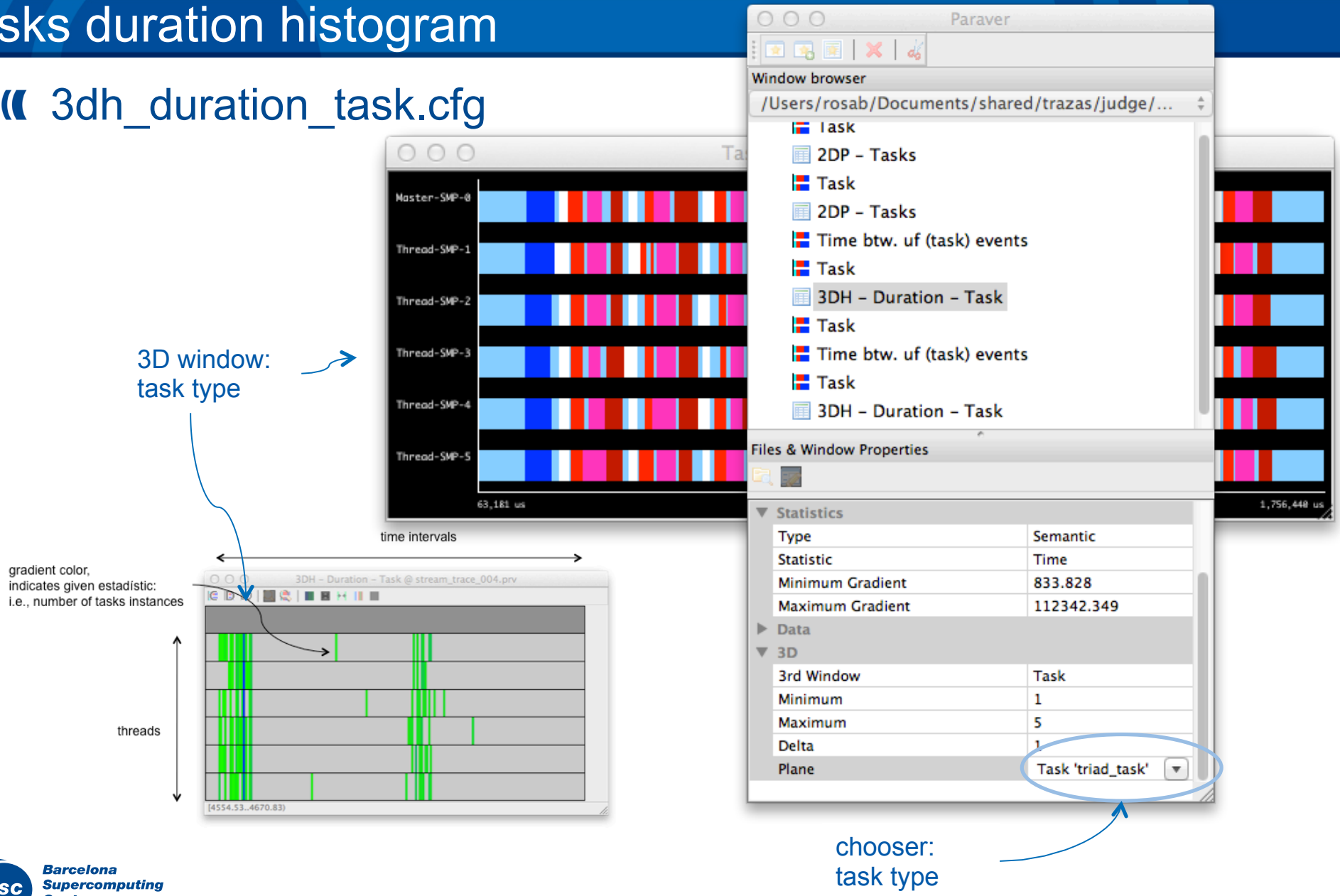
Tasks duration histogram

- 3dh_duration_task.cfg



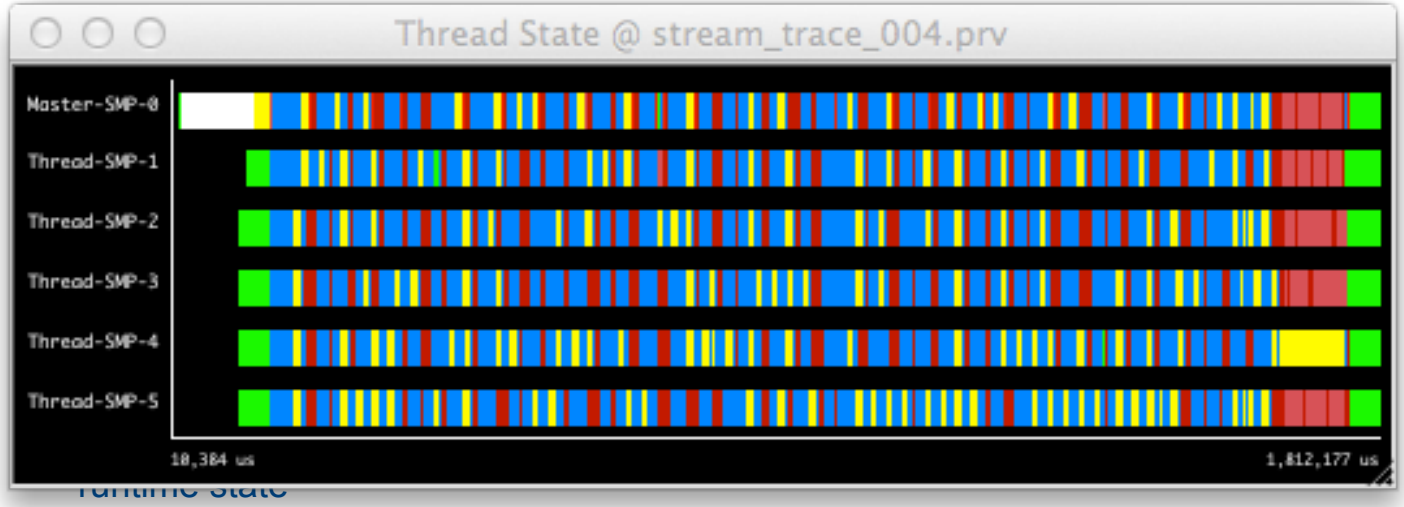
Tasks duration histogram

- 3dh_duration_task.cfg

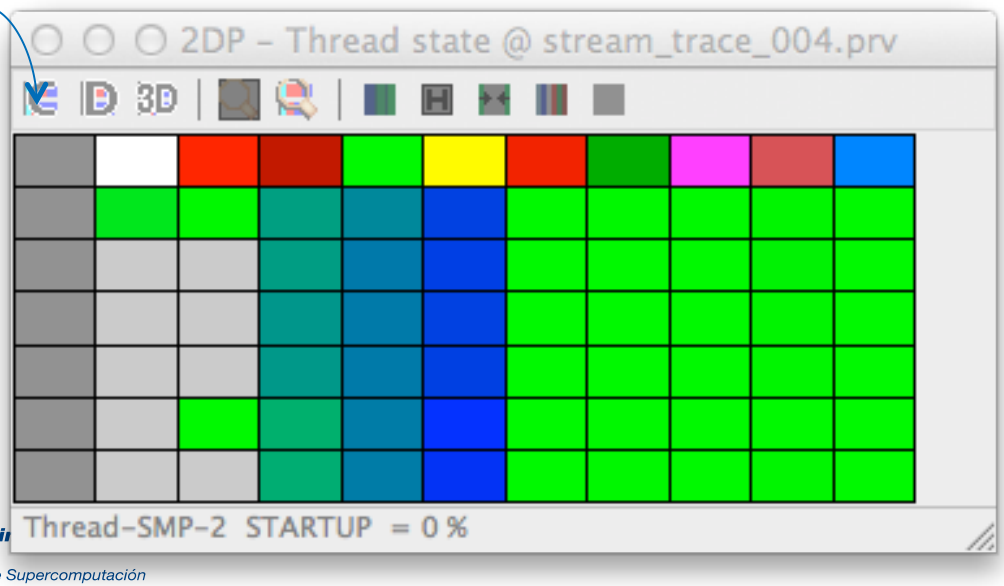


Threads state profile

- 2dp_threads_state.cfg



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



threads

Generating the task graph

- Compile with --instrument
- export NX_INSTRUMENTATION=graph
- export OMP_NUM_THREADS=1